Energistics Transfer Protocol (ETP) Specification v1.2

ETP Overview	Energistics Transport Protocol (ETP) is a data-transfer specification that enables the efficient transmission of data between applications, including realtime streaming and CRUD operations. ETP is a component in the Energistics Common Technical Architecture and can be used with other Energistics data model specifications, such as WITSML, RESQML, and PRODML.
Version of Standard	1.2
Version of Document	1.1
Date published	**September 27, 2021**
Prepared by	Energistics and the Architecture Team
Document type	Specification
Abstract	Describes how ETP works, identifies requirements, and provides examples of related message schemas. Produced in part from contents of the ETP UML model.
Language	U.S. English
Keywords	standards, energy, data, information, process, transfer protocol

Acknowledgements

Energistics would like to thank the Energistics Architecture Team, who drove ETP design, and the members of the WITSML, RESQML and PRODML Special Interest Groups (SIGs) who participate in the design, testing, review, and implementation of ETP.

Amendment History			
Standard Version	**Document Version**	**Date**	**Comment**
1.2	1.1	Sept 27, 2021	**Documentation udpate (clarification) only. No schema change.** For channel data protocols: • Content was added to clarify that the primary index is or must be first, for example, in an array of channel data. • For range replace messages, content was added to explicitly state expected data and order (similar to content for ***ChannelData*** messages). Chapters/protocols impacted: • Chapter **6**, **ChannelStreaming (Protocol 1)** (Sections: 6.1.1, 6.2.2 (Row 3), 6.3.4) • Chapter **7**, **ChannelDataFrame (Protocol 2)** (Sections: 7.2.2 (Row 8), 7.3.2, 7.3.6) • Chapter **19**, **ChannelSubscribe (Protocol 21)** (Sections 19.2.2 (Row 7), 19.3.5, 19.3.7, 19.3.11) • Chapter **20**, **ChannelDataLoad (Protocol 22)** (Sections 20.2.2. (Row 6), 20.3.6, 20.3.7) • Chapter **23**, **ETP Datatypes** (Section 23.33.7)
1.2	1.0	Sept. 9, 2021	Publication of ETP v1.2. Version v1.2 is an extensive redesign and expansion of ETP v1.1. For the summary of changes, see Section **2.1**.

Table of Contents

1 Introduction to ETP

Energistics Transfer Protocol (ETP) is a data transfer specification that enables the efficient transfer of data between two software applications (endpoints), which includes real-time streaming. ETP has been specifically envisioned and designed to meet the unique needs of the upstream oil and gas industry and, more specifically, to facilitate the exchange of data in the Energistics family of data standards, which includes: WITSML (well/drilling), RESQML (earth/reservoir modeling), PRODML (production), and EML (the data objects defined in Energistics *common*, which is shared by the other three domain standards). Initially designed to be the API for WITSML v2.0, ETP is now part of the Energistics Common Technical Architecture (CTA).

ETP defines a publish/subscribe mechanism so that data receivers do not have to poll for data and can receive new data as soon as they are available from a data provider, which reduces data on the wire and improves data transmission efficiency and latency. Additionally, ETP functionality includes data discovery, real-time streaming, store (CRUD) operations, and historical data queries, among others.

- For the list of protocols published in the current version of ETP and list of changes since the previous version, see Chapter **2**.
- For an overview of how ETP works, see Chapter **3**.
  - Section **3.1** is a big picture overview that defines the main concepts and constructs in ETP and how those "pieces" work together; both developers and business people who want a high-level understanding of "how ETP works" should find it useful.
  - The remaining sections in Chapter 3 are details for developers.

1.1 Working with Different Energistics Data Models

- ETP is supported for version 2.0 or higher of all Energistics domain data models (i.e., WITSML, RESQML, and PRODML, which are informally referred to as "the MLs"). In the download package and set of schemas, each ML has a folder named Energistics *common*, which contains a set of data objects shared by the domain standards; Energistics common has a namespace that begins *EML*.
- In a commercial implementation, you CAN NOT reference an object from an ML that has not yet been released.

1.2 Support for Multiple Versions of ETP

Beginning with ETP version 1.2, changes have been made to how ETP is versioned. This change was made so that it is easier for developers to implement multiple versions of ETP in the same codebase. The main change is the visible use of a 2-digit version number in the namespace.

- **ETP v1.2 Namespace:** *Energistics.Etp.v12* **EXAMPLE:** *Energistics.Etp.v12.Datatypes*
- **ETP v1.1 Namespace:** *Energistics* **EXAMPLE:** *Energistics.Datatypes*

To determine which version(s) of ETP are available from a given server and the capabilities of each version, see Section **4.3.**

IMPORTANT: Both endpoints in an ETP session MUST use the same version of ETP.

1.3 Overview of Supported Use Cases

Use cases for the business problems that Energistics standards help to solve are defined by the individual domain standards—WITSML, RESQML and PRODML. ETP defines how these domain standards "move data" in support of those use cases. But that "data movement" can be aggregated into higher level use cases, that can help you better understand the role of ETP, its design, and how to implement it.

ETP supports these data movement use cases:

1. **Rigsite Aggregation:** Several ETP clients and servers, some possibly without external internet access, connecting at a rigsite to exchange data.
2. **Rig-to-Shore:** A single data store located at a rig site exchanging data with an off-site data store, possibly in the cloud or a data center.
3. **Data Center Replication:** The contents of a central data store are (partially) replicated to another central data store.
4. **Real-Time Monitoring and Calculations:** The contents of a central data store are monitored in real-time, sometimes with calculated data values written back to the central data store.
5. **Data Import/Export:** Ad-hoc bulk imports and exports of data into / out of central data stores.
6. **Peer-to-Peer Data Sharing:** Data exchange between end-user applications running in end-user environments.

These use cases share some common features, which can include:

- **Potentially long-lived sessions:** An ETP session (which represents a single WebSocket connection) may be expected to last anywhere from minutes to months.
- **Dynamic data:** Over the lifetime of a session, many changes—including deletions, additions, authorizations and "de-authorizations"—may happen to data available in the session’s endpoints.

The variations in these use cases also fall into broad categories, which also impact the design and implementation of ETP. These variations include:

- **End-User Driven:** Scenarios where an end user is using an application that is an ETP client connected to an ETP server.
- **Machine-to-Machine:** Scenarios where a background service is operating an ETP client connected to an ETP server.
- **Partner Data Sharing:** Scenarios where the ETP clients and servers belong to unrelated companies.
- **Reverse Data Flows:** Scenarios where ETP clients act as data stores and ETP servers act as data customers.

For more information, about:

- High-level ETP data replication use cases and workflows, see **Appendix: Data Replication and Outage Recovery Workflows**.
- Domain-specific use cases, see the relevant domain documentation.

1.4 ETP Design Principles

As in all of its design efforts, Energistics aims to leverage existing relevant IT and industry standards; those used in ETP are referenced throughout this document. **EXAMPLES:**

- As its transport mechanism, ETP uses WebSocket, which is a protocol standardized by the Internet Engineering Task Force (IETF) as RFC 6455 (http:/tools.ietf.org/html/rfc6455).
- For serialization of messages, ETP follows a subset of the Apache Avro specification (http://avro.apache.org/docs/current/spec.html).
- For query functionality, ETP uses an OData-like syntax based on a subset of the Open Data Protocol (OData) query string syntax, specifically OData v4.0, which is an OASIS standard (https://www.oasis-open.org/standards#odatav4.0).

In designing ETP protocols, Energistics and its various technical working teams aim to honor well known design principles of software, such as:

1. Single responsibility: https://en.wikipedia.org/wiki/Single_responsibility_principle
2. Don't repeat yourself: https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

Additionally, ETP has these specific principles:

3. ETP communication is carried out through the asynchronous exchange of messages between two endpoints (e.g., a client and server) and only 2 endpoints (no multicasting).
4. ETP is organized into a set of sub-protocols, each of which has a specific purpose.
    a. **EXAMPLE:** Core (Protocol 0) is the ETP sub-protocol to create and manage an ETP session; Discovery (Protocol 3) is for finding data objects in a store; Store (Protocol 4) is for CRUD operations for data objects.
    b. **EXAMPLE:** Some sub-protocols have been developed to work with specific types of data. For example, Store (Protocol 4) operates on data objects, which includes channels. To add a channel, use Store (Protocol 4). However, to move the data in a channel, ETP provides these protocols: ChannelStreaming (Protocol 1) (for simple streamers like a sensor); ChannelDataFrame (Protocol 2) (for getting a set of channels in rows); ChannelSubscribe (Protocol 21) (for more subscribe and more sophisticated read/get than Protocol 1); and ChannelDataLoad (Protocol 22) (for write/put channel data operations).
5. Unless otherwise specified, message names, key words, etc. identified by this specification are case-INSENSITIVE.
    a. An important exception to this rule is the keys in Avro maps, which are ALWAYS case sensitive.
6. To allow for the smallest, most efficient binary transfers, ETP regularly uses a pattern of assigning numeric identifiers in addition to human-readable names. **EXAMPLES:**
    a. Each Protocol has both a human-readable name and a number (e.g., the Core protocol is Protocol 0, and the Store protocol is Protocol 4).
    b. Each ETP message type is has a human-readable message name and an assigned number.
    c. For high-frequency messages with small payloads, like streaming Channel data, longer identifiers, such as URIs, are replaced with integer identifiers, which are used in subsequent operations in the sub-protocol for an ETP session.
7. ETP includes the notion of roles. Roles govern behavior within an ETP sub-protocol and define sets of functionality that can be associated with a specific role. For each sub-protocol in ETP, this specification identifies the two allowable roles (which for most sub-protocols are "customer" and "store"). For more information on ETP roles, see Section **3.1.2.**
8. ETP implements “upsert” semantics for data objects, which means any change to a data object requires a full replacement of the data object. Partial updates of individual fields is not supported.

a. This principle does not extend to bulk data such as channel data, growing object parts, and array data. ETP provides support for adding, editing, or removing subsets of bulk data in specialized protocols.

1.4.1 Design Decisions for ETP v1.2

This section lists design decisions that were made for ETP v1.2, which implementers must be aware of.

1. Security and entitlements are intentionally outside the scope of ETP aside from a minimum set of authorization functionality to facilitate interoperability.
2. Support for eventual consistency relies heavily on endpoint-provided timestamps. This approach was chosen as a reasonable tradeoff between implementation complexity and the ability to automatically recover from most real-world causes of data outages.
3. ETP does NOT explicitly handle multi-master updates to a data object, for example, 2 applications trying to write to the same data object. In the current design, the "last write wins".
4. ETP does not provide any features to rate limit incoming messages.

1.5 Document Details

This specification is intended for IT and software professionals who want to implement ETP. A basic understanding of the technology and related general concepts is assumed. For a detailed explanation of key ETP and Energistics concepts, see Chapter **3**.

1.5.1 How to Use This Document (IMPORTANT: Read This!)

Each ETP sub-protocol has its own chapter. The intent is for each protocol-specific chapter to be the starting point for everything you need to know about each specific sub-protocol—including the messages in that protocol, basic message sequence, and functional requirements—but links to other relevant information—such as standard behavior across all of ETP, definitions, etc.—are provided.

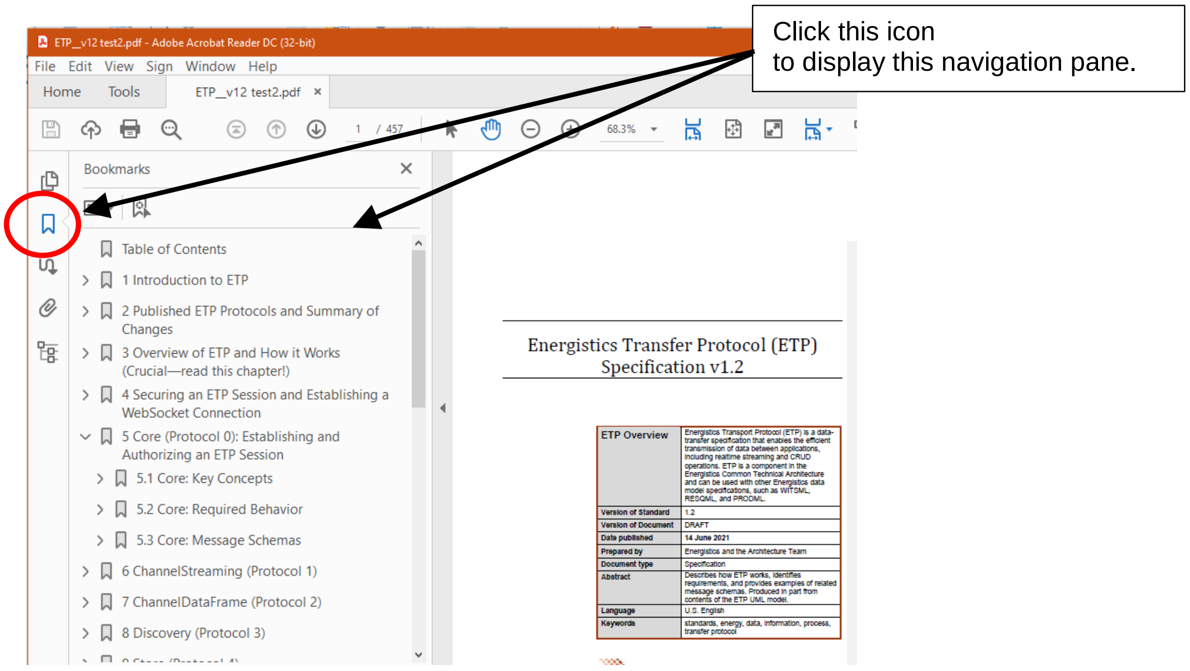
This document is organized as follows:

- Each sub-protocol defined by ETP has its own chapter, which contains all the key information about that sub-protocol, with each chapter structured in the same consistent way to include:
  - **Introduction** (purpose, scope, etc.) and relationship to other ETP sub-protocols.
  - **Message sequence for the main tasks performed in a sub-protocol**, including request and required response patterns; use of ETP-defined endpoint, data object and protocol capabilities; and error conditions and related error messages.
  - **Behavior requirements** (in addition to those defined in the message sequence section).
  - **Sample schemas of the messages defined in a sub-protocol** (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes definitions of each field in a schema.
- Chapter **3, Overview of ETP and How it Works,** has a lot of important principles and concepts that apply across ETP. The relevant parts of Chapter 3 are NOT repeated in the protocol-specific chapters. *To fully understand a protocol, you MUST use both Chapter 3 and the protocol's specific chapter.*

RECOMMENDATION: This document was designed to be used primarily as a PDF, with extensive linked cross references. If you choose to actually print a hard copy of this document (which many developers have said they will do), you will still need the PDF to navigate the links; see the next section.

1.5.2 Recommendation for Using the PDF

With 17 defined sub-protocols (each one with its own chapter) and additional chapters and appendixes, this ETP Specification is a complete, detailed, and lengthy document. To help more easily navigate the document, we recommend clicking on the circled icon in your PDF navigation pane (as shown in the screenshot below). When you do that, it displays the navigation pane shown, which makes it easier to explore the contents and move throughout the document. The document also provides extensive in-line links to related content.


Click this icon
to display this navigation pane.

ETP_v12 test2.pdf - Adobe Acrobat Reader DC (32-bit)
File Edit View Sign Window Help
Home Tools ETP_v12 test2.pdf
1 / 457
68.3%
Bookmarks
Table of Contents
1 Introduction to ETP
2 Published ETP Protocols and Summary of Changes
3 Overview of ETP and How it Works (Crucial—read this chapter!)
4 Securing an ETP Session and Establishing a WebSocket Connection
5 Core (Protocol 0): Establishing and Authorizing an ETP Session
5.1 Core: Key Concepts
5.2 Core: Required Behavior
5.3 Core: Message Schemas
6 ChannelStreaming (Protocol 1)
7 ChannelDataFrame (Protocol 2)
8 Discovery (Protocol 3)

Energistics Transfer Protocol (ETP)
Specification v1.2

ETP Overview	Energistics Transport Protocol (ETP) is a data-transfer specification that enables the efficient transmission of data between applications, including realtime streaming and CRUD operations. ETP is a component in the Energistics Common Technical Architecture and can be used with other Energistics data model specifications, such as WITSML, RESQML, and PRODML.
Version of Standard	1.2
Version of Document	DRAFT
Date published	14 June 2021
Prepared by	Energistics and the Architecture Team
Document type	Specification
Abstract	Describes how ETP works, identifies requirements, and provides examples of related message schemas. Produced in part from contents of the ETP UML model.
Language	U.S. English
Keywords	standards, energy, data, information, process, transfer protocol

1.5.3 Parts of this Document Are Created from the ETP UML Model

ETP has been designed and developed using UML® implemented with Enterprise Architect (EA), a UML modeling tool from Sparx Systems. The schemas and some of the content in this specification (the example message schemas for each protocol and Datatypes in Chapter **23**) have been generated from the UML model.

- If any discrepancy exists between the schema and the specification, the schema is the primary source (though all content should be consistent because it is produced from the same source.) **NOTE:** Only this document provides definitions of data fields in the schemas.
- Content in this specification should be considered "normative" unless otherwise specified.

1.5.4 Documentation Conventions

1. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in RFC 2119. (http://www.ietf.org/rfc/rfc2119.txt).
2. Text formatting conventions:
   a. References to other chapters or sections in this document are hyper-links that appear in **bolded blue text**.
      i. Links generated in Enterprise Architect (UML modeling tool from which schemas are generated) and HTTP links use the standard convention (blue text, underlined).

   b. Message names are in bold, italic text (and Pascal case); **EXAMPLE:** the ***OpenSession*** message.

   c. Field names are in italics (and camel case); **EXAMPLE:** the *serverInstanceId*.

3. **Energistics domain standards**—WITSML (well/drilling), RESQML (earth/reservoir modeling), PRODML (production operations and reporting)—are informally referred to as the "MLs".

   a. Name spaces for the MLs include the ML name and version number. **EXAMPLE:** witsml20.

   b. Each domain standard has a package of shared data objects defined in Energistics *common* (which, in this document, is always referred to as shown here: Energistics *common*). Objects defined in Energistics *common* have a namespace of eml plus the version of common **EXAMPLE:** For Energistics *common* v2.1 the namespace is eml21.

4. **Error codes.** For brevity in this specification, when an error condition is described, the text states "send error *Name* (*N)*" where *Name* and *N* are actual error code names and numbers, such as "send error EUNSUPPORTED_PROTOCOL (4)" as defined by this specification (see Chapter **24**). The error code is sent in the ***Protocol Exception*** message, which is defined in Core (Protocol 0) but is used in any protocol when an error occurs. For more information, see Section **3.7.2.1** and Section **5.3.8**.

5. **Extensive use of numbering.** In addition to chapter and section numbering, main steps, paragraphs, table rows, and key points are numbered in this document.

   a. In some cases (**EXAMPLE:** The task/message sequence section in each protocol-specific chapter) the numbers are used to show sequence.

   b. In other cases, items have been numbered for easy reference (i.e., when discussing with a colleague or reporting an issue, you can refer to "Section 3.7.3, Paragraph 2.b.ii"). Use of numbering makes it easier to link to very specific content; when possible, that has been done.

6. Energistics documentation is produced using U.S. English spelling and punctuation conventions.

1.6 ETP Resources Available for Download

ETP leverages existing information technology standards, for example, the WebSocket Protocol, which is published by the Internet Engineering Task Force (IETF) and OData, which is published by OASIS. Links to those and other relevant standards are provided elsewhere throughout this specification.

The table below list resources from Energistics that are available for implementing ETP, all of which are freely available to everyone, from the Energistics website: https://www.energistics.org/download-standards/. Resources are included in the ETP download, unless otherwise specified.

	Document/Resource	**Description**
1.	*ETP Specification v1.2* (This document)	Provides an overview or ETP, its business purpose, supported use cases, design, etc. It's located in the **doc** folder of the ETP download package. Defines key concepts, messages, field definitions, and behaviors of ETP. For full understanding of ETP, the specification MUST be used in conjunction with the schemas. **NOTE:** Only this document provides the definitions of the data fields in the schemas.
2.	Schemas	Avro schemas as described in this document. The download package organizes the ETP schemas into 2 main groups (folders) plus a standalone file: • **Protocols:** A folder for each ETP sub-protocol, which contains the message schemas for messages defined in those protocols. • **Datatypes:** A set of folders for the low-level data structures, which are used to define the ETP messages. It contains data types defined by both Avro and ETP. • **etp.apvr** file: Is a single file that contains all schemas.
3.	Proxy classes	The **src** folder contains proxy classes for the following: • C# • Java
4.	ETP DevKit (NOT in the ETP download)	A .NET library providing a common foundation and the basic infrastructure for implementing ETP. For more information and to download a copy, go to this link at the Energistics website: https://www.energistics.org/developer-resources/
5.	ML-specific implementation specifications (These will be made available when published.)	Each version of each Energistics domain standard (i.e., WITSML, RESQML and PRODML) has or will have its own implementation specification that provides any ML-specific details required to use a particular version of ETP with a particular version of an ML/data model.

2 Published ETP Protocols and Summary of Changes

The current version of ETP (ETP v1.2) includes the ETP sub-protocols listed in the table below. (Gaps in numbering in the table below mean that the protocol number has been assigned, but the protocol is not yet ready for publication.)

- For an overview of how ETP works—including key concepts, technology, and functionality patterns used throughout ETP—see Chapter **3**.
- For ETP design principles and decisions, see Section **1.4**.
- For the content that defines each of these protocols, see subsequent chapters in this specification. The table below links to the protocol-specific chapter.

This chapter also lists a summary of changes (see Section **2.1**).

Protocol Name and Number	Description
Core (Protocol 0): Establishing and Authorizing an ETP Session	Creates, manages and optionally authorizes ETP sessions.
ChannelStreaming (Protocol 1)	Minimalist streaming functionality for scenarios like smart sensors streaming data. For richer streaming functionality, see ChannelSubscribe (Protocol 21) and ChannelDataLoad (Protocol 22).
ChannelDataFrame (Protocol 2)	Gets channel data from a store in "rows". Supports the log on-disk use case.
Discovery (Protocol 3)	Enables store customers to enumerate and understand the contents of a store of data objects as a graph.
Store (Protocol 4)	Performs CRUD operations (create, read, update and delete) on data objects in a store.
StoreNotification (Protocol 5)	Allows store customers to receive notification of changes to data objects in the store in an event-driven manner, resulting from events/operations in Protocol 4.
GrowingObject (Protocol 6)	Manages the growing parts of data objects that are index-based (i.e., time and depth) other than channels. Also enables operations (adds and updates) on the growing data object header.
GrowingObjectNotification (Protocol 7)	Allows a store customer to receive notifications of changes to the growing parts of growing data objects in a store, in an event-driven manner, resulting from operations in Protocol 6.
Protocol 8: DEPRECATED	Deprecated and implemented as custom protocol, 2100.This protocol number will never be reused.
DataArray (Protocol 9)	Transfers large, binary arrays of data, which Energistics domain standards typically store using HDF5.
DiscoveryQuery (Protocol 13)	Query for resources with OData-like syntax (main discovery behavior is defined in Protocol 3).
StoreQuery (Protocol 14)	Query for data objects with OData-like syntax (main store behavior is defined in Protocol 4).
GrowingObjectQuery (Protocol 16)	Query for parts in a growing data object using OData-like syntax (main growing data object behavior is defined in Protocol 6).
Transaction (Protocol 18)	Provides high level support for transactions on operations in other protocols, especially Protocols 4 and 9 (typically associated with earth modeling/RESQML).
ChannelSubscribe (Protocol 21)	Provides read/get data behavior for channels with standard publish/subscribe behavior for customers to connect to a store (server) and receive new channel data as available (streaming).
ChannelDataLoad (Protocol 22)	Provides write/put data behavior for channels, allowing one endpoint to push/load/stream data to another endpoint.
Dataspace (Protocol 24)	Used to discover dataspaces in a store. After discovering dataspaces, use Discovery (Protocol 3) to discover objects in the dataspace.

Protocol Name and Number	Description
SupportedTypes (Protocol 25)	Enables store customers to discover a store's data model, to dynamically understand what object types are *possible* in the store at a given location in the data model (though the store may have no data for these object types), without prior knowledge of the overall data model and graph connectivity.
Protocol 26 – 1999	Undefined. Reserved for future use.
Protocol 2001+	Custom. Available for custom use by individual companies (not Energistics).
Protocol 2100: WitsmlSoap	In ETP v1.1, this protocol was published as Protocol 8. It is now a custom protocol published by an Energistics member company. .

2.1 Summary of Changes from ETP v1.1 to v1.2

The design changes from ETP v1.1 to v1.2 are significant. These changes were made in efforts to make the design more consistent and robust, based on feedback from real-world use, plus adding key functionality to support a broader range of use cases, include increased reliability (i.e., avoiding data loss). Additionally, the design was improved to make ETP work consistently across all Energistics domain standards, WITSML, RESQML and PRODML.

There are many more protocols in ETP v1.2, which is by design, in efforts to support adoption and implementation of ETP.

A conscious design decision was to not "overload" any one protocol with too much functionality and to avoid or minimize any interdependency of protocols; this approach allows implementers to implement only the functionality they need. But for consistency, when you implement an ETP protocol, you MUST implement all of it (i.e., you must support all messages and functional requirements).

EXAMPLES: A discovery process for dataspaces is in its own protocol (Dataspaces (Protocol 24)) because for many use cases, discovery of dataspaces is not needed. Also query behavior was put into separate "companion" protocols so that support of query behavior could be optional (e.g., DiscoveryQuery (Protocol 13) contains the query behavior/capabilities for Discovery (Protocol 3), and StoreQuery (Protocol 14) contains the query behavior/capabilities for Store (Protocol 4)).

For the complete list of ETP design principles, see Section **1.3**.

This section provides a summary of the key changes.

2.1.1 The Specification Document has been Reorganized and Improved

Key changes:

- Each sub-protocol defined by ETP has its own chapter, which contains all the key information about that sub-protocol, with each chapter structured in the same consistent way to include:
  - **Introduction** (purpose, scope, etc.) and relationship to other ETP sub-protocols.
  - **Message sequence for the main tasks performed in a sub-protocol**, including request and required response patterns; use of ETP-defined endpoint, data object and protocol capabilities; and error conditions and related error messages.
  - **Behavior requirements** (in addition to those defined in the message sequence section).
  - **Sample schemas of the messages defined in a sub-protocol** (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes definitions of each field in a schema.
- **New Chapter:** Chapter **3**, **Overview of ETP and How it Works** explains important concepts and patterns applicable across ETP. **RECOMMENDATION:** *Familiarize yourself with this chapter; complete understanding of each individual ETP sub-protocol relies on information in Chapter 3*.
- Other organizational changes and cross references in an effort to improve the usability of the document.

- **New Appendixes:**
  - **Appendix: Energistics Identifiers** documents the specific requirements for identifying Energistics data objects in the domain standards and in ETP, which is predominantly using URIs, which most use the formats specified in the appendix, which are referred to as the canonical Energistics URIs. **NOTE:** For ETP v1.2, this content supersedes the *Energistics Identifier Specification, v4.0*.
  - **Appendix: Data Replication and Outage Recovery Workflows** describes high-level workflows for the stated tasks and also provides additional information about why new features have been added to ETP v1.2 (for example, the *storeLastWrite* field, which is a key component for these workflows) how some of the sub-protocols and new features are intended to work.
  - **Appendix: Security Requirements and Rationale for the Current Approach** provides a high-level summary of requirements for the new security design and a brief explanation as to why the other security standards that were considered were not selected.

2.1.2 Things that Have Been Removed from ETP

This section lists things (functionality/features) that have been removed since ETP v1.1. The bulleted list below are the simpler changes; the sub-sections are for things that were removed and require more explanation.

- Nullable unions removed for strings, arrays and maps (and other types that are natively nullable).
- Growing object parts can no longer be streamed using ChannelStreaming (Protocol 1) (as was done in ETP v1.1). GrowingObject (Protocol 6) has now been significantly enhanced to handle all operations for growing object parts; see Section **2.1.3**).
- Retained notifications. Notifications are now only sent for changes that happen while a session is established. New mechanisms have been added to support efficiently discovering changes that happened while disconnected.
- Removed MIME types for object types and now use data object types that are based on OData qualified types. This change resulted in a change to the URI format. For more information, see Section **2.1.3.1** and **Appendix: Energistics Identifiers**.

2.1.2.1 Stability Indexes

Up to v1.1, ETP used stability indexes (a concept borrowed from Node.js API), which let us mark protocols on a scale of "Experimental" to "Stable". The idea was to allow evolutionary development of the specification, while allowing implementers to use with confidence the portions that are stable.

In reality, these indexes weren't as practical or helpful as was hoped, so they have been removed. Now if content is published in the specification, it is considered normative and ready for implementation. Draft content may be published in a separate document, so that it is available for people to review and test, until it is ready to be published.

2.1.2.2 Protocol 8 (WitsmlSoap) Deprecated/Moved to Custom Protocol

Protocol 8 has been deprecated and implemented as custom protocol, 2100. Protocol 8 (number) will never be reused.

2.1.2.3 Session Survivability Functionality

All previous behaviors associated with "remembering" things across connections have been removed. There is no more session state maintained between connections. ETP servers are now 'stateless'. Individual protocols describe behavior for reconnecting and "catching up". **NOTE:** The domain data, in some cases, is still required to remember some state, such as deletions or when something changed. Those behaviors are described in the individual protocols, in **Appendix: Data Replication and Outage Recovery Workflows**, and additional information may be provided in the companion ML-specific ETP implementation specifications.

2.1.3 Improved/Redesigned ETP Sub-Protocols and New Features

The table below lists protocols that were officially in ETP v1.1 and a summary of changes. However, the protocols may have been "experimental" (the former stability index as described in Section **2.1.2.1**) or they have been changed/improved significantly in ETP v1.2, as part of the overall design and an effort to implement consistent patterns across protocols and make them work for all Energistics data models.

Protocol	**Description of Change**
Core (Protocol 0)	• Some new data fields have been added to existing messages, for example, timestamps to support clock-based eventual consistency (data replication) workflows and changed or new fields to support new security behavior (see Section **2.1.3.5**). ○ ***RenewToken*** message has been renamed to ***Authorize*** and a new ***AuthorizeResponse*** message has been added; these messages are used for initial authorization of an ETP session and for renewal of Bearer Tokens. • New ***Ping*** and ***Pong*** messages to also support clock-based eventual consistency (data replication) workflows. • Support for multiple versions of ETP: ○ Server can support both ETP v1.1 and ETP v1.2. ○ Client can choose which version of ETP it wants to use. • CANNOT use different versions of ETP sub-protocols from different versions of ETP. (That is, an ETP session now is with one version of ETP and all sub-protocols in that version). • Placeholder support for exchanging data objects in JSON or other formats. • More granular object support. • More secure session identifiers. • Message compression support (vs. object compression support in ETP v1.1). • Addition of endpoint and data object capabilities (in addition to protocol capabilities). ○ WebSocket limits are exchanged and must be respected. • Error handling: ***ProtocolException*** messages now have 2 modes: ○ Single error. ○ Map of error messages relating back to a map of multiple request items (which, allows some of the requests to pass/fail (instead of the entire request failing)).
ChannelStreaming (Protocol 1)	Now for "simple streaming" only, so many messages have been removed and the message names and behavior have been simplified. The "standard streaming" capabilities have been moved to two new protocols (ChannelSubscribe (Protocol 21) and ChannelDataLoad (Protocol 22)), which are explained in Section **2.1.4**). • Removed all 'discovery' aspects previously in this protocols; all discovery operations are done using Discovery (Protocol 3). • Removed the notification aspects (channel status changes and notifications of added / removed channels); all relevant notifications are now done with StoreNotification (Protocol 5). • Data rate-throttling limits were removed. • Streaming of growing object parts is no longer allowed (as was the case in ETP v1.1).
Discovery (Protocol 3)	• Design change for the discovery operation to "walk" the data model as a graph. • Can now discover data objects (nodes on the graph) and relationships between them (edges that connect the nodes). • Changing this protocol was a major redesign to properly support cross-domain workflows and all Energistics data models. • Added support for discovering deleted objects • Moved support for dataspace discovery to Dataspaces (Protocol 24) and model discovery to SupportedTypes (Protocol 25).
Store (Protocol 4)	• Message names and functionality have been changed, significantly. Clear naming-convention patterns (see Section **3.4.2**).

Protocol	Description of Change
	• Messages can now operate on several objects at the same time: ○ ETP v1.1: ***GetObject*** message ○ ETP v1.2: ***GetDataObjects*** message • Robust support for sending and retrieving large data objects ○ Objects that exceed the WebSocket message constraints are sent using a set of new ***Chunk*** messages. • Can now create (but NOT update!) and read the complete growing data objects (data object "header" and its parts). • Behavior for data objects that contain other data objects using ByValue mechanism is clearly defined (e.g., Channels in Channel Sets). • Store role now sends success/confirmation messages in response to Put and Delete messages. • New per-data-object session capabilities let customers know what operations are supported for each object type and per-data-object-limits for sizes (e.g., how many contained data objects a container can have).
StoreNotification (Protocol 5)	This protocol was "experimental" in ETP v1.1. The design is significantly improved and consistent with other changes/patterns for ETP v1.2. • Subscriptions are now graph-based (were tree-based in v1.1). • Many more types of notifications are supported. • Notifications are only sent for changes that happen during the session. • The store can automatically subscribe the customer to notifications. • Robust handling of large data objects (using ***Chunk*** messages as in Store (Protocol 4). • In ETP v1.1, functionality that was previously in ChannelStreaming (Protocol 1) is now in StoreNotification, which includes notifications for channels added, removed and status changes.
GrowingObject (Protocol 6)	This protocol was "unstable" in ETP v1.1. The design is significantly improved and consistent with other changes/patterns for ETP v1.2. • Now the ONLY way to do CRUD operations on the PARTS of a growing data object. ○ In ETP v1.1, ChannelStreaming (Protocol 1) could be used to read parts; this behavior is no longer allowed. ○ Growing data objects and their parts can be ADDED and retrieved using Store (Protocol 4). • Supports operations on multiple parts. • Supports atomic range replace or delete of parts. • Supports operations on only the header of the growing data object (either add or updated). • Robust handling of large data objects. • Use of change annotations for more efficient recovery from unplanned outages (minimizing the need to "resend the entire data object".
DataArray (Protocol 9)	Previously published as Protocol 7 and as "experimental"; now Protocol 9. Messages names and functionality have been changed/added.

2.1.3.1 New URI Format

The URI format has been revamped. The use of MIME types have been removed and replaced with data types based on OData qualified types, which is reflected in the URI format. For all necessary information about the new format, see **Appendix: Energistics Identifiers**.

2.1.3.2 New Plural Messages

As a performance improvement based on testing of ETP v1.1, many protocol messages have been updated to allow multiple requests and responses within the same message, (which is typically done using array or map data structures). Errors can now also be communicated in the same way using these plural features in the ***ProtocolException*** message. Multipart messages are also used more widely to allow data that is too large to fit into a single WebSocket message to be sent.

For information on plural message design patterns and how they work, see Section **3.7**.

2.1.3.3 New ETP-Defined Response Messages

Most messages in all ETP sub-protocols now have an ETP-defined response messages. Many of these are called "success only" response messages and have been added to ETP to support more efficient operations of customer role software. For example, a customer now receives confirmation of when a data object has been successfully added to the store, helping the user of that application to know when they can begin additional operations on that data object.

For message-naming conventions, see Section **3.4.2**.

2.1.3.4 New Reliability Features

Reliability in ETP is about avoiding data loss. The approach for ETP reliability relies on the exchange of timestamps from the store’s clock, in several operations, in various ETP sub-protocols. Endpoints can track these timestamps, for about 24 hours, for channel data and growing data object parts and can track it for even longer for data objects.

The approach that ETP takes is eventual consistency, which is "...a characteristic of distributed computing systems such that the value for a specific data item will, given enough time without updates, be consistent across all nodes. Accordingly, the value across all nodes will be consistent with the last update that was made—eventually." (https://whatis.techtarget.com/definition/eventual-consistency#:~:text=Eventual%20consistency%20is%20a%20characteristic,that%20was%20made%20%2D%2D%20eventually).

The main use cases for ETP is server-to-server replication and catching up after unintended outage (e.g., a dropped satellite link). For a substantial overview for the intended workflows in ETP, see **Appendix: Data Replication and Outage Recovery Workflows**.

The new features added to support these workflows include those listed below, which are explained in detail in the relevant sections of this specification.

- Store-only timestamps (e.g., *storeCreated* and *storeLastWrite)* which are timestamps on the ***Resource*** record ONLY—NOT the ML-defined data object in the store (i.e., NOT the C*reation* or *LastUpdate* elements from the ***Citation*** element of the data object). For more information, see Section **3.12.5.1**.

  Several messages have one or both of these stamps and exchange them to track times (which is explained in relevant sections of the specification):

  - ***OpenSession*** and ***RequestSession*** messages in Core (Protocol 0)
  - ***Ping*** and ***Pong*** messages, which were added explicitly so that an endpoint can track the latest change time (or "high-water mark") or help determine if the store clock has changed; these message are defined in Core (Protocol 0), but may be used at any point in an operation.
  - ***Resource*** record and through it, the ***DataObject*** record (***DataObject*** record uses the ***Resource*** record) (see Sections **23.34.5** and **23.34.11**).
  - Notification messages in StoreNotification (Protocol 5) and GrowingObjectNotification (Protocol 7). (NOTE: Put operations in Store (Protocol 4) and GrowingObject (Protocol 6) may not have timestamps, but an endpoint can get them from notification messages.)
  - Change annotations, which are used in GrowingObject (Protocol 6) and ChannelSubscribe (Protocol 21)
- Ability to query for what has changed.

2.1.3.5 New Security Functionality and Requirements

ETP v1.1 supported Basic Authentication and JSON Web Tokens (JWT).

Based on concerns for the need for improved security from the Energistics community, security in ETP v1.2 has been redesigned. The new v1.2 approach still focuses on authorizing the connections between ETP applications.

Some of the biggest changes in v1.2:

- Allows authorization to happen at the ETP application layer instead of OR in addition to the HTTP transport layer.
- Supports authorization workflows for both endpoints in an ETP session (i.e., the client can authorize to the server and the server can authorize to the client).
- Has expanded the use of tokens to include any type of bearer token, not just JWT.
- Basic Authentication is no longer recommended.

For more information:

- About the requirements and how the new approach works, see Chapter **4 Securing an ETP Session and Establishing a WebSocket Connection**.
- About the specific changes to ETP v1.2 in support of this new approach, see Section **4.1.1.2**.
- About why and how the Energistics community arrived at the current approach, see **Appendix: Security Requirements and Rationale for the Current Approach**.

2.1.4 New ETP Sub-Protocols

Based on a conscious design approach (explained in the introduction to this Section **2.1**) the following new ETP sub-protocols were added to ETP v1.2.

New error codes have also been added to accommodate new behaviors; for the complete list of error codes defined in this version of ETP, see Chapter **24**.

Protocol	Description of Purpose and Features
ChannelDataFrame (Protocol 2)	Gets channel data from a store in "rows". Supports the log on-disk use case.
GrowingObjectNotification (Protocol 7)	Allows store customers to receive notification of changes to parts of growing data objects in the store in an event-driven manner, from events in Protocol 6 (GrowingObject). Where applicable, consistent in design with StoreNotification (Protocol 5)
DiscoveryQuery (Protocol 13)	Query behavior for discovery operations.
StoreQuery (Protocol 14)	Query behavior appended for store operations.
GrowingObjectQuery (Protocol 16)	Query behavior for parts within a growing data object.
Transaction (Protocol 18)	Handles messages associate with software application transactions, for example, end messages for applications that may have long, complex operations (typically associated with earth modeling/RESQML).
ChannelSubscribe (Protocol 21)	The "read/get" behavior for channel data, this protocol provides standard publish/subscribe behavior. • In ETP v1.1, some of this behavior was previously in ChannelStreaming (Protocol 1), which is now only for simple streamers. • Significant design to include efficiency and outage recovery. • Added previously missing functionality (from WITSML v1.x and ETP v1.1), including synchronization/historical change detection features.
ChannelDataLoad (Protocol 22)	The "write/put" behavior for channel data; this protocol allows an endpoint with the customer role to connect to an endpoint with the store role and push/load data to it.

Protocol	Description of Purpose and Features
	• New functionality for ETP v1.2. • Similar design to Protocol 21 (for consistency) and to include same/similar features for the "write" operations.
Dataspaces (Protocol 24)	Used to discover dataspaces in a store. After discovering dataspaces, use Discovery (Protocol 3) to discover objects in the dataspace.
SupportedTypes (Protocol 25)	Used to discover a store's data model, to dynamically understand what object types are possible in the store at a given location (though the store may have no data for these object types), without prior knowledge of the overall data model and graph connectivity.

2.1.5 Error Codes Have Been Significantly Revised

This list summarizes the changes. For the complete list of ETP-defined error codes in this version and for additional information on assigning and using codes, see Chapter 24.

- Codes are no longer scoped to individual protocols.
  - Any error code can be used in any appropriate error condition. (**NOTE:** This specification describes error conditions and the error code that MUST be used if that error condition occurs. However, the specification does NOT describe all possible error conditions that could occur. Implementers are encouraged to use their best judgement to apply the defined error codes for error conditions that are not explicitly defined.)
  - Error codes that were numbered based on the protocol in which they were defined have NOT been renumbered. (**EXAMPLE:** ENOTGROWING OBJECT (6001) is still error code 6001 but may be used wherever appropriate, for example, in GrowingObjectNotification (Protocol 7) or GrowingObjectQuery (Protocol 17).)
  - Implementers MAY specify custom error codes, which MUST be assigned negative code numbers.
- These codes have been deleted:

Deleted Error Code Name (v1.1)	Error Code to use now (v1.2)
EGROWING_PORTION_IGNORED (3005) **NOTE:** You can now add a growing object and its parts via Store (Protocol 4), but update to parts of a growing data object are no longer allowed in Protocol 4 (parts must be updated with GrowingObject (Protocol 6).	EUPDATEGROWINGOBJECT_DENIED (23) (For attempt to update parts using Protocol 4.)

- These codes have been renamed:

Former Error Code Name (v1.1)	New Error Code Name (v1.2)
EPERMISSION_DENIED (6)	EREQUEST_DENIED (6)
ETOKEN_EXPIRED (10)	EAUTHORIZATION_EXPIRED (10)
EINVALID_OBJECT (3002)	EINVALID_OBJECT (**14**)
ENOCASCADE_DELETE (3003)	ENOCASCADE_DELETE (**4003**)
EPLURAL_OBJECT (3004)	EPLURAL_OBJECT (**4004**)

- These new codes have been added:
  - ELIMIT_EXCEEDED (12)

- EMAX_TRANSACTIONS_EXCEEDED (15)
- EDATAOBJECTTYPE_NOTSUPPORTED (16)
- EMAXSIZE_EXCEEDED (17)
- EMULTIPART_CANCELLED (18)
- EINVALID_MESSAGE (19)
- EINVALID_INDEXKIND (20)
- ENOSUPPORTEDFORMATS (21)
- EREQUESTUUID_REJECTED (22)
- EUPDATEGROWINGOBJECT_DENIED (23)
- EBRACKPRESSURE_LIMIT_EXCEEDED (24)
- EBACKPRESSURE_WARNING (25)
- ETIMED_OUT (26)
- EAUTHORIZATION_REQUIRED (27)
- EAUTHORIZATION_EXPIRING (28)
- ENOSUPPORTEDDATAOBJECTTYPES (29)
- ERESPONSECOUNT_EXCEEDED (30)
- EINVALID_APPEND (31)
- EINVALID_OPERATION (32)
- ERETENTION_PERIOD_EXCEEDED (5001)
- ENOTGROWINGOBJECT (6001)

3 Overview of ETP and How it Works (Crucial—read this chapter!)

This chapter is crucial for developers who are implementing ETP. It provides an overview of the Energistics Transfer Protocol (ETP) and detailed explanation of how it works, which includes key concepts, functionality, patterns, and technology used throughout ETP.

IMPORTANT! The protocol-specific chapters in this specification provide details for each specific sub-protocol in the current version of ETP. Those chapters have been designed to work with this chapter (that is, this crucial information is NOT repeated in the protocol-specific chapters, only referenced). To understand how a sub-protocol works you need BOTH this chapter and the protocol-specific chapter. ***It is highly recommended you read this chapter first.***

Other related information:

- For the list of protocols published in this version of ETP and a summary of changes since the previous published version of ETP, see Chapter **2**.
- For the list of design principle and decisions for ETP, see Section **1.4**.
- For the list of ETP error codes, see Chapter **24**.

Specifically, this chapter:

- Provides **a "big picture" overview** of how ETP works (Section **3.1**). It provides important introductory material about ETP as it relates to communication protocols and how it uses endpoint roles. This section is also the top-down view of some of the bottom-up details provided in other sections of this chapter and Chapter **4**.
- Gives an **overview of ETP sessions** and how they work with HTTP and WebSocket connections (Section **3.2**).
- Defines and explains **server, protocol, and endpoint capabilities** which are ETP-defined parameters that are used to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle) (Section **3.3**).
- Describes the **ETP message approach** and related topics (Section **3.4**), including:
  - Overview of the Avro schemas that define ETP messages and how they are generated.
  - Architecture of messages (use of ETP-defined records to compose consistent messages).
  - General message types and related naming conventions.
- Describes the **ETP message format**, which includes a separate message header and message body that are transmitted separately on the wire to support more efficient processing (Section **3.5**).
- Describes **ETP extension mechanisms** (Section **3.6**).
- Explains some **common ETP message patterns** designed to optimize processing and data flow; these include plural messages and multipart requests, responses, and notifications (which are actually composed of multiple messages) (Section **3.7**).
- Describes **how ETP messages are serialized with Avro**, which supports binary and JSON encoding (Section **3.8**). **NOTE:** Currently, JSON encoding may be used only for internal testing and debugging.
- Describes **the message transport mechanism,** which is the WebSocket protocol (Section **3.9**).
- Gives an **overview of how data queries work** in ETP, using URI query string syntax and a tailored subset of OData (Section **3.10**).
- Provides an overview of change tracking and detection mechanisms in ETP (Section **3.11**)
- Describes how **common types of data** that are used extensively by oil and gas software—for example, variations of time (e.g., time stamps, elapsed time), units of measure (UOM) and ranges/intervals—are handled (Section **3.12**).
- Provides some troubleshooting tips (Section **3.13**).

3.1 ETP Overview: Big Picture

ETP is a communication protocol. Generically, a communication protocol can be thought of as a precise set of rules for the exchange of data between endpoints, usually in the form of messages. Message exchange in ETP is asynchronous.

ETP is a collection of messages, organized into sub-protocols (**Figure 1**, orange box). Each sub-protocol has a specific purpose, often deals with different types of data (e.g., data objects, channels, arrays, etc.), and the messages for each of these sub-protocols are designed to perform tasks specific to their respective purposes. For example:

- The purpose of Protocol 0 is to establish and manage the ETP session; its messages (such as ***RequestSession*** and ***OpenSession***) have been designed to do tasks required to create and manage a session.
- The purpose of Protocol 3 is to discover data objects and relationships between them in a store.
- The purpose of Protocol 4 is to perform CRUD operations for data objects in a store.
- The purpose of Protocol 5 is to subscribe to and receive notifications about changes to data objects.
- The purpose of Protocol 21 is to subscribe to channels and have the data streamed to you.
- For the complete list of ETP sub-protocols published in this version of ETP, see Chapter **2**, which contains links to the individual chapters for each protocol.

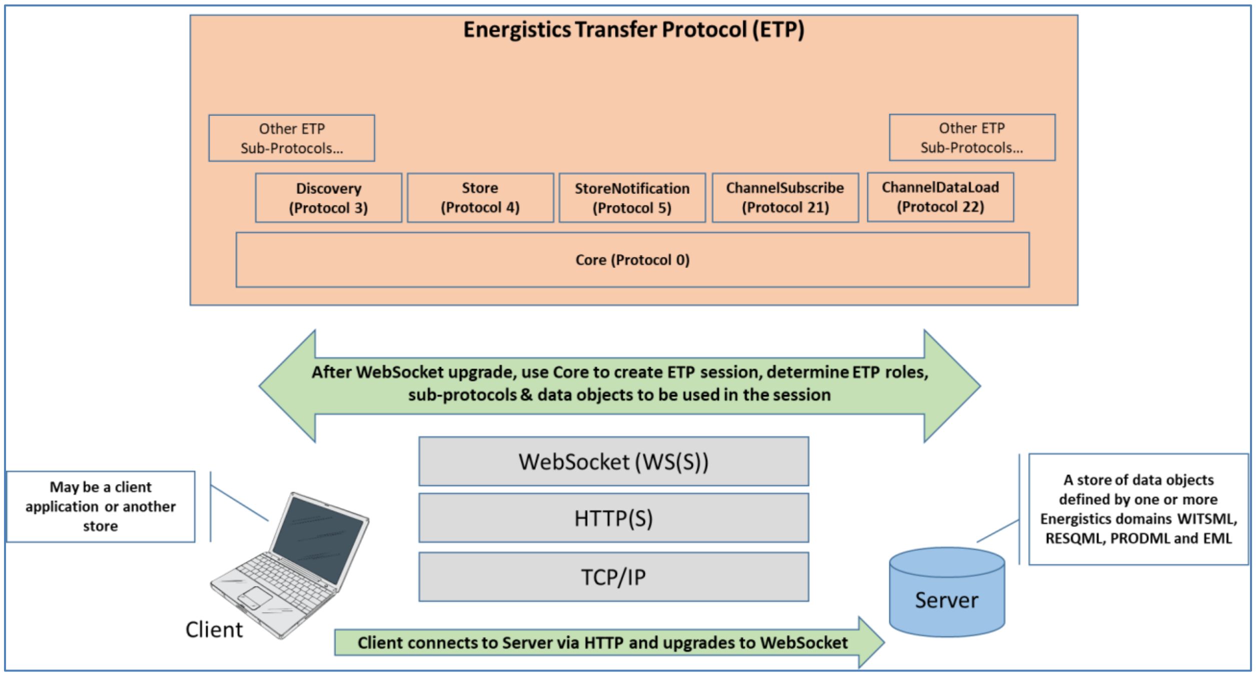

Energistics Transfer Protocol (ETP)
Other ETP Sub-Protocols...
Other ETP Sub-Protocols...
Discovery (Protocol 3)
Store (Protocol 4)
StoreNotification (Protocol 5)
ChannelSubscribe (Protocol 21)
ChannelDataLoad (Protocol 22)
Core (Protocol 0)
After WebSocket upgrade, use Core to create ETP session, determine ETP roles, sub-protocols & data objects to be used in the session
WebSocket (WS(S))
HTTP(S)
TCP/IP
May be a client application or another store
Client
A store of data objects defined by one or more Energistics domains WITSML, RESQML, PRODML and EML
Server
Client connects to Server via HTTP and upgrades to WebSocket

Figure 1: High-level schematic of ETP and how it works.

Like most modern communication protocols, ETP uses a layered approach and sits on top of the existing Transmission Control Protocol (TCP) layered model (**Figure 1**, gray boxes) (for a simplified stack diagram, see **Figure 2**). Thus, the concept of protocol is used in many contexts throughout this document, and the notion of a sub-protocol is used to discuss protocols that sit (somewhat un-intuitively) just *above* another protocol in the stack.

IMPORTANT! In this document, the terms *protocol* and *sub-protocol* are used interchangeably, about any layer, depending on context.

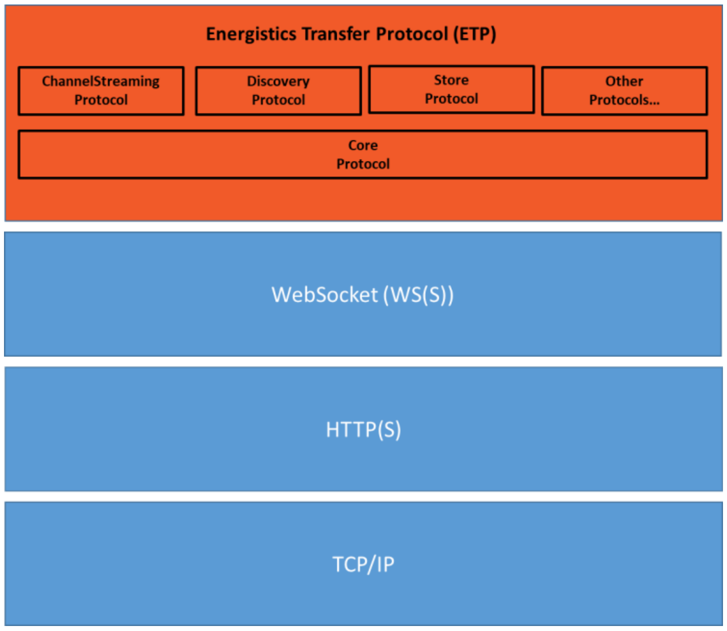


Figure 2: ETP protocol stack

Figure 2 shows that ETP is itself a sub-protocol of the WebSocket protocol and that ETP also has its own layers and sub-protocols, each designed to carry specific data that follows a specific pattern (in terms of size, frequency, and variability of data, as described above). Core (Protocol 0) has a direct connection to WebSocket and is agnostic to the various kinds of messages that are carried in each of its own sub-protocols. (For more information, see Section **3.9.1 How ETP is Bound to WebSocket**.) This layered approach allows for separation of concerns between the various parts of the stack and supports the adoption of future standards that may be developed lower in the stack.

3.1.1 Sub-Protocols defined by ETP

Each of the ETP sub-protocols has a name and numeric identifier; as in most cases, the name is for humans and the numeric identifier is for computers. For example, the Core protocol is Protocol 0, and the ChannelStreaming protocol is Protocol 1. For a complete list of ETP protocol names and numbers included in this version, see Chapter **2**. Each ETP schema header has a *protocol* field that shows the number of the protocol that defines that message (see **Figure 6** in Section **3.5.5**).

3.1.1.1 Custom Protocols

Optionally, organizations may develop and use their own custom protocols. ETP has designated a range of numbers (2000 or higher) to identify custom protocols.

3.1.1.2 Requirements for Supporting ETP Protocols

ETP has been designed for flexible implementation, so that a company can implement only the functionality it needs. However, for consistency and interoperability, implementers MUST observer these rules:

1. Core (Protocol 0) MUST be supported.
2. Support for other protocols is optional (depending on the functionality a company wants to implement in its software products).
3. If a software application supports an ETP sub-protocol, then it MUST support all messages and behaviors in that protocol.
    a. Protocols that a server supports MUST be listed in its ServerCapabilities record (for more information, see Section **4.3.1**).

b. Where ETP endpoint, protocol or data object capabilities allow, applications may advertise and impose certain limits on the functionality they support.

4. If a sending endpoint requests an action for a protocol that the receiving endpoint does not allow, the receiving endpoint MUST send either the specific error code defined by the relevant part of this specification or, if no specific error code is defined, EREQUEST_DENIED (6) or an appropriate custom error code.
RECOMMENDATION: Endpoints SHOULD supply an error message explaining why the request was denied. For example, for read-only servers (which do not allow Put operations), the explanation could be "Read-only server; operation not allowed."

   NOTE: Error codes are sent in a ***ProtocolException*** message, which is defined in Core (Protocol 0), but is used in the protocol where the error occurred. For more information about ***ProtocolException*** messages and how they work, see Section **3.7.2.1**.

3.1.2 Endpoints and Roles

Consistent with all TCP communication, ETP includes the fundamental roles of **client** and **server** (**Figure 1**, bottom). That is, in all ETP communications, one endpoint MUST be a server that is listening on a TCP port, and one endpoint MUST be a client that begins by connecting on that address and port.

The client connects to the server using WebSocket (ws or wss). (For details of this process and for information on ETP security, see Section **4.3**.)

IMPORTANT! Like WebSocket, ETP communication is strictly between two parties—with no allowance for multicast messages. That is, ETP allows for multiple sessions, but each session may have only 2 endpoints.

3.1.2.1 ETP Defines Two Roles for each Protocol

ETP also includes the notion of roles and defines 2 possible roles for each sub-protocol. Roles govern behavior within an ETP sub-protocol and define sets of functionality that can be associated with a specific role. These ETP roles are independent of the initial client server role, and the general direction of information flow is independent of this client/server relationship. This separation of direction of information flow from the client/server role is a business requirement to support WITSML use cases and workflows.

The assignment of these roles to each endpoint happens as a part of the ETP session negotiation, which happens in Core (Protocol 0) (see Chapter **5**). In general, the client begins by telling the server:

- which sub-protocol(s) it wants to use
- the named role for each sub-protocol that it wants the server to fulfill in this session. Each endpoint can have only one role per protocol, per session.

IMPORTANT: An endpoint's roles MUST be consistent across protocols within a single session. That is, in one session, an endpoint may not be a store in one protocol and a customer in another. If an endpoint needs to use different roles (e.g., both customer and store), it MUST use more than one session.

The server then responds to indicate it is able to fulfill its role; if it cannot fulfill the role, it MUST send a ***ProtocolException*** message to the client with an appropriate error code, for example, ENOROLE (1) or ENOSUPPORTEDPROTOCOLS (2).

This "client-goes-first" logic is predicated on the basis that a client knows why it is connecting to a server; whereas, a server that is capable of supporting two different roles has no way of knowing which one the client wants to use. An endpoint may only choose one role for each sub-protocol in each session.

3.1.2.2 Current Roles in ETP

The following table lists the pairs of roles currently used in ETP and the protocols in which they are used. Additionally, each protocol-specific chapter in this specification and each message schema identifies its two roles, and each section in this specification that describes an ETP message indicates the role that sends that message.

ETP Roles	Used In Protocols	
client, server	Core (0)	• The **server** role represents the endpoint that listened for and accepted the incoming WebSocket connection. • The **client** role represents the endpoint that connected to the **server** to establish the WebSocket connection.
producer, consumer	ChannelStreaming (Protocol 1)	• The **store** role represents a store of data objects and bulk data defined in the Energistics domain standards—WITSML, RESQML, or PRODML—and EML, which is the namespace for Energistics *common*, a set of data objects shared by the other domain standards. • The **customer** role represents any type of Energistics-aware software application wishing to read data from or write data to a **store**, including an application that is itself typically a server/store (for example, using ETP for data replication/synchronization workflows; for more information, see **Appendix: Data Replication and Outage Recovery Workflows**).
store, customer	All other ETP protocols	• The **producer** role represents a source that makes new data available over time for consumption. • The **consumer** role represents an application wishing to consume data produced by a **producer**.

3.1.2.3 Example of Roles and Their use from WITSML

Here is a more concrete example from WITSML of how roles work: ChannelSubscribe (Protocol 21) and ChannelDataLoad (Protocol 22) are for reading and writing channel data, respectively. Both have the same two roles: customer and store.

When establishing an ETP session (in Protocol 0), the "client" always specifies the role—but it can choose to be a store or customer on either protocol, depending on the use case/workflow. **EXAMPLE:**◦Simplistically, we could say that a rig aggregator is usually a store; a service company WITSML store can be a either a store or a customer, and a Web browser or desktop client is usually only a customer. For the rig aggregator/store interaction, either endpoint could be the "client" that initiates the ETP connection and then chooses its role, depending on whether it is going to "pull" data (using ChannelSubscribe (Protocol 21)) or "push" data (using ChannelDataLoad (Protocol 22)). (For more information, see **Appendix: Data Replication and Outage Recovery Workflows**, Section **26.2.4**.)

3.1.3 Session

ETP includes the notion of a session, which is an established WebSocket connection between a client and server that is open for a period of time. Each endpoint maintains information for the life of the session (as explained in other sections of this specification).

When the ETP session is established, the client and server (in their respective ETP protocol-specific roles) may begin using the sub-protocols and data objects negotiated in the session to perform the required operations. The operation of each ETP sub-protocol are covered in Chapters 5–22 of this document.

For more information about ETP sessions, see Section **3.2**.

IMPORTANT! ETP endpoints MUST have clocks. Workflows for reconnecting after a dropped connection and eventual consistency between stores are based on these endpoints being able to assess changes and retrieve changed (historical) data since a particular time.

- For more information about use of time and timestamps in ETP, see Section **3.12.5**.

- For more information about workflows based on using these timestamps, see **Appendix: Data Replication and Outage Recovery Workflows**.)

3.1.4 Data objects, Resources, and Identifiers (UUIDs, URIs, and UIDs)

Operations in ETP are performed on **data objects** that represent real-world business objects, like wells, horizons, or production volumes. These data objects are defined by the Energistics domain standards, WITSML, RESQML and PRODML.

However, for efficiency of operations, initial inquiries in ETP often return a **resource**, which is a lighter weight meta-object based on the content of the actual instance of a data object.

Energistics specifies and requires these main types of identifiers: UUID, URI, and UID. (For more information about Energistics identifiers, see **Appendix: Energistics Identifiers**.)

- **UUID**. In Energistics domain models, an instance of a data object is uniquely identified with a UUID. In most messages and records use of UUID must be of datatype ***Uuid*** (Section **23.6**).
- **URI**. In ETP, an instance of a data object MUST be identified with a URI.
  - Energistics specifies canonical URIs (e.g., for data objects, data spaces, and data object queries), which MUST be supported.
  - **IMPORTANT!** In most cases in this specification, when the customer has to provide a URI (for example, in a request message) it must be the canonical Energistics URI (Section **25.3.5**).
  - ETP also supports use of alternate URI formats. If an endpoint supports them, and their use is established in an ETP session, alternate URI formats may be used in subsequent requests in the ETP session.
  - For more information about rules and usage for URIs in ETP, see Section **3.7.4**.
- **UID**. In Energistics domain models, some data objects have one or more collections of sub-objects or parts. A UID uniquely identifies one sub-object or part within its collection. A UID may or may not be in the form of a UUID. **EXAMPLE:** A Trajectory has a collection of TrajectoryStations, and each TrajectoryStation has a UID that is different than the UID of all other TrajectoryStations in that Trajectory.
  - Some ETP sub-protocols use UIDs to refer directly to a specific part or sub-object within a data object.
  - Other IDs like message IDs, channel IDs and map keys are discussed elsewhere in this document.

3.1.5 ETP Messages

Each ETP sub-protocol defines a set of messages to be used to carry out operations designed for the sub-protocol. **EXAMPLE:** Store (Protocol 4) has messages designed to get objects from the store and messages to put objects in the store.

ETP messages:

- Are defined in Avro schemas and serialized using Avro (a system specifically designed for this purpose). For more information about messages, see Sections **3.4** (**ETP Message Approach**) and **3.5 (ETP Message Format)**.
- Are transported in accordance with the message framing of the WebSocket (WS) protocol, a protocol standardized by the Internet Engineering Task Force (IETF) as RFC 6455 (http:/tools.ietf.org/html/rfc6455), which allows for high-speed, full-duplex, binary communication between endpoints (primarily Web servers and browsers) using TCP and the standard HTTP(s) ports 80/443.

  NOTE: Like WebSocket, ETP communication is strictly between two parties—with no allowance for multicast messages. That is, ETP allows for multiple sessions, but each session may have only 2 endpoints. For more information on the WebSocket protocol, see Section **3.9**.

3.1.6 Security

In any communication protocol—especially one carrying sensitive, proprietary data—security is a major concern. With regard to security:

- ETP DOES NOT itself define any new security protocols. Rather, it relies on the available security mechanisms in its underlying protocols (HTTP, WebSocket, TLS, TCP, etc.).
- ETP DOES specify authorization methods (based on adoption and adaption of OAuth 2.0 and OpenID Connect Discovery v1.0), which MUST be supported by all servers for interoperability.
  - However, this approach DOES allow implementers to add custom behavior now and for future extensibility.
- ETP focuses only on authorizing the connections between ETP applications (not necessarily a device).

For more information, see Section **4.1**.

3.2 Sessions: HTTP, WebSocket and ETP

ETP includes the notion of a session, which is an established WebSocket connection between a client and server that is open for a period of time. Each endpoint maintains information for the life of the session (as explained in other sections of this specification). One benefit of a session is that it allows for a context within which various domain objects can be referenced in messages by abbreviated identifiers (which are often only meaningful/valid during a given session) as opposed to their character-based names. This approach allows for much smaller messages on the wire and more efficient processing of messages in code.

Some important facts about ETP sessions:

- An ETP session is created and maintained by ETP sub-protocol Core (Protocol 0). All messages referred to in this list are defined in Core (Protocol 0) and explained in Chapter **5**.
- Each ETP session is assigned a unique identifier (UUID), mainly to help in troubleshooting for endpoints/users. The store assigns the *sessionId*, in the ***OpenSession*** message.
- An ETP session is between 2 endpoints only; multicast messages are NOT allowed. Each endpoint assigns itself a unique instance ID (UUID) (*clientInstanceId* in the ***RequestSession*** message and *serverInstanceId* in the ***OpenSession*** message) for the session. These instance IDs can also be used for troubleshooting.
- Two endpoints may have 1 or more concurrent ETP sessions (and in fact, some workflows require more than 1 session). Each session is independent, with no built-in mechanism for cross-session coordination.
- Some identifiers (such as *messageId* and *channelId*) have context only within the scope of a session.

To establish an ETP session, a client MUST first connect to a server using WebSocket (ws or wss). To properly negotiate and establish the ETP session, ETP also specifies some optional functionality that may happen as part of the WebSocket connection.

For more information on:

- Establishing a WebSocket connection, see Section **4.3**.
- Establishing an ETP session, see Chapter **5**.

3.2.1 Why WebSocket for Transport?

When design work on ETP began (circa 2012), the Energistics Architecture Team evaluated available technology options for the ETP stack. The WebSocket protocol was selected because it was a message-based, bi-directional protocol approved by the Internet Engineering Task Force (IETF), and a protocol that could reliably deliver messages in the order in which they were sent.

3.3 Capabilities: Endpoint, Protocol, Server and Data Object

ETP defines parameters—which Energistics refers to as *capabilities*—that can be used to help prevent aberrant behavior, for example: sending oversized messages or sending more messages than an endpoint can handle.

A **capability** is a key-value pair that is composed of an ETP-defined keyword for a specific parameter (**EXAMPLES:** ChangeRetentionPeriod or ActiveTimeoutPeriod) and an endpoint-specific value for that parameter. These parameters and values are then used by the endpoints, as defined in this specification, for applicable interactions and required behavior. A client and a server can each specify its capabilities and related values.

- In some cases, ETP specifies a minimum, maximum and/or default value for the parameter.
- In some cases, the domain standard (WITSML, RESQML or PRODML) specifies appropriate parameter values.
- Individual usage guidance, default values (if applicable), and error messages if the parameter is violated are provided in this specification:
  - The "global" capabilities are documented below (see Section **3.3.2**).
  - The more specialized capabilities are documented in the protocol-specific chapters where they are used.

    NOTE: In the protocol-specific chapters, the behavior related to using the protocols is defined in the Required Behavior section (always numbered *N*.2, where *N* is the chapter number), which includes all of the operational behavior for a protocol.

IMPORTANT! The capabilities defined by ETP do not appear in any schemas; however, these capabilities ARE part of the ETP Specification. The lists of ETP-defined capabilities are maintained in the ETP Enterprise Architect model (which is used to produce message schemas and some content in this document). If these parameters are presented, they MUST be accepted and processed.

ETP has these main kinds of capabilities:

- **Endpoint**: Parameters that are applicable to an endpoint, in any protocol where it makes sense. For example, MaxWebSocketFramePayloadSize—the maximum size for a WebSocket frame that an endpoint can handle—applies to all ETP protocols that are implemented by the endpoint.
  - For the list of endpoint capabilities defined by ETP, see Section **23.4**.
  - For information about how endpoint capabilities work, see Section **3.3.1**.
- **Protocol**: Parameters that are applicable to one or more specific protocols. Each protocol-specific chapter identifies the applicable capabilities and how they are used to.
  - For the list of all protocol capabilities defined by ETP, see Section **23.5**.
  - For information about how protocol capabilities work, see Section **3.3.1**.
- **Data Object**: Parameters that allow an endpoint to specify capabilities for types of data objects. ETP includes commonly used capabilities (i.e., can the data object type be retrieved, saved or deleted) as well as some especially for ETP and Energistics data models, such as ActiveTimeOut period.
  - For the list of all data object capabilities defined by ETP, see Section **23.3**.
  - For information on how data object capabilities work, see Section **3.3.4**.

Individual ETP implementations MAY define custom or proprietary endpoint, protocol, or data object capabilities, which may include new parameters or different values for parameters specified by ETP. If an endpoint encounters a capability it does not know how to handle, it MUST IGNORE it.

The capabilities for each endpoint are exchanged and, when appropriate, negotiated when the session is established (see Chapter **5**). Servers also advertise their capabilities before clients connect with the ***ServerCapabilities*** record (see Section **4.3.1**).

3.3.1 How Protocol and Endpoint Capabilities Work

Observe these rules for capabilities:

1. ETP-defined capabilities do NOT appear in any schemas; however, they are part of the ETP Specification. (Links to the lists of all capabilities defined by ETP are available in the section **above**.)
2. When used, each parameter keyword MUST be used with the exact name shown in this document and an endpoint-specific, protocol-specific and/or data-object-specific value (key-value pair).
    a. In some cases, the same ETP-defined capability may be defined as one or more capability kind; **EXAMPLE:** MaxDataObjectSize may be an endpoint, protocol, AND data object capability. In these cases, guidance is provided about how all the different kinds are used together (i.e., the precedence for applying the kinds of capabilities).
3. If one or more of the defined capabilities is presented by an endpoint, the other endpoint in the ETP session MUST accept it (them) and process the value, and apply them to the behavior as specified in this document. That is, each ETP implementation MUST recognize and accept the capabilities defined in this specification that are relevant for the protocols, roles, objects and features the implementation supports.
    a. If an endpoint does NOT specify a value for a particular capability, then the other endpoint MUST use the default value (as specified in this document or a companion ML-specific ETP implementation specification) for that capability. **EXAMPLE:** The default value for ResponseTimeoutPeriod is 300 seconds. If a server does not specify a different value for its ResponseTimeoutPeriod, a client should expect to receive a response from a request within 300 seconds.
4. An endpoint MAY also use custom capabilities.
    a. If an endpoint does NOT understand a custom key word, it MUST ignore it (NOT send an error).
5. The client and server exchange endpoint, protocol, and data object capabilities in Core (Protocol 0) as part of establishing the ETP session. For details on session establishment, including rules for how capabilities for a session are determined, see Section **5.2**).
    a. Optionally, the client MAY "pre-discover" a server's capabilities through its ***ServerCapabilities*** record as part of establishing the WebSocket connection (see Section **4.3**).
        i. This pre-discovery is very important because after a WebSocket connection is made, it may not be possible to change certain parameters (**EXAMPLE:** WebSocket frame and message sizes). So it is important that the client understand these parameters BEFORE establishing the WebSocket connection.
    b. A client conveys its capabilities in the ***RequestSession*** message in Core (Protocol 0) as part of establishing the ETP session (see Chapter **5**).
    c. A server conveys its capabilities one of both of these methods: 1) in the ***ServerCapabilities*** (before creating the WebSocket connection; see Section **4.3.1**) and 2) in the ***OpenSession*** message in Core (Protocol 0), (see Chapter **5**).
        i. **NOTE:** Clients should be aware that, for various reasons, capability values sent in the ***OpenSession*** message may vary from those "advertised" in the ***ServerCapabilities*** record.
    d. Depending on the specifics of the capability, their exchange between endpoints serves as an advisory (here is my value; work with it, which is typically the case for server capabilities) OR a "pseudo-negotiation", where the lesser of two values may need to be used for successful operations between the two endpoints **(EXAMPLE:** MaxResponseCount (see Section◦**3.7.3.1**, Paragraph◦**4.d**).
        i. However, if one endpoint cannot comply with a specified constraint, it can always drop the connection.

3.3.2 "Global" Capabilities

ETP defines several capabilities that are used in all or most ETP sub-protocols, or that impact the operation of all or most ETP sub-protocols (such as MaxWebSocketMessagePayloadSize)—which makes them essentially "global" throughout ETP. Use of the capabilities are documented here and referenced in relevant chapters.

Additionally, ETP has two other categories of capabilities that are documented here:

- Capabilities that allow an endpoint to specify optional functionality for an ETP implementation (see Section **3.3.3 below**).
- Data object capabilities, which may be applied to any data object supported by an implementation (see Section **3.3.4 below**).

NOTE: All of the "global" capabilities listed here are endpoint capabilities (for definition, see **above**). However, not all endpoint capabilities are "global". **EXAMPLE: ChangeRetentionPeriod** is an endpoint capability but is used only by ETP sub-protocols that provide discovery of changes or deleted objects (i.e., Discovery (Protocol 3), GrowingObject (Protocol 6) and ChannelSubscribe (Protocol 21)), so is documented in relevant sub-protocol chapters.

3.3.2.1 ActiveTimeoutPeriod (Endpoint)

The WITSML domain has the notion of “active” data objects. ETP represents this with the *activeStatus* field on the ***Resource*** record. WITSML data objects that can be “active” usually also have a status element that reflects this, which, in WITSML 2.0, is typically named *GrowingStatus* or *IsActive*.

A data object that is “active” is one where updates are actively being made to the data object itself or to other data objects related to it. For channels and growing data objects, this field reflects updates to the data object’s data points or parts, respectively. For wellbores, this field reflects updates to channels or growing data objects associated with the wellbore.

- If updates are actively being made that cause a data object’s *activeStatus* to be set to true, the data object is said to be "active".
- If no such change occurs within its ActiveTimeoutPeriod, the object is said to be "inactive".
- A data object is “activated” by the first update that causes its *activeStatus* to be set to true when it was previously false.
- A data object is “deactivated” when its *activeStatus* is set to false after it was previously true.

The behavior for each data object and the relevant element in the WITSML data object that maps to the *activeStatus* field in the current version of ETP are defined in the relevant WITSML implementation specification. For WITSML 2.0, this is the *ETP v1.2 for WITSML 2.0 Implementation Specification*.

The ability to determine which data objects are "active" is important and useful in WITSML workflows. As such, ETP supports discovery and query behavior of channels and growing data objects based on their active status and displays the active status in the ***Resource***, which is returned in response messages in discovery and query operations.

A store sets a data object's *activeStatus* field based on activity (e.g., changes to parts of data points in the data object) or inactivity (e.g., no changes to parts of data points in the data object) in excess of an endpoint's or data object's value for the ActiveTimeoutPeriod capability.

- The general behavior related to exceeding this capability and setting the status to "inactive" is described below in this section.
- Behavior for setting the status to "active" is described in the protocols where that behavior occurs.

Changes in active status may also result in notifications being sent; that behavior is explained in the relevant protocol-specific chapters. The "scope" table below in this section provides a summary of the protocols where behavior is defined to set this status and the messages that display the status.

IMPORTANT: Not every store will be able to accurately track *activeStatus* over a long period of time. For example, if a store application restarts, the store may lose track of this information. **The minimum requirement to enable eventual consistency workflows is this:**

- If a store loses track of whether a given data object is “active” or “inactive”, the store MUST set the data object’s *activeStatus* to true and start the ActiveStatusTimeout.
- The store MUST also send any appropriate notifications caused by the change to *activeStatus*.

Name: Description	Type	Units Value Units	Defaults and/or MIN/MAX
ActiveTimeoutPeriod: (This is also a data object capability.) The minimum time period in seconds that a store keeps the active status (*activeStatus* field in ETP) for a data object as “active”, after the most recent update causing the data object’s active status to be set to true. For growing data objects, this is any change to its parts. For channels, this is any change to its data points. This capability can be set for an endpoint and/or for a data object. A data object-specific value overrides an endpoint-specific value.	long	second <number of seconds>	**Default:** 3,600 **MIN:** 60

SCOPE: This table summarizes protocols and messages that define behaviors related to changing and setting the *activeStatus* field, sending notifications about change in status and messages that either display the field or trigger changes to its. For details, see the relevant ETP-sub-protocol-specific chapters.

Protocols	Behavior	**Related Messages** (see note in SCOPE above)
Discovery (Protocol 3)	A customer can filter discovery operations on the *activeStatus* field (for relevant object types).	***GetResources*** ***GetResourcesResponse*** ***GetResourcesEdgesResponse***
Store (Protocol 4)	Behavior/conditions for when to set the field to "active" is specified.	***PutDataObjects*** (for growing data objects and channels)
StoreNotification (Protocol 5)	Behavior/conditions for when to send notifications based on changes to active status are specified.	***ObjectActiveStatusChanged***
GrowingObject (Protocol 6)	Behavior/conditions for when to set the field to "active" is specified.	All operations that change the parts in a growing data object
DiscoveryQuery (Protocol 13)	A customer can filter discovery operations on the *activeStatus* field (for relevant object types).	***FindResources*** ***FindResourcesResponse***
StoreQuery (Protocol 14)	Customer can query the *activeStatus* field (for relevant object types).	***FindDataObjectsResponse***
ChannelSubscribe (Protocol 21)		***GetChannelMetadataResponse*** provides the current activeStatus (on the ***ChannelMetadataRecord***)
ChannelDataLoad (Protocol 22)	Behavior/conditions for when to set the field to "active" is specified.	Operations that change channel data

REQUIRED BEHAVIOR:

1. For growing data objects, when no parts have been added, changed or deleted for the duration of the store's relevant ActiveTimeoutPeriod value, a store MUST set a growing data object's *activeStatus* field (and the data object element it maps to) to "inactive".
2. For Channel data objects, when no data points have been added, changed or deleted for the duration of the store's relevant ActiveTimeoutPeriod value, a store MUST set a channel's *activeStatus* field (and the data object element it maps to) to "inactive".
3. For other data objects (i.e., other than growing and channel data objects), when no updates have been made that cause the data object's *activeStatus* to be set to true for the duration of the store's relevant ActiveTimeoutPeriod value, a store MUST set the data object's *activeStatus* (and the data object element it maps to) to "inactive".
4. The relevant ActiveTimeoutPeriod capability is the data object capability for the type of data object affected, if set, or, if not set, it is the endpoint capability.
5. When setting a data object's *activeStatus* to "inactive", the store MUST NOT make the change sooner than the ActiveTimeoutPeriod after the most recent change that activated the data object.
    a. The store MUST make the change as soon as is practical after the ActiveTimeoutPeriod has elapsed. **RECOMMENDATION:** Change *activeStatus* within seconds after the ActiveTimeoutPeriod has elapsed.
6. **NOTIFICATION BEHAVIOR:** When a data object's *activeStatus* field changes, a store MUST send an ***ObjectActiveStatusChanged*** notification message for any relevant subscriptions. For more information, see Chapter◦**10◦StoreNotification (Protocol 5)**.

3.3.2.2 ChangePropagationPeriod (Endpoint)

Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
ChangePropagationPeriod: The maximum time period in seconds—under normal operation on an uncongested session—for these conditions: • after a change in an endpoint before that endpoint sends a change notification covering the change to any subscribed endpoint in any ETP session. • **if the change was the result of a message WITHOUT a positive response**, it is the maximum time until the change is reflected in read operations in any ETP session. • **If the change was the result of a message WITH a positive response**, it is the maximum time until the change is reflected in ETP sessions other than the session where the change was made. RECOMMENDATION: Set as short as possible (i.e., a few seconds).	long	second <number of seconds>	**Default:** 5 **MIN:**1 **MAX:** 600

SCOPE: All Protocols; All get, notification or data messages.

REQUIRED BEHAVIOR: The following rules assume normal operating conditions on an uncongested session. These rules also apply to the other endpoint within the ETP session where the change happened as well as other ETP sessions that may be active at the time. An endpoint MUST follow these rules:

1. After a change in the endpoint happens, the endpoint MUST send any notifications or data messages relating to the change no later than the endpoint's ChangePropagationPeriod value.
    a. It MAY send notifications or data messages sooner than this time.

2. When several changes to the object happen within this period, the endpoint MAY choose to send only a single notification for a data object, provided that both of these conditions are met:
   a. The endpoint does not exceed this limit.
   b. The notification accurately reflects the state of the affected object at the time the notification is sent, which MUST represent the most recent state of the object.

3.3.2.3 ResponseTimeoutPeriod (Endpoint)

Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
ResponseTimeoutPeriod: The maximum time period in seconds allowed between a request and the standalone response message or the first message in the multipart response message. The period is measured as the time between when the request message has been successfully sent via the WebSocket and when the first or only response message has been fully received via the WebSocket. When calculating this period, any *Acknowledge* messages or empty placeholder responses are ignored EXCEPT where these are the only and final response(s) to the request.	long	second <number of seconds>	**Default:** 300 **MIN:**60

SCOPE: All protocols; all request messages.

REQUIRED BEHAVIOR:
1. After receiving a request from a customer, a store MUST send the standalone response message or the first message in a multipart response no later than the value for the customer's ResponseTimeoutPeriod.
   a. If the store cannot respond within the customer's ResponseTimeoutPeriod, the store MAY cancel by sending error ETIMED_OUT (26).
   b. If the store's value for ResponseTimeoutPeriod is less than the customer's value, and the store exceeds its limit, then the store MAY cancel the response by sending error ETIMED_OUT (26).
2. If a customer receives an ETIMED_OUT error, it may indicate that the session has become congested or the store has encountered other "abnormal circumstances."

3.3.2.4 MaxDataObjectSize (Endpoint)

NOTE: MaxDataObjectSize is also a data object capability and a protocol capability. The REQUIRED BEHAVIOR section below, explains how the three types of capabilities work together.

Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
MaxDataObjectSize: (This is also a protocol and data object capability.) The maximum size in bytes of a data object allowed in a complete multipart message. Size in bytes is the size in bytes of the uncompressed string representation of the data object in the format in which it is sent or received. This capability can be set for an endpoint, a protocol, and/or a data object. If set for all three, here is how they generally work: • An object-specific value overrides an endpoint-specific value.	long	byte <number of bytes>	**MIN:** 100,000

Name: Description	Type	Units Value Units	Defaults and/or MIN/MAX
• A protocol-specific value can further lower (but NOT raise) the limit for the protocol. EXAMPLE: A store may wish to generally support sending and receiving any data object that is one megabyte or less with the exceptions of Wells that are 100 kilobytes or less and Attachments that are 5 megabytes or less. A store may further wish to limit the size of any data object sent as part of a notification in StoreNotification (Protocol 5) to 256 kilobytes.			

SCOPE: It must be used in the protocols and messages/operations listed in this table. For details, see the relevant ETP-sub-protocol-specific chapters.

Protocols	Messages/Operations
Store (Protocol 4)	Put and get operations
StoreNotification (Protocol 5)	If sending object data with notifications
GrowingObject (Protocol 6)	Operations on growing data object "headers" (i.e., "non-growing portion", which are data objects, just informally considered/called "headers" in relation to their respective parts
StoreQuery (Protocol 14)	FindDataObjectsResponse

REQUIRED BEHAVIOR:

1. In requests, a customer MUST limit the size of each data object to the value of the store's relevant MaxDataObjectSize protocol, object or endpoint capability.
   a. The limit that applies to a specific data object is the lesser of the global capability limit for that data object type, if set, and the protocol capability limit.
   b. The global capability limit for the object is the data object capability limit for the data object type if set, or, if not set, the endpoint capability limit.
2. If any data object in the request exceeds its relevant limit, a store must deny the entire request by sending error EMAXSIZE_EXCEEDED (17).
3. A store MUST limit the size of data objects in responses and notifications to the customer's protocol value for MaxDataObjectSize.
   a. If the store is sending the customer a notification about a data object that, when the data object is requested and including it in the message would exceed the customer's value for MaxDataObjectSize, the Store MUST instead send the notification without the associated data object data.
   b. If a data object exceeds the customer's MaxDataObjectSize value (limit), the customer MAY notify the store by sending error EMAXSIZE_EXCEEDED (17).

3.3.2.5 *MaxPartSize (Endpoint)*

Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
MaxPartSize: The maximum size in bytes of each data object part allowed in a standalone message or a complete multipart message. Size in bytes is the total size in bytes of the uncompressed string representation of the data object part in the format in which it is sent or received.	long	byte <number of bytes>	Min: 10,000

SCOPE: It must be used in the protocols and messages/operations listed in this table. For details, see the relevant ETP-sub-protocol-specific chapters.

Protocols	**Messages/Operations**
Store (Protocol 4)	Allowed get and put operations on growing data and its parts.
StoreNotification (Protocol 5)	If sending object data with notifications (growing object and its parts)
GrowingObject (Protocol 6)	Put and get operations on the parts of a growing data object
GrowingObjectNotification (Protocol 7)	If sending parts data with notifications
StoreQuery (Protocol 14)	***FindDataObjectsResponse*** (if the result of the query is a growing object and its parts)
GrowingObjectQuery (Protocol 16)	***FindPartsResponse*** (if the result of the query is a growing object and its parts)

REQUIRED BEHAVIOR:

1. In requests, a customer MUST limit the size of each data object part in requests to the Store's relevant MaxPartSize endpoint capability.
   a. If any data object part in the request exceeds its relevant limit, a store MUST deny the entire request by sending error EMAXSIZE_EXCEEDED (17).
2. In responses, a store MUST limit the size of data object parts and notifications to the customer's endpoint value for MaxPartSize.
   a. If a data object part exceeds the customer's MaxPartSize value (limit), the customer MAY notify the store by sending error EMAXSIZE_EXCEEDED (17).

3.3.2.6 *MaxSessionClientCount (Endpoint)*

For where this capability should be used in the WebSocket connection/ETP session establishment process, see Sections **4.3** and **5.2.1.1**.

Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
MaxSessionClientCount: The maximum count of concurrent ETP sessions that may be established for a given endpoint, by a specific client. If possible, the determination of whether this limit is exceeded should be made at the time of receiving the HTTP WebSocket upgrade or connect request based on the authorization	long	count <count of sessions>	**MIN:** 2 sessions

Name: Description	Type	Units Value Units	Defaults and/or MIN/MAX
details provided with the request. At the latest, it should be based on an authorized ***RequestSession*** message.			

REQUIRED BEHAVIOR:

1. If a new connection from a particular client may cause a server to exceed its value for MaxSessionClientCount endpoint capability, a server MAY refuse the incoming connection.
   a. If a server chooses to reject an incoming connection because it would exceed this limit:
      i. If it does this during the WebSocket connect or upgrade step, it SHOULD deny the connection or upgrade with HTTP 429: Too Many Requests.
      ii. If it does this on receiving a ***RequestSession*** message, it SHOULD deny the request by sending error ELIMIT_EXCEEDED (12).

3.3.2.7 MaxSessionGlobalCount (Endpoint)

For where this is used in the WebSocket connection process, see Section **4.3**.

Name: Description	Type	Units Value Units	Defaults and/or MIN/MAX
MaxSessionGlobalCount: The maximum count of concurrent ETP sessions that may be established for a given endpoint across all clients. The determination of whether this limit is exceeded should be made at the time of receiving the HTTP WebSocket upgrade or connect request. **NOTE:** Exposing this information may have security implications, so it should only be exposed if an implementation is comfortable with any potential associated risks.	long	count <count of sessions>	**MIN:** 2 sessions

REQUIRED BEHAVIOR:

1. If a new connection may cause a server to exceed its value for MaxSessionGlobalCount endpoint capability, a server MAY refuse the incoming connection.
   a. If a server chooses to reject an incoming connection because it would exceed this limit, it SHOULD reject the WebSocket request with HTTP 503: Service Unavailable.

3.3.2.8 MaxWebSocketFramePayloadSize (Endpoint)

Name: Description	Type	Units Value Units	Defaults and/or MIN/MAX
MaxWebSocketFramePayloadSize: The maximum size in bytes allowed for a single WebSocket frame payload. The limit to use during a session is the smaller of the client's and the server's value for MaxWebSocketFramePayloadSize, which should be determined by the limits imposed by the WebSocket library used by each endpoint.	long	byte <number of bytes>	N/A

SCOPE: All protocols; all request messages.

REQUIRED BEHAVIOR:

1. A client MUST NOT send a WebSocket frame that exceeds either its value or the server's value for MaxWebSocketFramePayloadSize.
2. A server MUST NOT send any WebSocket frame that exceeds either its value or the client's value for MaxWebSocketFramePayloadSize.
3. In either case, if the limit is exceeded, ETP behavior is undefined.
   a. The likely behavior is the WebSocket connection will be closed.

3.3.2.9 MaxWebSocketMessagePayloadSize (Endpoint)

Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
MaxWebSocketMessagePayloadSize: The maximum size in bytes allowed for a complete WebSocket message payload, which is composed of one or more WebSocket frames. The limit to use during a session is the smaller of the client's and the server's value for MaxWebSocketMessagePayloadSize, which should be determined by the limits imposed by the WebSocket library used by each endpoint.	long	byte <number of bytes>	N/A

SCOPE: All protocols; all request messages.

REQUIRED BEHAVIOR:

1. A client MUST NOT send a WebSocket message that exceeds either its value or the server's value for MaxWebSocketMessagePayloadSize.
2. A server MUST NOT send a WebSocket message that exceeds its value or the client's value for MaxWebSocketMessagePayloadSize.
3. In either case, if the limit is exceeded, ETP behavior is undefined.
4. If a store response to a customer request would exceed the limit, the store MUST try to send the response as a multipart message, where each message part does not exceed the limit.
   a. If the store cannot do so, it MUST deny the request and send error EMAXSIZE_EXCEEDED.
5. If a store notification to a customer would exceed the limit, the store MUST try to send the notification as separate, stand-alone notifications.
   a. If the store cannot do so, it MUST attempt to remove optional information (such as object data) so that the notification can be sent without exceeding the limit.
6. ETP behavior is undefined if this limit is exceeded. If an endpoint cannot send a message because doing so would exceed this limit, the most likely outcome is that the endpoint will drop the connection.

NOTE: One strategy for overcoming WebSocket limits communicated by this capability is use of ***Chunk*** messages; for more information, see Section **3.7.3.2**.

3.3.2.10 RequestSessionTimeoutPeriod (Endpoint)

Name: Description	Type	Units Value Units	Defaults and/or MIN/MAX
RequestSessionTimeoutPeriod: The maximum time period in seconds a server will wait to receive a ***RequestSession*** message from a client after the WebSocket connection has been established.	long	second <number of seconds>	**Default:** 45 **MIN:** 5

REQUIRED BEHAVIOR:

1. If a server does not receive a ***RequestSession*** message within this period, it MAY send error ETIMED_OUT (26) and close the WebSocket connection.
2. The server MUST NOT send the ***CloseSession*** message because no attempt was made to establish a session.

3.3.2.11 SessionEstablishmentTimeoutPeriod (Endpoint)

Name: Description	Type	Units Value Units	Defaults and/or MIN/MAX
SessionEstablishmentTimeoutPeriod: The maximum time period in seconds a client or server will wait for a valid ETP session to be established. **For a server:** • A valid session is established when it sends an ***OpenSession*** message to the client, which indicates a session has been successfully established. • The time period starts when it receives the initial ***RequestSession*** message from the client. **For a client:** • A valid session is established when it receives an ***OpenSession*** message from the server. • The time period starts when it sends the initial ***RequestSession*** message to the server.	long	second <number of seconds>	**Default:** 3,600 **MIN:** 60

REQUIRED BEHAVIOR:

1. If a session is not successfully established within this period, either endpoint MAY send error ETIMED_OUT (26) and then close the WebSocket.
2. The CloseSession message MUST NOT be sent because no session was established.

3.3.3 Support for ETP Optional Functionality

The endpoint capabilities listed in this section allow an endpoint to specify if it supports some optional ETP functionality.

Endpoint Capability	Description of Use/More Information
SupportsAlternateRequestUris	Energistics specifies canonical URIs (e.g., for data objects, data spaces, and data object queries), which MUST be supported. However, ETP also supports use of alternate URI formats.

Endpoint Capability	Description of Use/More Information
	For more information about how URIs are used in ETP, see **Section 3.7.4**. For definitions and information about required and optional URI formats, see **Appendix: Energistics Identifiers**.
SupportsMessageHeaderExtensions	Indicates of if an endpoint supports the use of a ***MessageHeaderExtension***, an optional structure that may be sent between the ETP ***MessageHeader*** and message body. For more information, see Section **3.6.2**.

3.3.4 Data Object Capabilities: How They Work

ETP specifies a set of capabilities that let an ETP implementation limit operations or specify capabilities at the data object level. For a simple listing of all data object capabilities defined in ETP, see Section **23.3**.

This section:

- Explains the "basic" data object capabilities and how they work (required behavior) for get, put, and delete operations, for all data objects (Section **3.3.4.1**).
- Lists, explains and provides references for the "specialty" data object capabilities, which are those caps that apply to particular kinds of data objects or have a more specialized purpose (Section **3.3.4.2**).

3.3.4.1 Data Object Caps for Get, Put and Delete

The section lists the main data object capabilities that apply to all data objects and explains the required behavior for using them.

Data Object Capabilities	Description of se/More Information
SupportsGet	Indicates whether get operations are supported for the data object type. Default: true
SupportsPut	Indicates whether put operations are supported for the data object type. If the operation can be technically supported by an endpoint, this capability should be true. Default: true
SupportsDelete	Indicates whether delete operations are supported for the data object type. If the operation can be technically supported by an endpoint, this capability should be true. Default: true

REQUIRED BEHAVIOR: The required behavior is the same for each of the capabilities listed in the table. (In the following instruction, <RequestType> may be *get*, *put* or *delete* and <*DataObjectCapability*> is one the corresponding Data Object Capabilities from the table above.)

1. A customer MUST NOT send a <RequestType> request for an object type where the <*DataObjectCapability*> value is false. (**EXAMPLE:** <RequestType>/<*DataObjectCapability*> = get, SupportsGet)
2. If a Store's <*DataObjectCapability*> value is false, the store MUST reject any <RequestType> request by sending error ENOTSUPPORTED (7). (**EXAMPLE:** SupportsPut, put)

3.3.4.2 "Specialty" Data Object Capabilities

This section lists data object capabilities that apply to particular kinds of data objects or have a more specialized purpose, with references to sections with more information:

- **ActiveTimeoutPeriod**, which is also defined as endpoint and protocol capabilities; see Section **3.3.2.1**.
- **MaxDataObjectSize**, which is also defined as endpoint and protocol capabilities, see Section **3.3.2.4**.
- **MaxContainedDataObjectCount** and **OrphanedChildrenPrunedOnDelete**, which both are used for container/contained data objects and explained in the chapters for the protocols in which they are used. For definitions of container and contained data objects, see Section **9.1.3**.
- **MaxSecondaryIndexCount** limits the number of secondary indexes that a Channel or Channel Set data object may have; its use is explained in Store (Protocol 4), where these data objects are created and updated. See Section **9.2.1.2**.

3.3.5 ADVISORY: Implication of Capabilities and Required Behavior for Stores

The minimum required values for the set of endpoint, data object, and protocol capabilities defined by ETP essentially defines the minimum required behavior for an ETP store. Implementers are advised to look at and evaluate the set of capabilities in their totality and consider them when developing an ETP store.

3.4 ETP Message Approach

ETP is a collection of messages that are organized into sub-protocols. Endpoints in an ETP session exchange messages asynchronously to communicate about and perform actions on data objects, which represent real world business objects such as wells, wellbores, logs, channels, earth models (and their constituent parts), etc. (For definition of data object, see Section **25.1**).

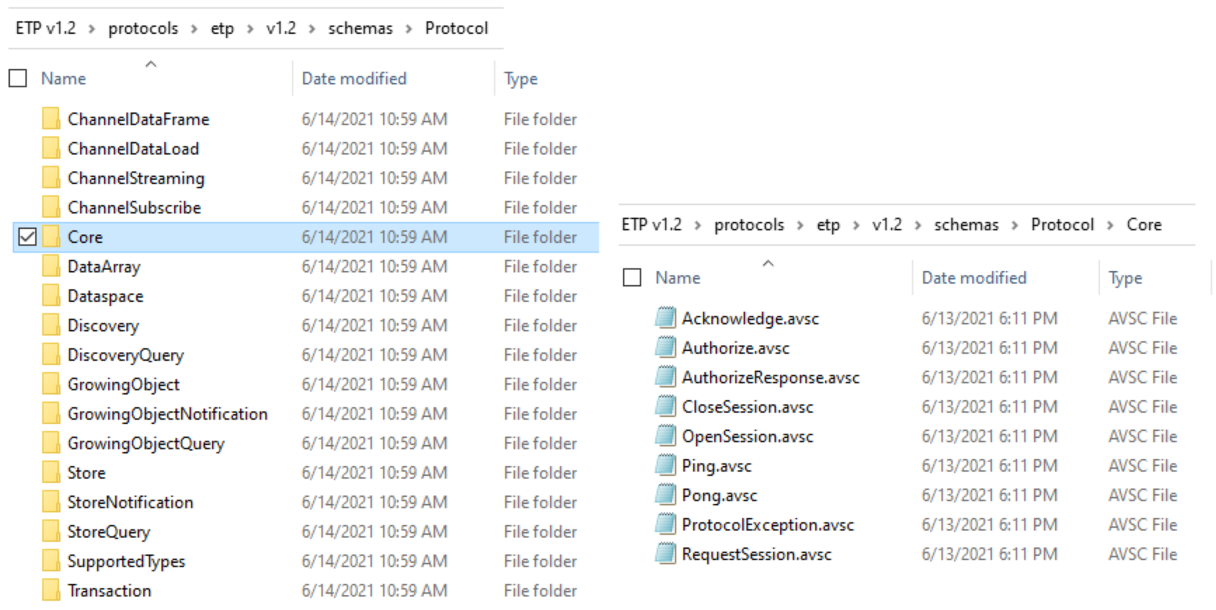
To support all of the varied functionality required, ETP was designed to support multiple message patterns (or styles of message transfer) especially with respect to message size, frequency, and complexity. For example, messages used to send channel data differ in design from messages used to transfer large files of complex array data. As such, ETP has separate protocols for these tasks (channel data and array data), and the message content in the different protocols vary according to the types of data they handle.

- For consistency, each ETP message is defined by an Avro schema and is composed of low-level data types, many of which are also defined in ETP (others are defined in Avro). (For more information about ETP data types, see Section **3.4.1.1** and Chapter **23**.)
  - ETP v1.2 is based on Avro v1.10, but remains compatible with Avro v1.8.2.
  - For an overview of the standard ETP message format, see Section **3.5**.
- To support easier implementation, there are some general types of ETP messages with standard naming conventions (see Section **3.4.2**).
- For more information about asynchronous transmission and how related messages are correlated, see Section **0**.

NOTE: This practice of segmenting protocols by message types is somewhat in contrast with previous Energistics data service specifications, where each domain's related standard (i.e., WITSML, PRODML and RESMQL) had service contracts that were unique to the domain objects. With ETP, the goal is to use a single set of protocols across multiple domains, with the differences in protocols focused more on the communication requirements of particular operations (e.g., real-time streaming vs. querying a store).

3.4.1 Messages are Defined by Avro Schemas

Each ETP message is defined by an Avro schema with the file name ***MessageName.avsc*** (where ***MessageName*** is the actual message). When you download ETP from the Energistics website, the package contains these schemas, organized by ETP sub-protocol (**Figure 3**). (**NOTE:** In the ETP download, the folders for each ETP sub-protocol are alpha ordered, not ordered by ETP sub-protocol number.)



ETP v1.2 › protocols › etp › v1.2 › schemas › Protocol

☐ Name	Date modified	Type
ChannelDataFrame	6/14/2021 10:59 AM	File folder
ChannelDataLoad	6/14/2021 10:59 AM	File folder
ChannelStreaming	6/14/2021 10:59 AM	File folder
ChannelSubscribe	6/14/2021 10:59 AM	File folder
☑ Core	6/14/2021 10:59 AM	File folder
DataArray	6/14/2021 10:59 AM	File folder
Dataspace	6/14/2021 10:59 AM	File folder
Discovery	6/14/2021 10:59 AM	File folder
DiscoveryQuery	6/14/2021 10:59 AM	File folder
GrowingObject	6/14/2021 10:59 AM	File folder
GrowingObjectNotification	6/14/2021 10:59 AM	File folder
GrowingObjectQuery	6/14/2021 10:59 AM	File folder
Store	6/14/2021 10:59 AM	File folder
StoreNotification	6/14/2021 10:59 AM	File folder
StoreQuery	6/14/2021 10:59 AM	File folder
SupportedTypes	6/14/2021 10:59 AM	File folder
Transaction	6/14/2021 10:59 AM	File folder

ETP v1.2 › protocols › etp › v1.2 › schemas › Protocol › Core

☐ Name	Date modified	Type
Acknowledge.avsc	6/13/2021 6:11 PM	AVSC File
Authorize.avsc	6/13/2021 6:11 PM	AVSC File
AuthorizeResponse.avsc	6/13/2021 6:11 PM	AVSC File
CloseSession.avsc	6/13/2021 6:11 PM	AVSC File
OpenSession.avsc	6/13/2021 6:11 PM	AVSC File
Ping.avsc	6/13/2021 6:11 PM	AVSC File
Pong.avsc	6/13/2021 6:11 PM	AVSC File
ProtocolException.avsc	6/13/2021 6:11 PM	AVSC File
RequestSession.avsc	6/13/2021 6:11 PM	AVSC File

Figure 3: ETP download folder structure of ETP protocols (left) and message schemas for Core protocol (right). This specification contains a chapter for each published protocol in the current version of ETP; for easy reference, each protocol-specific chapter also displays the message schemas for the subject protocol.

Some general points about the schemas:

1. All schemas—the ones in the download and those displayed in this specification—are generated from the ETP UML model, so they *should* be identical.
    a. The UML tool is Enterprise Architect; for information about the schema-generation process, see Section **3.4.1.2**.
    b. If there is a discrepancy between a schema (.avsc file) and the specification, the schema is the primary source.
2. The ETP schema download also includes an .avpr file, which is an aggregation of all .avsc files into a single file. The .avpr file is provided as a convenience to support alternate ways of working with these schemas.
3. *Energistics.Etp.v12* is the name of the root package, or namespace, for all messages and data types in ETP v1.2. It is not expected that this namespace will contain any types or classes directly; it is just a container for other namespaces.
4. For a general explanation of the contents of an Avro message schema, see Section **3.5.5**.

3.4.1.1 Messages are Composed of Data Types and Primitives Defined by Avro and ETP

For consistent design, ETP leverages Avro primitive data types (long, float, string, etc.) and defines other low-level data types (which are specified as Avro records, enumerations, etc.). **Figure 4** shows examples of some frequently used Avro records defined by ETP and the messages that use those records. For the complete list and definitions of data types, see Chapter **23**.

NOTE: The schemas and related documentation reference these data types and links are provided. Typically, to completely understand the content of a specific message, you must read the related data type documentation, especially for records and enumerations.

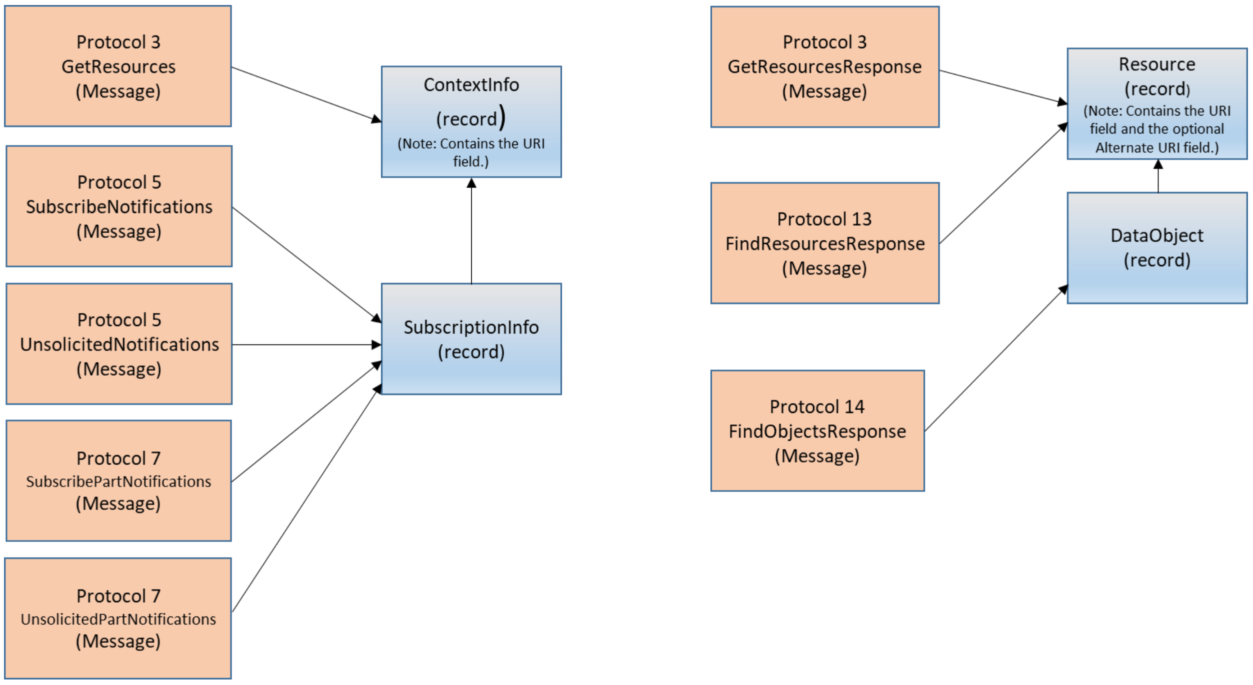


Figure 4: Examples of ETP-defined Avro records that are used by multiple messages and other records.

The following data types are composed of primitives that are defined by Avro or ETP:

- **Record:** Specifically, this is an Avro record, which is similar to a C or C++ struct or to a JSON object or JavaScript object. The record stereotype is used to designate low-level data types that are composed to create messages. **EXAMPLE:** In the figure above, the ***SubscriptionInfo*** record is used in several notification messages by different protocols, and the ***SubscriptionInfo*** record uses the ***ContextInfo*** record.

  NOTE: The key components of an ETP message, the header, body and optional header extension, are also defined as Avro records, each of which are composed of other Avro primitives and records.
- **Enumeration:** Enumerated values are defined in the schemas as a list of literal names and serialized on the wire as an integer value. Avro schemas do not allow a bespoke integer to be associated with a given enumeration, and so they are order dependent. **NOTE:** This order-dependency means that, for maximum interoperability, the ordering of enumerations must be consistent across ETP versions.
- **Union:** Used to represent a type that can be any one of a selected list of types. Union is similar to unions in C or C++ and more or less maps to the xsd:choice element in XML schemas.

3.4.1.2 How the Avro Schemas are Generated

The Avro schemas, in JSON form, are produced by a code-generation process in Enterprise Architect (EA), the tool used by Energistics to design ETP, and some additional custom scripts. This process creates one .avsc file per Avro record or enumeration, in a folder structure.

Another script is used to generate all of the schemas in a single Avro Protocol (.avpr) file. Note that while the .avpr format is a convenient way to place all of the schema in a single file, ETP DOES NOT use the Avro RPC protocol.

3.4.2 General Message Types and Naming Conventions

In general, ETP has these types of messages: request, response, notification, and data. To support implementation, ETP uses some general naming conventions for these types of messages, which are defined and described in the table below.

NOTE: Additionally, ETP has two messages that are defined in Protocol 0 but may be used in any of the ETP protocols; these include: ***ProtocolException*** and ***Acknowledge*** messages. For more information on these "universal" messages, see Section **3.7.2**.

Message Type	Convention	Examples
Request: Ask for specific data or action to be performed	Uses a verb/object construction. Verbs used: "get", "find" If a message has been designed to request and/or respond with/for multiple objects, the message name uses the U.S. English plural. MOST messages in ETP are plural messages. For more information about so-called plural messages, see Section **3.7.2**.	***GetResources*** (Protocol 3) ***GetDataObjects*** (Protocol 4) ***FindDataObjects*** (Protocol 14) ***TruncateChannels*** (Protocol 22)
Response: An ETP-defined answer to a request message.	A response message that corresponds to a specific type of request message and uses this convention: <request message name> + the word "Response" Two exceptions to this general rule: • Messages that can be either a response OR a notification are named with the Notification convention (examples: SubscriptionEnded) • Requests with multiple possible positive responses use a variation of the Response convention (examples: GetResourcesEdgesResponse, GetFrameResponseHeader, GetFrameResponseRows) **NOTE:** In ETP v1.2, "success only" response messages have been added to support more efficient operations of customer role software. These messages confirm that an operation (e.g., a put or delete operation) in a request have been completed successfully.	***GetResourcesResponse*** (Protocol 3) ***GetDataObjectsResponse*** (Protocol 4) ***FindDataObjectsResponse*** (Protocol 14) ***TruncateChannelsResponse*** (Protocol 22)
Notification: A notice from a store that a type of change has occurred. Endpoints can specify if they want to receive the actual data that has changed with the notification.	Begins with a noun (which is typically the word "object", "parts", "channels", "range" or "subscription" and ends with a verb in the past tense, such as "changed", "replaced", or "ended".	***ObjectChanged*** (Protocol 5) ***SubscriptionEnded*** (Protocol 5) ***PartsDeleted*** (Protocol 7) ***ChannelsTruncated*** (Protocol 21) ***ChannelsClosed*** (Protocol 22)
Data: Messages sent as part of a subscription, which typically includes streaming data, notifications, and unsolicted subscriptions.	Noun or phrase that describes the kind of data or notification.	***ChannelData*** (Protocol 1), (Protocol 21), (Protocol 22)

3.5 ETP Message Format and Basic Sequence Requirements

This section explains the format of a standard ETP message and general rules for how to process a message. For more information about specific types of messages and message patterns used in ETP, see Section **3.7**.

This section includes information for these topics:

- An overview of an ETP message (Section **3.5.1**)
- General requirements for the message format (Section **3.5.2**)
- A description and requirements for the basic response/request message sequence (Section **3.5.3**)
- Details about the message header, its data fields, how to populate them, and how to process a message header (Section **3.5.4**)
- Description of the general characteristics of the message body and the Avro schemas that define them (Section **3.5.5**)
- Mechanisms to limit the size of messages (Section **3.5.6**)
- How compression works (Section **3.5.7**)

NOTE: Not all error conditions and usage of related error codes are specified in this document. Implementers are encouraged to familiarize themselves with the general behaviors defined in this chapter and throughout the document, the available ETP-defined error codes defined (see Section **24.3**), and use their best judgements for their particular implementation.

3.5.1 Overview of an ETP Message

Each ETP message consists of one each:

- message header
- message body

Figure 5 shows the basic format of a standard ETP message. The message header and message body are encoded in separate Avro records (see Section **3.8**), which are sent sequentially on the wire in a single WebSocket message (see Section **3.9**).

ETP Message #1 Header	ETP Message #1 Body	ETP Message #2 Header	ETP Message #2 Body

Figure 5: ETP standard message format. Each message header and body is encoded in a separate Avro record.

This separate encoding of message header and message body enables the receiver to read and decode the ETP message header, independent of the ETP protocol or ETP message body itself. (This design is consistent with and supports software design best practices for modularity and efficiency, e.g., protocol-specific "handlers".)

NOTE: Unlike earlier Energistics standards based on SOAP and XML (e.g., WITSML v1.x), ETP has no concept of an ‘envelope’ schema that contains the entire ETP message. However, the WebSocket payload length field plays the same role in terms of defining the extent of the message content; for more information, see Section **3.9**.

3.5.2 General Requirements for ETP Message Format

Each ETP message MUST conform to these requirements:

1. Each ETP message MUST have one header and one body.

   a. The message header for all ETP-defined messages has a standard format defined by the ***MessageHeader*** schema, which MUST be used. For details about the content, use, and rules for processing the message header, see Section **3.5.4.**
   b. Each message body has a unique schema (one for each ETP-defined message). For more information about the message body, see Section **3.5.5**.
      i. For certain messages, the message body MAY be zero length (which is specified in the relevant message schemas).
      ii. The body of any message—except for those explicitly excluded in Core (Protocol 0)—MAY be compressed, regardless of role, based on the compression encoding negotiated during initialization of the ETP session (see Chapter **5**). For more information about compression, see Section **3.5.7**.
2. An ETP message—except for those explicitly excluded in Core (Protocol 0)—MAY include an optional message header extension (***MessageHeaderExtension***), which allows the sender to add additional contextual/extension data (e.g., such as information for open tracing) to any message.
   a. If used, the message header extension MUST be sent between the message header and the message body. For more information about using ***MessageHeaderExtension***, see Section **3.6.2**.
3. ETP provides several mechanisms to limit the size of messages (to respect WebSocket limits and to help with throughput and performance); for more information, see Section **3.5.6**.

3.5.3 General Sequence for ETP Request/Response Messages

This section explains the high-level basic sequence of how request and response messages are exchanged between the 2 endpoints in an ETP session. It provides some general rules that apply to virtually all message exchanges. (For information on the types of ETP messages and naming conventions, see Section **3.4.2**.)

Specific message flows and more complex patterns (such as multipart requests, responses and notifications) are described in Section **3.7**. Also, each protocol-specific chapter identifies key tasks performed in that protocol and the exact messages that must be exchanged to perform the task, and includes error scenarios, when to send a ***ProtocolException*** message, and which error codes to use.

The rules for the basic message sequence are as follows:

1. One endpoint (sender) sends a request message to the other endpoint (receiver).
   a. The message MUST be composed of one ***MessageHeade***r and a message body.
      i. For information on the content of and on how to populate the ***MessageHeader***, see Section◦**3.5.4**.
      ii. For information on the content of the message body, see Section **3.5.5**.
   b. The message is serialized using Avro; see Section **3.8**.
   c. A complete ETP message (header and body) is sent in a single WebSocket message.
      i. For more information about use of WebSocket with ETP, see Section **3.9**.
      ii. Each message MUST NOT exceed the MaxWebSocketMessagePayloadSize endpoint capability. For more information see Sections **3.3.2.9**.
   d. Each endpoint MUST send messages ordered by message ID. (For information on numbering in the *messageId* field, see **3.5.4.1**.)
2. When the receiver receives the message it MUST do the following:
   a. Wait to receive the entire WebSocket message before it begins processing any of the content. (That is, the receiver MAY NOT process the incoming message on a frame-by-frame basis.)

    i. A single WebSocket message is considered the lowest common "unit of work" that must be received to begin processing.

  b. After receiving the entire WebSocket message, the receiver MUST first attempt to de-serialize the ***MessageHeader*** (before the message body):

    i. If deserialization FAILS: The receiver MUST send error EINVALID_MESSAGE (19).The ***MessageHeader*** of the ***ProtocolException*** message (that contains the error code) MUST have *protocol* = 0, and *correlationId* = 0. (Because the receiver could not de-serialize the header, it does not know the protocol or message ID of the errant message. For information on the content of and on how to populate the ***MessageHeader***, see Section◦**3.5.4**.)

    ii. If the deserialization SUCCEEDS, the receiver MUST process the content of the ***MessageHeader***, some of which may require the receiver to take action even before processing the message body. For details on how to process the content of the ***MessageHeader***, see Section◦**3.5.4.2**.

  c. After de-serializing the header, the receiver MUST de-serialize and process the message body and respond to the request.

    i. For requirements unique to ***Acknowledge*** and ***ProtocolException*** messages, see Section◦**3.7.2**.

    ii. For more information on sequences for more complex ETP message patterns (for example, plural and multipart messages), see Section **3.7.3**.

    iii. The key tasks and related message sequences—including message-specific processing, error scenarios, when to send a ***ProtocolException*** message, and which error codes to use—are explained in the protocol-specific chapters (Chapters 5 through 22).

3.5.4 ETP Message Header

The ETP ***MessageHeader*** includes key identifying and usage data for each message; all ETP message schemes have a message header. This section explains:

- The content (data fields) in a message header and requirements for how the sender role populates these fields (see Section **3.5.4.1**).
- How the receiver role must process the information in the message header (see Section **3.5.4.2**).

For more information about:

- The ***MessageHeader*** schema with field definitions, see Section **23.25**.
- Any special processing requirements for ***ProtocolException*** messages, see Section **3.7.2.1**.
- Any special processing requirements for ***Acknowledge*** messages, see Section **3.7.2.2**.
- Using an optional ***MessageHeaderExtension***, see Section **3.6.2**.

3.5.4.1 Required Behavior for Populating a Message Header

Observe these rules and requirements for a ***MessageHeader***:

1. The ***MessageHeader*** and all of its fields are REQUIRED.
2. The ***MessageHeader*** MUST NOT be compressed.

The ***MessageHeader*** does all of the following:

3. **Identifies by its assigned sub-protocol number** (field name = *protocol*), **the ETP sub-protocol in which the message is being used** (e.g., Core (Protocol 0), ChannelStreaming (Protocol 1), ChannelDataFrame (Protocol 2), Discovery (Protocol 3), etc.).
    a. **EXCEPTION:** ETP messages are used exclusively in the protocol that they are defined in except for ETP “universal” messages (which include ***ProtocolException*** and ***Acknowledge*** messages), which may be used in any ETP sub-protocol even though they are defined in Core (Protocol 0).

For more information, see Section **3.7.2**.

4. **Identifies the type of message being sent** (field name = *messageType*). The combination of the *protocol* and *messageType* uniquely identifies the message type; that is, it defines the schema for the message body.
   b. ETP identifies message types by assigning an integer to each unique message in a sub-protocol.
      i. **EXAMPLE:** In ChannelStreaming (Protocol 1), messageType 2 is the ***ChannelData*** message, whose schema is shown in Section **6.3.3**.
5. **Identifies the message ID** (field name = *messageId*) **of each unique message in an ETP session**. Message IDs MUST observe these rules:
   a. Message IDs MUST be unique within a session, and for a given endpoint (i.e., client/server). The IDs used by a client and a server are completely independent of one another. Put another way, the "primary key" of any given message could be thought of as endpointType + messageId.
   b. Message IDs MUST be strictly increasing.
      i. Each endpoint MUST send messages ordered by message ID.
   c. To help with de-bugging and troubleshooting, ETP has adopted this numbering convention for message IDs, which endpoints MUST observe:
      i. The client side of the connection MUST use ONLY non-zero **even-numbered** message IDs.
      ii. The server side of the connection MUST use ONLY non-zero **odd-numbered** message IDs.
      iii. A message ID of 0 is invalid.
      iv. Message IDs ARE NOT required to be sequential or have any correlation between message IDs and any particular sub-protocol.
6. **Correlates related messages.** The ***MessageHeader*** has a correlation ID (field name = *correlationId*), whose purpose is somewhat context-sensitive, depending on the specific message.
   In general, its purpose is to correlate related messages. (**EXAMPLE:** In some cases the "related message" may be all messages that comprise a complete multipart request or it may be to correlate a response to the request message it is responding to.)
   This specification provides a correlation ID usage guideline for each ETP message (see the individual messages listed for each protocol). General correlation ID behaviors are:
   a. Not all messages use *correlationId*; when it is not used by a message, the *correlationId* MUST be ignored and should set to 0.
   b. For a single-message request, the correlationId MUST be set to 0.
   c. For a response message, the correlationId MUST be set to the messageId of the corresponding request message (i.e., the request that this response message is replying to).
   d. For usage unique to ***ProtocolException*** messages, see Section **3.7.2.1**.
   e. For behaviors for *correlationId* related to plural messages and multipart requests, responses, and notifications, see Section **3.7.3**.
7. **Has a** *messageFlags* **field, which acts as a bit-field and allows multiple Boolean flags to be set on a message**. These flags are currently defined:
   a. **0X02**: Message is the final message (a so-called "FIN bit") for all ETP message types (i.e., request, response, notification or data messages).
      i. This flag MUST always be set on the final message of any action—even if the action is composed of only a single message.
      ii. **EXAMPLES:** 1) Each ***ChannelData*** message (used in the channel streaming protocols) MUST have its FIN bit set; 2) For a multipart request such as ***ReplaceRange*** (in ChannelDataLoad (Protocol 22), the FIN bit MUST be set on the last message of the multipart request (so if the

request is composed of 5 ***ReplaceRange*** messages, the FIN bit is set on the fifth message).

   iii. For information on setting the FIN bit on multipart requests, responses and notifications, see Section **3.7.3.1**.

b. **0x08**: Message body (and optional ***MessageHeaderExtension***, if used) is compressed.

c. **0x10**: Sender is requesting an Acknowledge message. For more information on the ***Acknowledge*** message, see Section **3.7.2.2.**

d. **0x20**: Indicates that this message includes an optional message extension. In this version of ETP, the only message extension mechanism is the ***MessageHeaderExtension***. For more information on MessageHeaderExtension, see Section **3.6.2**.

e. **NOTE**: 0X01 and 0X04 (which were used in the previous version of ETP) are currently unused.

3.5.4.2 Required Behavior for Processing Message Headers

This section explains requirements for how receiving endpoints (receiver) MUST process message headers for each ETP message.

NOTE: The process below continues from Section **3.5.3** and is the details of step 2.b.**ii**; if the receiver correctly de-serialized the ***MessageHeader***, now it must process it as follows:

1. The receiver MUST inspect all header fields because these attributes determine processing requirements. The receiver MUST use or process the attributes as specified in steps 3 through 5.
2. The receiver MUST first inspect the *messageFlags* field to determine if the sender is requesting an ***Acknowledge*** message (abbreviated as "Ack"):
   a. For detailed information on when and how to send the ***Acknowledge*** message, see Section **3.7.2.2**.
   b. The remainder of message flags are explained in Step **5 below**.
3. The receiver MUST use the *protocol* number and *messageType* to determine which schema to use to interpret the message (or which handler to send the message to for such processing).
4. The receiver MUST inspect the *correlationId* to determine if this message is associated with another message.
   a. Each message schema section in this specification includes basic information about the correlation ID for that message.
   b. For rules on how to use correlation IDs with plural and multipart messages, see Section **3.7.3**.
5. The receiver MUST inspect the *messageFlags* and it MUST use the provided information for processing and/or perform the requested action, as described here:
   a. **0x02:** If true, the "FIN bit" indicates the final message of an action. That is, the receiver has all messages that comprise this request, response, notification or data. (For more information on requirements for setting the FIN bit in multipart requests, responses, and notifications, see Section **3.7.3.1**.)
      i. The FIN bit MUST always be set to true on the final message, even if an action (a request, response, notification or data) is composed of only 1 message.
   b. **0x08:** If true, the message body (and optional ***MessageHeaderExtension***, if used) is compressed. Before it can inspect the message body for protocol handling, the receiver MUST uncompress the remainder of the WebSocket payload, which is the message body (and optional ***MessageHeaderExtension***, if used), using the compression algorithm negotiated when the ETP session was established. For more information about compression, see Section **3.5.7**.
   c. **0x10:** If true, it indicates that the sender is requesting an ***Acknowledge*** message. Because an Ack must be sent first, if requested, behavior for processing this flag is explained in Section

3.7.2.2.

d. **0x20:** If true, it indicates that the message has a ***MessageHeaderExtension*** (optional header between the ***MessageHeader*** and ***MessageBody***). (For more information and requirements for processing a ***MessageHeaderExtension***, see Section 3.6.2.)

3.5.5 ETP Message Body

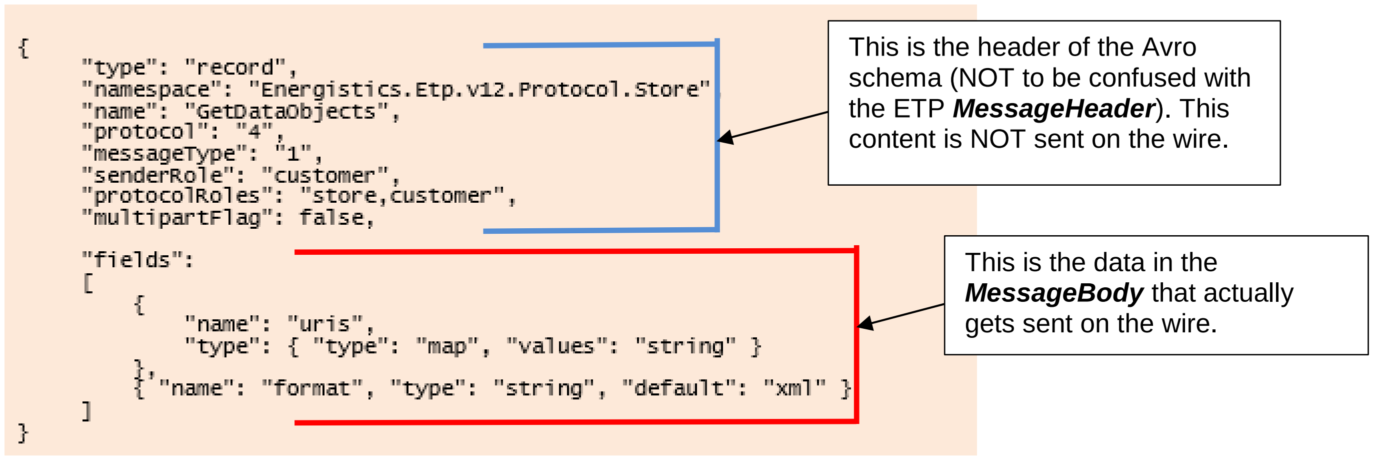
The ETP message body holds the actual content of a message. The body of any message (except those explicitly excluded in Core (Protocol 0)), MAY BE compressed, regardless of role, based on the compression encoding negotiated during initialization of the ETP session.

The specific content of each *messageType* is defined throughout this specification, in the individual sub-protocol that defines a particular message. **NOTE:** For some messages, ETP provides an extension mechanism in the message body; for more information, see Section 3.6.

Figure 6 is an example ETP message body schema (***GetDataObjects.avsc***).

The message body schema contains these general types of information:

1. Avro schema header information, which is completely separate from the ETP ***MessageHeader*** described in Section 3.5.4. This information is the content that is highlighted by the blue bracket in the figure; this content is NOT sent on the wire. Some of the fields on this header are defined and required by Avro Schema; other fields are defined by ETP, which are explained below the figure.
2. Content of the message—the data actually sent on the wire—is shown in the red bracket.



```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Store",
     "name": "GetDataObjects",
     "protocol": "4",
     "messageType": "1",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "uris",
             "type": { "type": "map", "values": "string" }
         },
         { "name": "format", "type": "string", "default": "xml" }
     ]
}
```

This is the header of the Avro schema (NOT to be confused with the ETP ***MessageHeader***). This content is NOT sent on the wire.

This is the data in the ***MessageBody*** that actually gets sent on the wire.

Figure 6: An example ETP message schema (GetDataObjects message from Store (Protocol 4)).

The Avro Schema header content contains:

1. Information required by the Avro specification (e.g., *type, namespace, name*).
2. Other fields specified by ETP to aid in processing or provide useful context about when and how to use this message. The ETP-specific information is defined in the UML model and appears in the message body and in this specification document. The ETP-defined information includes:
   a. *protocol*: the ETP sub-protocol number.
   b. *messageType*: the integer that uniquely identifies this message within the ETP sub-protocol in which it is defined.
   c. *senderRole*: the ETP endpoint role(s) that actually sends the message.
   d. *protocolRoles*: the two ETP endpoint roles allowed in this protocol.
   e. *multipartFlag*: a Boolean that shows whether or not a response, request or notification may be

implemented as a series of multiple related messages. Only messages where this value is true may be implemented as a multipart request or response.

3.5.6 Mechanisms to Limit Message Size

ETP has a few mechanisms that allow endpoints to specify limits with a goal of keeping message size as small as possible to respect WebSocket limits and to help with throughput and overall performance. The individual parameters and how they work are discussed in the context of the individual protocols in which they are used or other relevant sections in this specification. This section summarizes the available types of parameters and provides links to other relevant sections of this specification.

The mechanisms place limits on messages and message data, and these limits are communicated using the data object, endpoint and protocol capabilities. For more information see Section **3.3**.

Parameter Description	For More Information
WebSocket libraries. The WebSocket protocol is the transport mechanism for ETP. People implementing ETP are free to use any published WebSocket libraries specifically designed for implementing WebSocket, but should be aware that these libraries have limits for the WebSocket frame and message sizes—some of these limits were found to be as small as 128 kb. While ETP does not define these library-specific limits, it does provide a way for endpoints to share their WebSocket-related size limits. **EXAMPLES:** *MaxWebSocketFramePayloadSize* and *MaxWebSocketMessagePayloadSize*	• For information on how WebSocket works with ETP, see Section **3.9**.
Limits to the size of the data contained in ETP messages. ETP defines several capabilities that limit the size of data and objects that may be contained within ETP messages. **EXAMPLES:** *MaxDataArraySize, MaxDataObjectSize, MaxPartSize*	• For more information about ETP data object capabilities and how they work, see Section **3.3.4**. • Some data object capabilities may also be endpoint capabilities, see Section **3.3.1**.
Limits to the counts of items contained in ETP messages. ETP defines several capabilities that limit the count of items that may be contained within ETP messages. Most of the relevant capabilities are defined at the protocol level. **EXAMPLES:** *MaxResponseCount, MaxFrameResponseRowCount*	• Explained in the protocols where they are used.
Limits related to multipart requests, responses, and notifications. ETP provides endpoint capabilities that allow endpoints to specify limits on the maximum number of in-flight multipart message sequences (i.e., to help constrain maximum memory necessary to accumulate all the messages of multipart sequences) **EXAMPLES:** *MaxConcurrentMultipart*	• For more information, see Section **3.7.3**.

3.5.7 Message Compression

ETP supports use of compression. Endpoints may negotiate a supported compression algorithm when the ETP session is established. (For more information, see Chapter **5**.)

Observe these rules for use of compression:

1. If a client or server supports compression, it MUST support at least gzip.
2. If the client and server agree to use compression in a session, messages sent in the session MAY (but are NOT required to) be compressed.
3. When a message is compressed:
   a. a ***MessageHeader*** is NEVER compressed.

b. The remainder of the WebSocket payload after the ***MessageHeader*** is compressed, which includes:

   i. The message body.

   ii. If used, the optional ***MessageHeaderExtension*** (see Section **3.6.2**).

   iii. The 0x08 flag in its ***MessageHeader*** MUST be set to true.

3.6 ETP Extension Mechanisms

ETP specifies some extension mechanisms so that organizations have a standard way to add custom or proprietary data to ETP. These mechanisms include use of custom protocols and capabilities, message header extensions, message extensions, attribute metadata, and extensions to data objects, which are all described in this section. All mechanisms explained here are OPTIONAL.

3.6.1 Custom Protocols and Capabilities

Organizations may extend ETP by developing additional protocols beyond those defined in this specification; these are referred to as "private" protocols. Custom protocols use protocol numbers 2000+.

Custom protocols MAY require related protocol capabilities; or additional protocol and endpoint capabilities MAY BE added to the capabilities defined by ETP for the ETP-defined protocols. For information about ETP capabilities, see Section **3.3**.

For more information about creating custom protocols and assigning protocol numbers, contact Energistics.

3.6.2 MessageHeaderExtension

Use of message header extensions (***MessageHeaderExtension***) allows additional contextual information, about either the ***MessageHeader*** or the message body, to be sent with a specific message. It can be used by implementers to send system-wide, custom properties and contextual information that needs to be passed up and down a call stack. A common use case in cloud-native environments and other call stacks such as HTTP/Rest and gRPC is the requirement to pass tracing information (such as open tracing) down, and back up through a call stack.

If used, the sender indicates (using the designated bit in the *messageFlags* field of the standard ***MessageHeader***) that a ***MessageHeaderExtension*** is being sent, and then sends the ***MessageHeaderExtension*** between the standard ***MessageHeader*** and the ***MessageBody***.

WARNING: It is strongly recommended that message header extensions NOT be used with "one-way" notification messages or other high-throughput or streaming messages (such as ***ChannelData*** messages) due to potentially high overhead if their use is abused.

For the schema for the ***MessageHeaderExtension***, see Section **23.26**.

To use the MessageHeaderExtension, you MUST observe these rules:

1. To use message header extensions in an ETP session, the endpoints (client and server) MUST negotiate their use as part of establishing an ETP session (as described in Chapter **5**).

   a. If an endpoint supports use of message header extensions, it MUST set the endpoint capability *supportsMessageHeaderExtensions* to true (this capability can be specified in the ***RequestSession*** or ***OpenSession*** messages, as described in Chapter **5**).

   b. To successfully use message header extensions, BOTH endpoints in an ETP session must support their use.

   c. If only one endpoint (or no endpoints) in an ETP session supports use of message header extensions, then NEITHER endpoint may use them.

   NOTE: The next steps explain how to use a ***MessageHeaderExtension*** when exchanging messages, including error conditions.

2. When an endpoint needs to send a ***MessageHeaderExtension***, it MUST do both of the following:
    a. In the ***MessageHeader***, it MUST set the 0x20 bit in the *messageFlags* field to true (so the receiver "knows" a ***MessageHeaderExtension*** is being sent).
    b. Before sending the message body, it MUST send the ***MessageHeaderExtension*** (so the ***MessageHeaderExtension*** is sent between the ***MessageHeader*** and message body as shown in **Figure 7**).

ETP Message #1 Header (with 0x20 flag = true)	**Message Header Extension** (optional)	ETP Message #1 Body	ETP Message #2 Header (with 0x20 flag = true)	**Message Header Extension** (optional)	ETP Message #2 Body

Figure 7: ETP message format with optional MessageHeaderExtensions.

3. The endpoint that receives the ***MessageHeaderExtension*** MUST do at least one of the following:
    a. If the endpoint supports ***MessageHeaderExtension***, it MUST attempt to process it.
    b. If the ***MessageHeaderExtension*** contains keys that the receiver does not understand or is not interested it, it MUST ignore them (no error messages).
    c. If the endpoint does NOT support ***MessageHeaderExtension***, it MUST send error EINVALID_MESSAGE (19).
    d. If the ***MessageHeaderExtension*** flag is set to true AND the ***MessageHeaderExtension*** is omitted (not just an empty map, but the map is omitted entirely), it MUST send error EINVALID_MESSAGE (19).
        i. NOTE: Conditions 3.c and 3.d and 4.a will cause message de-serialization issues,
4. ETP permits only one ***MessageHeaderExtension*** per message.
    a. If an endpoint sends more than 1, the receiver MUST send error EINVALID_MESSAGE (19).
5. If a message containing a ***MessageHeaderExtension*** is compressed, then the ***MessageHeaderExtension*** MUST be compressed with the message body.

3.6.3 Data Attribute Metadata

ETP defines a mechanism that makes it possible to annotate (or "decorate") individual data points in a channel or channel frame, for example, with quality, confidence, or audit information. ETP itself does not currently define any specific attributes, values, or use, but the mechanism is available for definition and use by individual MLs or companies to specify and use.

The mechanism is a record named ***DataAttribute*** and the mechanics of its use are described in in relevant protocols, records and chapters where it occurs.

3.6.4 Message Extension

Several ETP data structures (**EXAMPLES:** Resource, ChannelMetadataRecord, etc.) allow an endpoint to send custom data in addition to the data in standard fields and records in the ETP schemas. This custom data is also informally referred to as "proprietary data or content".

In all cases, custom data is sent in a field named *customData* and are key-value pairs of custom key names and associated values. Observe these rules for specifying custom data:

1. The keys MAY BE both well-known (and thus, reserved) names as well as application- and vendor-specific names.
2. Keys are case sensitive.
3. The value MUST be one of the types specified in the Energistics data type *DataValue*.

3.6.5 Data Object Extension

Energistics domain standards MAY be extended similarly to ETP messages by using ExtensionNameValue elements on the data objects and specifying key-value pairs, each consisting of a custom key name and associated value. For more information, consult the relevant ML documentation.

3.7 ETP Message Patterns

To support easier implementation, the ETP design aims to use and re-use consistent messages and message patterns. ETP defines:

- "Universal" messages, which are defined in Core (Protocol 0) but may be used in any ETP protocol.
- Some specific data structures, messages, and design patterns to support maximum efficiency and performance in handling large volumes of data.

This section defines these constructs and explains how to use them. The section includes:

- Definitions for these terms: *universal message, map*, *Chunk message*, *multipart requests and responses*, and *plural messages*, (see Section **3.7.1)**.
- Usage rules for:
  - "Universal" messages, which include ***ProtocolException*** and ***Acknowledge*** (see Section **3.7.2**).
  - Plural messages, which allow multiple requests and/or responses in a single message (see Section **3.7.3**).
  - Multipart requests, responses, and notifications (which can be thought of as one large virtual request, response or notification that has been implemented as a group of related messages) (see Section **3.7.3.1**).

IMPORTANT! This specification defines which particular messages are plural (indicated by the message name and data structures in the message body) and which response, requests, and notifications MAY be implemented as a set of multiple related messages (indicated by the *multipartFlag* on the message schema set to true).

3.7.1 Message Patterns: Key Concepts and Definitions

ETP has some concepts and message patterns that have proven to be useful but may not be commonly used, so this section defines them and explains how they work together. Related usage rules for these constructs are referenced as appropriate.

3.7.1.1 "Universal" Messages (Definition)

By design, most message types in ETP are used only in a specific protocol, for a specific purpose. However, some messages may be used in any of the ETP protocols; these messages are informally referred to as "universal" messages. These universal messages are defined in Core (Protocol 0) (see Chapter **5**) and they include ***ProtocolException*** and ***Acknowledge*** messages; rules for using these messages are explained in Section **3.7.2**.

3.7.1.2 Plural Message (Definition)

A *plural message* is a general term for messages that allow multiple requests and/or responses to be included in a single message. Most messages in ETP are plural messages (indicated by message name being the U.S. English plural, **EXAMPLE:** ***PutDataObjects*** message). For rules on how to use plural messages, see Section **3.7.3**.

The general patterns for plural messages are:

- A single request message with a multipart response that follows one of these patterns:
  - A single request that allows multiple responses in one or more response messages, depending on the protocol and request (**EXAMPLE:** Get me all the channels associated with a specific

wellbore/ Discovery (Protocol 3) one ***GetResources*** message may return multiple ***GetResourcesResponse*** messages, which each contain an array of the channel identifiers.))

  – A single request message that contains a map of requests and allows one or more response messages that contain a map of responses. For definition of map, see Section **3.7.1.3**.

- Additionally, ***Chunk*** messages are defined in and may be used in several protocols. ***Chunk*** messages make it possible to sub-divide binary large objects (BLOBs) that are too big to fit in WebSocket messages into multiple smaller "chunks" that may be sent as a set of related messages—along with the standard response message (as specified in each sub-protocol where used). (For usage rules for Chunk messages, see Section **3.7.1.4**).

3.7.1.3 Map (Definition) and Use of "Small Keys"

A *map* (also known as an *associative array*) is a data structure that is a collection of key-value pairs. ETP makes extensive use of this data structure in request, response, and error (***ProtocolException***) messages.

One of the main benefits of using the map pattern in ETP is that it allows an endpoint to send a large number of requests in a single message, and for the other endpoint to respond to the requests that it can, send ***ProtocolException*** messages for requests it cannot process, and use the map keys to correctly correlate the responses and errors. Essentially maps are a batching mechanism. This pattern is used extensively and is explained in detail in Section **3.7.3**.

IMPORTANT: When map keys are assigned, they are only valid for the particular request and any associated response messages. In another request, the same values (e.g. URIs or channel IDs) may be associated with completely different keys.

REMINDER: Map keys are always case sensitive.

When assigning keys, the recommendation is to use the smallest viable identifier (e.g., 1, 2, 3…). **NOTE:**◦However, map keys are strings—not integers—because Avro does not allow integers. The keys MUST be unique within a map.

EXAMPLE: A customer, using an ETP request message with a map, sends 100 unique requests to a store and assigns keys 1–100. The store can successfully fulfill 97 of those requests, so it sends a positive response message to the customer that contains a map with the 97 requests it could fulfill (each identified by its customer-assigned map key) and a ***ProtocolException*** message with a map containing error codes for the 3 requests in error (each also identified by its unique customer-assigned key). The customer can then use those keys to determine further processing (e.g., to process the 97 correct responses and/or aim to resolve the 3 errors).

3.7.1.4 Chunk Message (Definition)

Some messages in Store (Protocol 4), StoreNotification (Protocol 5), and StoreQuery (Protocol 14) allow or require a data object to be sent with a message. If the size of the data object (bytes) is too large for the WebSocket message size (which for some WebSocket libraries can be quite small, e.g. 128 kb), you must subdivide the data object (binary large object/BLOB) and send it in "chunks" that are small enough for your WebSocket library to handle. Send these "chunks" using the ***Chunk*** message, which is defined in each of the protocols where it is used. For rules on how to use ***Chunk*** messages, see Section **3.7.3.2**.

3.7.1.5 Multipart Requests, Responses, and Notifications (Definition)

Multipart requests, responses and notifications refer to a pattern that makes it possible for an endpoint to break up something that is potentially large into a series of related, smaller messages.

IMPORTANT! Each "part" in a multipart request, response, or notification is a "complete" ETP message (i.e., each has a header and body as defined in Section **3.5**). To be precise, they are actually "multi-message" request, response, and notifications.

A multipart request, response or notification should be considered a single virtual request, response, or notification. Observe these rules for using multipart messages:

1. Only messages designated as multipart (i.e., in the Avro schema header, multipartFlag: true; see Section 3.5) may be implemented as multipart. If multipartFlag=false, the endpoint MUST send only one message of that type per request or response.
2. In the simplest usage, multipart requests, responses, and notifications are composed of 2 or more of the same type of message (**EXAMPLES**: For a multipart request, all ***ReplaceRange*** messages; for a multipart response, all ***GetResourcesResponse*** messages). For general rules on how to use multipart messages, see Section 3.7.3.1.

   However, multipart requests, responses, and notifications MAY also be composed of multiple TYPES of messages; for example, they may include:

   – One or more positive response messages with one or more ***ProtocolException*** message(s). This pattern is used with messages that contain a map data structure; for more information on how it works, see Section 3.7.3.

   – One or more types of positive responses, as defined by a specific ETP sub-protocol (**EXAMPLE:** In ChannelDataFrame (Protocol 2) the standard positive response behavior to a ***GetFrame*** request message is to return 1 (one) ***GetFrameResponseHeader*** message and 1 to *n* ***GetFrameResponseRows*** messages (where *n* is the number of messages that are needed to return all the rows that fulfill the request).

   – For store-related protocols (Protocols 4, 5, and 14) a positive response may have 1 or more associated ***Chunk*** messages. For rules on how to use ***Chunk*** messages, see Section 3.7.3.2.

Details of protocol-specific behavior are captured in the relevant protocol chapter in this specification.

3.7.2 ETP "Universal" Messages: Usage Rules

By design, most message types in ETP are used only in a specific protocol, for a specific purpose. However, some messages may be used in any of the ETP protocols; these messages are informally referred to as "universal" messages. How these so-called "universal" messages work is explained below in this section; the messages are defined Protocol 0 (Chapter 5); they include:

- ***ProtocolException*** message (see Section 3.7.2.1)
- ***Acknowledge*** message (see Section 3.7.2.2)

3.7.2.1 ProtocolException Message: Usage Rules

ETP defines a ***ProtocolException*** message (see Chapter 5) and a set of error codes (see Chapter 24). Follow these rules and guidelines for using and populating attributes in a ***ProtocolException*** message:

1. When errant behavior (as defined in this specification) occurs, you MUST send the ***ProtocolException*** message with the appropriate error code(s), which are also defined by this specification.
   a. For the list of ETP-defined error codes and guidance on their usage, see Chapter 24.
   b. This message should not be used to indicate general failures of low-level protocols (such as WebSocket, HTTP, or TCP/IP) on which ETP depends.
2. In many cases, this specification defines errant behavior/error conditions and specifies the error code that MUST be used when specific errant behavior occurs.
   a. In some general cases, not all instances of errant behavior are defined. For example, anytime an invalid argument is encountered, use error code EINVALID_ARGUMENT (5). **RECOMMENDATION:** For these general cases, implementers should familiarize themselves with the error codes and use their best judgement in using error codes.
   b. An endpoint MUST send only ONE error code per incident. If multiple errors are associated with an operation, the implementer must choose one error code only.
   c. Custom error codes MAY be defined but MUST use negative numbers. For more information, see Section 24.1.

3. ***MessageHeader***. Observe these rules for populating attributes in the ***MessageHeader*** of a ***ProtocolException*** message:
   a. In MOST CASES, the *protocol* attribute MUST be set to the ID of the protocol where the error occurred. **EXAMPLE:** If a ***ProtocolException*** message is being sent as a result of an error in Discovery (Protocol 3), then the *protoco*l attribute in its ***MessageHeader*** is set to 3.
      i. For the EXCEPTIONS to this rule, see 3.**c below**.
   b. In MOST CASES, the *correlationId* MUST be set to the *messageId* of the errant message that resulted in the ***ProtocolException*** message being sent.
      i. For the EXCEPTIONS to this rule, see 3.**c below**.
   c. These are the EXCEPTION cases to 3.**a** and 3.**b**. The *protocol* attribute SHOULD be set to 0 and the *correlationId* attribute SHOULD be set to 0:
      i. When a WebSocket message is received and the ETP ***MessageHeader*** cannot be de-serialized. (If the message cannot be de-serialized, the receiver has no way of knowing the messageId of the errant message.) The receiver MUST send error EINVALID_MESSAGE◦(19).
      ii. When sending a ***ProtocolException*** message with error code EBACKPRESSURE_WARNING (25) or EBACKPRESSURE_LIMIT_EXCEEDED (24). For more information, see Section **3.13.4**.
      iii. When sending a ***ProtocolException*** message with error EAUTHORIZATION_EXPIRING◦(28) or EAUTHORIZATION_EXPIRED (10).
      iv. **NOTE:** For **ii** and **iii**, the errant behavior is not associated with a particular message, so no *correlationId* is required.
   d. Set the *messageFlags*:
      i. The 0X08 bit MUST NEVER be set to true; a ***ProtocolException*** message MUST NOT be compressed.
      ii. All other flags follow standard usage (see Section **3.5.4.1**).
4. The ***ProtocolException*** message includes two main fields: *error* and *errors*.
   a. The *error* field MUST be populated for a single error code and related message.
   b. The *errors* field MUST be populated for a map of errors, which MUST be used in response to a request message that contains a map. (**NOTE:** A ***ProtocolException*** message may be implemented as multipart to accommodate many errors for large map requests.)
   c. In a single instance of a ***ProtocolException*** message, you MUST use one of the two error fields (either *error* or *errors*), but you MUST NOT use both in the same message.
      i. If both *error* and *errors* is populated, the behavior is undefined.
5. For brevity in this specification, the text used for raising an error is "send error *Name* (*N)*", where *Name* and *N* are actual error code names and numbers defined by ETP, such as "send error EUNSUPPORTED_PROTOCOL (4)". This text means send the ***ProtocolException*** message with the named error code. For the example shown, it means send the ***ProtocolException*** message with error code 4, which is named EUNSUPPORTED_PROTOCOL.
6. For additional information about using ***ProtocolException*** messages with plural and multipart messages, see Section **3.7.3**.

3.7.2.2 Acknowledge Message: Usage Rules

ETP offers the flexibility for an endpoint to request an ***Acknowledge*** message (informally referred to as an "Ack") for each ETP message sent.

Use the ***Acknowledge*** message (defined in Chapter **5**) in any protocol where specific acknowledgement of *receipt of a message* is needed. That is, *an Ack is confirmation that a message was received; it does NOT indicate that an action was completed.*

An Ack is a logically separate response message from any behavioral responses defined in specific ETP sub-protocols.

In certain cases, use of Acks is prohibited. Rules and requirements are specified in the following list. **NOTE:** This section documents behavior specific to Acks, which fits into the larger message sequence documented in Section **3.5.3**.

Follow these rules for requesting and sending Acknowledge messages:

1. An endpoint MUST send an ***Acknowledge*** message ONLY when the other endpoint in an ETP session has requested an Ack.
2. For an endpoint to request an Ack for a given message, it MUST set the 0x10 bit in the *messageFlags* field (bit map of flags in hexadecimal values) in the ***MessageHeader*** of the message that requires acknowledgement. For more information on available *messageFlags* and other fields in the ***MessageHeader***, see Section **3.5.4.**
   a. An ***Acknowledge*** message MUST NEVER have its 0x10 bit set (that is, an Ack MUST NOT be Acked—this would create an endless loop).
   b. Either endpoint/role MAY request an ***Acknowledge*** message.
3. The endpoint that receives the message (the receiving endpoint) MUST do the following:
   a. The receiver MUST first inspect the *messageFlags* field in the ***MessageHeader*** to determine if the sender is requesting an ***Acknowledge*** message. BOTH of these conditions must be true, to send an Ack:
      i. The 0x10 flag is true, indicating that the other endpoint is requesting an Ack.
      ii. The message is NOT an ***Acknowledge*** message.
   b. If both conditions are true, the receiving endpoint MUST immediately send an ***Acknowledge*** message, before any other processing of the ***MessageHeader*** and before any attempt to de-serialize and/or process an (optional) message header extension or the message body.
      i. This approach means an ***Acknowledge*** message always precedes any other messages sent in response (to the message that requested the Ack) i.e. Ack will precede any other message with the same *correlationId* as the Ack message, including any ***ProtocolException*** messages.
   c. The receiving endpoint MUST send the Ack in addition to any response message(s) specified by the sub-protocol (which may not be sent until much later, for example, depending on the size and complexity of the request).
   d. The endpoint that is sending the Ack MUST populate the ***Acknowledge*** message according to these rules:
      i. The message body has no content. Any meaning is embodied in the receipt of the Ack itself.
      ii. ***MessageHeader***. Observe these rules for setting values for fields in the ***MessageHeader***:
         1. The *protocol* field in the ***MessageHeader*** of the ***Acknowledge*** message MUST be set to the protocol ID of the ETP sub-protocol that sent the message to be acknowledged.

            EXAMPLE: If an ***Acknowledge*** message is being requested for a message in Discovery (Protocol 3), then the *protocol* field in its ***MessageHeader*** is set to 3.

            NOTE: Because ***Acknowledge*** is one of the messages defined in Protocol 0 that may be used in any protocol, *protocol* should only be set to 0 if you are acknowledging a

Protocol 0 message.)

2. The *correlationId* in the ***MessageHeader*** of the ***Acknowledge*** message MUST be set to the ID of the message (*messageId* field) that requested the acknowledgement.

   NOTE: Each message in ETP MUST have a *messageId* unique to the endpoint in an ETP session; for ETP message ID numbering requirements, see Section **3.5.4**.

3. Set the *messageFlags*. Observe these details for setting the *messageFlags on an* ***Acknowledge*** message:
   - The 0x02 bit (FIN bit) MUST ALWAYS be set to true. That is, an Ack is always only a single ETP message.
   - The 0x10 bit MUST NEVER be set to true; an Ack MUST NOT be acked.

4. Cautions for using ***Acknowledge*** messages:
   a. Consider use of Acks carefully. While ETP has no restrictions on their use, overuse of Acks can degrade performance. For example, in general it would not be good practice to request Acks for every ***ChannelData*** message in a streaming protocol (such as ChannelSubscribe (Protocol 21)). However, in some cases (for example, poor phone line connection), use of Acks on every part could be beneficial.
   b. **NOTE:** In ETP v1.1, the ***Acknowledge*** message was also used with the 0x04 bit to indicate a response of "no data" to a specific request. The "no data" responses have been clarified in ETP v1.2 and this 0x04 bit is no longer used.

3.7.3 Usage Rules for "Plural Messages"

ETP defines a message pattern that allows for multiple requests and responses in a single message; this pattern is informally referred to as a "plural message". Request, response and some notification messages can be plural. For a complete definition of plural message, see Section **3.7.1.2**.

NOTE: Multipart requests and responses are a specific type of plural messages (explained below). Those messages are subject to the design patterns and rules specified here AND the additional patterns and rules specified in Section **3.7.3.1.**

Design patterns and rules for plural messages:

1. Plural messages are indicated by use of plural words in the message names.
   (In U.S. English, plural words are usually indicated by adding the letter "s" to the end of an object name, for example, the ***GetDataObject***s and ***GetDataObject***s***Response*** messages in Store (Protocol 4).)
   a. Both request and response messages MAY be plural. However, in some cases, request messages (though having a plural name) may be a single request that allows a plural response.
2. By definition, a plural request message is one of these:
   a. **A Get or Find request WITHOUT a map.** These messages represent a single request that allows a plural response. In Discovery (Protocol 3) one ***GetResources*** message may return multiple ***GetResourcesResponse*** messages, each of which contains an array of resources. (In general, each request message has a corresponding response message indicated by name, e.g., ***GetResources*** and ***GetResourcesResponse***; for information about types of messages and naming conventions, see Section **3.4.2**.)
   b. **Any request message WITH a map.** The map is a collection of key-value pairs, which makes it easier to match requests with responses and/or errors. Each item in a map has an assigned map key that MUST be unique within the context of that map. **RECOMMENDATION:** Use the smallest viable key (e.g., 1, 2, 3…), which MUST be a string, NOT an integer. For a complete definition of *map*, see Section **3.7.1.3**.

   i. The endpoint making the request that contains the map MUST assign the map keys.

   ii. If a map is used in a multipart response or request, the keys MUST be unique across all messages of the multipart response/request. **EXAMPLE:** The ***OpenChannelsResponse*** messages in ChannelDataLoad (Protocol 22) is a multipart message with a map; it is a response that lists the channels a store can accept data for, each channel must be identified by a map key unique across all messages that the make up the response.

   iii. As previously specified (in Section **3.5.4**), the final message in a request or response MUST have its FIN bit set (*messageFlags* 0x02 in the ***MessageHeader***). For information on setting FIN bits on multipart requests, responses, and notifications, see Section **3.7.3.1**.

3. **A Get or Find request WITHOUT a map MUST have ONE of the following as a response:** a) the ETP-defined response message that contains items that the store could return, b) the ETP-defined response message with an empty array (no data was found that met the request criteria) or c) a ***ProtocolException*** message (PE) with the *error* field populated (i.e., a single error for the entire request) with the appropriate error code).
   EXAMPLE: In Discovery (Protocol 3), the possible responses to a ***GetResources*** request message are: a) one or more ***GetResourcesResponse*** message with the array of resources that the store could return, b) one ***GetResourcesResponse*** message with an empty array (the URI is valid but no data meeting the criteria specified in the request was found) or c) a ***ProtocolException*** message, e.g., if the URI in the request is malformed, the PE would contain in its *error* field EINVALID_URI (9).

   a. If the response is multipart, it is possible that an endpoint might begin sending response messages and THEN encounter an error (e.g., a server exception occurs). In this case, the server MUST do all of the following:

      i. Send a ***ProtocolException*** message with an appropriate error code.

      ii. Stop processing the request that caused the error.

      iii. Stop sending any additional response messages for that request.

   b. As previously specified (in Section **3.5.4**), the final message in a request or response MUST have its FIN bit set (*messageFlags* 0X02 in the ***MessageHeader***). For more information on setting FIN bits on multipart requests, responses, and notifications, see Section **3.7.3.1**.

4. **A map request MUST have as a response:** a) zero or more positive map response messages, b) zero or more map ***ProtocolException*** errors, and c) zero or one terminating, non-map ***ProtocolException*** error.

   a. If the response is multipart, it is possible that an endpoint might begin sending response messages and THEN encounter an error (e.g., a server exception occurs). In this case, the server MUST do all of the following:

      i. Send a terminating, non-map ***ProtocolException*** message with an appropriate error code.

      ii. Stop processing the request that caused the error.

      iii. Stop sending any additional response messages for that request.

   b. Otherwise, if no terminating errors are sent:

      i. Each key from the map in a map request MUST appear either as the key in a positive response map or as the key in the *errors* map in a ***ProtocolException*** message; that is, each request in a map was either successfully completed or results in an error, but not both.

      ii. If a request message results in both positive responses and errors, the number of returned positive responses and the number of errors in ***ProtocolException*** MUST equal the total number of request items.

   c. Terminating ***ProtocolException*** messages are intended for store-wide or request-wide failures that are unrelated to the success or failure of individual requests within the request message. In response to these errors, the customer MAY try the request again later or try to split it into smaller groups of individual requests. These errors will not help the customer correct errors within the individual requests.

   i. An example of a store-wide error could be a store losing its database connection.

   ii. Examples of request-wide errors could be an unhandled exception processing the request that prevents further processing or exceeding the endpoint's value for the MultipartMessageTimeoutPeriod capability.

d. Map ***ProtocolException*** messages are intended to provide specific failure reasons for individual requests within the request message. In response to these errors, the customer SHOULD attempt to correct the individual requests based on the specific error received. **EXAMPLE:** For EINVALID_OBJECT, the customer SHOULD attempt to fix issues with the object data.

e. Terminating ***ProtocolException*** messages are NOT a substitute for map ***ProtocolException*** messages.

   i. Customers are likely to simply retry requests that fail with a terminating ***ProtocolException*** message or send different subsets of the request to try to narrow down the potential problem.

f. If a store MUST send a terminating ***ProtocolException*** message, it SHOULD attempt to send all positive responses and all map error responses it is able to before sending the terminating ***ProtocolException*** message.

g. A store MUST NOT use a terminating ***ProtocolException*** message as a convenience mechanism to avoid sending map error responses, even if all map error responses to a request are the same.

5. The ***ProtocolException*** message(s) that contain the error responses to a map request MUST have:
   a. *protocolId* in the ***MessageHeader*** set to the protocol number that the request message was issued from. **EXAMPLE:** If the ***ProtocolException*** is in response to a ***GetResources*** message, the *protocolId* is "3" (for Discovery (Protocol 3).
   b. *correlationId* in the ***MessageHeader*** set to the request message (*messageId*) that it is a response to.
   c. An error code for each item in the *errors* map.
6. As previously specified (in Section **3.5.4**), the final message in a request or response MUST have its FIN bit set (*messageFlags* 0X02 in the ***MessageHeader***). For more information on setting FIN bits on multipart requests, responses, and notifications, see Section **3.7.3.1**.

3.7.3.1 Usage Rules for Multipart Requests, Responses, and Notifications

Multipart requests, responses, and notifications make it possible for an endpoint to break up something that is potentially large into a series of related, smaller messages. This set of messages should be considered a single virtual request, response, or notification. For detailed definitions, see Section **3.7.1.5**.

The patterns and rules specified here MUST be followed in addition to the "plural message" rules specified just above.

Design patterns and rules for multipart request, response, and notification messages:

1. Only ETP messages with multipartFlag = true may be implemented as a series of related messages. This flag appears in the header information of the Avro schema that defines each ETP message (see Section **3.4.1**) and in each section of this specification that defines a message.
   a. How to partition and group data for multipart requests, responses, and notifications is an implementation issue; the ETP Specification provides no guidance or recommendations on how to do this. Each implementer determines its own approach.
      i. Message sizes MUST honor each endpoint's MaxWebSocketMessagePayloadSize capability (see Section **3.3.2.9**).

b. To protect performance (e.g., throughput), ETP specifies capabilities so that endpoints can impose limits to size, concurrency, and duration of multipart requests, responses, and notifications. (For more information about capabilities and how they work, see Section **3.3**.) The capabilities pertaining to multipart messages include those listed here, with links to details of required behavior when using them:

   i. ResponseTimeoutPeriod (endpoint) (see Section **4.b below**)

   ii. MaxResponseCount (protocol) (see **4.d below**)

   iii. MaxConcurrentMultipart (endpoint) (see Section **3.7.3.1.1.1 below**)

   iv. MultipartMessageTimeoutPeriod (endpoint) (see Section **3.7.3.1.1.2 below**)

2. By definition, a multipart request, response, or notification MUST be bounded. (That is, a multipart request, response, or notification should be considered a single virtual request, response, or notification.)

3. As described in Section **3.5.4**, each message in an ETP session MUST be uniquely numbered (using the *messageId* field in the ***MessageHeade***r)—this rule applies to each message of a multipart response, request, or notification, related Chunk messages (if used, see Section **3.7.3.2**) and ***ProtocolException*** messages (if used, see Section **3.7.2.1**).

   a. For *messageId* requirements, see Section **3.5.4**.

4. A multipart response:

   a. MUST correlate to a specific request message.

      i. For usage rules for *correlationIds* (included in the ***MessageHeader*** of each ETP message), see numbers **6** and **7 below**.

   b. Data messages (such as ***ChannelData*** messages sent in Protocols 1, 21 or 22) ARE NOT responses and ARE NOT multipart; they are individual messages containing data that are sent as they become available.

      i. For types of messages and naming conventions, see Section **3.4.1**.

   c. MUST begin within the customer's ResponseTimeoutPeriod endpoint capability. That is, after receiving a request from a customer, a store MUST send the standalone response message or the first message in a multipart response no later than the value for the customer's ResponseTimeoutPeriod.

      i. If the store cannot respond within the customer's ResponseTimeoutPeriod, the store MAY cancel by sending error ETIMED_OUT (26).

      ii. If the store's value for ResponseTimeoutPeriod is less than the customer's value, and the store exceeds its limit, then the store MAY cancel the response by sending error ETIMED_OUT (26).

      iii. If a customer receives an ETIMED_OUT error, it may indicate that the session has become congested or the store has encountered other "abnormal circumstances."

   d. MAY include a combination of valid response messages and errors, which MAY specifically include:

      i. **Valid designated response messages** as defined by each ETP sub-protocol (**EXAMPLES:** in Store (Protocol 4) one or more ***GetDataObjectsResponse*** messages may be returned in response to a ***GetDataObjects*** request message; in ChannelDataFrame (Protocol 2), one ***GetFrameResponseHeade***r and one or more ***GetFrameResponseRows*** are returned in response to a ***GetFrame*** request message).

      ii. **Chunk message.** This message is used in 3 store-related protocols (Protocols 4, 5 and 14); it makes it possible for the store to attach potentially large data objects (which may be included with some request, response, and notification messages) as binary large objects (BLOBs) partitioned into manageable sized "chunks". For more information on how the Chunk message works, see Section **3.7.3.2**.

iii. **ProtocolException message(s).** The map construct makes it possible to submit multiple requests in a single message and have some requests pass and some requests fail (instead of the entire request failing); in this case, the response is a mix of valid response messages (for the requests the store could fulfill) and ***ProtocolException*** messages (for requests that resulted in errors).

- Errors MUST be sent in one or more ***ProtocolException*** messages, which MAY also be a series of multiple related messages. For more rules related to ***ProtocolException*** messages as part of a multipart response, see numbers **5, 8** and **9** below.

e. A Store MUST limit the total count of response items it returns in response to one non-map request to the customer's MaxResponseCount value. (The MaxResponseCount is an endpoint capability; it is the maximum total count of responses allowed in a complete multipart message response to a single non-map request.) **EXAMPLE:** A store must not return more than MaxResponseCount ***Resource*** records in response to a ***GetResources*** request message.

i. If the store's MaxResponseCount value is smaller than the customer's MaxResponseCount value, the store MAY send fewer response items.

ii. If the store's response exceeds this limit, the customer MAY notify the store by sending error ERESPONSECOUNT_EXCEEDED (30).

iii. If a store cannot send all responses to a request because it would exceed the lower of the customer's or the store's MaxResponseCount value, the store:

1. MUST terminate the multipart response by sending error ERESPONSECOUNT_EXCEEDED (30).
2. MUST NOT terminate the response until it has sent MaxResponseCount responses.

iv. **NOTE:** In some protocols, there are additional capabilities that limit the response count to specific requests. For example, in ChannelSubscribe (Protocol 21), MaxRangeDataItemCount limits the count of ***DataItem*** records sent in response to a ***GetRanges*** request. These capabilities are documented in the relevant protocols, but the behavior for these is as described here for MaxResponseCount: if the limit would be exceeded, the store MUST send ERESPONSECOUNT_EXCEEDED (30) and the store MUST NOT send this until it has sent the maximum number of responses allowed by the capability.

5. **Message Flags/FIN bit.** For all ETP messages, the ***MessageHeader*** contains a *messageFlags* attribute. This attribute acts as a bit-field and allows multiple Boolean flags to be set on a message (for the complete list of flags, see Section **23.25**). The 0x02 flag indicates that the message is the last message for a request, response or notification (a FIN bit). (**NOTE:** FIN bit is set for every "action"; that is, each message or set of messages that serve as a request, response, notification or data. For example: If a request message is composed of only a single message, its FIN bit MUST be set. Follow these rules to set the FIN bit for multipart requests, responses or notifications:

a. For a multipart request, you MUST set the FIN bit on the last message of the request ONLY.

b. For a multipart response, you MUST set the FIN bit on EITHER: the last message of the multipart response OR on a related ***ProtocolException*** message (per number **4 above**), depending on which message is sent last (see also, numbers **8** and **9** below.)

i. The final message in a multipart response MAY be an "empty" response message with the 0x02 flag set.

c. For multipart notifications, you MUST set the FIN bit on the last message of the multipart notification.

6. **CorrelationId:** For a multipart REQUEST or NOTIFICATION:

   a. The correlationId of the first message MUST be set to 0 and the correlationId of all successive messages in the same multipart request or notification MUST be set to the messageId of the first message of the multipart request or notification.

7. **CorrelationId:** For a multipart RESPONSE:
   a. The correlationId of each message that comprises the response MUST be set to the messageId of the request message.
   b. If the request message is itself multipart, the correlationId of each message of the multipart response MUST be set to the messageId of the FIRST message in the multipart request.
   c. When an endpoint receives a response message with the 0x02 flag set (which indicates it is the last part), the endpoint will NOT receive any additional response messages with the same correlationId. (i.e., after sending a message with the FIN bit set, the sending endpoint MUST NOT send any additional messages with the same correlationId).
   d. When an endpoint receives a notification message with the 0x02 flag set (which indicates it is the last part), the endpoint will NOT receive any additional notification messages with the same correlationId. (i.e., after sending a message with the FIN bit set, the sending endpoint MUST NOT send any additional messages with the same correlationId).
8. **ProtocolException messages.** If a response includes one or more ***ProtocolException*** messages:
   a. Its correlationId must be set to the correlationId of the request message that caused the error.
   b. If the ***ProtocolException*** message is the last part in the multipart response, then its 0x02 bit flag MUST be set (indicating it is the last part).
9. If a catastrophic error occurs in the middle of a multipart response:
   a. The sender MUST send a ***ProtocolException*** message with a single error EMULTIPART_CANCELLED (18) (in the *error* field) and:
      i. NO map keys are populated.
      ii. The FIN bit is set.
      iii. **NOTE:** This is the only situation in which a ***ProtocolException*** message that is part of a multipart response can have an empty map.
   b. The receiver MUST treat this situation as a cancellation of the entire operation (because multipart messages are treated as an atomic operation).
10. **Message content.** For a multipart request, response, or notification, only the content of the collection field (i.e., array or map) may vary between the message parts. The values for all other fields carrying metadata MUST be identical for all message parts of a multipart message. **EXAMPLES:**
   a. For GrowingObject (Protocol 6), a multipart ***ReplacePartsByRange*** request MUST contain the same values for *uri*, *format*, *deleteInterval*, and *includeOverlappingIntervals* within all message parts; the only data that may change among messages is the content of the *objectParts* collection.
11. **RECOMMENDATION:** To avoid sending more individual messages than necessary when sending multipart requests, responses and notifications, group together data, where possible. **EXAMPLE:** Group all error responses to a map request into a single ***ProtocolException*** message, if doing so does not exceed MaxWebSocketFramePayloadSize.
12. **WARNING:** Use of multipart requests, responses, and notifications in data-movement protocols (such as ***PutDataObjects*** in the Store (Protocol 4) may create mutating or racing conditions. Currently, ETP does not attempt to handle these conditions. Identifying and addressing these conditions is up to the developer/implementer. The safest thing for client applications to do now is to ensure they do not issue concurrent, competing requests to a store.

3.7.3.1.1 Multipart Message Capabilities

This section defines and specifies required behavior for endpoint capabilities that apply generally and exclusively to multipart messages.

3.7.3.1.1.1 MaxConcurrentMultipart (Endpoint)

DEFINITION: The maximum count of multipart messages allowed in parallel, in a single protocol, from one endpoint to another. The limit applies separately to each protocol, and separately from client to server and from server to client. The limit for an endpoint applies to the multipart messages that the endpoint can receive.

EXAMPLE: If an endpoint's MaxConcurrentMultipart is 5, then it can receive 5 multipart messages—each with any number of parts—at one time, in Store (Protocol 4) and another 5 messages in process in Discovery (Protocol 3). In Discovery (Protocol 3), this could be the multipart responses to 5 distinct ***GetResources*** requests.

Value units: <count of messages>

Min: 1

REQUIRED BEHAVIOR:

1. A customer MUST limit the count of multipart requests that it makes in parallel in a protocol to the store's value for MaxConcurrentMultipart.
    a. If the customer exceeds this limit, the store MUST deny multipart requests that exceeds this limit by sending error EMULTIPART_CANCELLED (18).
2. A Store MUST limit the count of multipart responses and notifications that it sends in parallel to the customer's value for MaxConcurrentMultipart.
    a. If sending a multipart response to a request would require a store to exceed the customer's limit, the store MUST abort the entire multipart response by sending error EMULTIPART_CANCELLED (18).
    b. If sending a multipart notification would require a store to exceed the customer's limit, the store MUST separate and send the multipart notification message content as distinct, "single" messages, with the FIN bit (*messageFlags* 0x02) set on each message.
    c. When sending notifications in this manner for notifications that would normally require the use of ***Chunk*** messages (see Section **3.7.3.2**) the store MUST NOT send associated object data, even if the customer requested the object data.

3.7.3.1.1.2 MultipartMessageTimeoutPeriod (Endpoint)

DEFINITION: The maximum time period in seconds—under normal operation on an uncongested session—allowed between subsequent messages in the SAME multipart request or response. The period is measured as the time between when each message has been fully sent or received via the WebSocket.

Value units: <count of seconds>

Max: 60 seconds

REQUIRED BEHAVIOR: This behavior is expected under normal operation on an uncongested session.

1. A customer MUST send each message in a multipart message no later than the store's value for MultipartMessageTimeoutPeriod after the previous message in the same multipart message.
    a. If the store's limit is exceeded, the store MAY cancel multipart requests by sending error ETIMED_OUT (26).
    b. If a customer receives error ETIMED_OUT, it may indicate that the session has become congested or the store has encountered other abnormal circumstances.
2. A store SHOULD send each message in a multipart message no later than the customer's value for

MultipartMessageTimeoutPeriod after the previous message in the same multipart message.

   a. If the customer's limit is exceeded, the customer MAY cancel the multipart response by sending error ETIMED_OUT (26).

3. If the store's value for MultipartMessageTimeoutPeriod is less than the customer's value and the store exceeds its limit, the store MAY cancel the rest of the response with ETIMED_OUT (26).

3.7.3.2 Sending Binary Large Objects (BLOBs) in ETP

Some messages in Store (Protocol 4), StoreNotification (Protocol 5), and StoreQuery (Protocol 14) allow or require a data object to be sent with the message. If the size of the data object is too large (bytes) for an endpoint's MaxWebSocketMessagePayloadSize (an endpoint capability negotiated for an ETP session) (**NOTE:** For some WebSocket libraries, frame and message sizes can be quite small, e.g. 128 kb), you must subdivide the data object (BLOB) and send it in "chunks" that are small enough for both endpoints to handle. Send these "chunks" using a set of related ***Chunk*** messages, which is defined in each of the protocols in which it appears (for example, in Chapters **9**, **10**, **16**).

Recommendations for using Chunk messages:

- Use ***Chunk*** messages to overcome data objects that are too large (bytes) for the negotiated WebSocket message size limit for an ETP session. This limit—MaxWebSocketMessagePayloadSize (and its companion MaxWebSocketFramePayloadSize)—is an endpoint capability negotiated when the ETP session is established. For more information, see Section **3.3.2**.
  - Use of ***Chunk*** messages DOES NOT address data objects that exceed the limits set by the MaxDataObjectSize capability (see Section **3.3.2.4**).
- Optimize use of ***Chunk*** messages, for example, by making these messages as large as your WebSocket limits allow.
- If not needed, DO NOT use ***Chunk*** messages.

The ***Chunk*** message is defined in each of the protocols listed above and it works the same in all cases. The ***Chunk*** message is implemented as part of a multipart request or response and thus MUST follow the rules defined in Section **3.7.3.1**.

Any message that allows or requires a data object to be sent, contains a field called *dataObjects*, which is a map composed of the ETP data type ***DataObject*** (which is defined in Section **23.34.5**). If the specific data object you are sending is too large, do the following:

1. Assign a BLOB ID to the data object that you want to send and enter it in the *blobId* field on the ***DataObject*** record.
   a. The *blobId* MUST be a UUID, and it MUST be unique within an ETP session.
   b. When you populate the *blobId* field, the *data* field (on the ***DataObject*** record) MUST be empty.
   c. Populating the *blobId* field means that the actual data will be sent in a set of ***Chunk*** messages (not in the ***DataObject***).
2. Divide the data object into appropriate sized "chunks" (to accommodate the negotiated WebSocket message size limit for the session) and send the content in multiple ***Chunk*** messages.
   a. Each ***Chunk*** message MUST contain the BLOB ID assigned to the data object.
   b. The correlationId for each ***Chunk*** message MUST be the *messageId* of the message that resulted in the need to create chunks. (The message varies by ETP sub-protocol; **EXAMPLE:** In Store (Protocol 4) the ***PutDataObjects*** message and the ***GetDataObjectsResponse*** message may result in the need for a set of ***Chunk*** messages.
      i. All ***Chunk*** messages associated with a given "parent message" belong to the multipart request or response. **EXAMPLE:** If a ***PutDataObjects*** operation is putting 3 data objects that all must be sent using ***Chunk*** messages, then the ***PutDataObjects*** message and all ***Chunk*** messages

(for all 3 data objects) are part of the same multipart request.

c. The data object MUST be partitioned and each ***Chunk*** message MUST be sent in order, as indicated by the *messageId* (described in Section **3.5.4**).

   i. The last ***Chunk*** message for the data object MUST have the *final* field set to true. Because a ***Chunk*** message MUST be sent in the context of another request, response or notification message, which may be multipart itself, the ***Chunk*** message has its own *final* flag field (in the body of the ***Chunk*** message), indicating the last chunk for one data object.

   ii. The receiver of the messages uses the *blobId*, *messageId* and *final* fields to re-assemble the data object in its correct order.

d. ***Chunk*** messages for different objects MUST NOT be interleaved within the context of one multipart message operation.

   i. If more than one data object must be sent using ***Chunk*** messages, you MUST finish sending all chunks for each data object before sending the chunks for the next data object.

3. If a ***Chunk*** messages is the last message in a multipart request, response or notification, the sender MUST set the FIN bit in the message header. **EXAMPLE:** In the example on 2.b.**i above**, the FIN bit MUST be set on the last ***Chunk*** message of the third data object.

3.7.4 How and "Where" URIs are Used in ETP (General Usage Rules)

For information on data objects, resources, and Energistics identifiers, see **Appendix: Energistics Identifiers**.

IMPORTANT Information About URIs:

- In most cases, when this document refers to a URI it is referring to the canonical Energistics URI, which is explained in **Appendix: Energistics Identifiers**, Section **25.3.5**.
- For rules on encoding URIs, see Section **3.12.2**.

Most messages in ETP require one or more URIs. However, because of how ETP is designed (use of record structures to compose messages as explained in Section **3.4.1.1**), WHERE you specify the URI may vary. The possible options:

- In a *uri* or *uris* field directly in a message, for example, as the map value in the *uris* field of the ***GetDataObjects*** message in Store (Protocol 4).
- In a ***Resource*** record (see Section **23.34.11**), which is used to construct several response messages in Discovery (Protocol 3) and the query protocols (see **Figure 4** above).
- In the ***ContextInfo*** record (see Section **23.34.15**), which is referenced from the ***SubscriptionInfo*** record (Section **23.34.16**), which are used to construct several messages in the notification protocols (StoreNotification (Protocol 5) and GrowingObjectNotification (Protocol 6)) (see **Figure 4** above).

EXAMPLE: A typical ETP workflow may be as follows:

1. The customer role, using Discovery (Protocol 3), sends a ***GetResources*** message with the default dataspace URI **eml:///** (which is specified in the ***ContextInfo*** record).
2. The store role responds with the ***GetResourcesResponse*** message, which provides an array of ***Resource*** records, each of which MUST specify a canonical Energistics URI for each resource returned.
   a. If the store supports alternate URIs (as defined in Section **25.3.9** with usage rules explained below in Section **3.7.4.1**), the store MAY also return one or more alternate format URIs in the *alternateUris* field of a ***Resource*** record.
3. The customer can then use one or more of the returned URIs (either the canonical or an alternate URI) in other messages for other actions. For example, it can use a returned canonical Energistics URI for a resource to get the associated data object form the store, using the ***GetDataObjects***

message in Store (Protocol 4).

3.7.4.1 Rules for Using Alternate URI Formats in ETP

For flexibility, ETP supports use of alternate URI formats (in addition to the required canonical URI format). You MUST observe these rules for using alternate URI formats:

1. If a server supports alternate URI formats, it MUST indicate that support by setting the endpoint capability named *SupportsAlternateRequestUris* to true.
    a. Endpoint capabilities are discovered in the ***ServerCapabilities*** record (when establishing the HTTP/WebSocket connection; see Section **4.3**) and exchanged in Core (Protocol 0) in the ***RequestSession*** and ***OpenSession*** messages (see Chapter **5**).
    b. If a store sets the *SupportsAlternateRequestUris* to false, a customer MUST only use the canonical Energistics URI, else the server MUST send error EINVALID_URI (9).
2. All alternate URIs MUST follow the rules specified in Section **25.3.9**.
3. If a store supports alternate URIs:
    a. The store MUST return its allowed alternate URIs in Discovery (Protocol 3) in the ***GetResourcesResponse*** message (***Resource*** record) (see Section **23.34.11**).
    b. The store is expected to support these alternate URIs in all protocols that the store supports where alternate URIs are allowed.
    c. There is no expectation that alternate URIs can be used in a different store.
4. A customer SHOULD only send/use alternate URIs (e.g., in other protocols/messages) that it received from the store (i.e., an alternate URI that the store returned in Discovery (Protocol 3) in the ***GetResourcesResponse*** message).

3.7.4.2 Rules for when Alternate URIs MAY Be Used and when Canonical URIs MUST Be Used

Canonical Energistics URIs must be used in some messages even when the use of alternate URIs has been negotiated for a session. In the following messages, canonical Energistics URIs MUST ALWAYS be used:

1. All Discovery (Protocol 3) requests (i.e., ***GetResources*** and ***GetDeletedResources***).
2. All GetSupportedTypes (Protocol 25) requests (i.e., ***GetSupportedTypes***).
3. All Put and Delete operations in Store (Protocol 4) and Dataspace (Protocol 24) (e.g., ***PutDataObjects*** in Store (Protocol 4) and ***DeleteDataspaces*** in Dataspace (Protocol 24).
4. All query protocol requests (e.g., ***FindResources*** in DiscoveryQuery (Protocol 13).
5. All response messages. **EXCEPTION:** Store-supported alternate URIs in the *alternateUris* field on ***Resource*** records.
6. All notification messages. **EXCEPTION:** Store-supported alternate URIs in the *alternateUris* field on ***Resource*** records.

In all other request messages, if use of alternate URIs has been negotiated for the session, then alternate URIs MAY be used.

For the specific rules for individual message, see the documentation for each message that uses URIs.

3.8 Avro Serialization

The serialization of messages in ETP follows a subset of the Apache Avro specification (http://avro.apache.org/docs/current/spec.html). Avro is a system for defining schemas and serializing data objects according to those schemas. It was developed as a part of the Hadoop® project to provide a flexible, high-speed serialization mechanism for processing big data. Avro was selected for ETP after a review of several similar serialization systems.

NOTE: Unlike XML, Avro has no concept of a well-formed vs. valid document or a generic document node model; thus, it is not possible to de-serialize an Avro document without knowledge of the schema of that document. As such, ETP provides capabilities for a client to discover which versions of ETP a server supports, and clients request a specific ETP version, with associated schemas, when establishing the WebSocket connection.

ETP v1.2 is based on Avro v1.10 but remains compatible with Avro v1.8.2.

ETP uses only this subset of the Avro functionality:

1. ETP **does** define all messages using the Avro schema file format. The formal definitions of these schemas are defined in UML class models using Enterprise Architect (EA). The Avro schema files are generated from this UML model.
2. ETP **does** serialize all messages on the wire in accordance with the Avro serialization rules.
3. ETP **does not** use the Avro RPC facility.
4. ETP **does not** use the Avro container file facility.
5. ETP **does** use the additional schema attributes (permissible in Avro) to define message and protocol metadata (as described in Section **3.5.5**).
6. ETP makes minimal use of Avro logical types. Similar ETP types predate the equivalent Avro logical type and/or were considered better suited to ETP use cases.

3.8.1 Supported Data Encoding

The Avro specification supports the use of both **binary** and **JSON** (JavaScript Object Notation) encoding of data. ETP also supports the use of both, with the following caveats:

1. ETP production implementations MUST use binary encoding.
2. JSON encoding of ETP messages is for internal testing and debugging only; JSON encoding MUST NOT be used for ETP production implementations.
    a. JSON has significant limitations with encoding some data (for example, NaN in doubles) and does not meet wire-size or performance requirements for ETP.
    b. Avro has certain conventions on how to construct JSON messages; however, these conventions are not commonly supported by Avro libraries or commonly used JSON libraries. This lack of support has proven problematic to interoperability; therefore, you cannot expect consistent JSON representation of messages across different implementations.
3. JSON support is not required for ETP compliance, and its presence MUST NOT be relied on.
4. For more information about encoding rules in ETP, see Section **3.12.1**.

3.9 WebSocket Transport

ETP is designed to use the WebSocket protocol for transport. The WebSocket protocol was selected because it guarantees reliable, in-order delivery of messages.

A full description of WebSocket is beyond the scope of this document. In brief, WebSocket is a protocol, standardized by the Internet Engineering Task Force (IETF) as RFC 6455 (http:/tools.ietf.org/html/rfc6455), which allows for high-speed, full-duplex, binary communication between endpoints (primarily Web servers and browsers) using TCP and the standard HTTP(s) ports 80/443. This approach allows communications to easily and safely cross many corporate firewalls.

Like WebSocket, ETP communication is strictly between two parties, with no allowance for multicast messages.

3.9.1 How ETP is Bound to WebSocket

ETP is bound to WebSocket in these main ways:

1. ETP is considered a sub-protocol of WebSocket as defined in Sections 1.9 and 11.5 of RFC 6455.
2. ETP sessions start with and may use optional headers in the WebSocket opening handshake.
3. An ETP message (the message header, optional message header extension (if used), and message body) maps directly to a WebSocket message (Figure 8), which in turn is composed of the "application data" sections of WebSocket data frames. As shown in the figure, both the header and body of an ETP message are sent in the same WebSocket message. In most cases, these details are invisible to developers, because developers use a vendor-supplied library to interface with WebSocket.

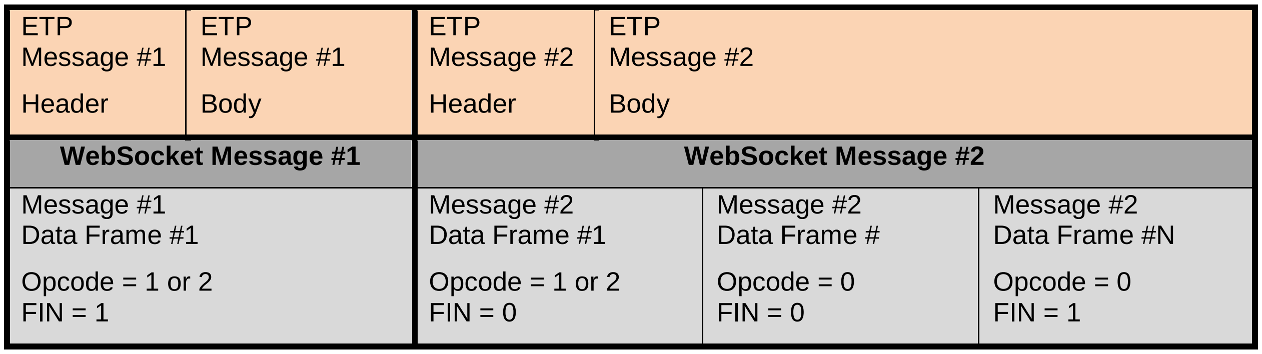

ETP Message #1 Header
ETP Message #1 Body
ETP Message #2 Header
ETP Message #2 Body
WebSocket Message #1
WebSocket Message #2
Message #1 Data Frame #1 Opcode = 1 or 2 FIN = 1
Message #2 Data Frame #1 Opcode = 1 or 2 FIN = 0
Message #2 Data Frame # Opcode = 0 FIN = 0
Message #2 Data Frame #N Opcode = 0 FIN = 1

Figure 8: ETP standard message layout mapped to WebSocket message layout.

NOTE: Unlike earlier Energistics standards based on SOAP and XML (e.g., WITSML v1.x), ETP has no concept of an 'envelope' schema that contains the entire ETP message (however, the WebSocket payload length field plays the same role in terms of defining the extent of the message content).

3.9.1.1 WebSocket Message Fragmentation

The WebSocket message itself may be fragmented into multiple WebSocket data frames per the WebSocket protocol (See "Message #2" in **Figure 8**). At the WebSocket layer, a WebSocket message does not have its own header, but each WebSocket frame that comprises a WebSocket message does have its own WebSocket frame header. The WebSocket frame header is described in RFC 6455 (https://tools.ietf.org/html/rfc6455).

You MUST use a WebSocket library that handles fragmentation or do the fragmentation yourself. Some WebSocket libraries implement fragmentation and some do not. Some libraries do fragmentation but leave you to reassemble the message yourself.

3.9.1.2 Limits to WebSocket Message Sizes

WebSocket libraries set their own limits for message size and some can be relatively small (e.g., 128 kb). ETP protocols and messages have been designed to work with these limits, for example, ETP requests and responses can be composed of multiple related messages, to keep individual message sizes small.

Additionally, ETP provides a mechanism for endpoints to convey their WebSocket-related limits; these limits include these endpoint capabilities:

- MaxWebSocketFramePayloadSize, for detailed definition and required behavior, see Section◦**3.3.2.8**.
- MaxWebSocketMessagePayloadSize, for detailed definition and required behavior, see Section◦**3.3.2.8**.

Server endpoint limits can be discovered via pre-session discovery (see Section **4.3.1**). A server may be informed of the client limits via query parameters when establishing a WebSocket connection (see Section **4.3**).The limits established for the WebSocket connection are also exchanged when establishing the ETP session in Core (Protocol 0) (see Chapter **5**). For general information about ETP-defined capabilities and how they are used, see Section **3.3**.

3.9.2 ETP Uses Asynchronous Exchange of Messages

All ETP communication is carried out through the asynchronous exchange of messages. This approach is distinct from the request/response pattern normally associated with HTTP, and the “RPC style” associated with many SOAP implementations (including all WITSML versions before v2.0). Of course, many use cases still require a request on the part of one endpoint and expect a response of some sort from the other.

The ETP design includes specific response messages to support specific uses cases. Additionally, for most ETP messages (except as noted in this specification), an endpoint can optionally request an Acknowledge message by setting a flag in the message header (for more information on how to request an Acknowledge message, see Section **3.7.2.2**).

NOTE: Implementers should always consider message sending and receipt to be happening asynchronously. **RECOMMENDATION:** Handle processing using state machines that model the various timings of message exchange that could occur.

3.9.2.1 How ETP Ensures Messages are Correctly Correlated

The WebSocket protocol guarantees the delivery of messages in the same order that they were dispatched. To ensure messages are correlated correctly, ETP uses several mechanisms, which include:

1. All messages within a session MUST be numbered (*messageId* in the ***MessageHeader*** Avro record). Message numbers MUST be integers, MUST be unique for each endpoint within an ETP session, and MUST be increasing. For all message numbering requirements, see Section **3.5.4**.
2. A *correlationId* is included in each message header, which designates relationships between messages (for example, the response that correlates to a request or the multiple "part" messages that comprise a multipart request message).
    a. *CorrelationId* is not required in all cases; the individual schemas and this specification indicate *correlationId* usage requirements. For more information, see Section **3.5.4**.
3. In some cases, mechanisms other than *correlationId* are used to correlate messages and are explained accordingly in this specification. These mechanisms include:
    a. UUID in the original message and later correlated messages (**EXAMPLES:** *requestUuid* and *blobId*).
    b. Map keys for messages that use maps.
4. Various ETP sub-protocols may impose specific ordering and numbering of certain messages, which is specified in the required behavior section of each protocol-specific chapter.

3.10 URI Query String Syntax with OData

For query functionality, ETP uses an OData-like syntax based on a subset of the Open Data Protocol (OData) query string syntax, specifically OData v4.0. OData is an OASIS standard (https://www.oasis-open.org/standards#odatav4.0).

OData was selected because it is a widely known (introduced in 2007) and maturing standard. Many client and server libraries are available for all major platforms. Its URL conventions also work well with ETP URIs.

OData resources:

- OASIS URL Conventions: http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part2-url-conventions.html
- OData.net libraries: http://www.odata.org/libraries/
- Linq to Querystring features: http://linqtoquerystring.net/features.html

For more information on query capabilities used in ETP, see Chapter **14** and the individual protocols that include query behavior (which include the word "query" in their name).

3.11 Tracking and Detecting Changes in an ETP Store

ETP provides several mechanisms for applications to track and detect changes to data in an ETP store. These mechanisms are described in detail in other parts of the specification, but this section summarizes them. **Appendix: Data Replication and Outage Recovery Workflows** provides an overview of how these features can be used to support eventual consistency between two data stores.

The main change detection features are:

- **Timestamps**: ETP requires stores to provide current timestamps and timestamps associated with data changes. Various messages allow customers to query for data that has changed since a given timestamp, which allows customers to discover relevant data changes they are interested in using timestamps from the store's clock. For more information, see Section **3.12.5**.
- **Store-Managed Fields**: ETP requires stores to track certain fields for every data object: *storeLastWrite*, *storeCreated*, and, for certain data objects, *activeStatus*. Stores must update these fields in response to data changes and ETP store operations. Customers can use these fields to query or filter for data objects of interest. For more information, see Section **3.12.4**.
- **Notifications**: ETP provides protocols that allow customers to subscribe to change notifications to data objects, channel data and growing data object parts. Once subscribed, customers automatically receive notifications of new, modified or deleted data. For more information about notifications, see Chapter **10**, **StoreNotification (Protocol 5)**, Chapter **12**, **GrowingObjectNotification (Protocol 7)**, and Chapter **19**, **ChannelSubscribe (Protocol 21)**.
- **Tombstones**: ETP requires stores to retain minimal information about deleted data objects after they have been deleted. Stores must track these so-called tombstones for a period of time specified by the ChangeRetentionPeriod endpoint capability. These tombstones allow customers to discover which data objects have been deleted recently. For more information about tombstones, see Section **8.2.1.2**.
- **Change Annotations**: ETP requires stores to track which data intervals have changed in channels and growing data objects. Stores must track this information for a period of time specified by the ChangeRetentionPeriod endpoint capability. Customers can request the change annotations for channels and growing data objects to determine if there are changes they may be interested in. For more information about change annotations, see Section **11.1.4**.

3.11.1 Benefits of Change Tracking and Detection Features

These features are designed to:

1. Minimize the effort a customer must put into getting new data,
2. Minimize the latency before a customer receives new data, and
3. Minimize the amount of data a customer needs to request if it is only interested in new data.

As such, stores are strongly encouraged to retain any change information (especially store-managed fields, tombstones, and change annotations) for as long as is feasible.

3.11.2 "Relaxed" Change Tracking and Detection Behavior for Some Stores

However, not all stores can track these changes accurately over a long period of time. **EXAMPLE:** Some stores are end-user applications without a persistent data store for ETP information, and other stores are implemented as an API over an existing, legacy data store. To support these types of stores, ETP allows some change detection behavior to be "relaxed"; **EXAMPLES:** Provided a store meets certain requirements when doing so, the store may retain changes for shorter periods and/or set change times on a best endeavors basis. The specific ways this "relaxed behavior" is allowed is documented in the relevant sections of the specifications.

3.11.3 Some Important Points About Change Detection

- Implementing "relaxed" change detection behavior WILL result in additional queries from customers, higher latency in getting new data to customers, and more data being requested from customers. Relaxed change detection should only be done where it is unavoidable.
- When implementing "relaxed" change detection behavior, stores MUST still correctly retain (subject to where appropriate to the ChangeRetentionPeriod capability) all relevant change information (store-managed fields, change annotations, and tombstones) as long as there is at least one session connected.
- If a store loses track of relevant change information when no sessions are connected (e.g., the store application is restarted and there is no persistent store for the information), the store MUST track the "start" timestamp from which it is able to provide relevant change information. This may be the timestamp when the application started. Stores MUST provide this timestamp in the ***RequestSession*** or ***OpenSession*** to allow customers to make informed decisions when connecting to the store. For more information, see Chapter **5**.

3.12 How to Handle Commonly Used Types of Data in ETP

Software applications for upstream oil and gas have common types of data that are used extensively, for example, variations of time (e.g., time stamps, elapsed time), units of measure (UOM) and ranges/intervals. This section explains how ETP handles these types of data.

3.12.1 Data Model as a Graph

Data objects in an ETP store are connected to each other through relationships forming a graph. A graph is a mathematical structure used to model pairwise relations between objects (https://en.m.wikipedia.org/wiki/Graph_theory). In this context, a graph is made up of *nodes* (which are also called *points* or *vertices*) and the *lines* (also called *links* or *edges*) that connect the nodes (**Figure 9**).

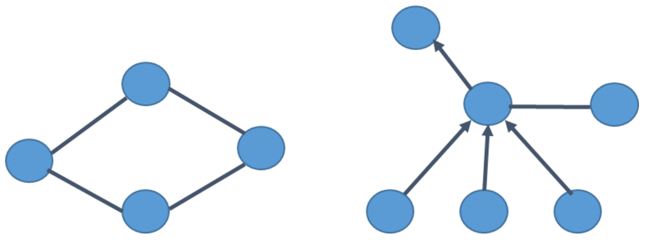


Figure 9: Examples of simple graphs: left image is an undirected graph and right image is a directed graph.

ETP has been designed to navigate Energistics data models as graphs where:

- Nodes represent data objects in a data model (WITSML, RESQML, PRODML or EML (i.e., Energistics *common*) (For the definition of data object, see Section **25.1**).
- Lines (directed links between nodes) represent relationships between those data objects. A data object can have multiple distinct references to other data objects (as specified in the various domain models).

For a complete explanation of graphs and how they work in ETP, see Section **8.1.1**.

3.12.1.1 Benefit of Graphs in ETP

Discovery (Protocol 3), StoreNotification (Protocol 5) and the query and other notification protocols have been designed to work with these graphs of data objects. These protocols use as key inputs graph concepts that allow a customer endpoint to precisely specify—in a single request—the desired subset of data objects from the graph for operations in an ETP session. This ability significantly reduces traffic on the wire. Conversely, if the graph concepts are not understood and related inputs not used properly, related operations will be highly inefficient.

RECOMMENDATION: Read Section **8.1.1** and make sure you understand the related inputs as specified in the respective messages where they are used.

3.12.2 Encoding Rules for ETP

Observe these rules:

1. Strings in Avro must be UTF-8 encoded.
2. Any XML, JSON or other text-based content included with an ETP message must be UTF-8 encoded.
3. For UUIDs:
    a. For use in ETP messages—with the exception of string representation of data objects that may be conveyed with a message (see next bullet)—ETP uses the ***Uuid*** datatype (Section **23.6**) to send UUIDs. The ***Uuid*** data type is encoded as an array of 16 bytes in big-endian format. **EXAMPLE:** The UUID "00112233-4455-6677-8899-aabbccddeeff" is encoded as the byte array [0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff] in the ***Uuid*** data type.
    b. For use in Energistics domain standards, for string representation of a data object, a UUID MUST be serialized using Microsoft Registry Format; that is, with dashes inside the UUID and without curly braces.

3.12.3 Serialization of URIs

As mentioned in Section **3.7.4**, URIs are an important and frequently used identification mechanism in ETP. Of particular note for the serialization of URIs in Avro/ETP:

1. Sending endpoints MUST URL encode (i.e., percent encode, as per section 2 of RFC 3986) any components that contain reserved characters for the URI scheme.
2. Receiving endpoints MUST decode all incoming URIs.

3.12.4 "Store-Managed" Fields

ETP has several data fields that are referred to as "store managed", which means only a store can update these fields. Operations performed by the customer role can require that the store update these fields. The behaviors for when the store must update these fields is specified in the relevant protocol-specific chapters. **EXAMPLES:**

- *storeCreated* and *storeLastWrite* (For more information, see Section **3.12.5.2.**)
- *activeStatus* (For more information, see Section **3.3.2.1**.)

Sometimes these fields are elements on data objects such as a WITSML 2.0 Channel's *GrowingStatus* element, and sometimes these are fields on ETP records such as *storeLastWrite* on a ***Resource*** record. In some cases, the field on ETP records maps to elements on data objects, and the ETP store manages both, such as *activeStatus* on ***Resource*** and *GrowingStatus* on a WITSML 2.0 Channel.

3.12.5 Time

Time and timestamps are important component of data acquisition and transfer related to oil and gas operations and in ETP.

3.12.5.1 Time Data Types

Time/date is often used as both an index and a filter. This table lists and explain various types of time and how they are handled in ETP.

Type of Time	How Handled in ETP	Examples in ETP
Timestamp	Must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	Most times in ETP use a timestamp with this format.

Elapsed Time	MUST be the number of microseconds from 0 and serialized as an Avro long.	ChannelIndexKind in ChannelMetadata

3.12.5.2 *New in ETP v1.2: storeCreated and storeLastWrite*

ETP workflows to "catch up" or "recover" after unplanned disconnects and for eventual consistency between stores are based on times designated by the endpoint with the store role in the relevant protocols. For more information, see Section **3.11**, **Tracking and Detecting Changes in an ETP Store**, and **Appendix: Data Replication and Outage Recovery Workflows**.

To implement these workflows, ETP defines 2 timestamp fields—*storeCreated* and *storeLastWrite*—which both are on the ETP ***Resource***, ***Dataspace***, and ***DataArrayMetadata*** records only (NOT on the data object, dataspace or data array in an underlying store). By carrying this data on these records only (not the data object, dataspace or data array), it separates transport properties from data object properties.

Field Name	Definition
storeCreated	The time that the data object, dataspace or data array (that the Resource, Dataspace, or DataArrayMetadata represents) was created in the store, which IS NOT the same as the *creation* field in the ***Citation*** in Energistics *common*. Its main purpose is for use in workflows for eventual consistency between 2 stores. Specifically, this field helps with an important edge case: on reconnect, an endpoint can more easily determine if while disconnected a data object, dataspace or data array was modified OR deleted and recreated. (Each of these scenarios would require different actions.) It is a timestamp as defined in Section **3.12.5.1 above**.
storeLastWrite	The last time the data object, dataspace or data array was written in a particular store, which IS NOT the same as the *lastChanged* element on a data object's ***Citation*** element. This *storeLastWrite* field may be the last time the data object, dataspace or data array was saved to a database or the last time a file was written (depending on the store). • For ANY CHANGES to a data object or its data (E.g., parts of a growing data object or channel data in a channel data object) a store MUST update *storeLastWrite*. – Put operations using Store (Protocol 4). – Part additions or updates using GrowingObject (Protocol 5) – Channel data updates using ChannelStreaming (Protocol 1) or ChannelDataLoad (Protocol 22) – Updates to values in a data object's associate data array(s) using DataArray (Protocol 9) • It is a timestamp as defined in Section **3.12.5.1 above**.

Important usage points about these timestamp fields:

1. **Times used should be from the "store" role.** In many operations in ETP, the customer role sends request messages, which contain timestamps (e.g., as a filter, "send me resources for objects that have changed since time *X*"). To ensure no data is "missed" (e.g., when reconnecting after a disconnect, because customer and store clocks may be different), a time in a customer request SHOULD be based on a time it received from the store (e.g., in a resource or metadata record or a notification message). This approach of using the source time eliminates the need to synchronize clocks between endpoints.
    a. In ETP, the customer is sometimes "pulling" data from a store and may send requests to the store to see what created or modified data needs to be pulled. In other situations, the customer is "pushing" data to a store and may send requests to the store to see what data is missing or outdated that still needs to be pushed. In both situations, the timestamps in the request messages should be based on timestamps previously received from the store.
2. **To determine if the sender's clock has changed:** Though ETP has no need to synchronize clocks,

an endpoint's clock time can change. When this happens, it may disrupt ongoing data transfer operations. ETP does not provide explicit features to recover from this scenario, but it is possible in some cases to detect when this has happened by using the ***Ping*** and ***Pong*** messages. ***Ping*** and ***Pong*** are defined in Core (Protocol 0) and can be used at any time during an ETP session. For more information, see **26 Appendix: Data Replication and Outage Recovery Workflows**.

3. **On reconnect, request changes for a time a bit earlier than the latest known timestamp**. Because ETP is asynchronous and multiple messages can be sent in response to an action or operation, there is no guarantee to timing on when you will receive a particular response. **RECOMMENDATION:** On reconnecting after a session was disconnected, a customer should use the store's value for ChangePropagationPeriod endpoint capability (which is in seconds) to request changes that many seconds before the last store change time the customer knew about before it was disconnected (e.g., if a store's value for ChangePropagation Period is 300 seconds, the customer should request changes 300 seconds before the last store change time the customer knew about before it was disconnected).
   EXAMPLE: If a container data object is deleted, and it requires pruning of orphan data objects, that operation might trigger multiple messages with the same change timestamp, which may not all be sent at exactly the same time. If the session disconnects in the middle of receiving these messages (e.g., the receiving endpoint sees only the first 2 of 3 generated messages) won't be aware of all of the changes that happened at that timestamp. By adding requesting changes a bit earlier than the latest timestamp it has, an endpoint can account for this possibility and ensure greater probability that it doesn't "miss" any data.

4. Not every store will be able to accurately track creation and modification time over a long period of time. **The minimum requirements to enable eventual consistency workflows are that:**

   a. When a data object, dataspace or data array is created in a store, the store MUST set both *storeCreated* and *storeLastWrite* (for creation) to the same timestamp, which is equal to or more recent than the actual time the data object, dataspace or data array was created or modified. This requirement applies even when a data object, dataspace or data array was created or modified by something other than an ETP store operation.

   b. When a data object, dataspace or data array is created or modified in a store, the store MUST set *storeLastWrite* equal to or more recent than the actual time the data object, dataspace or data array was created or modified. This requirement applies even when a data object, dataspace or data array was created or modified by something other than an ETP store operation.

   c. When the creation or modification happens **through ETP store operation**, the store MUST set the timestamps equal to the actual creation or modification time as part of the store operation.

   d. *storeCreated* MUST always be equal to or more recent than the actual time the data object, dataspace or data array was created.

   e. *storeLastWrite* MUST always be equal to or more recent than the actual time the data object, dataspace or data array was modified.

   f. *storeLastWrite* MUST always be equal to or more recent than *storeCreated* for any given data object, dataspace or data array.

      With these rules, stores MAY use a more recent time for *storeLastWrite* and *storeCreated* if necessary under certain circumstances. **EXAMPLE:** If a store application is restarted and it loses track of previously known *storeCreated* and *storeLastWrite* timestamps, it may choose to initialize all *storeCreated* and *storeLastWrite* timestamps to the time at which the store application started.

      However, for optimal support of eventual consistency workflows, both *storeCreated* and *storeLastWrite* SHOULD always be equal to the actual creation or modification time. Choosing a different time may lead customers to request more data that would otherwise not be necessary.

      IMPORTANT: The store MUST send appropriate ***ObjectChanged*** notifications in response to ANY change to *storeLastWrite* and *storeCreated*, including those changes that are not in response to an ETP store operation.

3.12.6 Units of Measure (UOM)

Accurate use, exchange, and conversion of units of measure (UOM) in upstream oil and gas software are crucial. EXAMPLE: Errors in units of measure can cause serious problems for the accuracy and integrity of earth and reservoir models and the decisions that are based on those models. This table lists how units of measure are handled in ETP.

Object Type	Where to get the UOMs	Usage in ETP
Channel	Units of measure for channel data and channel indexes are sent in the channel protocols using the ***ChannelMetadataRecord*** (see Section **23.33.7**) and ***IndexMetadataRecord*** (see Section **23.33.6**) records within ***ChannelMetadata*** and ***OpenChannelResponse*** messages.	ChannelStreaming (Protocol 1) ChannelSubscribe (Protocol 21) ChannelDataLoad (Protocol 22)
Data Object	Energistics has a Unit of Measure (UOM) standard, which is used by all Energistics domain standards. When data objects are included in an ETP message, applicable units of measure must be sent with that data in accordance with the ML that defines the data object and the UOM standard.	Any protocol in ETP where data objects may be sent or received, which include: Store (Protocol 4) StoreNotification (Protocol 5) GrowingObject (Protocol 6) DataArray (Protocol 9) StoreQuery (Protocol 13)'
Part or Range	Growing data objects contain parts, which have indexes and other content. Units for part indexes are sent using the ***PartsMetadataInfo*** record (see Section **23.34.17**). Units for growing object part content and indexes are also included directly in the part data, which is handled in the same way as data object content described above. When working with growing object parts and channels, it is common to be interested in a range of it (instead of individual parts or data points). When defining a range, the units of measure MUST also be specified in the ***IndexInterval*** record (see Section **23.34.8**).	Any protocol in ETP where parts of growing data objects may be sent or received, which include: GrowingObject (Protocol 6) GrowingObjectNotification (Protocol 7) GrowingObjectQuery (Protocol 16)

3.12.7 Use of PWLS

The Practical Well Log Standard (PWLS), which is stewarded for the industry by Energistics, categorizes the marketing names for logging tools and the obscure mnemonics used for the measurements, using plain English. PWLS provides an industry-agreed list of logging tool classes and a hierarchy of measurement properties and applies all known mnemonics to them.

In general, the main goal of PWLS is to support the classification of well log property measurement data—usually referred to as *curves* or *channels*—that are commonly used by oil and gas companies, and to use this classification as a tool to support general queries over large populations of channels.

With this type of classification, PWLS makes it possible to do queries such as "give me all the gamma ray logs" and have a store return all gamma ray logs (channels), from all vendors, regardless of the variety of vendor mnemonics used to identify gamma ray data.

For more information about PWLS and to download a copy of it, go to the Energistics website: https://www.energistics.org/download-standards/

PWLS is implemented in the Energistics domain data models (version 2.0+)—WITSML, RESQML and PRODML—through the Property Kind Dictionary (PropertyKindDictionary data object), which contains all known property kinds (element name = *PropertyKind*). The PropertyKindDictionary is published as part of Energistics *common* (namespace = EML), which itself is versioned and published with each of the domain standards.

EXAMPLE: WITSML v2.0 requires that if a WITSML implementation supports the Channel data object, that WITSML implementation MUST support the PropertyKind data object (i.e. the contents of the PropertyKindDictionary). Messages in ETP that handle the Channel data object have a field that MUST reference a PropertyKind data object, by specifying a URI to the specific property. For WITSML v2.0, the PropertyKindDictionary is in the Energistics common v2.1 ancillary folder.

For ETP, the Properties in the Property Kind Dictionary are used as follows:

1. Discovery (Protocol 3): Because relevant PropertyKind data objects must be available for an ETP store that supports Channel objects, endpoints can discover the available PropertyKind data objects, and then use the discovered/desired values to do discovery or query operations on the ETP store, **EXAMPLE:** Give me all the channels with property equal to property kind "gamma ray".
2. ChannelSubscribe (Protocol 21): When getting metadata about a channel from a store, the store MUST populate the *channelClassUri* field (in the ***ChannelMetadataRecord***; see Section **23.33.7**) with the URI for the appropriate PropertyKind data object.
   a. **NOTE:** In ETP v1.2, the ***IndexMetadataRecord*** and ***AttributeMetadataRecord*** also provide fields where the URI of a property kind MAY be entered. The field is optional in this version of ETP because the current published domain model (WITSML v2.0) does not have this field. In future versions of ETP (and WITSML) this field may be required.
3. DiscoveryQuery (Protocol 13) and StoreQuery (Protocol 14), see item **1** in this list.

NOTE: Version PWLS v3.0 was published in March 2021. At the time ETP v1.2 was published, the current published version of the PropertyKindDictionary published in Energistics *common* was based on earlier drafts of PWLS v3.0. The PropertyKindDictionary based on PWLS v3.0 will be updated and published in the next version of Energistics *common.*

3.12.8 Value and Range Endpoint Comparisons in Requests

Value comparisons are used in ETP in these main contexts:

- Range requests, which are request messages that operate on an index range (typically time or depth) of channel data or growing object parts.
- Requests that filter results based on *storeLastWrite* or *storeCreated* values.
- OData queries against values in data objects.

The value comparisons used in these contexts are sometimes inclusive (e.g., greater than or equal) and sometimes exclusive (e.g., strictly greater than). The specific comparison used depends on the context and the specific ETP request message.

The following table explains the general rules for how value comparisons are handled. The chapters for the relevant individual protocols also specify this information.

ETP Protocol or Use Case	Inclusive or Exclusive Comparison
The *storeLastWriteFilter*, which is available in Discovery (Protocol 3), Dataspaces (Protocol 24), and relevant query protocols	EXCLUSIVE (this is a "greater than" (GT) operation)
Range Request: Growing data object (Protocol 6)	Specific overlap/inclusive behavior is defined in the protocol (see Chapter 11)
Range Requests: ChannelSubscribe (Protocol 21) and ChannelDataLoad (Protocol 22), where the ranges are specified in the	INCLUSIVE (this is a "greater than or equal" (GTE) operation).

ETP Protocol or Use Case	Inclusive or Exclusive Comparison
Query Protocols	As specified by the subset of OData operators that ETP uses.

3.12.9 Nullable Values

In the ETP Avro schemas, nullable types are specified as an Avro union with "null" and the type, e.g., a nullable long is defined like this (schema truncated for clarity):

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
    "name": "ChannelMetadataRecord",
    "fields":
    [

        { "name": "startIndex", "type": ["null", "long"] },
        { "name": "endIndex", "type": ["null", "long"] },

```

NOTE: Arrays, maps, strings and the bytes data type are NOT nullable unless they are in an Avro union that includes “null”. When these data types are not in such a union, to send a “null” value for these data types, you MUST send:

- for arrays, a zero length array
- for maps, an empty map
- for strings, an empty string
- for bytes, a zero length array

Binary encoding of nullable types is specified in the Avro Specification chapter “Data Serialization”, sub-heading “Unions”. For example, the “startIndex” in the example above with the value “16” must be encoded like: 02, 20, (hex). The value 0x02 is the zig-zag encoded value 1 representing the “long” type in the union, followed by the zig-zag encoded value.

3.13 Troubleshooting

Data acquisition and transfer related to oil and gas operations presents many, frequently occurring challenges. ETP has been designed to help address many of these challenges. This section highlights commonly occurring problems and ways to deal with them. In some cases, the method to address an issue is quite specific (**EXAMPLE:** Section **3.13.4**) in other cases more of a heuristic (**EXAMPLE:** Section **3.13.3**).

3.13.1 ETP-defined Capabilities

One of the ways that ETP helps to address challenges associated with data transfer in oil and gas operations is by setting limits with capabilities—endpoint, data object, and protocol capabilities—which are explained in Section **3.3**.

How these capabilities are used and defined behavior when the limits they specify are violated are explained in the individual protocol chapters where they are used. This is the general approach in this document for including behavior related to capabilities:

- Endpoint and data object capabilities that have global or widespread usage are documented Section **3.3** with some references in protocol-specific chapters. **RECOMMENDATION:**◦Implementers should learn these high-level capabilities and all cases where they may apply.
  - Each protocol-specific chapter has a Section *x*.2.3, which lists the capabilities most relevant for that chapter and links to other information.
- Protocol capabilities are explained in their protocol-specific chapters. Generally, the capabilities are explained in Section x.2.1 (message sequence for main tasks, so in the context of where they are used) and/or Section x.2.2 (general requirements for more general behavior within a protocol).

- The complete list and definitions of endpoint, data object and protocol capabilities are included in Chapter **23**.

3.13.2 Trying to Do Too Many Operations at the Same Time

Multipart requests, responses and notifications require dedicated pools of resources to handle correctly. To limit the number of these messages that can be done in parallel to keep the resources necessary to handle them bounded, ETP allows endpoints to use endpoint capabilities notifications as described in Section **3.7.3.1.1.1**.

WARNING: Use of multipart requests, responses, and notifications in data-movement protocols may create mutating or racing conditions. Currently, ETP does not attempt to handle these conditions. Identifying and addressing these conditions is up to the developer/implementer. The safest thing for client applications to do now is to ensure they do not issue concurrent, competing requests to a store.

"Stop Operation" Messages. For potentially large, long-running, or complex operations (e.g., streaming channel data), ETP sub-protocols define "stop", "cancel" or "unsubscribe" messages. These messages typically allow either endpoint to stop an operation for any reason. To terminate long-running operations that exceed limits defined in endpoint or protocol capabilities, endpoints may also send appropriate errors as defined elsewhere in this specification.

3.13.3 Always an Option: Drop the Connection

While more "elegant' options have been designed in ETP, if operations aren't proceeding as expected, an ETP endpoint always has the option to simply drop the connection.

If this happens, the other endpoint SHOULD do its best to treat the dropped connection as a shutdown of the ETP session (as described in Section **5.2.1.2**). If needed, endpoints should then use processes describe elsewhere in this document, to reconnect and catch up on data that may have been missed while disconnected. For more information, see **Appendix: Data Replication and Outage Recovery Workflows**.

3.13.4 Receiver not Receiving Messages Fast Enough

ISSUE: The receiver is not reading data fast enough from the session, so the sender is not able to send data at the expected frequency. The sender SHOULD do the following:

1. If the sender starts to detect sending backpressure (e.g., its queues of outgoing messages are starting to fill up), the sender MAY send error EBACKRESSURE_WARNING (25).
   a. This error code is one of the few cases when sending a ***ProtocolException*** message, where the correlationId in the MessageHeader MUST be set to 0. For more information, see Section **3.7.2.1**.
2. If the sender's sending queue capacity is exhausted and it is imminently unable to send a message to the receiver, the sender MUST do the following:
   a. Attempt to send error EBACKPRESSURE_LIMIT_EXCEEDED (24).
   b. Attempt to send the ***CloseSession*** message (from Core (Protocol 0).
   c. Close the connection, regardless of whether the ***ProtocolException*** or the ***CloseSession*** messages could actually be sent.
3. To resume data transmission, the connection and ETP session must be re-established and any ETP sub-protocol-specific outage recovery processes followed (to catch up on missed data). (All of these processes are documented in other parts of this specification.)
   a. Before reconnecting, any receiver issues that caused the previous congestion SHOULD be addressed.

3.13.5 Authorization Expiring

ETP specifies an authorization scheme based on adaptation of existing security IT standards. (For information, see Chapter **4 Securing an ETP Session and Establishing a WebSocket Connection**.) ETP specifies the following:

1. An endpoint SHOULD remain authorized with the other endpoint (as required by the respective endpoints when the ETP session was established) for the duration of the ETP session.
   a. An endpoint MUST re-authorize with the other endpoint BEFORE the current authorization expires.
      i. For the high-level workflow on how an endpoint gets a bearer token, see Section **4.1.2**.
   b. As needed, either endpoint CAN send the ***Authorize*** message (as described in Section **5.2.1.1)** at any time, to remain authorized for the duration of the session.
   c. After the initial authorization, the authorization method and security principal MUST not change and the scope MUST not be reduced.
   d. The authorization for each endpoint may have very different expirations, so each endpoint may re-authorize to the other at different times.
2. If an endpoint's authorization will expire "soon", the other endpoint MAY send error EAUTHORIZATION_EXPIRING (28).
   a. For more information, see the detailed text on the error code in Section **24.3**.
3. During an ETP session, if an endpoint's authorization expires, the other endpoint MUST:
   a. Send error EAUTHORIZATION_EXPIRED (10).
   b. Send the ***CloseSession*** message.

4 Securing an ETP Session and Establishing a WebSocket Connection

This chapter explains:

- ETP security (authorization) requirements, schemes, and workflows—which have changed significantly in ETP v1.2 (Section 4.1).
- Pre-requisites for establishing a WebSocket connection (Section 4.2).
- How a client does the following (Section 4.3):
  - Establishes a WebSocket connection.
  - "Pre-discovers" information about the server (e.g., version(s) of ETP it supports, how and where to get authorization information, functionality within ETP, and payload limits for WebSocket frames and messages), information that is helpful to the client in creating the WebSocket connection and ETP session.

After establishing a WebSocket connection, a client can establish an ETP session, which is explained in Chapter **5 Core (Protocol 0): Establishing and Authorizing an ETP Session**.

4.1 ETP Security

In any communication protocol—especially one carrying sensitive or proprietary data—security is a major concern. Implementers are encourage to follow best practices appropriate for oil and gas operations, the industry, and specific implementations. Basic Authentication is NOT recommended for production implementations.

The previous version of ETP supported 2 security schemes: Basic Authentication and basic support for JSON Web Tokens (JWT). The Energistics community has since determined that these schemes were not sufficient.

As in previous version of ETP, the focus for "security" is on authorization of connections. ETP also continues to leverage and adapt existing security standards and best practices (not invent its own).

Beginning with ETP v1.2, ETP has new security schemes. This section explains the new security approach and how it works. (**NOTE:** For background information including the list of requirements and how and why this approach was selected, see **Appendix: Security Requirements and Rationale for the Current Approach**.)

Based on requirements, security for ETP v1.2 had these design goals:

- Use common mechanisms for HTTP resources and ETP resources.
  - Some HTTP resources may be protected, for example, the well-known endpoint for pre-session discovery. We did not want to use 2 different mechanisms.
  - Adopt and re-use HTTP Auth schemes.
- Allow for flexible extensibility as security protocols evolve.
- Allow any type of bearer token to be used (NOT just JWT)
- To better allow for end-to-end scenarios, allow authorization to happen at the ETP application layer, not just the HTTP transport layer.
- As in WITSML, ETP v1.2 specifies auth schemes that MUST be implemented by all servers for interoperability, but allows for additional schemes (client certificates, for example) to be used by agreement between specific parties.

4.1.1 Overview of the Approach

Security for ETP leverages relevant parts of existing security standards to specify an approach that is based on existing best practices and can provide minimum standard requirements. The approach also allows for implementers to develop custom enhanced functionality now, and for ETP to expand its functionality and requirements in the future.

Some basic concepts used in this security scheme are introduced here and explained further below in this section:

- A "bearer token" is a general term for a mechanism that if the "bearer" presents the token, it can gain access to a resource. In the context of IT security there are many types of bearer tokens. The term "access token" refers specifically to a bearer token issued by an authorization service that authorizes access to a specific resource. The bearer does not know the contents of the access token, nor does it need to; the bearer only needs to know how to get the access token and present it to the resource it needs access to. The required ETP workflow uses access tokens.
- This security scheme requires an authorization server, which issues access tokens for an ETP endpoint. An authorization server may be separate from the ETP endpoint and may have a different authority, or it may be a relative URI to the ETP endpoint.

Security in ETP v1.2 has these requirements and characteristics:

These items are numbered for easy reference.

1. Leverages existing security standards, which includes those listed below. **NOTE:** Implementers are responsible for reading and understanding these existing standards.
    a. If the ETP server is supporting TLS, it MUST support v1.2 or greater (https://www.rfc-editor.org/rfc/rfc8996) for authentication, confidentiality, and integrity.
        i. Implementers SHOULD use TLS to secure as much of the network traffic as possible. It is highly recommended that all production ETP servers use the secure WebSocket protocol (i.e., wss).
    b. ETP servers MUST support the usage of OAuth 2.0 bearer tokens (https://rfc-editor.org/rfc/rfc6750).
    c. Authorization servers MUST support OpenID Connect Discovery v1.0 (https://openid.net/connect/).
    d. Authorization servers MUST support the client_secret_basic token_endpoint_auth_method as required by https://www.rfc-editor.org/rfc/rfc6749#section-2.3.1.
    e. Authorization servers MUST support the OAuth 2.0 client credentials grant type (https://www.rfc-editor.org/rfc/rfc6749.html#section-4.4).
    f. Authorization servers SHOULD provide the expires_in property with successful token responses (https://www.rfc-editor.org/rfc/rfc6749#section-5.1).
    g. Also, some aspects of ETP v1.2 security are modeled on those present in Third-Party Token-Based Authentication and Authorization for Session Initiation Protocol (SIP) (https://www.rfc-editor.org/rfc/rfc8898) and the generic HTTP Authentication framework it is based on (https://www.rfc-editor.org/rfc/rfc7235).
2. Focus is only on authorizing the connections between ETP applications (not necessarily a device), and NOT the domain objects within a store.
   GOAL: A client endpoint can connect to and establish a session with an ETP server endpoint.
    a. Authorization of fine-grained resources (e.g., a well, project, etc.) is NOT in scope for this specification.
    b. Because ETP supports the use case for an ETP store to connect as a client with the customer role to another ETP store, ETP security DOES allow for a both a client to authorize to a server AND for the server to authorize to the client (which is explained in relevant sections of this specification).
3. Uses OAuth 2.0 and OpenID Connect 1.0 to define behaviors for non-interactive acquisition and

usage of bearer tokens for application authorization.

   a. Focus on non-interactive authorization ONLY. (**NOTE:** Some security schemes also allow for interactive authentication and authorization; however, interactive workflows are NOT in scope for ETP v1.2).

4. Now allows any type of bearer token (not just JWT as in ETP v1.1).
   a. Tokens are typically opaque to clients. They are issued by authorization servers and presented to endpoints.
   b. A server MUST support the workflow to get a bearer token as specified in Section **4.1.2**.
   c. **NOTE:** Some usage profiles of bearer tokens (e.g., JWT) can self-generate tokens without an authorization server (which is the reason they were used for ETP v1.1). However, JWTs were determined to be too restrictive and not sufficiently extensible, so now ETP v1.2 can be used with any type of bearer token.
5. Allows implementers to add custom behavior now and future extensibility by using constructs similar to the HTTP generic challenge and response authentication framework from RFC 7235, which include these:
   a. The ***AuthorizeResponse*** message in Core (Protocol 0) acts as an extensible equivalent to the HTTP WWW-Authenticate challenge.
   b. The ***Authorize*** message in Core (Protocol 0) acts as an extensible equivalent to the HTTP Authorization response.
   c. ETP adopts authz_server and scope params specified in RFC 8898 to convey bearer token issuer discovery of an ETP application's bearer token requirements and configured authorization server(s).
      i. However, discovery of an authorization server's configuration uses OpenID Connect Discovery 1.0.
      ii. ETP defines the following methods for how an endpoint MAY specify its authz_server and scope params:
         1. In the AuthorizationDetails endpoint capability (see Section **4.1.3**), which an endpoint MAY discover as part of the establishing the WebSocket connection (see Sections **4.3** and **4.3.1**) or MAY be exchanged in the *endpointCapabilities* field (e.g., in the ***RequestSession*** message).
         2. In the ***AuthorizeResponse*** message, as part of establishing the ETP session (see Section **5.2.1.1**).

4.1.1.1 Authorization Options: Transport Layer or Application Layer

ETP allows and provides the functionality to authorize as follows:

- At the HTTP transport layer, as part of creating the WebSocket connection (see Section **4.3**).
- At the ETP application layer, as part of establishing the ETP session (see Section **5.2.1.1**).

IMPORTANT! Individual ETP implementations MAY choose to authorize at one or both layers.

For example, the application layer may choose to accept the transport layer authorization and not require additional authorization at the application layer, or it may choose to still require application layer authorization in addition to the transport level authorization.

However, if using transport layer authorization exclusively, there is no mechanism to update the authorization to extend the lifetime of the connection. The ETP design does not prevent an implementation from accepting transport layer authorization on initial connection and application layer authorization at later points, but when doing so care must be taken to maintain equivalent levels of security.

Implementations should be guided in the approach they take by any requirements or limitations of any middleware they want to participate in handling incoming connections.

4.1.1.2 Changes to ETP to Implement this New Approach

The following changes have been made to ETP:

- Added new endpoint capability (AuthorizationDetails) that allows a server to identify its authorization server(s) and required scope parameters. The client uses this information to get a bearer token as described in Section **4.1.2**. For more information on the AuthorizationDetails capability, see Section **4.1.3**.
- Revised and added messages in Core (Protocol 0) to allow endpoints to exchange authorization information as part of ETP session negotiation workflow. The ***RenewSecurityToken*** was renamed to ***Authorize*** and the ***AuthorizeResponse*** message was added. See Chapter **5**.
- In ETP v1.2, Basic Authentication is no longer recommended for use. Basic Authentication is as described in RFC-7617 (https://www.rfc-editor.org/rfc/rfc7617) for HTTP connections. As in HTTP, basic authentication has many security issues, especially when used on an insecure connection (i.e., not TLS).

4.1.2 High-Level Workflow for Getting a Bearer Token

This section explains the high-level workflow for getting a bearer token. ETP servers MUST support authorization with a bearer token using this workflow (i.e., client connecting to a server). If a client requires authorization, it MUST support this workflow.

NOTES:

- The provisioning of endpoints with an authorization server is outside the scope of this specification, but it is a prerequisite that both a resource server and confidential client must be provisioned beforehand.
- The steps below are written so they can be used by either the client or server. The "endpoint" is working to be authorized to the "peer endpoint".

Follow these steps to get a bearer token:

1. An endpoint gets a peer endpoint's capabilities as described in Section **4.3.1**.
2. The *endpointCapabilities* field MUST contain an endpoint capability named AuthorizationDetails, which contains this information:
    a. (REQUIRED) The URI of the authorization server (authz_server) that the endpoint uses to construct a well-known OpenID Connect Discovery URI.
        i. An authorization server may be separate from the peer's ETP endpoint and may have a different authority, or it may be a relative URI to the peer's ETP endpoint.
    b. (OPTIONAL, as required by the authz_server) Any additional scope or other parameters required for the endpoint to get what it needs to make a token request from the peer endpoint's authorization server.
        i. An endpoint SHOULD include the peer endpoint-provided parameters when making a token request to the corresponding authorization server. Some parameters may be dynamic or not cacheable.
    c. For details of the AuthorizationDetails endpoint capability, see Section **4.1.3**.
3. An endpoint performs OpenID Connection Discovery on the authorization server that it wishes to make a token request to.
4. An endpoint uses the token_endpoint property from the discovered OpenID Provider Configuration Document to perform a client credentials grant to acquire an access token.
    a. For details, see https://www.rfc-editor.org/rfc/rfc6749.html#section-4.4.

b. The access token is an opaque bearer token (i.e., the client does not know its content nor does it need to).

c. An endpoint MUST acquire the bearer token from one of the peer endpoint's authorization servers.

5. An endpoint then presents the bearer token to the peer ETP endpoint as described in Section **4.3**.

4.1.3 Contents of the AuthorizationDetails Capability and How it is Used

AuthorizationDetails is an endpoint capability that works like all endpoint capabilities in ETP (i.e., it may be discovered during pre-session discovery in the ***ServerCapabilities*** and is also provided in the *endpointCapabilities* field on the ***RequestSessio***n and ***OpenSession*** messages (for more information about endpoint capabilities, see Section **3.3**).

This section provides details of the AuthorizationDetails endpoint capability and its content.

1. The AuthorizationDetails endpoint capability contains an ArrayOfString with WWW-Authenticate style challenges. **NOTE:** The ***AuthorizeResponse*** message also contains a similar array of string with WWW-Authenticate style challenges.
2. To support the required authorization workflow (to enable an endpoint to acquire an access token with the necessary scope from the designated authorization server), the AuthorizationDetails endpoint capability MUST include at least one challenge with the Bearer scheme which must include the 'authz_server' and 'scope' parameters.

   a. The 'authz_server' parameter MUST be a URI for an authorization server to enable the endpoint to acquire any other needed metadata about the authorization server using OpenID Connect Discovery.

   b. Here are 2 example AuthorizationDetails endpoint capabilities:

   Example with 1 challenge: JSON style string escaping, they are array of strings so they are Avro strings

```
["Bearer authz_server=\"https://YourAuthServer/Path\"]
```

   Example with 3 challenges:

```
["Basic",

"Bearer authz_server=\"https://YourAuthServer/Path\" scope=\"openid\",

"Bearer authz_server=\"https://YourOtherAuthServer/SomePath\"
scope=\"yourRequiredScopes\""]
```

3. An ETP server MUST have the AuthorizationDetails endpoint capability, which must meet the requirements of Point 2 above.
4. If an ETP client does NOT need to authorize ETP servers, it MAY omit the AuthorizationDetails.

4.1.4 ETP Security Requirements for Establishing a WebSocket Connection

All servers and clients that implement ETP MUST observe these rules:

1. Specific vendors, service companies, and operators MAY also implement any other appropriate security mechanisms (such as SAML tokens), but they are not required by ETP and may lead to interoperability issues.
2. In all cases, the client MUST use the **authorization** request header defined by RFC 7235 (https://tools.ietf.org/html/rfc7235). Servers MUST support this method.

4.1.4.1 Authorization for Browser-Based Clients ONLY

Because the HTML5 WebSocket API definition does not allow access to the request headers, it is not possible for browser-based clients to add request headers when they make a WebSocket connection (as specified in Section **4.3**, Step **4**). Therefore, ETP defines these additional rules for authorization:

1. Browser-based clients MUST use ETP application layer authorization.

4.2 Prerequisites for Establishing a WebSocket Connection

A client application needs the following information for the ETP server it is trying to connect to—this information is typically sent out of band of this connection process, for example, as part of a contract or statement of work:

1. URL of the ETP endpoint.
2. Authentication details for an ETP endpoint's Authorization Server.
   a. The provisioning of endpoints with authorization servers is outside the scope of this specification, but it is a prerequisite that any confidential clients must be provisioned for the respective resource servers of the endpoints beforehand.
   EXAMPLE: An OAuth 2.0 client_id and client_secret for the ETP client to an Authorization Server of the ETP server and if required a similar set of credentials for the ETP server to an Authorization Server of the ETP client.
3. **OPTIONAL:** The version(s) of ETP that the server supports. ETP provides a way to query the endpoint for this information as part of the process of establishing the WebSocket connection; see Section **4.3**.
4. **OPTIONAL:** The payload size limitations of the WebSocket of ETP that the server supports. ETP provides a way to query the endpoint for this information as part of the process of establishing the WebSocket connection; see Section **4.3**.

4.3 How a Client Establishes a WebSocket Connection to an ETP Server

Before a client and server application can establish an ETP session, the client MUST create a WebSocket connection.

The basic process—including optional and required steps—is listed here:

1. **OPTIONAL:** Client gets the server's ServerCapabilities for the version of ETP that it wants to use. (For information on how to do this and available options, see Section **4.3.1**).
   a. The first time a client connects to a server it SHOULD get the server's ServerCapabilities, which provides information about the version(s) of ETP it supports as well as the specific protocols, object types, encoding and formats, etc. **IMPORTANT:** The ServerCapabilities also provides information about the WebSocket payload size limits—which cannot be changed once the WebSocket connection is established.
   b. Beginning with v1.2:
      i. ETP requires support of the ServerCapabilities; if a client requests the ServerCapabilities, then a server MUST provide it.
      ii. The ServerCapabilities contains the endpoint capability AuthorizationDetails. If a server requires authorization during this WebSocket connection process, the client MUST get the AuthorizationDetails capability because it contains the information and requirements the client needs to get an authorization token. For more information, see Sections **4.1.2** and **4.1.3**.
   c. Each supported version of ETP has its own requirements. For a version other than ETP v1.2, see the appropriate version of the specification.
2. If the client received a ***ServerCapabilities***, it MUST use the information received to inform the

WebSocket connection request and "prepare" the ETP session request. **EXAMPLES**:

a. The AuthorizationDetails endpoint capability has the information the client needs to find the authorization server and get a bearer token. For the high-level workflow for how to get the bearer token, see Section **4.1.2**.

b. **For** endpoint capabilities MaxWebSocketFramePayloadSize and MaxWebSocketMessagePayloadSize, the client can compare these values to its own max payload limits and then use them in the WebSocket connection request as described in step **4**.

3. Client connects to the ETP server using WebSocket.

   a. ETP servers MUST support HTTP/1.1 and RFC6455 (https://tools.ietf.org/html/rfc6455) and MAY support HTTP/2 and RFC8441 (https://tools.ietf.org/html/rfc8441).

   b. If the server requires transport layer authorization, it must use RFC6750.

4. **Establish the WebSocket connection.** To do this, the client begins with the standard WebSocket handshake, and specifies the necessary attributes listed in the table below. This list of attributes includes both standard and ETP-custom ones.

   a. Some of the attributes are REQUIRED (**RQD**) as indicated in the table.

   b. Some of the attributes MAY be specified as EITHER a header property (**HP**) or a query string (**QSP**) but SHOULD not be specified as both in a WebSocket request.

      i. If the same attribute is specified as both a header property and a query string parameter, the server MUST process ONLY the header property.

      ii. HTML5 Web browser clients cannot currently add custom headers to a WebSocket request, so they MUST include these options as query string parameters. (For more information, see Section **4.3.2**.)

   c. All protocol header names and values MUST be lower case.

Attributes Specified for use by ETP in WebSocket Handshake

Attribute and Description	RQD	HP or QSP	Values and Description
sec-websocket-protocol List the versions of ETP that the client wants to use, ordered by the client's use preference. This is a standard header property defined by the WebSocket specification. **NOTE:** Unlike custom headers, HTML5 browser-based clients CAN set this header property.	Yes	HP	**etp12.energistics.org**, **energistics-tp** Indicates that ETP v1.2 is the client's first choice, but it will accept ETP v1.1.
etp-encoding Specifies the Avro encoding style to be used for the life of the connection. This is an ETP custom header property. **NOTE:** If a browser-based client uses this option; it MUST be set it as a query string parameter; see Section **4.3.2**.	No	HP or QSP	**binary** or **json** The default is **binary**, which MUST be used for ETP production implementations. JSON is for internal testing and debugging only.

Attributes Specified for use by ETP in WebSocket Handshake			
Attribute and Description	**RQD**	**HP or QSP**	**Values and Description**
MaxWebSocketFramePayloadSize Maximum number of bytes allowed for a WebSocket frame payload (which is determined by the library you use to implement WebSocket). **NOTE:** To determine appropriate values for these payload query string parameters, a client should compare its own maximum values with those of the server (which the client gets in the ServerCapabilities) and use the smaller of the two values. **NOTE:** If a browser-based client uses this option; it MUST be set it as a query string parameter; see Section **4.3.2**.	No	HP or QSP	**Type:** integer **Default:** 4194304
MaxWebSocketMessagePayloadSize Maximum number of bytes allowed for a WebSocket message payload (which is composed of multiple WebSocket frames, which is determined by the library you use to implement WebSocket). **NOTE:** To determine appropriate values for these payload query string parameters, a client should compare its own maximum values with those of the server (which the client gets in the ***ServerCapabilities***) and use the smaller of the two values. **NOTE:** If a browser-based client uses this option; it MUST be set it as a query string parameter; see Section **4.3.2**.	No	HP or QSP	**Type:** integer **Default:** 16777216

5. The server establishes the WebSocket (ws/wss) connection by responding to the client's WebSocket handshake with the latest version of ETP that it supports (based on the clients preference list).
   a. If the *sec-websocket-protocol* header value is not present or does not match any version that the server supports, then the server must send error HTTP status code 400 (Bad Request).
   b. For the custom header of **etp-encoding**:
      i. If this header is not present, the encoding is assumed to be binary.
      ii. For HTML5 Web browser clients that send **etp-encoding** as a query string parameter, servers MUST accept and process this value.
   c. If the server does not support the requested **etp-encoding**, it MUST reject the connection request with HTTP status code 400 (Bad Request). The client can try again (if it wishes) with the

alternate **etp-encoding** value.

6. Possible errors:
   a. If a new connection may cause the server to exceed its value for MaxSessionGlobalCount endpoint capability (for definition, see Section **3.3.2.7**), then the Server MAY refuse any incoming connections.
      i. If a server chooses to reject an incoming connection because it would exceed this limit, it SHOULD reject the WebSocket request with HTTP 503: Service Unavailable.
   b. If a new connection may cause the server to exceed its value for MaxSessionClientCount endpoint capability (for definition, see Section **3.3.2.6**), then the Server MAY refuse any incoming connections.
      i. If a server chooses to reject an incoming connection because it would exceed this limit, it SHOULD reject the WebSocket request with HTTP 429: Too Many Requests.
7. For information on how to establish the ETP session, see Chapter **5**.

4.3.1 Requirements for Getting and Using an ETP ServerCapabilities

The ServerCapabilities is the ETP mechanism—which may be a binary or a JSON object—that clients use to “pre-discover” the capabilities of a server out-of-band of the WebSocket connection, using a simple HTTP GET of this object. The information received is used to inform the WebSocket upgrade request and "prepare" for the ETP session request.

For the definition of ServerCapabilities and related capabilities it may contain, see Section **3.3**.

Here are the requirements for getting and using the ServerCapabilities:

1. Beginning with v1.2, ETP requires support of the ServerCapabilities; if a client requests the ServerCapabilities record, then a server MUST provide it.
2. The URL for a server's capabilities is derived by appending the string ‘.well-known/etp-server-capabilities’ to the HTTP-equivalent URL of the ETP WebSocket endpoint.

   EXAMPLE: If you have an ETP server listening at wss://etp.sample.org:8080, then the server capabilities document is retrieved with an HTTP GET from https://etp.sample.org:8080/.well-known/etp-server-capabilities.

   a. The URL scheme of the ***ServerCapabilities*** MUST match the "TLS-ness" of the WebSocket connection. That is:
      i. If the WebSocket address is at ws://, then the document MUST be at http://.
      ii. If the WebSocket is at wss://, then the document MUST be at https://.
3. ETP provides the query parameters listed here, which may be appended to the .well-known/etp-server-capabilities endpoint. If more than one is used they must be separated by an ampersand.

   EXAMPLE: A client might use **GetVersions** to determine which versions of ETP a server supports, then use **GetVersion** to request the ServerCapabilities for the specific version of ETP that it wants to use.

   a. ***Query parameter for a list of supported ETP versions***:

```
paths:
/.well-known/etp-server-capabilities:
parameters:
in: query
name: GetVersions
type: boolean
required: false
default: false
```

```
description: If GetVersions=true, return a JSON array of strings
containing the supported ETP versions of this server. The possible
string values are the same as those that are used with the Sec-
WebSocket-Protocol header when performing the WebSocket handshake (see
Section 4.3, Step 4).
```

A server supporting ETP v1.1 and ETP v1.2 would return this response***:***

```
 [
  "energistics-tp",
  "etp12.energistics.org"
]
```

b. ***Query parameter for the specific capabilities of each version of ETP:***

```
in: query
name: GetVersion
type: string
required: false
default: energistics-tp
description: If GetVersions=false, return a JSON object encoded using
the requested ETP version's ServerCapabilities schema. The
supportedProtocols field must only contain entries for the requested ETP
version. The string parameter is one of the values that are used with
the Sec-WebSocket-Protocol header when performing the WebSocket
handshake (see Step 4 in Section 4.2).
get:
summary: Get the server capabilities of this ETP server.
```

c. ***Query parameter for the format of specific capabilities of each version of ETP:***

```
in: query
name: $format
type: string
required: false
default: binary
description: This controls the format used in the response when
GetVersions=false. The Avro binary encoding of the requested ETP
version's ServerCapabilities schema is used when $format=binary and a
Avro JSON encoding is used when $format=json. An ETP Server MUST support
the binary format and may support the JSON format. NOTE: The $format
query parameter does not apply when GetVersion=energistics-tp.
```

d. **Examples of Valid Query Parameters Appended to the ServerCapabilities Endpoint:**

Query parameter to determine which versions of ETP a server supports:

```
/.well-known/etp-server-capabilities?GetVersions=true
```

NOTE: If a GetVersion query parameter is included when GetVersions=true, it must be ignored.

Query parameter for an ETP v1.1 server capabilities:

```
/.well-known/etp-server-capabilities

/.well-known/etp-server-capabilities?GetVersions=false
```

```
/.well-known/etp-server-capabilities?GetVersion=energistics-tp

/.well-known/etp-server-capabilities?GetVersions=false&GetVersion=energistics-tp
```

NOTE: The above for ETP v1.1 are all technically equivalent due to the way the default query parameter values are defined to be backwards compatible with ETP v1.1. The version with query string omitted is what ETP v1.1 clients should issue.

Query parameter for an ETP v1.2 server capabilities:

```
/.well-known/etp-server-capabilities?GetVersion=etp12.energistics.org&$format=binary

/.well-known/etp-server-capabilities?GetVersions=false&GetVersion=etp12.energistics.org
```

NOTE: The above for ETP v1.2 are both technically equivalent due to the default for the GetVersions query parameter being GetVersions=false and the default for the $format parameter being $format=binary.

e. If these query string parameters are omitted:
   i. Then the .well-known endpoint MUST fallback to ETP 1.1 behavior (i.e., return the ServerCapabilities for ETP v1.1).
   ii. If the server supports ONLY ETP v1.2, then the server MUST throw error HTTP 400. **RECOMMENDATION:** Provide a brief, but appropriate explanation such as: *This server supports only ETP v1.2+*.

4. The returned ***ServerCapabilities***:
   a. MUST have one of these content types:
      i. **avro/binary** (This is the default and MUST be supported.)
      ii. **application/json** (i.e., must be a JSON document, if 'json' specified in $format query parameter; see Paragraph 3.c, **above**). The content MUST match an Avro JSON serialization of the Energistics.Etp.v12.Datatypes.ServerCapabilities structure defined in Section **23.19**.

   NOTE: Additional endpoint capabilities, protocol capabilities and data object capabilities MAY be present in the ServerCapabilities, but they are not part of the ETP specification. If an Energistics domain standard has particular requirements, they are documented in that domain standard's ETP implementation specification (which is a companion to this main ETP specification). A client MUST ignore any capabilities it does not understand.

   b. MAY be a protected resource, but it is not required to be so.
      i. If the ***ServerCapabilities*** is a protected resource the HTTP Auth MUST provide a WWW-Authenticate challenge in its 401 response, subject to the same requirements as Section **4.1.3**.
5. Resources are ETP endpoints and subject to normal HTTP behavior (for details, see the HTTP specification), for example:
   a. The server capabilities request or response MAY be compressed as per the HTTP standard for any HTTP resource.
   b. Redirects are allowed and MUST be observed.
6. Optionally, a server MAY provide an additional, human-readable and non-standardized endpoint to provide custom information about an ETP server.
   a. The URL for this endpoint is derived by appending the string '.well-known/etp-server-info' to the HTTP-equivalent URL of the ETP WebSocket end point.

i. The content of this endpoint is likely to be a branded HTML page with marketing or support information about the server and is intended to be accessible by an end user from a Web browser.

4.3.2 How Browser-based Clients use Query Parameters Instead of Header Properties

Because the HTML5 WebSocket API definition does not allow access to the request headers, it is not possible for browser-based clients to add request headers when they make a WebSocket connection request (as specified in Section **4.3**, step 4). Therefore, if a browser-based client wants/needs to use these header properties, it MUST specify them as query string parameters.

ETP defines these additional rules, which browser-based clients MUST observe and all servers MUST support:

1. The client MAY provide the header property information (e.g., **etp-encoding**) in the query string parameter of the WebSocket upgrade request.
2. All parameters provided on the query string must be URL-encoded.

   EXAMPLE:

   ```
   etp-encoding: binary
   ```

   would appear in the query string as:

   ```
   &etp-encoding=binary
   ```

5 Core (Protocol 0): Establishing and Authorizing an ETP Session

ProtocolID: 0

Defined Roles: client, server

ETP includes the notion of a session, which is an established connection between a client and server that is open for a period of time. Use Core (Protocol 0) to create, authorize, and manage an ETP session.

Before establishing an ETP session, a client must establish a WebSocket connection to an ETP server (see Section **4.3**). When the WebSocket connection has been made, then the client can use Core (Protocol 0) to authorize (optionally) and establish the ETP session.

NOTE: A client may also optionally discover information about the server, before ever connecting to it. All of this pre-session discovery behavior is explained in Chapter **4**. Section **4.1** also provides an overview of security and authorization used in ETP. Beginning in ETP v1.2, authorization may occur as part of establishing the ETP session, which is explained in this chapter.

Core (Protocol 0) has these responsibilities:

- Authorizes endpoint(s) (if required).
- Establishes and closes the ETP session.
- Negotiates the sub-protocols, data objects, compression type, message encoding and object formats to be used in a session.
- Allows endpoints in the session to inform each other of capabilities (limits) and endpoint-specific values for endpoints, data objects, and protocols that will be used during a session.
- Allows endpoints to exchange timestamps from their respective local clocks.
- Defines messages that may be used in any of the ETP sub-protocols. These so-called "universal" messages include ***ProtocolException*** and ***Acknowledge*** messages.

This chapter includes main sections for:

- Key ETP concepts that are important to understanding how this protocol is intended to work (see Section **5.1**).
- Required behavior, which includes:
  - Description of the message sequence for main tasks, along with required behavior, use of capabilities, and possible errors (see Section **5.2.1**).
  - Other functional requirements (not covered in the message sequence) including use of endpoint and protocol capabilities for preventing and protecting against aberrant behavior (see Section **5.2.2**).
  - Definitions of the endpoint and protocol capabilities used in this protocol (see Section **5.2.3**).
- Sample schemas of the messages defined in this protocol (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes definitions of each field in a schema (see Section **5.3**).

5.1 Core: Key Concepts

This section explains concepts that are important to understanding how this protocol works.

5.1.1 ETP Session

ETP includes the notion of a session, which is a negotiated context within a WebSocket connection between a client and server that is open for a period of time. Each endpoint maintains information for the life of the session (as explained in other sections of this specification).

ETP uniquely identifies each session by assigning a UUID. However, the session identification is only to help with debugging and troubleshooting. ETP does NOT maintain session state between WebSocket connections or provide any means to resume a prior session (i.e., there is no session survivability).

For important facts about ETP sessions, see Section **3.2**.

5.1.2 Security and Authorization

For information about requirements and approaches for security and authorization in ETP, see Chapter **4**.

5.2 Core: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, and identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the task, including error scenarios and resulting error codes.
- **General Requirements.** Identifies high-level and/or protocol-wide general behavior that must be observed (in addition to behavior specified in Message Sequence), including usage of relevant endpoint capabilities.
- **Capabilities.** Lists and defines the relevant parameters that set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- Both endpoints have a common version of ETP implemented, because both endpoints in an ETP session must use the same version of ETP.
- Client must have credentials for the ETP server endpoint it wants to connect and must have created a WebSocket connection as described in Section **4.3**.

5.2.1 Core: Message Sequences

This section explains the basic message sequence for main tasks to be done using this protocol and includes related key behaviors and possible errors. The following General Requirements section provides additional requirements and rules for how this protocol works.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section.

Core (Protocol 0): Basic Message-Response flow by ETP Role	
Messages (sent by Client)	**Response Message (from Server)**
RequestSession: Request for a new ETP session This and ***OpenSession*** serve as a negotiation or protocols, roles, data objects, etc. to be used in the session.	***OpenSession***: Positive response to OpenSession This and OpenSession serve as a negotiation or protocols, roles, data objects, etc. to be used in the session.
CloseSession (May be sent by either role.)	***CloseSession*** (May be sent by either role.)
ProtocolException (multipart) (May be used in any protocol, and sent by either role.)	***ProtocolException*** (multipart) (May be used in any protocol, and sent by either role.)
Acknowledge (May be used in any protocol, and sent by either role.)	***Acknowledge*** (May be used in any protocol, and sent by either role.)

Core (Protocol 0): Basic Message-Response flow by ETP Role	
Messages (sent by Client)	**Response Message (from Server)**
***Authorize*:** Used to provide authorization information or to exchange with the other endpoint to learn how to get authorization information. May be sent by either role.	***AuthorizeResponse*:** Used to convey information about how to get authorizations information or that the receiving endpoint has been authorized. May be sent by either role.
Ping (optional): "Heartbeat" message to re-establish the latest times between a customer and a store, in support of data synchronization workflows. **NOTE:** Either endpoint can send a ***Ping*** message. The other endpoint MUST respond with a ***Pong*** message.	***Pong*** (required): Response "heartbeat" message to re-establish the latest times between a customer and a store, in support of data synchronization workflows.

5.2.1.1 To establish and (optionally) authorize an ETP session:

Requirements

1. If a server requires authorization, it MAY be done at the transport layer (as part of establishing the WebSocket connection, which is explained in Section **4.3**), at the application layer, or both layers (depending on requirements of individual implementations). Instructions for authorizing in the application layer are included in the procedure below.
2. An ETP session MUST be established within the client's and server's respective values for their SessionEstablishmentTimeoutPeriod endpoint capability (for definition, see Section **5.2.3**).
   a. If a session is not successfully established within this period, either endpoint may send error ETIMED_OUT (26) and then close the WebSocket. The ***CloseSession*** message MUST NOT be sent because no session was established.
   b. The SessionEstablishmentTimeoutPeriod begins with the first ***RequestSession*** message.

Process steps

1. After the WebSocket connection has been established (as described in Section **4.3**), the client MUST send a ***RequestSession*** message (Section **5.3.1**) to the store.
   a. The client MUST send a ***RequestSession*** message within the server's value for the RequestSessionTimeoutPeriod endpoint capability (for definition, see Section **5.2.3**).
      i. If a server does not receive a ***RequestSession*** message within this period, it MAY send error ETIMED_OUT (26) and close the WebSocket connection. The ***CloseSession*** message MUST NOT be sent, because no attempt was made to establish a session.
   b. The field names on the ***RequestSession*** message are listed here for easy reference and context in this message sequence. For complete definitions, purposes and usage requirements, see Section **5.3.1**.
      i. applicationName
      ii. applicationVersion
      iii. clientInstanceId
      iv. *requestedProtocols* is the list of protocols that the client wants to use in this ETP session. For each protocol being requested, this field includes the protocol number, version, the role that the client wants the server to take in the session, and protocol capabilities with the client's values for them. The roles MUST be consistent. That is, if a client requests one role in one protocol, it MAY NOT request the other role in another protocol. **EXAMPLE:** The client may not request the store role in Store (Protocol 4) and the customer role in StoreNotification (Protocol 5). If the client requests inconsistent roles, the server MUST reject the request with EINVALID_OPERATION (32).
      NOTE: Core (Protocol 0) MUST NOT be listed in this field.

v. *supportedDataObjects*, which includes for each data object being requested, its *qualifiedType* and *dataObjectCapabilities* with the client's values for them. For more information about data object capabilities, see Section **3.3.4**.

vi. *supportedCompression*

vii. *supportedFormats*

viii. c*urrentDateTime*

ix. *earliestRetainedChangeTime*

x. *endpointCapabilities,* with the client values specified. **NOTE:** If the client requires the server to authorize to it (*serverAuthorizationRequired* field = true) this field should include the client's AuthorizationDetails endpoint capability (for definition, see Section **5.2.3**).

xi. *serverAuthorizationRequired.* A flag that if set to true means the client is indicating that the server MUST authorize with the client. **NOTE:** This field is intended for clients that are ETP stores. Clients MAY use this in other scenarios, but servers are not required to support use of this field in all cases.

c. This request MUST NOT exceed the server's value for MaxSessionClientCount endpoint capability.

   i. A server SHOULD check for this limit at the time of the WebSocket connection request (see Section **4.3**). However, it's possible that the server may not be able to determine if it must reject a specific client until the client has been authorized and it receives a ***RequestSession*** message with the client's client instance ID (*clientInstanceId*).

   ii. A server MAY refuse any incoming connections if a new connection from a particular client may cause it to exceed its value for MaxSessionClientCount endpoint capability.

   iii. If a server chooses to reject an incoming connection because it would exceed this limit, it SHOULD reject the request with ELIMIT_EXCEEDED (12).

2. The server MUST respond with one of the following:
   a. If the client already authorized (when it created the WebSocket connection) and the server requires no additional authorization, continue with Step **9**.
   b. If the server requires the client to authorize, then it MUST send error EAUTHORIZATION_REQUIRED (28).
      i. For the client to authorize to the server it MUST send the ***Authorize*** message (see Section **5.3.4**) with the *authorization* field populated with an equivalent HTTP Authorization header value (i.e., bearer token) issued by the server's authorization server.
      ii. Steps 3 and 4 explain the possible scenarios (**CASE 1** and **CASE 2**) and steps for the client to get the bearer token.
3. **CASE 1: The client got a token during the WebSocket connection process** (described in Section **4.3**):
   a. The client MUST send the ***Authorize*** message with the *authorization* field populated with the token.
   b. **If the server accepts that token,** (i.e., the client has satisfied all requirements for authorization), go to Step **5**.
   c. **If the server DOES NOT accept the token,** then it MUST send error EAUTHORIZATION_REQUIRED (28).
      i. The client MUST continue with step 4.
4. **CASE 2: The client DOES NOT have a token.** In this case, the client and server MUST exchange ***Authorize*** and ***AuthorizeResponse*** messages so the client can get the information needed to get a valid token, which is explained in steps a – d.
   a. The client MUST send the ***Authorize*** message with the *authorization* field blank (empty string).

   b. The server MUST send the ***AuthorizeResponse*** message with the *success* flag set to false and the *challenges* field MUST contain the challenges needed and the metadata with the location of the authorization server. (**NOTE:** This MUST be the same information that is specified in an endpoint's AuthorizationDetails endpoint capability, as described in Section **4.1.3**.)
      i. Depending on the specific details of the authorization requirements, the client and server MAY require several exchanges before the client has the information needed to get a valid authorization.
   c. The client uses the information in the ***AuthorizeResponse*** message to get a bearer token using the workflow described in Section **4.1.2**.
   d. After the client has acquired the bearer token, the client MUST send the ***Authorize*** message with the *authorization* field populated with a valid equivalent HTTP Authorization header value (i.e., a bearer token) accepted by the server.
   e. If the server receives too many unsuccessful attempts and there is no other valid authorization for connection, the server MAY send error EAUTHORIZATION_EXPIRED (10) and disconnect the WebSocket.

5. When the server receives the ***Authorize*** message with valid authorization information from the client, it MUST send the ***AuthorizeResponse*** message with the *success* flag set to true.
6. The client MUST re-send the ***RequestSession*** message.
   a. The client MUST send the ***RequestSession*** message within the server's value for the SessionEstablishmentTimeoutPeriod endpoint capability.
      i. If a server does not receive a ***RequestSession*** message within this period, it MAY send error ETIMED_OUT (26) and close the WebSocket connection. The ***CloseSession*** message MUST NOT be sent, because no attempt was made to establish a session.
   b. If the *serverAuthorizationRequired* flag is set to **false**, continue with Step **9**.
   c. If the *serverAuthorizationRequired* flag is set to **true**, then the client is requiring that the server authorize.
      i. For the server to authorize to the client, the server MUST send the ***Authorize*** message (see Section **5.3.4**) with the *authorization* field populated with an equivalent HTTP Authorization header value (i.e., bearer token) issued by the client's authorization server.
         1. If BOTH the server and the client require authorization, the client MUST authorize to the server first, then the server MUST authorize to the client. These MUST be sequential (NOT concurrent) operations.
      ii. The client MUST provide the metadata and challenges that comprise the AuthorizationDetails (as defined in Section **4.1.3**) to the server.
      iii. ETP specifies the 2 methods below for the client to provide the metadata and challenges to the server; endpoints MUST support BOTH methods, but use ONLY ONE method in this workflow:
         1. **METHOD A:** Populate the AuthorizationDetails endpoint capability (in the ***RequestSession*** message's *endpointCapabilities* field) with the metadata and challenges. Continue with Step **7**.
         2. **METHOD B:** Use the ***Authorize*** and ***AuthorizeResponse*** messages to iterate and provide the metadata and challenges (as described in Step 4 above, where the client is authorizing to the server). Continue with Step **8**.
7. **For METHOD A:** The server MUST use the metadata and challenges to get a valid HTTP Authorization header (i.e., a bearer token), as described in Section **4.1.2**.
   a. After the server has acquired the bearer token, the server MUST send an ***Authorize*** message with the *authorization* field populated with an equivalent HTTP Authorization header value (i.e., bearer token) accepted by the client.

   b. The client MUST respond with the ***AuthorizeResponse*** message with the *success* flag set to true.
   c. Continue with Step **9**.
8. **For METHOD B:** The server MUST use the same process described above (in Step **4**, for the client) to exchange ***Authorize*** and ***AuthorizeResponse*** messages to get the information needed to get a valid HTTP Authorization header and then use that information to get a bearer token as described in Section **4.1.2**.
   a. After the server has acquired the bearer token, the server MUST send an ***Authorize*** message with the *authorization* field populated with an equivalent HTTP Authorization header value (i.e., bearer token) accepted by the client.
   b. The client MUST respond with the ***AuthorizeResponse*** message, with the *success* flag set to true.
   c. Continue with Step **9**.
9. If the server supports at least one of the requested protocols, then the server MUST respond with the ***OpenSession*** message, indicating which of the requested protocols and roles it can support.
   a. When the server has sent the ***OpenSession*** message, the session is established and authorized (per the requirements of the two endpoints in a particular session).
   b. The field names on the ***OpenSession*** message are listed here for easy reference and context in this message sequence. For complete definitions, purposes and usage requirements, see Section **0**.
      i. applicationName
      ii. applicationVersion
      iii. serverInstanceId
      iv. supportedProtocols **NOTE:** Core (Protocol 0) MUST NOT be listed in this field.
      v. supportedDataObjects
      vi. supportedCompression
      vii. supportedFormats
      viii. sessionId
      ix. currentDateTime
      x. earliestRetainedChangeTime
      xi. endpointCapabilities
   c. Possible errors:
      i. If the server supports NONE of the requested protocols, it MUST send error ENOSUPPORTEDPROTOCOLS (2) and drop the connection.
      ii. If the server supports NONE of the roles for each protocol that the client requested, it MUST send error ENOROLE (1) and drop the connection.
      iii. If the server supports NONE of the formats for data objects that the client requested, the server MUST send error ENOSUPPORTEDFORMATS (28) and drop the connection.
      iv. For additional requirements and information, see Section **5.2.2**, Rows **5** and **6** of the table.
10. Based on the information in the ***OpenSession*** message, the client "decides" whether to terminate the session or proceed with operations that it connected to the server to perform.
   a. If the client required authorization but the server fails to do so (but sends the ***OpenSession*** message anyways) the client MUST:
      i. Send error EAUTHORIZATION_EXPIRED (10).
      ii. Send the ***CloseSession*** message (Section **5.3.3**).

b. If a client cannot correctly process an ***OpenSession*** message, it MUST do the following:
   i. Send error EREQUEST_DENIED (6).
   ii. Send the ***CloseSession*** message.
c. If the client wants to terminate the session for any reason, it MUST send the ***CloseSession*** message.
   i. If an error condition causes the client to close the session, it MUST first send a ***ProtocolException*** message (Section **5.3.8**), with an appropriate error code (for the list of codes see Section **24.3**) and then send the ***CloseSession*** message.
   ii. The ***CloseSession*** message also contains a *reason* field that SHOULD include a brief reason for why the client is closing the session.
   iii. The server MUST respond by terminating the ETP session and doing a clean shutdown of the WebSocket connection.
d. If the client wants to proceed, it begins operations by sending a message from the ETP sub-protocols that was agreed upon (*supportedProtocols*), for operations on one or more of the data objects agreed on in *supportedDataObjects*, etc.

5.2.1.2 To end an ETP session:

1. Either role can terminate an ETP session for any reason by sending a ***CloseSession*** message (Section 5.3.3), which has an optional reason field to indicate why the session is being closed.
   a. If an error condition causes an endpoint to close the session, then the endpoint MUST first send an associated ***ProtocolException*** message (Section **5.3.8**) with an appropriate error code (see Section **24.3**) before sending the ***CloseSession*** message.
2. The receiver of the ***CloseSession*** message MUST respond by terminating the ETP session and doing a clean shutdown of the WebSocket connection.

5.2.2 Core: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) some rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ProtocolException messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI format, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Section **5.2.1.1**. b. In general, the list of supported data objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	ETP Session	1. To establish an ETP session, a client MUST connect to a server as described in Section **4.3** and Section **5.2.1.1**. a. The process described in Section **5.2.1.1** is somewhat of a negotiation between the client and server (based on the types of information exchanged in the ***RequestSession*** and ***OpenSession*** messages) to determine key factors for the operations that can take place in the session. Rows **5**–**6** in this table explains some of those key negotiations and rules for them. b. Other "negotiations" are explained in the context of the field definitions on those messages. 2. Both endpoints in an ETP session MUST use the same version of ETP. 3. A server MUST assign a unique ID (a UUID) to an ETP session (in the *sessionId* field of the ***OpenSession*** message). 4. The main purpose of the sessionId is to help in debugging and troubleshooting. 5. ETP has no session survivability. If the underlying WebSocket connection is dropped, the ETP session ends.

Row#	Requirement	Behavior
		a. Clients MUST establish new connections using the process described in Section **4.3** and Section **5.2.1.1**.
3.	Capabilities-related behavior	1. Relevant ETP-defined endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3**. a. For the list of global capabilities and related behavior, see Section **3.3.2**. 3. The capabilities listed in Section **5.2.3** MUST BE used in this ETP sub-protocol. Additional details for how to use the protocol capabilities are included below in this table and in Section **5.2.1 Core: Message Sequences**.
4.	Use of clocks in ETP endpoints	1. ETP endpoints MUST have a clock. 2. As part of establishing an ETP session, a client and server exchange their respective *currentDateTime*. a. The purpose of this field is part of the behavior for eventual consistency between 2 stores. For more information, see **Appendix: Data Replication and Outage Recovery Workflows**. 3. Optionally, an endpoint can send a **Ping**, a messages used to re-establish the latest times between a customer and a store, in support of data synchronization workflows. a. If an endpoint sends a ***Ping*** message, the other endpoint in the ETP session MUST respond with a ***Pong*** message. 4. For more information about timestamps, see Section **3.12.5**.
5.	Protocol Negotiation	1. A client and server determine which protocols they will use in an ETP session using the *requestedProtocols* and *supportedProtocols* fields (on the ***RequestSession*** and ***OpenSession*** messages, respectively), which for each protocol includes protocol number, version, client-requested role, and protocol capabilities. a. The negotiated protocols are essentially the intersection of *requestedProtocols* and *supportedProtocols.* b. These fields MUST NOT list Core (Protocol 0). 2. In addition to the rules specified in Section **5.2.1.1,** the server in its ***OpenSession*** response message: a. MUST NOT change the version of a requested protocol. If the server cannot support the exact version requested, then the server MUST treat the requested protocol(s) as 'unsupported'. b. MAY offer to support only some of the requested protocols. c. MUST NOT offer to support any additional protocols. d. MUST NOT change the requested role for each protocol and MUST fill only one role (the one specified by the client). 3. If the server response does not provide adequate functionality, then the client MAY send the ***CloseSession*** message immediately. 4. During the ETP session, endpoints MUST use only the supported protocols that were negotiated. a. If a client tries to use a protocol not included in the *supportedProtocols*, field, the server MUST send error EUNSUPPORTED_PROTOCOL (4).
6.	Negotiation of supported data objects and related capabilities	1. A client and server determine which data objects they will use in an ETP session using the *supportedDataObjects* fields (same name on the ***RequestSession*** and ***OpenSession*** messages). The negotiated supported data objects are essentially the intersection of the two fields. 2. For each data object in these fields, each endpoint MUST list its supported data objects, and for each MUST include: a. A qualifiedType

Row#	Requirement	Behavior
		b. DataObjectCapabilities, which indicates the operations permitted on a particular data object during that ETP session. (For more information about dataObjectCapabilities, see Section 3.3.4.) 3. During the ETP session, endpoints MUST use only the supported data objects that were negotiated and MUST only perform operations (i.e., get, delete, and put) and honor limits specified on the capabilities for each data object type. a. If the intersection between the client's requested list of data objects and server's supported list of data objects is empty, the server MUST send error ENOSUPPORTEDDATAOBJECTTYPES (29).
7.	Messages that MUST NEVER be compressed.	1. If in the ***MessageHeader*** record, the *protocol* field = 0, the message MUST NEVER be compressed. a. **NOTE:** ***ProtocolException*** and ***Acknowledge*** messages are defined in Core (Protocol 0); however, they may be used in any protocol, so their *protocol* field is rarely 0. 2. If an endpoint receives a message with *protocol*=0 that is compressed, it MUST send error ECOMPRESSION_NOTSUPPORTED (13).
8.	Authorization renewal and expiration	4. An endpoint SHOULD remain authorized with the other endpoint (as required by the respective endpoints when the ETP session was established) for the duration of the ETP session. a. An endpoint MUST re-authorize with the other endpoint BEFORE the current authorization expires. i. For the high-level workflow on how an endpoint gets a bearer token, see Section 4.1.2. b. As needed, either endpoint CAN send the ***Authorize*** message (as described in Section 5.2.1.1) at any time, to remain authorized for the duration of the session. c. After the initial authorization, the authorization method and security principal MUST not change and the scope MUST not be reduced. d. The authorization for each endpoint may have very different expirations, so each endpoint may re-authorize to the other at different times. 5. If an endpoint's authorization will expire "soon", the other endpoint MAY send error EAUTHORIZATION_EXPIRING (28). a. For more information, see the detailed text on the error code in Section 24.3. 6. During an ETP session, if an endpoint's authorization expires, the other endpoint MUST: a. Send error EAUTHORIZATION_EXPIRED (10). b. Send the ***CloseSession*** message.

5.2.3 Core: Capabilities

The table below lists key capabilities for this protocol.

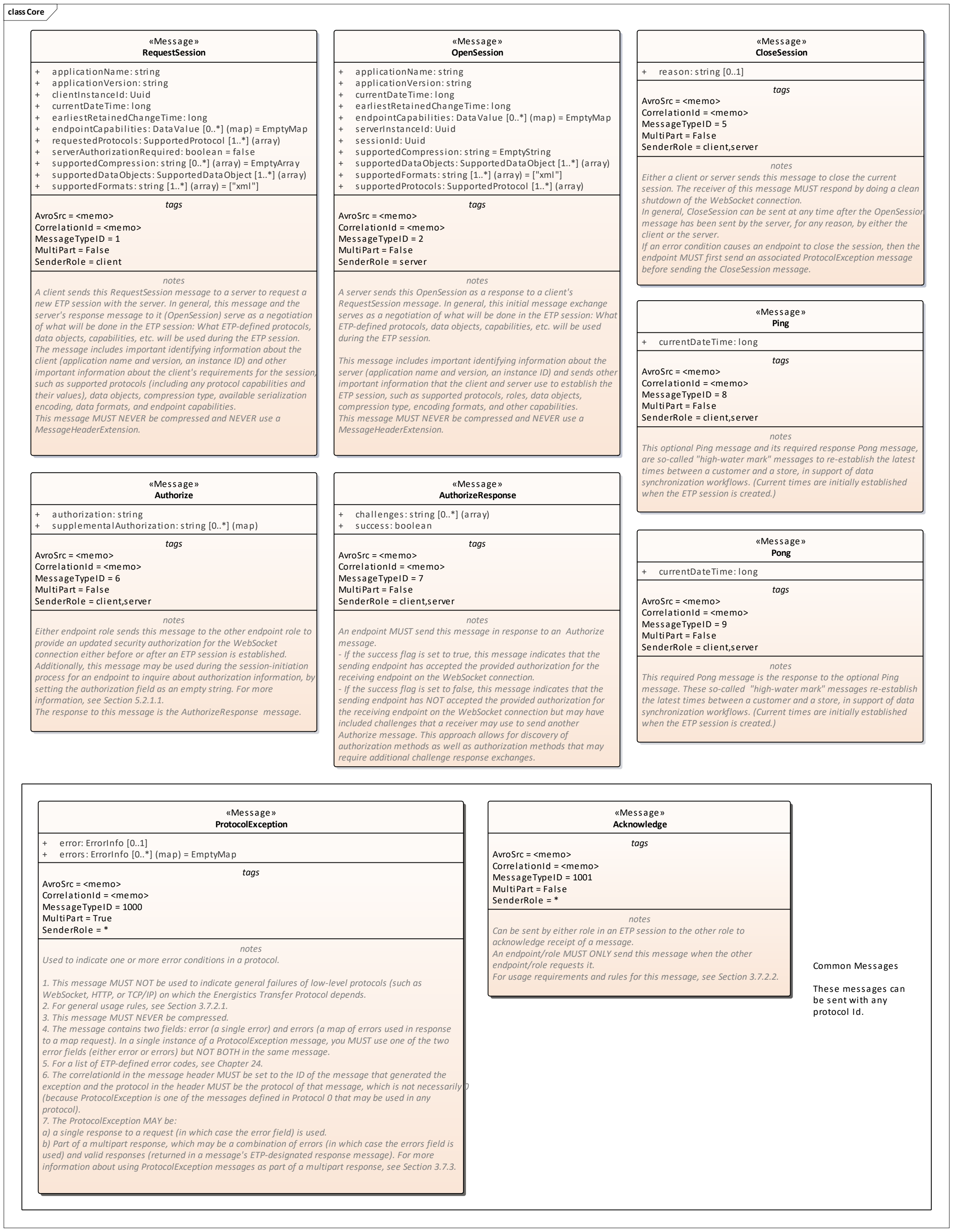
- For protocol-specific behavior for using these capabilities in this protocol, see Section **5.2.1 Core: Message Sequences**.
- For definitions of endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section **3.3**.

Core (Protocol 0): Capabilities			
Name: Description	**Type**	**Units** **Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of each endpoint capability, see Section **3.3**.)			

Core (Protocol 0): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section **3.3.2**. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP Server.**			
RequestSessionTimeoutPeriod: The maximum time period in seconds a server will wait to receive a ***RequestSession*** message from a client after the WebSocket connection has been established.	long	seconds <number of seconds>	**Default:** 45 **Min:** 5
SessionEstablishmentTimeoutPeriod: The maximum time period in seconds a client or server will wait for a valid ETP session to be established. **For a server**: • A valid session is established when it sends an ***OpenSession*** message to the client, which indicates a session has been successfully established. • The time period starts on receiving the initial ***RequestSession*** message from the client. **For a client:** • A valid session is established when it receives an ***OpenSession*** message from the server. • The time period starts when it sends the initial ***RequestSession*** message to the server.	long	seconds <number of seconds>	**Default:** 60 **Min:** 5
AuthorizationDetails: 1. Contains an ArrayOfString with WWW-Authenticate style challenges. 2. To support the required authorization workflow (to enable an endpoint to acquire an access token with the necessary scope from the designated authorization server), the AuthorizationDetails endpoint capability MUST include at least one challenge with the Bearer scheme which must include the 'authz_server' and 'scope' parameters. a. The 'authz_server' parameter MUST be a URI for an authorization server to enable the endpoint to acquire any other needed metadata about the authorization server using OpenID Connect Discovery. 3. An ETP server MUST have the AuthorizationDetails endpoint capability, which must meet the requirements of Point 2 above. 4. If an ETP client does NOT need to authorize ETP servers, it MAY omit the AuthorizationDetails.			
Protocol Capabilities			
NONE			

5.3 Core: Message Schemas

This section provides a figure that displays all messages in Core (Protocol 0). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each



class Core

```
«Message»
RequestSession
+   applicationName: string
+   applicationVersion: string
+   clientInstanceId: Uuid
+   currentDateTime: long
+   earliestRetainedChangeTime: long
+   endpointCapabilities: DataValue [0..*] (map) = EmptyMap
+   requestedProtocols: SupportedProtocol [1..*] (array)
+   serverAuthorizationRequired: boolean = false
+   supportedCompression: string [0..*] (array) = EmptyArray
+   supportedDataObjects: SupportedDataObject [1..*] (array)
+   supportedFormats: string [1..*] (array) = ["xml"]
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1
MultiPart = False
SenderRole = client
notes
```

A client sends this RequestSession message to a server to request a new ETP session with the server. In general, this message and the server's response message to it (OpenSession) serve as a negotiation of what will be done in the ETP session: What ETP-defined protocols, data objects, capabilities, etc. will be used during the ETP session.
The message includes important identifying information about the client (application name and version, an instance ID) and other important information about the client's requirements for the session, such as supported protocols (including any protocol capabilities and their values), data objects, compression type, available serialization encoding, data formats, and endpoint capabilities.
This message MUST NEVER be compressed and NEVER use a MessageHeaderExtension.

```
«Message»
OpenSession
+   applicationName: string
+   applicationVersion: string
+   currentDateTime: long
+   earliestRetainedChangeTime: long
+   endpointCapabilities: DataValue [0..*] (map) = EmptyMap
+   serverInstanceId: Uuid
+   sessionId: Uuid
+   supportedCompression: string = EmptyString
+   supportedDataObjects: SupportedDataObject [1..*] (array)
+   supportedFormats: string [1..*] (array) = ["xml"]
+   supportedProtocols: SupportedProtocol [1..*] (array)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 2
MultiPart = False
SenderRole = server
notes
```

A server sends this OpenSession as a response to a client's RequestSession message. In general, this initial message exchange serves as a negotiation of what will be done in the ETP session: What ETP-defined protocols, data objects, capabilities, etc. will be used during the ETP session.

This message includes important identifying information about the server (application name and version, an instance ID) and sends other important information that the client and server use to establish the ETP session, such as supported protocols, roles, data objects, compression type, encoding formats, and other capabilities.
This message MUST NEVER be compressed and NEVER use a MessageHeaderExtension.

```
«Message»
CloseSession
+   reason: string [0..1]
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 5
MultiPart = False
SenderRole = client,server
notes
```

Either a client or server sends this message to close the current session. The receiver of this message MUST respond by doing a clean shutdown of the WebSocket connection.
In general, CloseSession can be sent at any time after the OpenSession message has been sent by the server, for any reason, by either the client or the server.
If an error condition causes an endpoint to close the session, then the endpoint MUST first send an associated ProtocolException message before sending the CloseSession message.

```
«Message»
Ping
+   currentDateTime: long
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 8
MultiPart = False
SenderRole = client,server
notes
```

This optional Ping message and its required response Pong message, are so-called "high-water mark" messages to re-establish the latest times between a customer and a store, in support of data synchronization workflows. (Current times are initially established when the ETP session is created.)

```
«Message»
Authorize
+   authorization: string
+   supplementalAuthorization: string [0..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 6
MultiPart = False
SenderRole = client,server
notes
```

Either endpoint role sends this message to the other endpoint role to provide an updated security authorization for the WebSocket connection either before or after an ETP session is established.
Additionally, this message may be used during the session-initiation process for an endpoint to inquire about authorization information, by setting the authorization field as an empty string. For more information, see Section 5.2.1.1.
The response to this message is the AuthorizeResponse message.

```
«Message»
AuthorizeResponse
+   challenges: string [0..*] (array)
+   success: boolean
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 7
MultiPart = False
SenderRole = client,server
notes
```

An endpoint MUST send this message in response to an Authorize message.
- If the success flag is set to true, this message indicates that the sending endpoint has accepted the provided authorization for the receiving endpoint on the WebSocket connection.
- If the success flag is set to false, this message indicates that the sending endpoint has NOT accepted the provided authorization for the receiving endpoint on the WebSocket connection but may have included challenges that a receiver may use to send another Authorize message. This approach allows for discovery of authorization methods as well as authorization methods that may require additional challenge response exchanges.

```
«Message»
Pong
+   currentDateTime: long
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 9
MultiPart = False
SenderRole = client,server
notes
```

This required Pong message is the response to the optional Ping message. These so-called "high-water mark" messages re-establish the latest times between a customer and a store, in support of data synchronization workflows. (Current times are initially established when the ETP session is created.)

```
«Message»
ProtocolException
+   error: ErrorInfo [0..1]
+   errors: ErrorInfo [0..*] (map) = EmptyMap
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1000
MultiPart = True
SenderRole = *
notes
```

Used to indicate one or more error conditions in a protocol.

1. This message MUST NOT be used to indicate general failures of low-level protocols (such as WebSocket, HTTP, or TCP/IP) on which the Energistics Transfer Protocol depends.
2. For general usage rules, see Section 3.7.2.1.
3. This message MUST NEVER be compressed.
4. The message contains two fields: error (a single error) and errors (a map of errors used in response to a map request). In a single instance of a ProtocolException message, you MUST use one of the two error fields (either error or errors) but NOT BOTH in the same message.
5. For a list of ETP-defined error codes, see Chapter 24.
6. The correlationId in the message header MUST be set to the ID of the message that generated the exception and the protocol in the header MUST be the protocol of that message, which is not necessarily 0 (because ProtocolException is one of the messages defined in Protocol 0 that may be used in any protocol).
7. The ProtocolException MAY be:
a) a single response to a request (in which case the error field) is used.
b) Part of a multipart response, which may be a combination of errors (in which case the errors field is used) and valid responses (returned in a message's ETP-designated response message). For more information about using ProtocolException messages as part of a multipart response, see Section 3.7.3.

```
«Message»
Acknowledge
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1001
MultiPart = False
SenderRole = *
notes
```

Can be sent by either role in an ETP session to the other role to acknowledge receipt of a message.
An endpoint/role MUST ONLY send this message when the other endpoint/role requests it.
For usage requirements and rules for this message, see Section 3.7.2.2.

Common Messages

These messages can be sent with any protocol Id.

Figure 10: Core message schemas

5.3.1 Message: RequestSession

A client sends this RequestSession message to a server to request a new ETP session with the server. In general, this message and the server's response message to it (OpenSession) serve as a negotiation of what will be done in the ETP session: What ETP-defined protocols, data objects, capabilities, etc. will be used during the ETP session.

The message includes important identifying information about the client (application name and version, an instance ID) and other important information about the client's requirements for the session, such as supported protocols (including any protocol capabilities and their values), data objects, compression type, available serialization encoding, data formats, and endpoint capabilities.

This message MUST NEVER be compressed and NEVER use a MessageHeaderExtension.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: client

Field Name	Description	Data Type	Min	Max
applicationName	The string by which the client identifies itself, normally a software product or system name. The format is entirely application dependent. Vendors are encouraged to identify their company name as part of this string.	string	1	1
applicationVersion	The version of the application identified in *applicationName*.	string	1	1
clientInstanceId	A UUID that a client assigns itself to uniquely identify this instance of the client in an ETP session. It must be of the type Uuid. See also, *serverInstanceId* and *sessionId* (in the OpenSession message).	Uuid	1	1
requestedProtocols	The ETP sub-protocol(s) and associated information for each sub-protocol that the client expects to communicate on for the ETP session. It is an array of SupportedProtocol records, each of which identifies a sub-protocol ID, the role for the sub-protocol it expects the server to fill, and name-value pairs of protocol capabilities. **NOTES:** 1. An ETP sub-protocol MUST appear only once in this array. 2. Each sub-protocol MUST specify only one role (for the server). 3. Core (Protocol 0) MUST NOT be included in this list. 4. Requested roles MUST be consistent across protocols in an ETP session. **EXAMPLE:** An endpoint CANNOT request to be customer in one protocol and store in another, in the same ETP session.	SupportedProtocol	1	*
supportedDataObjects	The data objects that the client wants to use in this session and the information for each. It is an array of SupportedDataObject records. This field MUST be populated. Client and server use this field (in ***RequestSession*** and ***OpenSession*** messages respectively) to negotiate the objects that will be used during the session and determine the data object capabilities for each.	SupportedDataObject	1	*
supportedCompression	An array of compression algorithms supported by the client, in order of preference. An empty array indicates compression is not supported.	string	0	*

Field Name	Description	Data Type	Min	Max
	• If a client or server supports compression, it MUST support at least gzip. • If compression is used during an ETP session, a ***MessageHeader*** is NEVER compressed. Only content following the ***MessageHeader*** (the message body and optional ***MessageHeaderExtension***, if used) are compressed.			
supportedFormats	An array of data formats supported by the client, in order of preference. The format(s) are used when sending data objects or growing data object parts in ETP messages, so the receiver of messages knows how to deserialize the array of bytes representing the data object or growing data object part in the message. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats. **Default:** xml **NOTE:** An endpoint indicates in the message, per request and response, which format it wants to use or is being used.	string	1	*
currentDateTime	The current date and time of the endpoint's system clock. When establishing an ETP session, each endpoint indicates its current date and time. The purpose of this field is part of the behavior for eventual consistency between 2 stores. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
earliestRetainedChangeTime	When the endpoint is a store, the endpoint MUST set this to the earliest timestamp that customers may use to request retained change information, such as deleted resources and change annotations. For some stores, if the store has not yet been running longer than its value for the ChangeRetentionPeriod capability, the value in this field MAY be more recent than the value for ChangeRetentionPeriod. Customers should not request and stores will not provide retained change information from before this timestamp.	long	1	1
endpointCapabilities	A map of key-value pairs of endpoint-specific capability data (i.e., constraints, limitations). The names, defaults, optionality, and expected data types are defined by this specification. These endpoint capabilities are exchanged in this and the ***OpenSession*** message between the 2 endpoints for use in applicable protocols as defined in relevant chapters in this specification. • Map keys are capability names, which are case-sensitive strings. For ETP-defined capabilities, the name must spelled exactly as listed in EndpointCapabilityKind. • Map values are of type DataValue. • For more information about capabilities and rules for using them, see Section **3.3**.	DataValue	0	*
serverAuthorizationRequired	A flag that if set to true means the client is indicating that the server MUST authorize with the client. **NOTE:** This field is intended for clients that are ETP stores. Clients MAY use this in other scenarios, but servers are not required to support use of this field in all cases**.**	boolean	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Core",
     "name": "RequestSession",
     "protocol": "0",
     "messageType": "1",
     "senderRole": "client",
     "protocolRoles": "client, server",
     "multipartFlag": false,

     "fields":
     [
         { "name": "applicationName", "type": "string" },
         { "name": "applicationVersion", "type": "string" },
         { "name": "clientInstanceId", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         {
             "name": "requestedProtocols",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.SupportedProtocol" }
         },
         {
             "name": "supportedDataObjects",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.SupportedDataObject" }
         },
         {
             "name": "supportedCompression",
             "type": { "type": "array", "items": "string" }, "default": []
         },
         {
             "name": "supportedFormats",
             "type": { "type": "array", "items": "string" }, "default": ["xml"]
         },
         { "name": "currentDateTime", "type": "long" },
         { "name": "earliestRetainedChangeTime", "type": "long" },
         { "name": "serverAuthorizationRequired", "type": "boolean", "default": false },
         {
             "name": "endpointCapabilities",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

5.3.2 Message: OpenSession

A server sends this OpenSession as a response to a client's RequestSession message. In general, this initial message exchange serves as a negotiation of what will be done in the ETP session: What ETP-defined protocols, data objects, capabilities, etc. will be used during the ETP session.

This message includes important identifying information about the server (application name and version, an instance ID) and sends other important information that the client and server use to establish the ETP session, such as supported protocols, roles, data objects, compression type, encoding formats, and other capabilities.

This message MUST NEVER be compressed and NEVER use a MessageHeaderExtension.

Message Type ID: 2

Correlation Id Usage: MUST be set to the *messageId* of the ***RequestSession*** message that initiated the creation of the session.

Multi-part: False

Sent by: server

Field Name	Description	Data Type	Min	Max
applicationName	The string by which the server identifies itself, normally a software product or system name. The name may or may not include a version. The format is entirely application dependent. Vendors are encouraged to identify their company name as part of this string.	string	1	1
applicationVersion	The version of the application identified in *applicationName*.	string	1	1
serverInstanceId	A UUID that a server assigns itself to uniquely identify the instance of the server in an ETP session. It must be of the type Uuid. See also: sessionId and clientInstanceId (in the RequestSession message).	Uuid	1	1
supportedProtocols	The ETP sub-protocols and associated information for each sub-protocol that the server will support in response to the client's request. It is an array of SupportedProtocol records , each of which identifies the protocol IDs that the server will support for this session, the role it will use for each protocol (as assigned by the client in the RequestSession message), and key-value pairs of related capabilities. **NOTES:** 1. This array MUST be all or a subset of the protocols that the client requested in the *supportedProtocols* field of the ***RequestSession*** message. 2. A server may be capable of supporting both roles in an ETP sub-protocol, but in any given session it MUST fill only one role (the one requested by the client in the ***RequestSession*** message). 3. Core (Protocol 0) MUST NOT be included in this list.	SupportedProtocol	1	*
supportedDataObjects	A list of the data objects that the client wants to use in this session and the information for each as specified in SupportedDataObject. This field MUST be populated. Client and server use this field (in RequestSession and OpenSession messages respectively) to negotiate the objects that will be used during the session and determine the "capabilities" for each (get, put, del).	SupportedDataObject	1	*
supportedCompression	The compression algorithm supported by both client and server with the highest preference specified by the client (in the ***RequestSession*** message). An empty string indicates there were no mutually supported compression options. **EXAMPLE:** "gzip" • If a client or server supports compression, it MUST support at least gzip. • If compression is used during an ETP session, a ***MessageHeader*** is NEVER compressed. Only content following the ***MessageHeader*** (the message body and optional ***MessageHeaderExtension***, if used) are compressed.	string	1	1
supportedFormats	An array of data formats supported by the client, in order of preference. The format(s) are used when sending data objects or growing data object parts in ETP messages, so the receiver of messages knows how to deserialize the	string	1	*

Field Name	Description	Data Type	Min	Max
	array of bytes representing the data object or growing data object part in the message. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats. **Default:** xml **NOTE:** An endpoint indicates in the message, per request and response, which format it wants to use or is being used.			
currentDateTime	The current date and time of the endpoint's system clock. When establishing an ETP session, each endpoint indicates its current date and time. The purpose of this field is part of the behavior for eventual consistency between 2 stores. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
earliestRetainedChangeTime	When the endpoint is a store, the endpoint MUST set this to the earliest timestamp that customers may use to request retained change information, such as deleted resources and change annotations. For some stores, if the store has not yet been running longer than its value for the ChangeRetentionPeriod capability, the value in this field MAY be more recent than the value for ChangeRetentionPeriod. Customers should not request and stores will not provide retained change information from before this timestamp.	long	1	1
sessionId	An ID (UUID) that the server assigns to uniquely identify an ETP session; It must be of the type Uuid. The sessionId is only to help with debugging and troubleshooting. ETP does NOT maintain session state (i.e., there is no session survivability).	Uuid	1	1
endpointCapabilities	A map of key-value pairs of endpoint-specific capability data (i.e., constraints, limitations). The names, defaults, optionality, and expected data types are defined by this specification. These endpoint capabilities are exchanged in this and the ***RequestSession*** message between the 2 endpoints for use in applicable protocols as defined in relevant chapters in this specification. • Map keys are capability names, which are case-sensitive strings. For ETP-defined capabilities, the name must spelled exactly as listed in EndpointCapabilityKind. • Map values are of type DataValue. • For more information about capabilities and rules for using them, see Section **3.3**. Additionally, the ServerCapabilities may list a server's endpoint capabilities, though they may vary from the ones listed here for various reasons.	DataValue	0	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Core",
     "name": "OpenSession",
     "protocol": "0",
     "messageType": "2",
     "senderRole": "server",
     "protocolRoles": "client, server",
     "multipartFlag": false,

     "fields":
     [
         { "name": "applicationName", "type": "string" },
         { "name": "applicationVersion", "type": "string" },
         { "name": "serverInstanceId", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         {
             "name": "supportedProtocols",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.SupportedProtocol" }
         },
         {
             "name": "supportedDataObjects",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.SupportedDataObject" }
         },
         { "name": "supportedCompression", "type": "string", "default": "" },
         {
             "name": "supportedFormats",
             "type": { "type": "array", "items": "string" }, "default": ["xml"]
         },
         { "name": "currentDateTime", "type": "long" },
         { "name": "earliestRetainedChangeTime", "type": "long" },
         { "name": "sessionId", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         {
             "name": "endpointCapabilities",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

5.3.3 Message: CloseSession

Either a client or server sends this message to close the current session. The receiver of this message MUST respond by doing a clean shutdown of the WebSocket connection.

In general, ***CloseSession*** can be sent at any time after ***OpenSession*** message has been sent by the server, for any reason, by either the client or the server.

If an error condition causes an endpoint to close the session, then the endpoint MUST first send an associated ***ProtocolException*** message before sending the ***CloseSession*** message.

Message Type ID: 5

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: client,server

Field Name	Description	Data Type	Min	Max
reason	The reason for requesting the session be closed.	string	0	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Core",
     "name": "CloseSession",
     "protocol": "0",
     "messageType": "5",
     "senderRole": "client,server",
     "protocolRoles": "client, server",
     "multipartFlag": false,

     "fields":
     [
         { "name": "reason", "type": "string", "default": "" }
     ]
}
```

5.3.4 Message: Authorize

Either endpoint role sends this message to the other endpoint role to provide an updated security authorization for the WebSocket connection either before or after an ETP session is established.

Additionally, this message may be used during the session-initiation process for an endpoint to inquire about authorization information, by setting the *authorization* field as an empty string. For more information, see Section **5.2.1.1**.

The response to this message is the AuthorizeResponse message.

Message Type ID: 6

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: client,server

Field Name	Description	Data Type	Min	Max
authorization	• If providing authorization information to an endpoint, this field MUST be populated with the content of the equivalent HTTP Authorization header (NOT just a token). **EXAMPLES:** ○ bearer {TOKEN} ○ bearer cn389ncoiwuencr • If this message is being used to inquire about an endpoint's supported authorization mechanisms, this field MUST be empty.	string	1	1
supplementalAuthorization	A map of strings for additional authorization information that may be required in the future for new authorization mechanisms.	string	0	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Core",
     "name": "Authorize",
     "protocol": "0",
     "messageType": "6",
     "senderRole": "client,server",
     "protocolRoles": "client, server",
     "multipartFlag": false,

     "fields":
     [
         { "name": "authorization", "type": "string" },
         {
             "name": "supplementalAuthorization",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

5.3.5 Message: Ping

This optional ***Ping*** message and its required response Pong message, are so-called "high-water mark" messages to re-establish the latest times between a customer and a store, in support of data synchronization workflows. (Current times are initially established when the ETP session is created.)

Message Type ID: 8

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: client,server

Field Name	Description	Data Type	Min	Max
currentDateTime	The current date and time of the endpoint's system clock. When establishing an ETP session, each endpoint indicates its current date and time. If there has been no activity (messages exchanged), this establishes a new current time for the endpoints. The purpose of this field is part of the behavior for eventual consistency between 2 stores. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Core",
     "name": "Ping",
     "protocol": "0",
     "messageType": "8",
     "senderRole": "client,server",
     "protocolRoles": "client, server",
     "multipartFlag": false,

     "fields":
     [
         { "name": "currentDateTime", "type": "long" }
     ]
}
```

5.3.6 Message: Pong

This required Pong message is the response to the optional Ping message. These so-called "high-water mark" messages re-establish the latest times between a customer and a store, in support of data synchronization workflows. (Current times are initially established when the ETP session is created.)

Message Type ID: 9

Correlation Id Usage: MUST be set to the *messageId* of the ***Ping*** message that this is a response to.

Multi-part: False

Sent by: client,server

Field Name	Description	Data Type	Min	Max
currentDateTime	The current date and time of the endpoint's system clock. When establishing an ETP session, each endpoint indicates its current date and time. If there has been no activity (messages	long	1	1

Field Name	Description	Data Type	Min	Max
	exchanged), this establishes a new current time for the endpoints. The purpose of this field is part of the behavior for eventual consistency between 2 stores. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).			

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Core",
     "name": "Pong",
     "protocol": "0",
     "messageType": "9",
     "senderRole": "client,server",
     "protocolRoles": "client, server",
     "multipartFlag": false,

     "fields":
     [
         { "name": "currentDateTime", "type": "long" }
     ]
}
```

5.3.7 Message: AuthorizeResponse

An endpoint MUST send this message in response to an Authorize message.

- If the *success* flag is set to true, this message indicates that the sending endpoint has accepted the provided authorization for the receiving endpoint on the WebSocket connection.
- If the *success* flag is set to false, this message indicates that the sending endpoint has NOT accepted the provided authorization for the receiving endpoint on the WebSocket connection but may have included challenges that a receiver may use to send another Authorize message. This approach allows for discovery of authorization methods as well as authorization methods that may require additional challenge response exchanges.

Message Type ID: 7

Correlation Id Usage: MUST be set to the *messageId* of the ***Authorize*** message that this is a response to.

Multi-part: False

Sent by: client,server

Field Name	Description	Data Type	Min	Max
success	If the *success* flag is set to true, this message indicates that the receiving endpoint has been successfully authorized (to the endpoint that is sending the message).	boolean	1	1
challenges	Contains the challenge needed and possibly metadata on where to get authorization information. This field MUST only be non-empty when the success attribute is false.	string	0	*

Avro Source

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Protocol.Core",
    "name": "AuthorizeResponse",
    "protocol": "0",
    "messageType": "7",
    "senderRole": "client,server",
    "protocolRoles": "client, server",
    "multipartFlag": false,

    "fields":
    [
        { "name": "success", "type": "boolean" },
        {
            "name": "challenges",
            "type": { "type": "array", "items": "string" }
        }
    ]
}
```

5.3.8 Message: ProtocolException

Used to indicate one or more error conditions in a protocol.

1. This message MUST NOT be used to indicate general failures of low-level protocols (such as WebSocket, HTTP, or TCP/IP) on which the Energistics Transfer Protocol depends.
2. For general usage rules, see Section **3.7.2.1**.
3. This message MUST NEVER be compressed.
4. The message contains two fields: *error* (a single error) and *errors* (a map of errors used in response to a map request). In a single instance of a ***ProtocolException*** message, you MUST use one of the two error fields (either *error* or *errors*) but NOT BOTH in the same message.
5. For a list of ETP-defined error codes, see Chapter **24**.
6. The *correlationId* in the message header MUST be set to the ID of the message that generated the exception and the protocol in the header MUST be the protocol of that message, which is not necessarily 0 (because ***ProtocolException*** is one of the messages defined in Protocol 0 that may be used in any protocol).
7. The ProtocolException MAY be:
    a. A single response to a request (in which case the *error* field is used).
    b. Part of a multipart response, which may be a combination of errors (in which case the *errors* field is used) and valid responses (returned in a message's ETP-designated response message). For more information about using ***ProtocolException*** messages as part of a multipart response, see Sections **3.7.3**.

Message Type ID: 1000

Correlation Id Usage: MUST be set to the *messageId* of the message that caused the exception to be raised.

Multi-part: True

Sent by: *

Field Name	Description	Data Type	Min	Max
error	Field for use for a single error code and related message. In a single instance of a ***ProtocolException*** message, you MUST use one of the two error fields (either *error* or *errors*), but you MUST NOT use both in the same message.	ErrorInfo	0	1

Field Name	Description	Data Type	Min	Max
	If both fields are populated, the behavior is undefined.			
errors	A map of errors, which MUST be used in response to a request message that contains a map. In a single instance of a ***ProtocolException*** message, you MUST use one of the two error fields (either *error* or *errors*), but you MUST NOT use both in the same message. If both fields are populated, the behavior is undefined.	ErrorInfo	0	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Core",
     "name": "ProtocolException",
     "protocol": "0",
     "messageType": "1000",
     "senderRole": "*",
     "protocolRoles": "client, server",
     "multipartFlag": true,
     "fields":
     [
         { "name": "error", "type": ["null", "Energistics.Etp.v12.Datatypes.ErrorInfo"] },
         {
             "name": "errors",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.ErrorInfo" },
"default": {}
         }
     ]
}
```

5.3.9 Message: Acknowledge

Can be sent by either role in an ETP session to the other role to acknowledge receipt of a message.

An endpoint/role MUST ONLY send this message when the other endpoint/role requests it.

For usage requirements and rules for this message, see Section **3.7.2.2**.

Message Type ID: 1001

Correlation Id Usage: MUST be set to the *messageId* of the message whose receipt is being acknowledged.

Multi-part: False

Sent by: *

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Core",
     "name": "Acknowledge",
     "protocol": "0",
     "messageType": "1001",
     "senderRole": "*",
     "protocolRoles": "client, server",
     "multipartFlag": false,

     "fields":
     [

     ]
}
```

6 ChannelStreaming (Protocol 1)

ProtocolID: 1

Defined Roles: producer, consumer

Use ChannelStreaming (Protocol 1) to stream channel-oriented data from an endpoint that is a "simple" producer (i.e., a sensor) to a consumer endpoint. Beginning in ETP v1.2, Protocol 1 is used only for so-called "simple streamers" (in previous versions of ETP, Protocol 1 included all channel streaming behavior).

The main use case that this protocol supports is basic "WITS-like", one-way data streaming from a sensor or other relatively "dumb" device. There is no real "back and forth" between the endpoints—that is, the consumer cannot discover available channels nor specify which channels it wants; the producer simply sends any data it has. Also, there is no flow control—except for "stop". In its simplest form, this protocol supports a workflow of connect and begin receiving data. Reliability (data transmission without loss) cannot be guaranteed.

Other ETP sub-protocols that may be used with or instead of ChannelStreaming◦(Protocol 1):

- **ChannelSubscribe (Protocol 21)** (see Chapter 19): Has the "get/read" behavior for channel data, allowing a customer to connect to a store and to "listen" for changes in channel data that require a notification (or data updates) to be sent while connected. Functionality includes standard publish/subscribe behavior that was previously included in Protocol 1, though has been enhanced in the current version of ETP.
- **ChannelDataLoad (Protocol 22)** (see Chapter 20): New in ETP v1.2, this protocol provides the "put/write" behavior for channel data. Protocol 22 "pushes" data from the customer role endpoint to the store role endpoint. Main use cases include rig acquisition systems or any case in which you want to load a lot of data to a system or store.
- **ChannelDataFrame (Protocol 2)** (see Chapter 7) allows a customer endpoint to get channel data from a store in a row-orientated 'frame' or 'table' of data. In oil and gas jargon, the general use case that Protocol 2 supports is typically referred to as getting a "historical log". (In ETP jargon you are actually getting a frame of data from a ChannelSet data object; for more information, see Section 7.1.1).

This chapter includes main sections for:

- Key ETP concepts that are important to understanding how this protocol is intended to work, specifically definitions of channels and related constructs in ETP (see Section 6.1).
- Required behavior, which includes:
  - Description of the message sequence for main tasks, along with required behavior and possible errors (see Section 6.2.1).
  - Other functional requirements not covered in the message sequence (see Section 6.2.2).
  - Definitions of the endpoint capabilities used in this protocol (see Section 6.2.3). For this basic protocol, there are no protocol capabilities.
- Sample schemas of the messages defined in this protocol (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes documentation for each field in a schema (see Section 6.3).

6.1 Channels: Key Concepts

This section explains concepts that are important to understanding how ETP channel streaming protocols work, which includes ChannelStreaming (Protocol 1), ChannelSubscribe (Protocol 21), and ChannelDataLoad (Protocol 22). Unless otherwise specified, you can assume the information below applies to all channel streaming protocols.

6.1.1 Channel Definition and its Design in Energistics Standards

In oil and gas jargon, a channel (in its simplest form) is series of indexes, each with an associated value. The index is traditionally time or depth and the "associated value" is often a measurement (of something other than time or depth, e.g., pressure, porosity, etc.) at a particular index. (Though channel values can also be other kinds of data, such as comments, which can also be streamed.)

In Energistics domain standards (WITSML, RESQML and PRODML) and ETP a channel is described by a named data object in the system, identified by a URI. The Channel data object includes the identifying information about the channel, such as its name/mnemonic, and some metadata about the channel like when it was created and last updated, status, etc., which is much of the information that is defined for all Energistics data objects. As a data object, operations such as adding, deleting, and updating are done using Store (Protocol 4), and a customer can subscribe to receive notifications of these operations using StoreNotification (Protocol 5).

The Channel data object does not include the actual index/value pairs, just the descriptive information. In ETP, the channel data is moved using one of the channel protocols (i.e., ChannelStreaming (Protocol 1), ChannelSubscribe (Protocol 21) (see Chapter **19**) or ChannelDataLoad (Protocol 22) (see Chapter **20**)).

In ETP, "updates" to a Channel data object DO NOT include updates to the channel data. Updates to the channel data is done using the ETP streaming protocols.

Additionally, ChannelDataFrame (Protocol 2) makes it possible to view several channels in a set of channels as a table or frame of data. For more information, about how channels may be organized together, see Section **7.1.1**.

The ETP channel streaming protocols do not define any specific meaning or behaviors to specific URIs. Defining meaning is the responsibility of the Energistics domain standards.

NOTE: In WITSML v1.4.1.1 terms, a channel is comparable to a log curve; more generally, it is comparable to a tag in a process historian. In WITSML v2.0, the v1.4.1.1 constructs (such as log curve) have been redesigned similarly to ETP as channels, which can be grouped into channel sets and logs. However, ETP channel streaming protocols handle individual channels; that is, whether or not the channel is part of a channel set or log is irrelevant to how it is handled in a channel streaming protocol.

6.1.1.1 About Indexes and Channel Data

ETP defines several channel index kinds (for the list and definitions, see Section **23.33.2**); which includes several variations for time and depth (e.g., dateTime, elapsed time, MD, TVD, etc.) to support oil and gas workflows and to be consistent with the Energistics domain standards.

Additionally, ETP supports use of secondary indexes, which provide additional context for the associated values. A secondary index is one or more indexes in addition to the primary one. In ETP, the primary index is always the first index. For example, if the primary index of a channel is time, the secondary index may be depth. When channel data is ordered by the primary index, secondary index values may or may not be ordered. For example, when channel data is ordered by time as the primary index, a secondary index of hole depth may be ordered while a secondary index of bit depth would not be ordered. On endpoints that support it, secondary indexes may also be used for filtering on range and frame operations.

NOTES:

1. Support for secondary index is optional and is considered advanced query functionality that most ETP servers will not support.
2. If a store supports write operations, it MUST support at least 1 secondary index. That is, a store's value for the MaxSecondaryIndexCount data object capability MUST be at least 1.

6.1.2 Metadata for Channels, Indexes and Attributes

To make channel streaming more efficient, ETP is designed so that endpoints can exchange relevant metadata about channels once, at the beginning of the ETP session. This approach allows each endpoint to use the metadata to "set up" (e.g., for the receiving endpoint to interpret and understand what channels it will be receiving, relevant units of measure (UOM), etc.), and then as new data points are produced, the sending endpoint simply streams the new data, which at a minimum is typically the latest index and value at that index.

The main types of metadata include those listed here:

- **Channel metadata** is exchanged in the ***ChannelMetadataRecord*** (see Section **23.33.7**), which is a standard ETP structure (Avro record) that contains the metadata for one channel. Various messages in the channel streaming protocols—for example: ***ChannelMetadata*** message in ChannelStreaming (Protocol 1); ***GetChannelMetadataResponse*** message in ChannelSubscribe (Protocol 21); and the ***OpenChannelsResponse*** message in ChannelDataLoad (Protocol 22)—send one ***ChannelMetadataRecord*** record per channel.
- **Index metadata** is the metadata about the index(es) in one channel, which includes information such as the index kind (time, depth, scalar or elapsed time) and direction (increasing or decreasing). The ***IndexMetadataRecord*** (Section **23.33.6**) is an ETP datatype (Avro record) sent in the ***indexes*** field of the ***ChannelMetadataRecord***.
- **Attribute metadata.** ETP provides an Avro record (***DataAttribute***, Section **23.23**) that allows an endpoint to pass attributes associated with individual channel data points (sometimes referred to as "decorating" individual points). These attributes are typically metadata for things such as quality, confidence, audit information, etc. **NOTE:** ETP simply provides the structure for passing such data. ETP does NOT specify the content and usage, which may be specified by individual MLs (in the relevant implementation specification) or may be custom.

  Consistent with the established pattern (for channels and indexes), ETP defines attribute metadata in the ***AttributeMetadataRecord*** record (Section **23.24**), which is the information needed to interpret/understand the data attributes that may be sent in an ETP session.

6.1.3 What Data is Sent When Streaming Channels

As explained above, in its most basic form streaming channel data is sending an index and a data value at that index, which are sent in ***ChannelData*** messages (for all 3 channel streaming protocols). There are more options though to cover all scenarios and to help reduce message size on the wire.

NOTE: For streaming data, ETP does NOT send null data values. However, if a new data value has the same index as the previous data value, then the index MAY be null, which indicates it is the same as the previous index. **EXCEPTION:** If channel data values are arrays, then the arrays MAY contain null values, but at least one array value MUST be non-null and the entire array CANNOT be null.

The ***ChannelData*** message uses the ETP datatype ***DataItem*** (Section **23.33.5**), which specifies these fields:

- *channelId*: the identifier of the channel for this point, as received in a ***ChannelMetadataRecord***.
- *indexes*: the value of the index(es) for the data value.
- *value*: the value of the data point. This field must be one of the types specified in the ETP datatype ***DataValue*** (Section **23.30**)—which includes options to send a single data value (of various types such as integers, longs, doubles, etc.) OR arrays of values. For more information about options for sending arrays, see Section **6.1.3.1 below**.
- *valueAttributes*: any qualifiers, such as quality, accuracy, etc., attached to this data point.

6.1.3.1 Sending an Array in a Data Value

When sending an array as a data point value, the data should be encoded as a 1D array and additional information must be provided so that the receiver endpoint can reconstruct the 1D array into its original dimensions.

The information for reconstructing the array is specified in the *axisVectorLengths* field on the ***ChannelMetadataRecord***, by encoding the positional information (as an absolute 'start' offset) between it and the length of each subarray that Avro will encode onto the wire. For more information see, Section **23.33.7**.

In some situations, it's possible that individual values within an array of data could be null. ETP offers these approaches for specifying null values in an array:

- Sparse arrays. Using *axisVectorLengths*, specify the number of 'skips' (which indicate null values) in addition to the 'start' offsets.
- For arrays of Int, Long or Boolean, ETP specifies a corresponding nullable type (i.e., ArrayOfNullableInt, ArrayOfNullableLong, ArrayOfNullableBoolean).
- For arrays of double or float values, use "NaN" to specify null values.

You must observe these rules when specifying null values in an array:

1. The base type specified in ***ChannelMetadataRecord*** for the array type must be the base, non-nullable array type (see ArrayofLong example in next item).
2. The underlying type must be consistent for the same channel or other usage. i.e., if you start sending ArrayofLong, you must only use ArrayofLong, ArrayofLongNullable, SparseArray with Long.

6.1.3.2 Reducing Channel Data Message Size on the Wire

Sending every data point in its own ETP message introduces a sizeable overhead on the wire. When sending channel data, following some best practices and taking advantage of some ETP features for channel data will help minimize the overhead.

Implementers should follow these best practices when sending channel data using the ***ChannelData***, ***GetRangesResponse***, and ***ReplaceRange*** messages:

1. Send values from related data channels together and in index order rather than channel-by-channel. **EXAMPLE:** If a single piece of equipment produces measurement values for 5 channels, then send values for all 5 channels together for one index followed by values for all 5 channels for the next index and so on.
2. Include as many ***DataItem*** records within a single message as possible without introducing unnecessary latency. That is, send as much available data as possible in every message without pausing to allow new data to accumulate and sending it in batches.
3. Group together ***DataItem*** records within a message, when they have the same primary and secondary indexes, and order them within each group by index values. That is, order data within groups by “row” rather than by “column” or on a channel-by-channel or basis.
4. When a ***DataItem***’s indexes and index values are the same as the previous ***DataItem*** record in the message, set the *indexes* field in ***DataItem*** to a zero length (empty) array.

6.1.4 "Simple Streamer" vs. "Standard Streamer"

Typically, in oil and gas operations we think of technology that streams data (so-called data "streamers") in two broad categories:

- A "simple streamer" refers to a basic device such as a sensor, that does nothing more than send data; by definition, it does not maintain history, and the receiver does NOT have any control over what the simple streamer sends.
- A "standard streamer" refers to a client or server application with capabilities beyond the simple streamer, such as maintaining history changes or supporting requests for specific ranges of data in a channel.

Protocol 1 supports behavior for a simple streamer only; for options for reading and writing data with standard streamers, see ChannelSubscribe (Protocol 21) and ChannelDataLoad (Protocol 22).

IMPORTANT: When eventual consistency is important with channel data (i.e., when an endpoint needs to ensure it gets all channel data—including corrections to previously streamed data—even if the connection is occasionally dropped), then you MUST use the protocols for standard streaming, which are ChannelSubscribe (Protocol 21) and ChannelDataLoad (Protocol 22). The simple streaming protocol (ChannelStreaming (Protocol 1)) does not support eventual consistency.

6.1.5 Organizing Channels into ChannelSets and Logs

While Energistics data models (e.g., WITSML) allow users to organize channels into channel sets and logs, a channel is an Energistics data object that is acted upon independently. The ETP channel streaming protocols ((Channel Streaming (Protocol 1), ChannelSubscribe (Protocol 21) and ChannelDataLoad (Protocol 22)) only operate on Channels, and whether or not a Channel is included in a ChannelSet or Log does not affect the behavior defined by these protocols. In contrast with this, ChannelDataFrame (Protocol 2) only supports reading data from the Channels within a ChannelSet. For more information about channel sets and logs, see Section **7.1.1**. For the definition of data object, see Section **25.1**.

6.2 ChannelStreaming: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, and identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the task, including error scenarios and resulting error codes.
- **General Requirements.** Identifies high-level and/or protocol-wide general behavior that must be observed (in addition to behavior specified in Message Sequence), including usage of protocol and endpoint capabilities.
- **Capabilities.** Lists and defines the parameters that set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**
- Contextual information (such as what wellbore/well the data is for) most likely will come out of band of ETP because simple streamers don't usually "know" these things and have no functionality to discover them.

6.2.1 ChannelStreaming: Message Sequence

This section explains the basic message sequence for the main task to be done using this protocol and includes related key behaviors and possible errors. The following Required Behavior section provides additional requirements and rules for how this protocol works.

The following table lists all messages defined in this sub-protocol, and the basic request/response usage patterns per ETP role. The detailed content of each message is explained in the Message Schema section.

ChannelStreaming (Protocol 1): Basic Message-Response flow by ETP Role	
Messages sent by Consumer	**Response/Message sent by Producer**
StartStreaming: Indicates that consumer is ready to receive data.	***ChannelMetatdata***: Indicates the channel(s) it may send data for and metadata for each.
	ChannelData: Contains the data the producer has for each channel; the producer keeps sending these messages as new data becomes available for as long as the consumer is connected.

ChannelStreaming (Protocol 1): Basic Message-Response flow by ETP Role	
Messages sent by Consumer	**Response/Message sent by Producer**
	TruncateChannels: Sent to reset the end index of a channel to allow streaming to resume from the new end index; used to correct "index jump" errors in previously sent data.
StopStreaming: Indicates that the consumer wants the producer to stop sending data.	

6.2.1.1 Main Message Sequence for Simple Streamers

Because this protocol is for so-called simple streamers, only one message sequence is required/described. This section describes the basic message sequence, related key behaviors, and possible errors.

1. The consumer sends a ***StartStreaming*** message (Section **6.3.1**) to the producer. The ***StartStreaming*** message indicates that the consumer is ready to receive data.
2. On receipt of the StartStreaming message, the producer MUST send at least one ***ChannelMetadata*** message (Section **6.3.3**), which indicates the channels it will stream.
    a. ***ChannelMetadata*** includes ***ChannelMetadataRecord*** records, which have the necessary contextual information (indexes, units of measure, etc.) that the customer needs to correctly interpret channel data.
    b. For many producers, this ***ChannelMetadata*** message MAY BE the only such message sent. However, if additional channels appear on the producer over time, the producer MUST send additional ***ChannelMetadata*** messages (for the new channels) before sending any data for the new channel.
    c. The producer MUST assign the channel an integer identifier that is unique for the session in this protocol. This identifier will be used instead of the channel URI to identify the channel in subsequent messages in this protocol for the session. This identifier is set in the *id* field in the ***ChannelMetadataRecord***. **RECOMMENDATION:** Use the smallest available integer value for a new channel identifier.
    IMPORTANT: If the channel is deleted and recreated during a session, it MUST be assigned a new identifier.
    d. A simple streamer always streams all its available channels. That is, it does NOT accept requests to stream individual channels.
    e. The producer MUST NOT send any data until it receives the ***StartStreaming*** message from the consumer.
    f. If the producer cannot handle the request, the producer MUST deny the request and MUST send error EREQUEST_DENIED (6).
3. After it sends the ***ChannelMetadata*** message, the producer MUST begin streaming ***ChannelData*** messages (see Section **6.3.4**) and MUST continue streaming new data points as long as the consumer is connected or until the consumer sends the ***StopStreaming*** message.
    a. A good assumption is that the producer begins streaming data from its "current" index, though there are no guarantees.
    b. When it receives a ***StartStreaming*** message, a producer may not have any channel data to send. In this scenario, the producer will not immediately send any ***ChannelData*** messages, but, when data becomes available, it MUST start sending ***ChannelData*** messages.
4. As needed, the producer MAY send ***TruncateChannels*** messages (e.g., to correct erroneous "index jumps") (Section **6.3.5**). When a TruncateChannels message is sent:

a. The consumer MUST:

  i. Reset its end index to the *newEndIndex* specified in the ***TruncateChannels*** message, and

  ii. Delete or disregard any data it previously received that was after the *newEndIndex.*

b. When the producer resumes sending ***ChannelData*** messages (for the channel whose index it corrected), the index for each new ***DataItem*** record MUST be greater than the new end index in ***TruncateChannels*** message.

5. If new channels appear on the producer, it MUST send the metadata for the new channels before sending any data for the channels.

   a. To do so, the producer sends additional ***ChannelMetadata*** messages, which MUST include a ***ChannelMetadataRecord*** for each new channel and MAY include a ***ChannelMetadataRecord*** for ALL channels it is currently streaming.

   b. The producer MAY then send ***ChannelData*** messages for the new channels.

6. To stop streaming, a consumer MAY send a ***StopStreaming*** message (Section **6.3.2**) to the producer or simply disconnect.

   a. Upon receipt of the **StopStreaming** message, the producer MUST stop sending ***ChannelData*** messages.

6.2.2 ChannelStreaming: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1), rows for general requirements for this protocol, and (possibly) some rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that must be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**.
2.	No Session Survivability	1. If a session is interrupted, a consumer MUST re-establish the session (as described in Section **5.1**) and restart ChannelStreaming (Protocol 1) (as described in Section **6.2.1.1**). 2. After the consumer reconnects, a simple streamer is NOT required to send data that would have been sent while the consumer was disconnected.
3.	Data order for streaming data	1. Streaming data points (in the ***ChannelMetadata*** messages) MUST be sent in primary index order for each channel, both within one message and across multiple messages. a. Primary index order is always as appropriate for the index direction of the channel (i.e., increasing or decreasing).

Row#	Requirement	Behavior
		2. The index values for each data point are in the same order as their corresponding ***IndexMetadataRecord*** records in the corresponding channel's ***ChannelMetadataRecord*** record, and the primary index is always first.
4.	Index metadata	1. A channel data object's index metadata MUST be consistent: a. The index units and vertical datums MUST match the channel's index metadata. 2. When sending messages, the producer MUST ensure that all index metadata and data derived from index metadata are consistent in all fields in the message, including in XML or JSON object data or part data. a. **EXAMPLE:** The *uom* and *depthDatum* in an ***IndexInterval*** record MUST be consistent with the channel's index metadata.
5.	No null values sent	1. In streaming data, ETP does NOT send null values.

6.2.3 ChannelStreaming: Capabilities

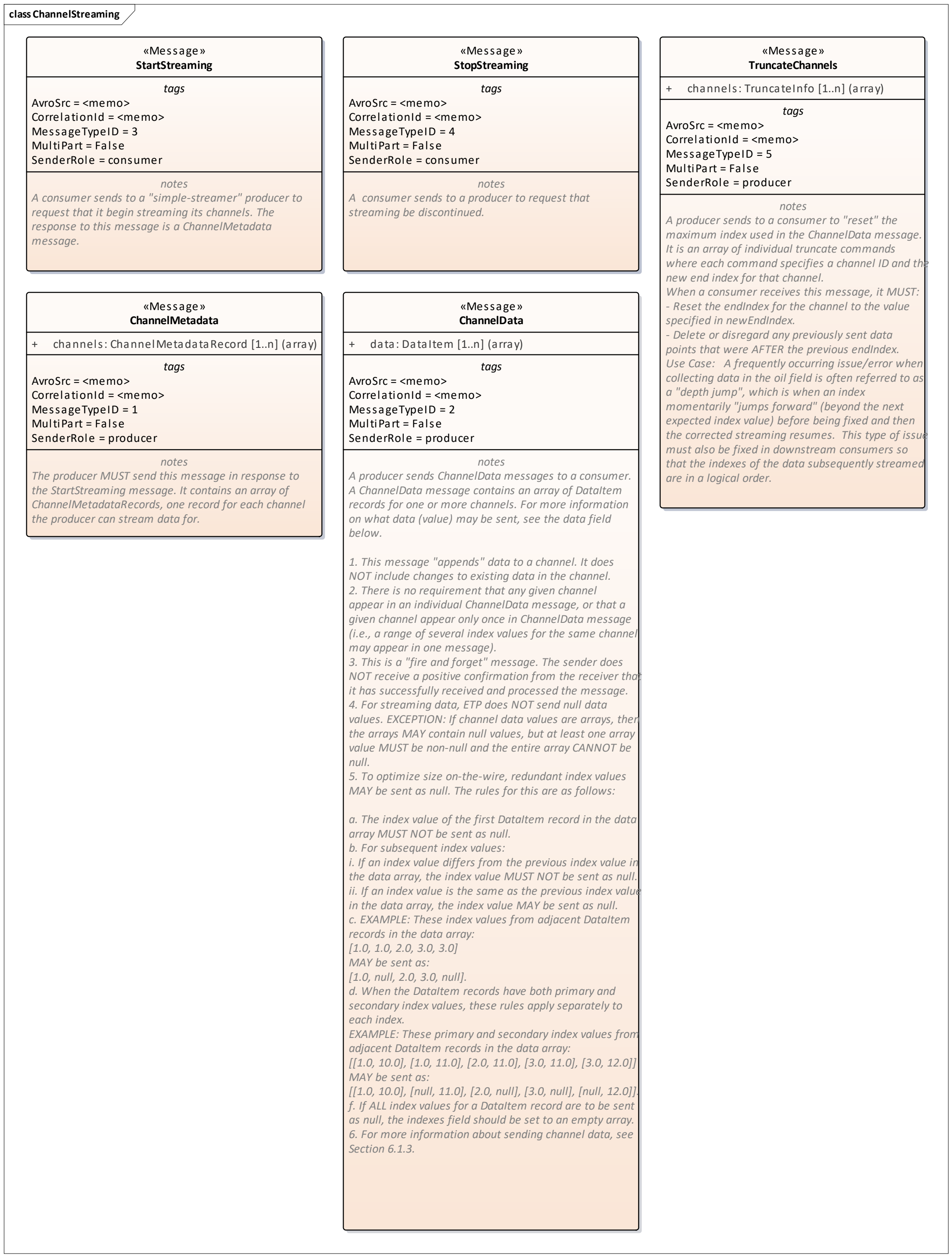
Intentionally designed for "simple streamers", ChannelStreaming (Protocol 1) has no protocol capabilities, only the endpoint capabilities shared by most of the ETP sub-protocols.

- For general information about the types of capabilities and how they may be used, see Section 3.3.

ChannelStreaming (Protocol 1): Capabilities			
Name: Description	**Type**	**Units** **Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section 3.3.2. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP** Server.			

6.3 ChannelStreaming: Message Schemas

This section provides a figure that displays all messages defined in ChannelStreaming (Protocol 1). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.



«Message»
StartStreaming

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 3
MultiPart = False
SenderRole = consumer

notes
A consumer sends to a "simple-streamer" producer to request that it begin streaming its channels. The response to this message is a ChannelMetadata message.

«Message»
StopStreaming

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 4
MultiPart = False
SenderRole = consumer

notes
A consumer sends to a producer to request that streaming be discontinued.

«Message»
TruncateChannels

+ channels: TruncateInfo [1..n] (array)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 5
MultiPart = False
SenderRole = producer

notes
A producer sends to a consumer to "reset" the maximum index used in the ChannelData message. It is an array of individual truncate commands where each command specifies a channel ID and th new end index for that channel.
When a consumer receives this message, it MUST:
- Reset the endIndex for the channel to the value specified in newEndIndex.
- Delete or disregard any previously sent data points that were AFTER the previous endIndex.
Use Case: A frequently occurring issue/error when collecting data in the oil field is often referred to as a "depth jump", which is when an index momentarily "jumps forward" (beyond the next expected index value) before being fixed and then the corrected streaming resumes. This type of issu must also be fixed in downstream consumers so that the indexes of the data subsequently streamed are in a logical order.

«Message»
ChannelMetadata

+ channels: ChannelMetadataRecord [1..n] (array)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1
MultiPart = False
SenderRole = producer

notes
The producer MUST send this message in response to the StartStreaming message. It contains an array of ChannelMetadataRecords, one record for each channel the producer can stream data for.

«Message»
ChannelData

+ data: DataItem [1..n] (array)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 2
MultiPart = False
SenderRole = producer

notes
A producer sends ChannelData messages to a consumer. A ChannelData message contains an array of DataItem records for one or more channels. For more information on what data (value) may be sent, see the data field below.

1. This message "appends" data to a channel. It does NOT include changes to existing data in the channel.
2. There is no requirement that any given channel appear in an individual ChannelData message, or that a given channel appear only once in ChannelData message (i.e., a range of several index values for the same channel may appear in one message).
3. This is a "fire and forget" message. The sender does NOT receive a positive confirmation from the receiver tha it has successfully received and processed the message.
4. For streaming data, ETP does NOT send null data values. EXCEPTION: If channel data values are arrays, ther the arrays MAY contain null values, but at least one array value MUST be non-null and the entire array CANNOT be null.
5. To optimize size on-the-wire, redundant index values MAY be sent as null. The rules for this are as follows:

a. The index value of the first DataItem record in the data array MUST NOT be sent as null.
b. For subsequent index values:
i. If an index value differs from the previous index value in the data array, the index value MUST NOT be sent as null.
ii. If an index value is the same as the previous index valu in the data array, the index value MAY be sent as null.
c. EXAMPLE: These index values from adjacent DataItem records in the data array:
[1.0, 1.0, 2.0, 3.0, 3.0]
MAY be sent as:
[1.0, null, 2.0, 3.0, null].
d. When the DataItem records have both primary and secondary index values, these rules apply separately to each index.
EXAMPLE: These primary and secondary index values from adjacent DataItem records in the data array:
[[1.0, 10.0], [1.0, 11.0], [2.0, 11.0], [3.0, 11.0], [3.0, 12.0]]
MAY be sent as:
[[1.0, 10.0], [null, 11.0], [2.0, null], [3.0, null], [null, 12.0]].
f. If ALL index values for a DataItem record are to be sent as null, the indexes field should be set to an empty array.
6. For more information about sending channel data, see Section 6.1.3.

Figure 11: ChannelStreaming: message schemas

6.3.1 Message: StartStreaming

A consumer sends to a "simple streamer" producer to request that it begin streaming its channels. The response to this message is a ChannelMetadata message.

Message Type ID: 3

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: consumer

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelStreaming",
     "name": "StartStreaming",
     "protocol": "1",
     "messageType": "3",
     "senderRole": "consumer",
     "protocolRoles": "producer,consumer",
     "multipartFlag": false,

     "fields":
     [

     ]
}
```

6.3.2 Message: StopStreaming

A consumer sends to a producer to request that streaming be discontinued.

Message Type ID: 4

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: consumer

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelStreaming",
     "name": "StopStreaming",
     "protocol": "1",
     "messageType": "4",
     "senderRole": "consumer",
     "protocolRoles": "producer,consumer",
     "multipartFlag": false,

     "fields":
     [

     ]
}
```

6.3.3 Message: ChannelMetadata

The producer MUST send this message in response to the StartStreaming message. It contains an array of ChannelMetadataRecords, one record for each channel the producer can stream data for.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: producer

Field Name	Description	Data Type	Min	Max
channels	The list of channels with metadata for each; the fields for each channel are defined in the ChannelMetadataRecord. The URIs MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.	ChannelMetadataRecord	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelStreaming",
     "name": "ChannelMetadata",
     "protocol": "1",
     "messageType": "1",
     "senderRole": "producer",
     "protocolRoles": "producer,consumer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "channels",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.ChannelMetadataRecord" }
         }
     ]
}
```

6.3.4 Message: ChannelData

A producer sends ChannelData messages to a consumer.

A ***ChannelData*** message contains an array of DataItem records for one or more channels. For more information on what data (value) may be sent, see the *data* field below.

1. This message "appends" data to a channel. It does NOT include changes to existing data in the channel.
2. There is no requirement that any given channel appear in an individual ***ChannelData*** message, or that a given channel appear only once in ***ChannelData*** message (i.e., a range of several index values for the same channel may appear in one message).
3. This is a "fire and forget" message. The sender does NOT receive a positive confirmation from the receiver that it has successfully received and processed the message.
4. For streaming data, ETP does NOT send null data values. **EXCEPTION:** If channel data values are arrays, then the arrays MAY contain null values, but at least one array value MUST be non-null and the entire array CANNOT be null.
5. The index values in each DataValue record are in the same order as their corresponding IndexMetadataRecord records in the corresponding channel's ChannelMetadataRecord record, and the primary index is always first.
6. To optimize size on-the-wire, redundant index values MAY be sent as null. The rules for this are as follows:

   a. The index value of the first ***DataItem*** record in the *data* array MUST NOT be sent as null.

   b. For subsequent index values:

      i. If an index value differs from the previous index value in the *data* array, the index value MUST NOT be sent as null.

ii. If an index value is the same as the previous index value in the *data* array, the index value MAY be sent as null.

c. **EXAMPLE:** These index values from adjacent ***DataItem*** records in the *data* array:

[1.0, 1.0, 2.0, 3.0, 3.0]

MAY be sent as:

[1.0, null, 2.0, 3.0, null].

d. When the ***DataItem*** records have both primary and secondary index values, these rules apply separately to each index.

e. EXAMPLE: These primary and secondary index values from adjacent ***DataItem*** records in the *data* array:

[[1.0, 10.0], [1.0, 11.0], [2.0, 11.0], [3.0, 11.0], [3.0, 12.0]]

MAY be sent as:

[[1.0, 10.0], [null, 11.0], [2.0, null], [3.0, null], [null, 12.0]].

f. If ALL index values for a ***DataItem*** record are to be sent as null, the *indexes* field should be set to an empty array.

6. For more information about sending channel data, see Section **6.1.3**.

Message Type ID: 2

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: producer

Field Name	Description	Data Type	Min	Max
data	Contains the data points for channels, which is an array of DataItem records. Note that the value must be one of the types specified in ***DataValue*** (Section **23.30**)—which include options to send a single data value (of various types such as integers, longs, doubles, etc.) OR arrays of values. For more information, see Section **6.1.3**.	DataItem	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelStreaming",
     "name": "ChannelData",
     "protocol": "1",
     "messageType": "2",
     "senderRole": "producer",
     "protocolRoles": "producer,consumer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "data",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.DataItem" }
         }
     ]
}
```

6.3.5 Message: TruncateChannels

A producer sends to a consumer to "reset" the maximum index used in the ***ChannelData*** message. It is an array of individual truncate commands where each command specifies a channel ID and the new end index for that channel.

When a consumer receives this message, it MUST:

- Reset the endIndex for the channel to the value specified in *newEndIndex*.
- Delete or disregard any previously sent data points that were AFTER the previous endIndex.

Use Case: A frequently occurring issue/error when collecting data in the oil field is often referred to as a "depth jump", which is when an index momentarily "jumps forward" (beyond the next expected index value) before being fixed and then the corrected streaming resumes. This type of issue must also be fixed in downstream consumers so that the indexes of the data subsequently streamed are in a logical order.

Message Type ID: 5

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: producer

Field Name	Description	Data Type	Min	Max
channels	Contains an array of TruncateInfo structures, which each indicate the channel ID and its new end index.	TruncateInfo	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelStreaming",
     "name": "TruncateChannels",
     "protocol": "1",
     "messageType": "5",
     "senderRole": "producer",
     "protocolRoles": "producer,consumer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "channels",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.TruncateInfo" }
         }
     ]
}
```

7 ChannelDataFrame (Protocol 2)

ProtocolID: 2

Defined Roles: store, customer

A customer uses ChannelDataFrame (Protocol 2) to get channel data from a store in a row-orientated 'frame' or 'table' of data. In oil and gas jargon, the general use case that Protocol 2 supports is typically referred to as getting a "historical log". (In ETP jargon you are actually getting a frame of data from a ChannelSet data object; for more information, see Section **7.1.1**).

With this protocol, a customer endpoint gets rows of data, where one row consists of a primary index value, all associated secondary index values from the ChannelSet's secondary indexes, and all associated data and attribute values from the ChannelSet's channels. Being able to retrieve data in a frame simplifies logic for customer role software applications when dealing with data as a "log" rather than individual channels.

This protocol supports several use cases, including:

- Exporting data from a store to an on-disk representation of a log.
- Querying "aligned" data for calculations like mechanical specific energy (MSE).

NOTES:

1. Protocol 2 supports get/read functionality only. To put/write channel data for individual channels, use ChannelDataLoad (Protocol 22) (see Chapter **20**); you CANNOT put "rows" in a channel set.
2. This protocol SHOULD NOT be used to poll for realtime data. Instead use ChannelSubscribe (Protocol 21) (see Chapter **19**) or for "simple streamers" use ChannelStreaming (Protocol 1) (see Chapter **6**).
3. ChannelDataFrame (Protocol 2) allows stores to introduce a delay between when they receive new channel data and when they make the data available for consumption using Protocol 2. This delay is intended to help ensure customers receive "complete" rows of data from a store because new data for channels in a channel set may arrive in the store at different times from different sources.

This chapter includes main sections for:

- Key ETP concepts that are important to understanding how this protocol is intended to work (see Section **7.1**).
- Required behavior, which includes:
  - Description of the message sequence for main tasks, along with required behavior, endpoint, data object and protocol capabilities usage, and possible errors (see Section **7.2.1**).
  - Other functional requirements (not covered in the message sequence) including use of additional endpoint, data object, and protocol capabilities for preventing and protecting against aberrant behavior (see Section **7.2.2**).
  - Definitions of the endpoint and protocol capabilities used in this protocol (see Section **7.2.3**).
- Sample schemas of the messages defined in this protocol (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes documentation for each field in a schema (see Section **7.3**).

7.1 ChannelDataFrame: Concepts

This section explains concepts that are important to understanding how ChannelDataFrame (Protocol 2) works.

7.1.1 Channel, Channel Set, Log and Frame

This section defines the concepts of **channel**, **channel set**, **log** and **frame**. The first 3 objects are also Energistics data objects (defined in ETP and/or WITSML)—Channel, ChannelSet and Log—which are named objects in a store, each uniquely identified by a URI.

- A **channel** is a series of values, usually measured or calculated, that are referenced to one or more indexes, usually time or depth. Channel data is a tuple of index values, data values, and attribute values; ETP supports that a "data value" may actually be an array of data values. A channel can be thought of as a "column" of data points. For more detailed definition of channel, see Section **6.1.1**.
- A **channel set** is an Energistics construct and is used to group channels with a compatible primary index for some purpose; users of a system compose channel sets. A channel set may also include secondary indexes, when all channels in the channel set also have compatible secondary indexes. 'Compatible' simply means that all of the channels in the set have the same kind of index (e.g., all time or all depth, or others as defined in ChannelIndexKind, see Section **23.33.2**, are all primary or all secondary, all have the same unit, all have the same direction, and use a common datum (for depth indexes). ***ChannelDataFrame (Protocol 2) acts on ETP ChannelSets; use Protocol 2 to get data from multiple channels in a pre-specified channel set at a given index value, as a "row"; consecutive rows constitute the frame.***
- In ETP, a **log** is a container for one or more **channel sets**, where the channel sets are not required to have compatible indexes. That is, it is a set of sets that may have different index types. A log is represented by the Log data object; Logs are composed by users. Note that the ETP approach differs from the traditional definition (e.g., in WITSML v1.x) where a log is analogous to a channel set. ***Because an ETP Log is a "set of sets" that may have different types of indexes, Protocol 2 works only on ETP ChannelSets.***

7.1.2 Support for Secondary Indexes

ETP provides support for secondary indexes both on channels and channel sets. As stated in Section **6.1.1.1**, support for secondary indexes is optional, advanced functionality, but if a store supports write operations, it MUST support at least one secondary index on a channel or channel set data object.

The ***IndexMetadataRecord*** record (Section **23.33.6**) is used to provide important metadata about both primary and secondary indexes. When used for secondary indexes:

- The *direction* field MUST be populated based on the order that secondary index values will appear when data is ordered by the primary index. For example, if a secondary index's values will be monotonically increasing when *direction* for the primary index is Decreasing, *direction* for the secondary index would be Increasing. If secondary index values are unordered when data is ordered by the primary index, the *direction* field for the secondary index MUST be se to Unordered.
- The optional *filterable* field MAY be set to true, which allows an endpoint to specify if a particular secondary index can be filtered on in various request messages in some ETP sub-protocols.

If an index is filterable, then the customer endpoint can use it to filter on ***GetFrame*** requests in this sub-protocol.

The ***GetFrame*** request includes these fields for using secondary indexes:

- *includeAllChannelSecondaryIndexes*. Flag that allows the customer to request that secondary indexes from Channel data objects that are NOT secondary indexes in the containing ChannelSet data object be included in the response.

  NOTE: Some of these secondary indexes from Channel data objects may have null values.

- *requestedSecondaryIntervals.* (Optional) For channels/channel sets that indicate an index is "filterable" (i.e., the *filterable* field on the ***IndexMetadataRecord*** is set to true) this option allows the customer to request that results be filtered on secondary indexes.

  This is an array of the secondary intervals (as defined in IndexInterval) on which the customer wants to filter.

EXAMPLE: If your primary interval is time, this field could be a depth interval on which filtering is being requested.

7.2 ChannelDataFrame: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.
- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**
- Customer must have the URIs of the ChannelSet data objects it's interested in; these URIs are typically found using Discovery (Protocol 3) (Chapter 8). (They may also come "out of band" of ETP, for example, by email.)
  - If the ChannelSet data object for the desired set of channels does not exist, the customer must create it using Store (Protocol 4).

7.2.1 ChannelDataFrame: Message Sequences

This section explains the basic message sequence for main tasks to be done using this protocol and includes related key behaviors and possible errors. The following General Requirements section provides additional requirements and rules for how this protocol works.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section.

ChannelDataFrame (Protocol 2): Basic Message-Response flow by ETP Role	
Message (sent by customer)	**Response Message** (sent by store)
GetFrameMetadata (optional): Request to get in ETP format the list of indexes and channels that compose a frame (ChannelSet).	***GetFrameMetadataResponse*** (multipart): The relevant list of indexes and channels that define the frame for the entire ChannelSet.
GetFrame: Request for a frame of data from an existing channel set.	***GetFrameResponseHeader*** (multipart): The FIRST response to the GetFrame message; it specifies the channel IDs (the "column headings" or position in a row) for the subsequent rows of data that the store will return in one or more GetFrameResponseRows messages. ***GetFrameResponseRows*** (multipart): Contains a "frame", which is an array of one or more rows of data whose "column headings" are specified in the corresponding GetFrameResponseHeader message.
CancelGetFrame: Request to stop sending data for a previous GetFrame request.	A final ***GetFrameResponseRows*** message with the FIN bit set.

7.2.1.1 To get row-oriented frames of data

1. Optionally, a customer sends a store a ***GetFrameMetadata*** message (Section **7.3.5**) with the URI of the channel set to get frame metadata for.
   a. **Purpose of this message:** to get in ETP format the list of indexes and channels that compose the ChannelSet from which the customer wants to get a frame.
   b. The message is optional because the customer may already have the information needed to retrieve a frame, e.g., out-of-band of the ETP session.
2. If the store successfully returns frame metadata for the channel set URI, it MUST send one or more ***GetFrameMetadataResponse*** messages (Section **7.3.6**), each of which contains arrays of index and/or frame channel metadata.
   a. **Purpose of this message:** Provides the relevant list of indexes and channels that define the frame for the entire ChannelSet data object.
3. If the store does NOT successfully return frame metadata, it MUST send a non-map ***ProtocolException*** message with an appropriate error, such as EREQUEST_DENIED (6).
4. The customer sends a store the ***GetFrame*** message (Section **7.3.1**).
   a. **Purpose of this message:** The request for the store to send row-oriented 'frames' (tables) of data for an existing ChannelSet data object.
   b. The ***GetFrame*** message provides options for filtering data including support for secondary indexes. For more information, see Section **7.1.2**.
5. If the store has no data that fulfills the request, it MUST respond with one of these:
   a. An "empty" ***GetFrameResponseHeader*** message, with the FIN bit set.
   b. A ***GetFrameResponseHeader*** message and a ***GetFrameResponseRows*** message with the *frame* field set to an empty array and the FIN bit set.
6. If the store returns frame data for the request, it MUST:
   a. Respond FIRST by sending a ***GetFrameResponseHeader*** message (Section **7.3.2**), which specifies the URIs of each Channel in the ChannelSet in the order that data points will be returned in each subsequent row of data (sent in one or more ***GetFrameResponseRows*** messages) and the index metadata for the index values in the frame, in the order that the index values will be returned in each row of data.
   b. Subsequently send ***GetFrameResponseRows*** messages (Section **7.3.3)** for the rows of data.
      i. The store MUST include any new or changed channel data in responses to ***GetFrame*** message requests no later than the store's value for the FrameChangeDetectionPeriod endpoint capability, after the data is changed or added. (In other words, updates to channels are not guaranteed to be visible in responses in less than this time.)
      ii. The store MUST limit the total count of rows returned in response to a request to the customer's value for the MaxFrameResponseRowCount protocol capability.
         1. The customer MAY notify the store of responses that exceed this limit by sending error ERESPONSECOUNT_EXCEEDED (30).
         2. If the store's value for MaxFrameResponseRowCount protocol capability is smaller than the customer's value, then the store MAY further limit the total count of rows to its value for MaxFrameResponseRowCount protocol capability.
      iii. If the store cannot send all rows in response to the request because it would exceed the lower of the customer's or the store's MaxFrameResponseRowCount value, it MUST send error ERESPONSECOUNT_EXCEEDED (30) to terminate the response.
         1. The store MUST NOT send error ERESPONSECOUNT_EXCEEDED (30) until it has sent

the maximum allowed rows in response to the customer's request.

c. The multipart response to one ***GetFrame*** request is composed of the ***GetFrameResponseHeader*** and all subsequent ***GetFrameResponseRows*** messages and, if an error occurs, a ***ProtocolException*** message.

   i. The "FIN bit" (0x02 flag in the message header) MUST be set on the last message of the multipart response; that last message may be either a ***GetFrameResponsesRows*** message or a ***ProtocolException*** message.

   ii. For more information on required behavior for multipart responses and requests, see Section **3.7.3.1**.

7. If the store does NOT successfully return frame data or an empty positive response, it MUST send a non-map ***ProtocolException*** message with an appropriate error, such as EREQUEST_DENIED (6).

   a. If in the ***GetFrame*** message, the customer requests that data be filtered by secondary index values and the store's value for SupportsSecondaryIndexFiltering protocol capability is false or the *filterable* field on the ***IndexMetadataRecord*** record for the secondary index is false, the store MUST deny the request and send error ENOTSUPPORTED (7).

7.2.1.2 To cancel a GetFrame operation

1. The customer MUST send to the store a ***CancelGetFrame*** message (Section **7.3.4**), which includes the request UUID (*requestUuid*) of the ***GetFrame*** operation to be cancelled.
2. If the store has not already finished responding to the request that is being canceled, the store MUST:

   a. Send a final ***GetFrameResponsesRows*** message with the FIN bit set; this final message MAY be empty (i.e. have the *frame* field set to an empty array).

   b. Stop sending ***GetFrameResponsesRows*** messages for the specified request operation.

7.2.2 ChannelDataFrame: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) some rows with additional requirements for specific types of operations.

Row #	Requirement	Behavior
1.	ETP-wide behavior that must be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.**

Row #	Requirement	Behavior
		a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter **5**. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant ETP-defined endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3**. b. For the list of global capabilities and related behavior, see Section **3.3.2**. 3. The capabilities listed in Section **7.2.3** MUST BE used in this ETP sub-protocol. Additional details for how to use the protocol capabilities are included below in this table and in Section **7.2.1 ChannelDataFrame: Message Sequence.**
3.	Message Sequence See Section **7.2.1**.	1. The Message Sequence section above (Section **7.2.1**) describes requirements for the main tasks listed there and also defines required behavior.
4.	Plural Messages (which includes maps)	1. This protocol uses plural messages. For detailed rules on handling plural messages (including ProtocolException handling), see Section **3.7.3**.
5.	Do NOT use this protocol for realtime streaming	1. This protocol SHOULD NOT be used to poll for realtime data. Instead use ChannelSubscribe (Protocol 21) (see Chapter **19**) or for "simple streamers" use ChannelStreaming (Protocol 1) (see Chapter **6**).
6.	Channel order in messages	1. The order of the channels in each row of the ***GetFrameResponsesRows*** message MUST be the same as the order specified in the ***GetFrameResponseHeader*** message. 2. The position (array index) of each channel in the 'points' array in ***FrameRow*** MUST be the same in each message of the multipart response. Position/Channel mapping is indicated by the channelURI's array. a. Unlike channel data in ETP (which MUST NOT have null values), a ***FrameRow*** MAY have null values (i.e., if a channel that composes the row has no data value at a specific index).
7.	Rows include ALL channels in a channel set	1. There is no mechanism to 'subset' the channels. This protocol returns ALL data for all channels for a given Channel Set. a. To get a subset of channels, you must create a ChannelSet data object that contains only the desired Channel data objects, using Store (Protocol 4) (see Chapter **9**).
8.	Indexes in rows	1. When the *includeAllChannelSecondaryIndexes* field in the ***GetFrame*** message is false, each row MUST contain an index value for the

Row #	Requirement	Behavior
		primary index and each secondary index in the frame's channel set. In this case, index values MUST NOT be null. a. The index values in each row are in the same order as their corresponding ***IndexMetadataRecord*** records in the ***GetFrameResponseHeader*** message, and the primary index is always first. 2. When *includeAllChannelSecondaryIndexes* is false, each row MUST contain an index value for the primary index and each secondary index of each channel in the channel set. a. There should only be one value per 'compatible' index. b. The index values in each row are in the same order as their corresponding ***IndexMetadataRecord*** records in the ***GetFrameResponseHeader*** message, and the primary index is always first. c. Secondary index values MUST NOT be null for secondary indexes that are secondary indexes on the channel set. d. Secondary index values MAY be null for secondary indexes that are not secondary indexes on the channel set.
9.	Index Metadata	1. When sending messages, both the store AND the customer MUST ensure that all index metadata and data derived from index metadata are consistent in all fields in the message. a. **EXAMPLE:** The *uom* and *depthDatum* in an ***IndexInterval*** record MUST be consistent with the channel set's frame index metadata. b. A store MUST reject requests with inconsistent index metadata with an appropriate error such as EINVALID_OBJECT (14) or EINVALID_ARGUMENT (5).
10.	Row order for responses	1. Rows MUST be sent in primary index order for the frame, both within one message and across multiple messages. Primary index order is always as appropriate for the index direction of the frame's primary index (i.e., increasing or decreasing).

7.2.3 ChannelDataFrame: Capabilities

The table below lists key capabilities for this protocol. The protocol capabilities are all defined here.

- For protocol-specific behavior relating to using these capabilities in this protocol, see Sections **7.2.1◦ChannelDataFrame: Message Sequences** and **7.2.2 ChannelDataFrame: General Requirements**.
- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section **3.3**.

ChannelDataFrame (Protocol 2): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of each endpoint capability, see Section **3.3**.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section **3.3.2**. Behavior associated with other endpoint capabilities are defined in relevant chapters.			

ChannelDataFrame (Protocol 2): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
EXAMPLE: The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP** Server.			
Protocol Capabilities			
FrameChangeDetectionPeriod: The maximum time period in seconds for updates to a channel to be visible in ChannelDataFrame (Protocol 2). Updates to channels are not guaranteed to be visible in responses in less than this period. (**EXAMPLE:** If your requested range includes rows that just received new data, the store may not return those rows. The store may be allowing time to potentially receive additional values for the rows before including them in responses.) The intent for this capability is that ChannelDataframe rows are complete, and not 'partially updated'. ChannelDataFrame (Protocol 2) should not be used to poll for realtime data.	long	seconds <number of seconds>	**Default:** 60 **Min:** 1 **Max:** 600
MaxFrameResponseRowCount: The maximum total count of rows allowed in a complete multipart message response to a single request.	Long	count <count of rows>	**MIN:** 100,000
SupportsSecondaryIndexFiltering: Indicates whether an endpoint supports filtering requested data by secondary index values. If the filtering can be technically supported by an endpoint, this capability should be true.	Boolean	N/A	N/A

7.3 ChannelDataFrame: Message Schemas

This section provides a figure that displays all messages defined in ChannelDataFrame (Protocol 2). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.

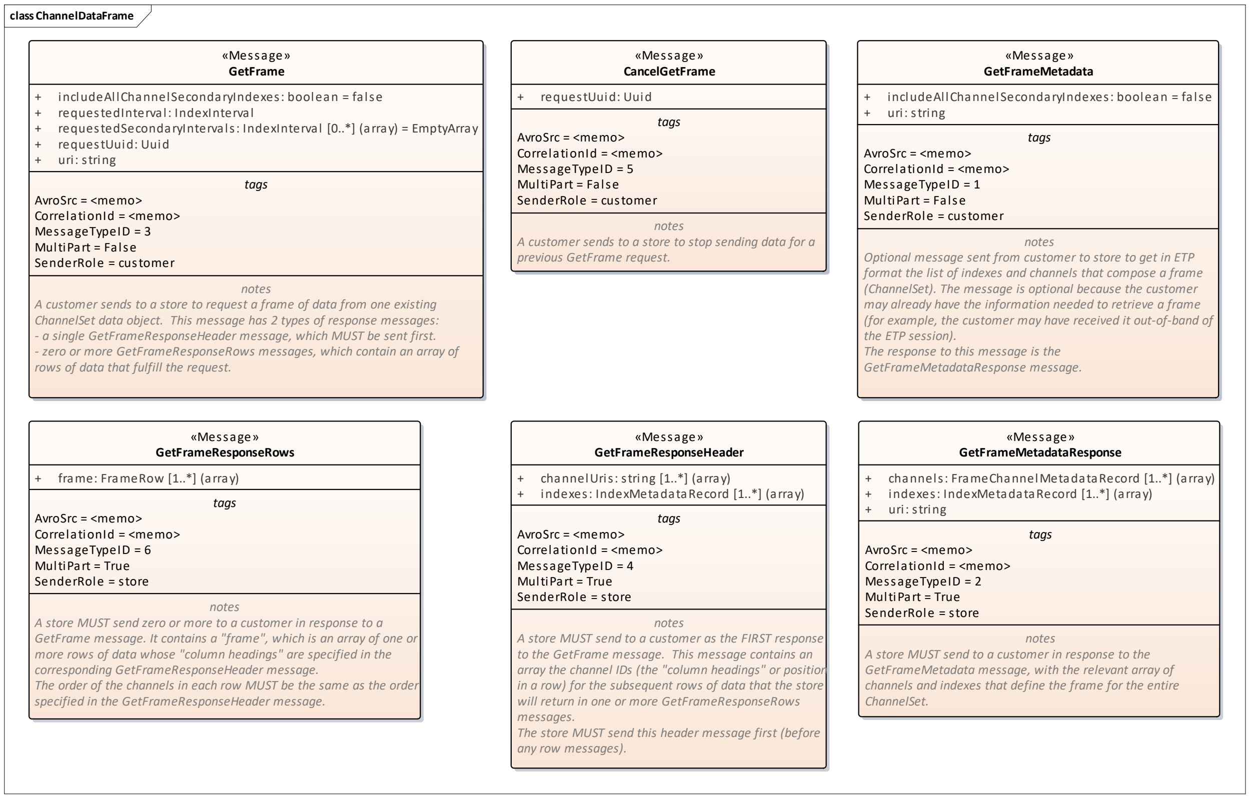

class ChannelDataFrame
«Message»
GetFrame
+ includeAllChannelSecondaryIndexes: boolean = false
+ requestedInterval: IndexInterval
+ requestedSecondaryIntervals: IndexInterval [0..*] (array) = EmptyArray
+ requestUuid: Uuid
+ uri: string
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 3
MultiPart = False
SenderRole = customer
notes
A customer sends to a store to request a frame of data from one existing ChannelSet data object. This message has 2 types of response messages:
- a single GetFrameResponseHeader message, which MUST be sent first.
- zero or more GetFrameResponseRows messages, which contain an array of rows of data that fulfill the request.
«Message»
CancelGetFrame
+ requestUuid: Uuid
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 5
MultiPart = False
SenderRole = customer
notes
A customer sends to a store to stop sending data for a previous GetFrame request.
«Message»
GetFrameMetadata
+ includeAllChannelSecondaryIndexes: boolean = false
+ uri: string
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1
MultiPart = False
SenderRole = customer
notes
Optional message sent from customer to store to get in ETP format the list of indexes and channels that compose a frame (ChannelSet). The message is optional because the customer may already have the information needed to retrieve a frame (for example, the customer may have received it out-of-band of the ETP session).
The response to this message is the GetFrameMetadataResponse message.
«Message»
GetFrameResponseRows
+ frame: FrameRow [1..*] (array)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 6
MultiPart = True
SenderRole = store
notes
A store MUST send zero or more to a customer in response to a GetFrame message. It contains a "frame", which is an array of one or more rows of data whose "column headings" are specified in the corresponding GetFrameResponseHeader message.
The order of the channels in each row MUST be the same as the order specified in the GetFrameResponseHeader message.
«Message»
GetFrameResponseHeader
+ channelUris: string [1..*] (array)
+ indexes: IndexMetadataRecord [1..*] (array)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 4
MultiPart = True
SenderRole = store
notes
A store MUST send to a customer as the FIRST response to the GetFrame message. This message contains an array the channel IDs (the "column headings" or position in a row) for the subsequent rows of data that the store will return in one or more GetFrameResponseRows messages.
The store MUST send this header message first (before any row messages).
«Message»
GetFrameMetadataResponse
+ channels: FrameChannelMetadataRecord [1..*] (array)
+ indexes: IndexMetadataRecord [1..*] (array)
+ uri: string
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 2
MultiPart = True
SenderRole = store
notes
A store MUST send to a customer in response to the GetFrameMetadata message, with the relevant array of channels and indexes that define the frame for the entire ChannelSet.

Figure 12: ChannelDataFrame: message schemas

7.3.1 Message: GetFrame

A customer sends to a store to request a frame of data from one existing ChannelSet data object. This message has 2 types of response messages:

- a single GetFrameResponseHeader message, which MUST be sent first.
- zero or more GetFrameResponseRows messages, which contain an array of rows of data that fulfill the request.

Message Type ID: 3

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uri	The URI of the ChannelSet data object from which you want to retrieve "rows" of data. If both endpoints support alternate URIs for the session, this MAY be an alternate URI. Otherwise, this MUST be a canonical Energistics data object URI. For more information, see **Appendix: Energistics Identifiers**.	string	1	1

Field Name	Description	Data Type	Min	Max
includeAllChannelSecondaryIndexes	Flag that allows the customer to request that secondary indexes from Channel data objects that are NOT secondary indexes in the containing ChannelSet data object be included in the response. **NOTE:** Some of these secondary indexes from Channel data objects may have null values.	boolean	1	1
requestedInterval	The interval of the ChannelSet data object over which data is being requested as specified in IndexInterval, which includes the start and end indexes (that define the interval) and The units and depth datum MUST match those in the ***IndexMetadataRecord*** for the channel set's primary index. The start and end indexes are INCLUSIVE.	IndexInterval	1	1
requestUuid	A UUID assigned to this request by the customer. If the customer later must cancel this request, it refers to this *requestUuid* in the CancelGetFrame message. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1
requestedSecondaryIntervals	(Optional) For channels/channel sets that indicate an index is "filterable" (i.e., the *filterable* field on the IndexMetadataRecord is set to true) this option allows the customer to request that results be filtered on secondary indexes. This is an array of the secondary intervals (as defined in IndexInterval) on which the customer wants to filter. **EXAMPLE:** If your primary interval (*requestedInterval* field above) is time, this field could be a depth interval on which filtering is being requested.	IndexInterval	0	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataFrame",
     "name": "GetFrame",
     "protocol": "2",
     "messageType": "3",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "includeAllChannelSecondaryIndexes", "type": "boolean", "default": false },
         { "name": "requestedInterval", "type":
"Energistics.Etp.v12.Datatypes.Object.IndexInterval" },
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         {
             "name": "requestedSecondaryIntervals",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.Object.IndexInterval" }, "default": []
         }
     ]
}
```

7.3.2 Message: GetFrameResponseHeader

A store MUST send to a customer as the FIRST response to the GetFrame message. This message contains an array the channel IDs (the "column headings" or position in a row) for the subsequent rows of data that the store will return in one or more GetFrameResponseRows messages.

The store MUST send this header message first (before any row messages).

Message Type ID: 4

Correlation Id Usage: MUST be set to the *messageId* of the ***GetFrame*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
channelUris	An array of channel URIs that indicates the position of each channel in the response FramePoints array in FrameRow (i.e., it specifies the "column headings" for each subsequent "row" returned.) These MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.	string	1	*
indexes	An array of IndexMetadataRecord records, with metadata for the data indexes in the frame, in the order in which they will appear in the frame. The record for the primary index MUST always be the first record in the array.	IndexMetadataRecord	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataFrame",
     "name": "GetFrameResponseHeader",
     "protocol": "2",
     "messageType": "4",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "channelUris",
             "type": { "type": "array", "items": "string" }
         },
         {
             "name": "indexes",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.IndexMetadataRecord" }
         }
     ]
}
```

7.3.3 Message: GetFrameResponseRows

A store MUST send zero or more to a customer in response to a GetFrame message. It contains a "frame", which is an array of one or more rows of data whose "column headings" are specified in the corresponding GetFrameResponseHeader message.

The order of the channels in each row MUST be the same as the order specified in the GetFrameResponseHeader message.

Message Type ID: 6

Correlation Id Usage: MUST be set to the *messageId* of the ***GetFrame*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
frame	An array of rows with each row containing the content defined in FrameRow.	FrameRow	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataFrame",
     "name": "GetFrameResponseRows",
     "protocol": "2",
     "messageType": "6",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "frame",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.FrameRow" }
         }
     ]
}
```

7.3.4 Message: CancelGetFrame

A customer sends to a store to stop sending data for a previous GetFrame request.

Message Type ID: 5

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
requestUuid	The UUID (assigned by the customer in the GetFrame message) of the request that is being canceled. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataFrame",
     "name": "CancelGetFrame",
     "protocol": "2",
     "messageType": "5",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" }
     ]
}
```

7.3.5 Message: GetFrameMetadata

Optional message sent from customer to store to get in ETP format the list of indexes and channels that compose a frame (ChannelSet). The message is optional because the customer may already have the information needed to retrieve a frame (for example, the customer may have received it out-of-band of the ETP session).

The response to this message is the GetFrameMetadataResponse message.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uri	The URI of a channel set (ChannelSet) (or any other Energistics domain data object that can describe a "frame"). If both endpoints support alternate URIs for the session, this MAY be an alternate URI. Otherwise, this MUST be a canonical Energistics data object URI. For more information, see **Appendix: Energistics Identifiers**.	string	1	1
includeAllChannelSecondaryIndexes	Flag that allows the customer to request that secondary indexes from Channel data objects that are NOT secondary indexes in the containing ChannelSet data object be included in the response.	boolean	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataFrame",
     "name": "GetFrameMetadata",
     "protocol": "2",
     "messageType": "1",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "includeAllChannelSecondaryIndexes", "type": "boolean", "default": false }
     ]
}
```

7.3.6 Message: GetFrameMetadataResponse

A store MUST send to a customer in response to the GetFrameMetadata message, with the relevant array of channels and indexes that define the frame for the entire ChannelSet.

Message Type ID: 2

Correlation Id Usage: MUST be set to the *messageId* of the ***GetFrameMetadata*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
uri	The URI of a channel set (ChannelSet data object) (or any other Energistics domain data object that can describe a "frame"). This MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
indexes	An array of IndexMetadataRecords for each channel that composes the frame. The record for the primary index MUST always be the first record in the array.	IndexMetadataRecord	1	*
channels	An array of metadata for each channel (as defined in FrameChannelMetadataRecord) that comprises the frame.	FrameChannelMetadataRecord	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataFrame",
     "name": "GetFrameMetadataResponse",
     "protocol": "2",
     "messageType": "2",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         { "name": "uri", "type": "string" },
         {
             "name": "indexes",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.IndexMetadataRecord" }
         },
         {
             "name": "channels",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.FrameChannelMetadataRecord" }
         }
     ]
}
```

8 Discovery (Protocol 3)

ProtocolID: 3

Defined Roles: store, customer

Customers of a store use Discovery (Protocol 3) to enumerate and understand the contents of a store. The store represents a database or storage of data object information; Discovery uses Energistics domain data models to navigate the store as a graph. (For more information on graph concepts and how this works in ETP, see Section **8.1**).

IMPORTANT! The main benefit of data model as graph is the ability to efficiently and precisely identify—often with a single request—the set of data objects that you are interested in. For more information on graphs and how ETP design leverages them, see Section **8.1.1**.

In Discovery, a customer and a store exchange discrete request and response messages that allow the customer application to request information and "walk the graph" to discover the store's content, which includes the data objects (nodes on the graph) and the relationships between them (the edges between the nodes).

Since the previous version of ETP, Discovery (Protocol 3) has been significantly redesigned with the 2 main goals of the redesign being:

- A single discovery protocol that works consistently across all Energistics domain models.
- The ability to reduce the number of messages required (i.e. reduce the back and forth between endpoints) to get all data objects of interest, thereby reducing traffic on the wire.

Additionally, Discovery (Protocol 3) contains messages for discovering deleted resources. This and other behavior defined here are functionality required to support workflows for eventual consistency between 2 stores.

Other ETP sub-protocols that may be used with Discovery (Protocol 3):

- If more than one dataspace exists on an endpoint and a customer needs to navigate dataspaces to find a particular store, the customer MUST use Dataspaces (Protocol 24) (see Chapter **21**).
  - When the customer finds the particular dataspace it wants, then the customer MUST use Discovery (Protocol 3) to discover and enumerate the content of the dataspace. **NOTE:** ETP stores MUST always have a default dataspace; for more information, see Section **8.2.2**.
- If a customer wants to dynamically discover a store's data model (i.e., understand what object types are *possible* in the store at a given location whether or not there is any data in the store), without prior knowledge of the overall data model and graph connectivity, the customer MUST use SupportedTypes (Protocol 25) (see Chapter◦**22**).
- To filter on property values within a data object, use DiscoveryQuery (Protocol 13) (see Chapter **15**). This includes discovery of planned vs. actual objects; for more information, see Section **8.2.2.** **NOTE:** For some widely used use cases, the ***GetResources*** message in Discovery (Protocol 3) provides a few filters at the message level; see Section **8.2.1.1**.
- For information about workflows for eventual consistency between stores, see **Appendix: Data Replication and Outage Recovery Workflows**.

This chapter includes main sections for:

- **Key ETP concepts** that are important to understanding how this protocol is intended to work (see Section **8.1**).
- **Required behavior**, which includes:
  - Description of the message sequence for main tasks, along with required behavior and possible errors (see Section **8.2.1**).
  - Other required behavior (not covered in the message sequence) including use of capabilities for preventing and protecting against aberrant behavior (see Section **8.2.2**).

- Definitions of the key endpoint and protocol capabilities used in this protocol (see Section 8.2.3).

- **Sample schemas of the messages defined in this protocol** (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes documentation for each field in a schema (see Section 8.3).

8.1 Discovery: Key Concepts

This section explains concepts that are important to understanding how Discovery (Protocol 3) works.

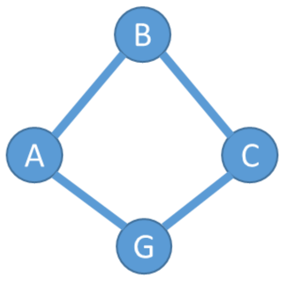
8.1.1 Data Model as a Graph

Discovery (Protocol 3) has been developed to work with data models as graphs. This section provides a general definition of a graph and how it works, and identifies key concepts, which are used as inputs for a customer to formulate a discovery request.

IMPORTANT! When understood and used properly, these inputs allow customers to specify precisely the desired set of objects in a single request, thereby reducing traffic on the wire. Conversely, if the graph concepts are not understood and not used properly, related operations will be highly inefficient.

A graph is a mathematical structure used to model pairwise relations between objects (https://en.m.wikipedia.org/wiki/Graph_theory). In this context, a graph is made up of *nodes* (which are also called *points* or *vertices*) and the *lines* (also called *links* or *edges*) that connect the nodes (**Figure 13**).

In some instances, the "direction" of the lines in the graph (which node points to another node) is not relevant; these are referred to as *undirected graphs* (Figure 13, left). In other graphs, the direction is important; these are referred to as *directed graphs* (Figure 13, right).



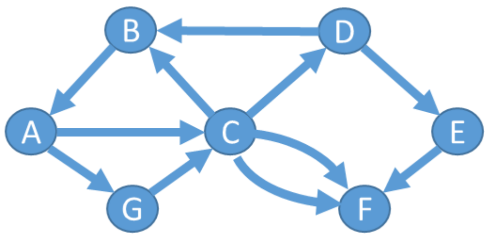


Figure 13: Examples of graphs: left image is an undirected graph and right image is a directed graph.

8.1.1.1 Energistics Data Models

For Energistics:

- Nodes represent data objects in a data model (WITSML, RESQML, PRODML or EML (i.e., Energistics *common*) (For the definition of data object, see Section **25.1**).
- Lines (directed links between nodes) represent relationships between those data objects. A data object can have multiple distinct references to other data objects (as specified in the various domain models).

For example, a wellbore may reference the well it is in and may reference multiple channels of data and/or channel sets of data collected about the wellbore. Or a 3D grid may reference hundreds of properties, and reference the faults and horizons used to derive the grid structure.

In some instances, a graph may have an obvious structure, for example a graph can be a tree or hierarchy, such as the traditional well, wellbore, log hierarchy. But in other cases, such as earth and reservoir modeling, objects may be in many-to-many relationships, so often there is no obvious "structure" or pattern (beyond the relationships). **Figure 14** shows how a set of data objects and the relationships among them form a directed multigraph.

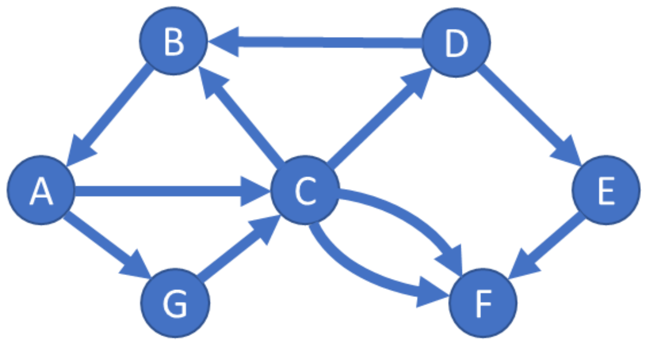


Figure 14: A set of data objects and the relationships among them form a directed multigraph.

8.1.1.1.1 Links and Relationships: Sources and Targets

The "direction" of links in some Energistics data models is relevant and is used in the Discovery protocol. Figure 15 shows a directed link between nodes A and C, which represents a relationship between data object A and data object C.

Figure 15: Node C is the "target" of the directed link from A to C; node A is the "source" of the directed link from A to C.

In graph theory terminology:

- Data object A is the “source” of the directed link from A to C.
- Data object C is the “target” of the directed link from A to C.
- By extension, data object A is the “source” of the relationship between A and C.

Relationships in Energistics data models are specified using 2 main constructs:

- eml:DataObjectReference (DOR)
- ByValue containment

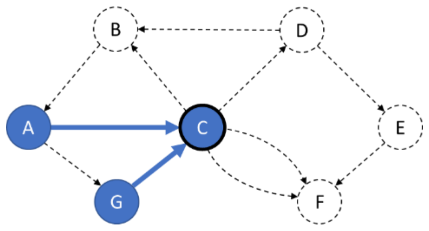
In general, these rules apply for specifying sources and targets:

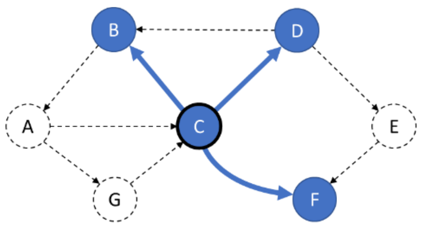
- For DORs, the data object that contains the DOR is the source, and the object that it "points to" is the target. **RECOMMENDATION:** DORs should be to one or more data objects in the SAME dataspace. Technically, DORs do not prohibit referencing a data object in another dataspace (i.e., a URI to the data object can be specified). However, some aspects of ETP functionality (e.g., notifications) may not work as designed.
- For ByValue containment, a contained object is the source, and the "container" is the target.

NOTES:

1. The “source” of a relationship between two data objects may be ML-specific.
2. For more information on these topics, see the relevant ML's ETP implementation specification.

Figure 16 is a more complex example of sources and targets relative to node C.





Sources are nodes with directed links **to** C.
Nodes A and G are sources of C.
C is the target of these relationships.

Targets are nodes with directed links **from** C.
Nodes B, D and F are targets of C.
C is the source of these relationships

Figure 16: More examples of targets and sources relative to node C.

8.1.1.1.2 Types of Relationships: Primary and Secondary

Energistics defines these types of relationships between data objects:

- **Primary**: Primary relationships are the "most important" relationships between data objects. For example, primary relationships may be used to organize or group data objects, such as organizing Channels into ChannelSets or organizing ChannelSets into Logs. In some cases, all relationships between data objects are important so all relationships are primary. Common characteristics of a primary relationship:
  - One end of the relationship is mandatory; that is, one object cannot exist (as a data object in the system) without the other. In the above example: A ChannelSet cannot exist without at least 1 Channel.
  - Relationships where one data object "contains" one or more other data objects (i.e., a by-value relationship), indicated with the ByValue construct in XML, such as ChannelSets containing Channels.
- **Secondary**: Secondary relationships are "less important" relationships between data objects and may provide additional contextual information about a data object to improve understanding. For example, the reference from a Channel to a Wellbore. Common characteristics of a secondary relationship:
  - Both ends of the relationships are optional.
- **GENERAL RULE:** ML-specific rules determine whether a relationship is primary or secondary. The ML rules to apply are determined by one of the data objects in the relationship.
  - In the case of relationships based on DORs, the type of relationship is determined by the ML of the data object that has the DOR.
  - In the case of by-value relationships, the relationship type is determined by the ML of the data object containing other data objects by value.

    EXAMPLES:
    A WITSML v2.0 ChannelSet contains WITSML v2.0 Channels by value and a ChannelSet has a DOR pointing to a WITSML Wellbore. So the WITSML rules determine whether the relationships between ChannelSet and Wellbore and ChannelSet and Channel are primary or secondary.
    A RESQML v2.0.1 obj_WellboreFeature has a DOR pointing to a WITSML Wellbore, so the RESQML rules determine whether the relationship between an obj_WellboreFeature and a Wellbore is primary or secondary.
- **NOTES:**
  1) For the ML-specific rules on which relationships are primary and which are secondary, see the ML-specific ETP implementation guide.
  2) Future versions of the ML data models will label the type or relationship (Primary or Secondary).

8.1.1.1.3 How to Use this Information in Discovery (Protocol 3)

Based on the notion of data model as a graph, the ***GetResources*** message (see Section **8.3.1**) was designed so the customer, using a single message, can specify relevant criteria for a discovery operation, for a very specific set of nodes (data objects) and optionally edges (relationships). This relevant criteria includes:

- *context*, which includes:
  - Starting position in the data model (which node) as specified by its URI (*uri*)
  - How many "levels" from that starting node the operation should explore (*depth*).
  - Which types of data objects to include (*dataObjectTypes*)
  - Which type of relationship, Primary or Secondary, to navigate (*navigableEdges*)
  - Whether to include an additional set of secondary targets and/or sources in the operation (*includeSecondaryTargets*, *includeSecondarySources*)
- Which "direction" in the graph that the operation should proceed (targets or sources) and whether or not to include the starting point (self) in the results (*scope*).
- Whether or not to provide counts of targets and sources (*countObjects*)

The ability to specify context and scope improves the efficiency of this version of ETP by making it possible to discover "more" of the model (i.e., a larger portion of the graph) with fewer discovery requests.

EXAMPLE: For a WITSML data model, if you specify the URI of a particular well, with an appropriate context and scope, it's possible to discover "everything" (all wellbores, logs, channels, BHA runs, etc.) related to the well in just one discovery request.

EXAMPLE: For a RESQML data model, a user wants to discover all horizon interpretations and all 2D grid representations of a particular horizon. These relationships are specified as DORs, with the horizon interpretations pointing to the horizon feature, and each 2D grid representation pointing to the horizon interpretation that it represents. The GetResources message for this example must specify:

- The context URI of the horizon feature with a level of '2'.
- The scope is "sources" (the data objects that point to the "self" specified in the horizon feature URI)
- The dataObjectTypes are "resqml20.obj_HorizonInterpretation, resqml20.obj_Grid2dRepresentation" (which are the specific types of data objects of interest).

8.1.1.1.4 Logic of the Discovery Operation

The basic logic of the ETP discovery operation MUST work as follows:

1. Based on the URI and depth (specified in *context*), a store determines an initial candidate set of nodes and edges.
2. If *includeSecondaryTargets* and/or *includeSecondarySources* flags are set to true, the store must expand the initial candidate set to include the secondary nodes and edges, as appropriate (i.e., depending on which flag(s) are set to true).
   a. If BOTH *includeSecondaryTargets* flag and *includeSecondarySources* are set to true, then the store MUST apply them "simultaneously" (not in sequence) so the candidate set is expanded once, not twice.
3. The store must remove from the set any node types that are not specified in *dataObjectTypes*.
4. Then the store must remove edges that are not connected to a node in the final set of nodes (post Step 3).

8.2 Discovery: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.
- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**

8.2.1 Discovery: Message Sequences

This section explains the basic message sequence for main tasks to be done using this protocol and includes related key behaviors and possible errors. The following General Requirements section provides additional requirements and rules for how this protocol works.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section.

Discovery (Protocol 3): Basic Message-Response flow by ETP Role	
Message (sent by customer)	**Response Message** (sent by store)
GetResources: Request message to enumerate contents of a store; it specifies details for the starting point and extent of the discovery operation.	***GetResourcesResponse*** (multipart): List of resources (graph nodes/data objects) that meet criteria specified in the request. ***GetResourcesEdgesResponse*** (optional, multipart): List of edges (relationship between nodes) that meet criteria in request, if edges requested.
GetDeletedResources: Request message to enumerate deleted objects in a store. (Used in data synchronization workflows.)	***GetDeletedResourcesResponse*** (multipart): List of deleted resources that a store can return information for.

8.2.1.1 *To discover data objects in a store and optionally the relationships between them:*

1. The customer sends a ***GetResources*** message (see Section **8.3.1**) to the store.

   For all details of all fields in this message, see the section referred to above. This table summarizes the fields in the ***GetResources*** message and referenced records and how they may impact the Discovery operation and message sequence.

Field	Description
context	REQUIRED. Context defines the parts of the data model that the customer wants the store to discover as defined in the ***ContextInfo*** record, which includes these fields:

Field	Description
uri	The URI of the dataspace or data object from which to start discovery. This MUST be the canonical Energistics URI as defined in **Appendix: Energistics Identifiers**. • Often, the first ***GetResources*** message that a customer sends contains a canonical dataspace URI, typically the default dataspace URI: eml:/// • A customer may also begin discovery by specifying the canonical URI for a specific data object. For more information about URIs used in Discovery (Protocol 3), see Section **8.2.2**.
depth	The depth in the graph (data model) from the "starting" URI. RECOMMENDATION: For maximum efficiency in discovery and notification operations, understand how the graph is intended to work and specify an appropriate value here ((i.e., for Discovery (Protocol 3) DO NOT simply set depth =1 and iterate)). For more information, see Section **8.1.1**.
dataObjectTypes	The types of data objects that you want to discover. This MUST be the set or a subset of the *supportedDataObjects* negotiated for the current ETP session. For more information, see Chapter **5** and Section **8.2.2.**
navigableEdges	Edges in a graph represent the relationships between the nodes (data objects). This field indicates the type of edges (relationships) to be navigated during the discovery operation. Choices are Primary, Secondary or Both. For more information about these types of relationships, see Section **8.1.1.1.2**. Only edges of the specified type are navigated during discovery. Use of this field helps to exclude unwanted objects being returned in Discovery.
includeSecondaryTargets	If true, the initial candidate set of nodes is expanded with targets of secondary relationships of nodes in the initial candidate set of nodes. The edges for these contextual relationships are also included. NOTE: This flag and *includeContextualSources* MUST be applied "simultaneously" (not in sequence) so the candidate set is expanded once, not twice. For more information, see Section **8.2.2**, row **10**.
includeSecondary Sources	If true, the initial candidate set of nodes is expanded with sources of contextual relationships of nodes in the initial candidate set of nodes. The edges for these contextual relationships are also included. For more information, see Section **8.2.2**, row **10**.

Field	Description
scope	Scope is specified in reference to the URI entered in context (row above). It indicates which direction in the graph that the operation should proceed (targets or sources) and whether or not to include the starting point (self) in the results. For definitions of sources and targets, see Section **8.1.1.1.1.** **NOTE:** Specifying an appropriate *context* and *scope* can significantly reduce the number of ***GetResources*** messages/back-and-forth between endpoints required to discover particular resources.

Field	Description
countObjects	If true, the store provides counts of sources and targets for each type of resource identified by the discovery operation. Default is false.
storeLastWriteFilter	Use this to optionally filter the discovery on a date when a data object was last written in a particular store. The store returns resources whose *storeLastWrite* date/time is greater than the date/time specified in this filter field. Purpose of this field is part of the behavior for eventual consistency between 2 stores.
activeStatusFilter	Use this to optionally filter the discovery for data objects that are currently "active" or "inactive" as defined in *ActiveStatusKind*.
includeEdges	If true, the store returns "edges" (relationships between the nodes). Default is false.

2. The store MUST respond with one or more of the messages listed below (column 1, Message Name), based on criteria in the table (column 2 (Positive or Error and Required or Optional) and column 3 (Description of conditions and related behavior).
   a. For information on the logic of the discovery operation, see Section **8.1.1.1.4**.

Message Name	Pos/Error RQD/OPT	Description
GetResourcesResponse (see Section **8.3.2**)	Pos/RQD	1. If the store successfully returns resources that meet the criteria specified in the ***GetResources*** message, the store MUST return one or more ***GetResourcesResponse*** messages and MUST observe these rules: a. Each message MUST contain an array of resources (i.e., a ***Resource*** record for each resource). **NOTE:** Data objects returned may vary depending on the *navigableEdges* specified (in ContextInfo, in the *Context* field on the ***GetResources*** message. i. A store MUST limit the total count of responses to the customer's value for the MaxResponseCount protocol capability. (**NOTE:** This protocol capability is a single limit that applies to the total combined count of ***GetResourcesResponse*** messages and ***GetResourcesEdgesResponse*** messages (if any). See the next row in this table.) ii. If the store exceeds the customer's MaxResponseCount value, the customer MAY send error ERESPONSECOUNT_EXCEEDED (30). iii. If a store's MaxResponseCount value is less than the customer's MaxResponseCount value, the store MAY further limit the total count of responses (to its value). iv. If a store cannot return all responses to a request because it would exceed the lower or the customer's or the store's value for MaxResponseCount, the store MUST terminate the multipart message with error ERESPONSECOUNT_EXCEEDED (30). v. A store MUST NOT send ERESPONSECOUNT_EXCEEDED (30) until it has sent MaxResponseCount responses. b. If the filter *dataObjectTypes* field is populated in the ***GetResources*** message, the store MUST first traverse the graph and then filter on *dataObjectTypes* (i.e., navigate to the full depth and scope, then filter response by the *dataObjectTypes*). (For more information, see Section **8.2.2**, Row **10**. c. If the store supports alternate URI formats (formats other than the canonical Energistics URI), it MAY return them in the *alternateUris* field (on the ***Resource*** record, in this message).

Message Name	Pos/Error RQD/OPT	Description
		For more information about URI formats, see **Appendix: Energistics Identifiers**. 2. If the store has no data objects that meet the criteria specified in the ***GetResources*** message, the store MUST send a ***GetResourcesResponse*** message with the FIN bit set and the *resources* field set to an empty array.
GetResourcesEdge Response (Section **8.3.3**)	Pos/OPT	1. If the store successfully returns edges that meet the criteria in the ***GetResources*** message, the store MUST return one or more ***GetResourcesEdgesResponse*** messages, each of which has an array of edges. 2. If the store has no edges that meet the criteria specified in the ***GetResources*** message, the store MUST NOT send a ***GetResourcesEdgesResponse***. **IMPORTANT:** The store MUST NOT return edges unless both: a. *includeEdges* is true. b. At least one resource is also returned. 3. When sending edges, it is RECOMMENDED that the store: a. Send the resources first, in breadth-first search order. b. Then send edges (i.e., after sending the resource records (nodes) that define each end of the edge). **NOTE:** Edges are defined by specifying the URI of each of its 2 nodes (source and target). c. Include an edge in the response if EITHER end (or both) are present in the ***GetResourcesResponse*** message (i.e., even if one node in the relationship is filtered out by *dataObjectTypes* filter, the store SHOULD still return a related edge). 4. The MaxResponseCount protocol capability (described in the row above) is a single limit that applies to the total combined count of ***GetResourcesResponse*** and ***GetResourcesEdgesResponse*** messages.
ProtocolException	Error/RQD	If the store does NOT successfully return resources or edges, it MUST send a non-map ***ProtocolException*** message with an appropriate error. **EXAMPLES:** • If the dataspace or data object specified by the URI in the context does not exist, the store MUST send error ENOT_FOUND (11). • If the request contains a malformed URI, the store MUST send error EINVALID_URI (9). For more information: • About ***ProtocolException*** messages and how they work, see Section **3.6.2.1**. • About how plural messages work with ***ProtocolException*** messages, see Section **3.8.1**.

3. Based on the response to a ***GetResources*** message and the specific data the customer is looking for, the customer MAY iteratively send additional ***GetResources*** messages until it discovers the desired resource(s).

   EXAMPLE: If the customer started with the URI eml:///, it might want to use one of the returned resources and continue discovering from there, so it would send another ***GetResources*** message using the URI of the desired resource.

   d. If the customer and store both support alternate URI formats and the store returned them in the **GetResourcesResponse** message, then the customer MAY use alternate URIs to make subsequent requests. (For more information and rules about using alternate URIs, see the *uri* field on the **Resources** record.)

8.2.1.2 To discover deleted objects in a store:

As part of the workflow for eventual consistency between 2 stores (aka, replication workflows), an endpoint may need to discover deleted objects, which are often called *tombstones*.

If a data object has been deleted within a store's ChangeRetentionTime capability (for more information, see Section **8.2.3**), the customer may follow these steps to discover tombstones. Otherwise, the customer may have to do a costly and inefficient full scan/compare. For more information, see **Appendix: Data Replication and Outage Recovery Workflows**.

Stores MUST support retention of tombstones and should set the ChangeRetentionPeriod to be as long as practical, ideally weeks. The longer it is, the less likelihood that a customer must perform an inefficient full scan.

Not every store will be able to accurately track and retain tombstones through the full ChangeRetentionPeriod. For example, some stores may lack a persistent data store for ETP-specific information so will lose this information if the store application restarts. **The minimum requirement to enable eventual consistency workflows is that:** If a store loses track of tombstones, the store MUST set *earliestRetainedChangeTime* in ***RequestSession*** or ***OpenSession*** to indicate the earlies timestamp that tombstones ARE available from.

To discover deleted objects in a store:

1. The customer MUST send a ***GetDeletedResources*** message (Section **8.3.4**) to the store.

   For all details of all fields in this message, see the section referenced above. This table summarizes the fields and how they may impact the message sequence.

Field	Description
dataspacesUri	REQUIRED. The URI of the dataspace where the objects were deleted. **NOTE:** Tombstones for deleted objects most likely no longer have sufficient history to put them in a context/scope of the data model, so the discovery MUST be done at the dataspace level. For more information, see **Appendix: Data Replication and Outage Recovery Workflows**.
deleteTimeFilter	Optionally, specify a delete time. 1. A customer MUST NOT request deleted resources with a deleteTimeFilter that is older than the store's ChangeRetentionPeriod endpoint capability. For more information, see Section **8.2.3**. 2. A store MUST deny any request that exceeds its value for ChangeRetentionPeriod and send error ERETENTION_PERIOD_EXCEEDED (5001).
dataObjectsType	Optionally, filter for the types of data objects you want.

2. The store MUST respond with one or more of these messages (column 1, Message Name), based on criteria in the table (column 2 (Positive or Error and Required or Optional) and column 3 (Description of conditions):

Message Name	Pos/Error RQD/OPT	Description
GetDeletedResourcesResponse (Section **8.3.5**)	Pos/RQD	1. If the store successfully returns deleted resources that meet the criteria specified in the ***GetDeletedResources*** message, the store MUST return one or more ***GetDeletedResourcesResponse*** messages that contain an array of deleted resources. a. A store MUST return any resources that were deleted more recently than its ChangeRetentionPeriod. i. The rules for the MaxResponseCount protocol capability described in Section **8.2.1.1** (first row of table), MUST also be observed for ***GetDeletedResourcesResponse*** messages.

Message Name	Pos/Error RQD/OPT	Description
		2. If the store has no deleted resources that meet the criteria specified in the ***GetDeletedResources*** message, the store MUST send a ***GetDeletedResourcesResponse*** message with the FIN bit set and the *deletedResources* field set to an empty array. 3. **NOTE:** For more information on how a customer can use these deleted resources to establish consistency with the store, see **Appendix: Data Replication and Outage Recovery Workflows**.
ProtocolException	Error/RQD	1. If the store does NOT successfully return deleted resources, it MUST send a non-map ***ProtocolException*** message with an appropriate error, such as EREQUEST_DENIED (6). a. If in the ***GetDeletedResources*** message, the *dataspacesUri* field is NOT a valid dataspace URI, send EINVALID_URI (9). b. If the URI is valid but does not refer to a deleted resource in the store, send error ENOT_FOUND (11). 2. For more information about ***ProtocolException*** messages and how they work, see Section **3.6.2.1**. 3. For more information about how plural messages work with ***ProtocolException*** messages, see Section **3.8.1**.

8.2.2 Discovery: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. Possible optional behaviors may also be defined. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) some rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter **5**. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session.

Row#	Requirement	Behavior
		c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in this specification. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3**. a. For the list of global capabilities and related behavior, see Section **3.3.2**. 3. Section **8.2.3** identifies the capabilities most relevant to this ETP sub-protocol. If one or more of the defined capabilities is presented by an endpoint, the other endpoint in the ETP session MUST accept it (them) and process the value, and apply them to the behavior as specified in this document. a. Additional details for how to use these capabilities are included in Section **8.2.1 Discovery: Message Sequence**.
3.	Message Sequence See Section **8.2.1**.	1. The Message Sequence section above (Section **8.2.1**) describes requirements for the main tasks listed there and also defines required behavior.
4.	Plural messages (which includes maps)	1. This protocol uses plural messages. For detailed rules on handling plural messages (including ProtocolException handling), see Section **3.7.3**.
5.	Discoverable data objects	1. Discovery MUST return only data objects whose type (*dataObjectTypes* specified in the *Context* field on the ***GetResources*** message) are in the set of negotiated *supportedTypes* (as indicated on the ***OpenSession*** message in Core (Protocol 0)) for the current ETP session. a. If a customer specifies a data object type in a request for a data object type that was not negotiated for the current ETP session, the store MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16). 2. For more information about data objects, resources, and identifiers, see Section **Appendix: Energistics Identifiers**.
6.	Discovery of planned vs. actual data objects	All top-level data objects in Energistics domain models derive from AbstractObject, which is in Energistics common (namespace = eml). AbstractObject has a field named ExistenceKind, which may have a value of "planned" or "actual". To discover planned vs actual data objects in a store: 1. The customer MUST use DiscoveryQuery (Protocol 13) (see Chapter **15**) and filter on ExistenceKind (specifying planned or actual) for AbstractObject.
7.	Dataspaces in discovery operations	1. "eml:///" refers to the default dataspace. a. A default dataspace ALWAYS exists, but it might be empty. b. If the default dataspace has no content, the store MUST return an empty array in the ***GetResourcesResponse*** message. 2. If other dataspaces exist, a client MUST: a. First use Dataspaces (Protocol 24) to discover the available dataspaces in the store (see Chapter **21**). b. Then use Discovery (Protocol 3) to discover and enumerate the content of a particular dataspace.

Row#	Requirement	Behavior
		3. All ETP stores MUST support discovery of dataspace URIs, including "eml:///", for all dataspaces supported by the store. a. When discovering a dataspace URI, the *scope* and *depth* fields (in the ***GetResources*** message) MUST be ignored.
8.	URI formats	1. For information about all URIs and allowed formats, see **Appendix: Energistics Identifiers**. 2. In a ***GetResources*** message, the customer MUST use one of the following: a. To discover the root contents of a dataspace, use the dataspace's canonical URI: "eml:///" i. Other root URIs MAY also be used, **EXAMPLE:** eml:///dataspace(some-dataspace-id)/ b. To discover the resource for a specific data object, use the data object's canonical Energistics URI. 3. Alternate URI formats: a. Alternate URIs are NOT allowed as URIs in the ***GetResources*** message. b. If the store DOES NOT support alternate URI formats, it MUST respond in the ***GetResourcesResponse*** message with only canonical Energistics URIs. c. If a store DOES support alternate URI formats, it MAY send alternate URIs in the ***GetResourcesResponse*** message. (**NOTE:** The *alternateUris* field is specified in the ***Resource*** record; a positive ***GetResourcesResponse*** message contains an array of ***Resource*** records.) d. For usage rules for alternate URIs, see Section **3.7.4.1**.
9.	How to navigate a data model	1. The customer MUST specify the details of how to navigate the data model in the ***GetResources*** message (see Section **8.3.1**), based on information explained in Section **8.1.1.1**. a. The mechanisms in the data object that specify the relationships between/among other data objects are DataObjectReference (DOR) and ByValue containment. For more information, see Section **8.1.1.1.1**. i. To navigate the graph, an endpoint MUST inspect the DOR and ByValue references. ii. Stores are not required to navigate relationships across dataspaces. 2. Currently, Energistics has 2 ways to specify DORs: a. The "old format" (which will be phased out over time) uses the previous Energistics URI format and a contentType (mime type) in the DOR format (see ObjectReference in the *Energistics Identifier Specification* v4.0 in Energistics Online). b. The "new format" replaces the contentType with the dataObjectType (which is the semantic equivalent of the OData qualifiedEntityType) and is based on the new Energistics URI format, which is explained in **Appendix: Energistics Identifiers** of this document (for information on the dataObjectType, see Section **25.3.7.1**). c. **Implications for ETP behavior related to the 2 formats for DORs:** A store MUST accept both the "old format" and the "new format". However, a store MUST return ONLY the "new format". (That is, a Store must be able to "understand" all DORs (both old format and new format) to traverse the graph in ETP for Discovery (Protocol 3) and notification subscriptions in Protocol 5.) 3. For information on how to navigate "across" different data models, see Row **11 below**.

Row#	Requirement	Behavior
10.	Logic of the discovery operation	For convenience this section is repeated from Section **8.1.1.1.4**. All field names refer to fields on the ***GetResources*** message. The basic logic of the ETP discovery operation MUST work as follows: 1. Based on the URI and depth (specified in *context*), a store determines an initial candidate set of nodes and edges. 2. If *includeSecondaryTargets* and/or *includeSecondarySources* flags are set to true, the store must expand the initial candidate set to include the secondary nodes and edges, as appropriate (i.e., depending on which flag(s) are set to true). a. If BOTH *includeSecondaryTargets* flag and *includeSecondarySources* are set to true, then the store MUST apply them "simultaneously" (not in sequence) so the candidate set is expanded once, not twice. 3. The store must remove from the set any node types that are not specified in *dataObjectTypes*. 4. Then the store must remove edges that are not connected to a node in the final set of nodes (post Step 3).
11.	Navigating data objects between different Energistics data models	Increasingly, software applications use data objects from various Energistics data models. For example, both earth modeling (RESQML) and production (PRODML) applications use the well and wellbore defined in WITSML. When working in applications that include objects from more than one Energistics data models, an application MUST observe these rules: 1. The domain model (ML) where the DOR resides or the data object that contains other data objects resides determines the attribute of the *navigableEdges* (i.e., whether the relationship is primary or secondary). a. **EXAMPLE:** In RESQML, all relationships are primary, so when a RESQML wellbore feature references a WITSML wellbore, the *navigableEdge* (relationship) is primary. b. **NOTE:** This is consistent with the general rule specified in Section **8.1.1.1.2**: The data object that has the DOR defines the primary or secondary relationship. 2. If the discovery operation has been specified so that it "continues" into the new data model, observe the *navigableEdges* as defined in that data model. **EXAMPLE:** In the example in 1.a above, any continued discovery MUST follow WITSML rules. 3. For exceptions to relationship rules, see the ML-specific ETP implementation guides.
12.	Requirements for use of PWLS	Practical Well Log Standard (PWLS) is an industry standard stewarded by Energistics. It categorizes the obscure mnemonics used for oilfield data and relates them to the marketing names for logging tools that make those measurements using plain English. PWLS provides an industry-agreed list of logging tool classes and a hierarchy of measurement properties and applies all known mnemonics to them. For more information, see Section **3.12.7**. 1. If an ETP store supports the WITSML Channel data object, then it MUST support PropertyKind data objects (which are an implementation of PWLS). 2. Endpoints MUST be able to discover property kind data objects (to determine available property kinds) and use the returned property kinds in relevant Discovery, Store and Query operations.

8.2.3 Discovery: Capabilities

The table below lists key capabilities for this protocol. The protocol capabilities are all defined here. For this protocol, one particularly crucial endpoint capability is also defined here.

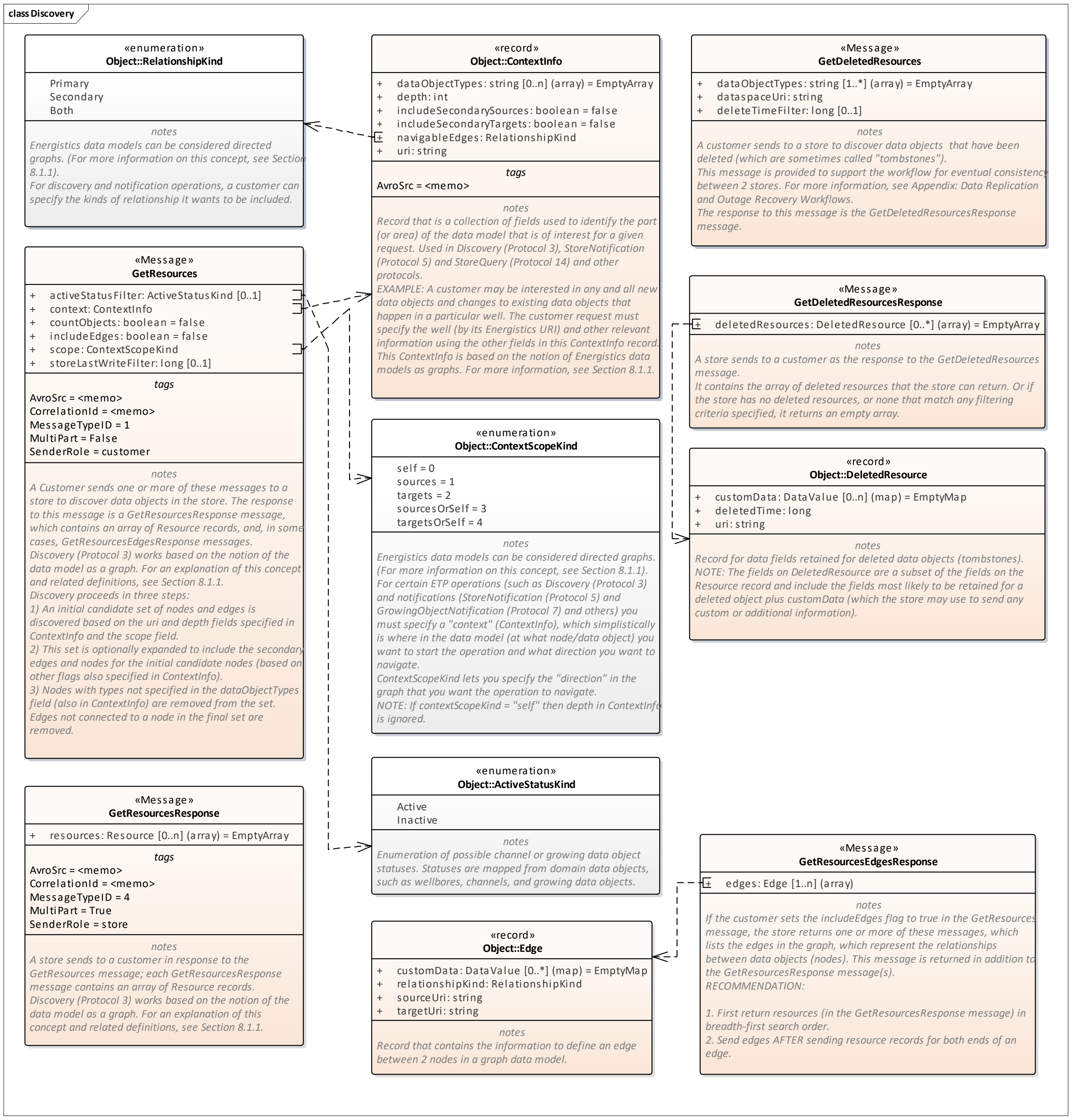
- For protocol-specific behavior for using these capabilities in this protocol, see Sections **8.2.1◦Discovery: Message Sequence** and **8.2.2 Discovery: General Requirements.**
- For definitions of endpoint and data object capabilities, see the links in the table.

- For general information about the types of capabilities and how they may be used, see Section **3.3**.

Discovery (Protocol 3): Capabilities			
Name: Description	**Type**	**Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions and usage rules, see Section **3.3**.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section **3.3.2**. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP** Server.			
ChangeRetentionPeriod: The minimum time period in seconds that a store retains the canonical URI of a deleted data object and any change annotations for channels and growing data objects. **RECOMMENDATION:** This period should be as long as is feasible in an implementation. When the period is shorter, the risk is that additional data will need to be transmitted to recover from outages, leading to higher initial load on sessions.	Long	Seconds **Value units:** <number of seconds>	**Default:** 86,400 **MIN:** 86,400
Protocol Capabilities			
MaxResponseCount: The maximum total count of responses allowed in a complete multipart message response to a single request.	Long	count **Value units:** <count of responses>	**MIN:** 10,000

8.3 Discovery: Message Schemas

This section provides a figure that displays all messages defined in Discovery (Protocol 3). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.



«enumeration»
Object::RelationshipKind

Primary
Secondary
Both

notes
Energistics data models can be considered directed graphs. (For more information on this concept, see Section 8.1.1).
For discovery and notification operations, a customer can specify the kinds of relationship it wants to be included.

«record»
Object::ContextInfo

+ dataObjectTypes: string [0..n] (array) = EmptyArray
+ depth: int
+ includeSecondarySources: boolean = false
+ includeSecondaryTargets: boolean = false
+ navigableEdges: RelationshipKind
+ uri: string

tags
AvroSrc = <memo>

notes
Record that is a collection of fields used to identify the part (or area) of the data model that is of interest for a given request. Used in Discovery (Protocol 3), StoreNotification (Protocol 5) and StoreQuery (Protocol 14) and other protocols.
EXAMPLE: A customer may be interested in any and all new data objects and changes to existing data objects that happen in a particular well. The customer request must specify the well (by its Energistics URI) and other relevant information using the other fields in this ContextInfo record.
This ContextInfo is based on the notion of Energistics data models as graphs. For more information, see Section 8.1.1.

«Message»
GetDeletedResources

+ dataObjectTypes: string [1..*] (array) = EmptyArray
+ dataspaceUri: string
+ deleteTimeFilter: long [0..1]

notes
A customer sends to a store to discover data objects that have been deleted (which are sometimes called "tombstones").
This message is provided to support the workflow for eventual consistency between 2 stores. For more information, see Appendix: Data Replication and Outage Recovery Workflows.
The response to this message is the GetDeletedResourcesResponse message.

«Message»
GetResources

+ activeStatusFilter: ActiveStatusKind [0..1]
+ context: ContextInfo
+ countObjects: boolean = false
+ includeEdges: boolean = false
+ scope: ContextScopeKind
+ storeLastWriteFilter: long [0..1]

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1
MultiPart = False
SenderRole = customer

notes
A Customer sends one or more of these messages to a store to discover data objects in the store. The response to this message is a GetResourcesResponse message, which contains an array of Resource records, and, in some cases, GetResourcesEdgesResponse messages.
Discovery (Protocol 3) works based on the notion of the data model as a graph. For an explanation of this concept and related definitions, see Section 8.1.1.
Discovery proceeds in three steps:
1) An initial candidate set of nodes and edges is discovered based on the uri and depth fields specified in ContextInfo and the scope field.
2) This set is optionally expanded to include the secondary edges and nodes for the initial candidate nodes (based on other flags also specified in ContextInfo).
3) Nodes with types not specified in the dataObjectTypes field (also in ContextInfo) are removed from the set. Edges not connected to a node in the final set are removed.

«Message»
GetDeletedResourcesResponse

+ deletedResources: DeletedResource [0..*] (array) = EmptyArray

notes
A store sends to a customer as the response to the GetDeletedResources message.
It contains the array of deleted resources that the store can return. Or if the store has no deleted resources, or none that match any filtering criteria specified, it returns an empty array.

«enumeration»
Object::ContextScopeKind

self = 0
sources = 1
targets = 2
sourcesOrSelf = 3
targetsOrSelf = 4

notes
Energistics data models can be considered directed graphs. (For more information on this concept, see Section 8.1.1).
For certain ETP operations (such as Discovery (Protocol 3) and notifications (StoreNotification (Protocol 5) and GrowingObjectNotification (Protocol 7) and others) you must specify a "context" (ContextInfo), which simplistically is where in the data model (at what node/data object) you want to start the operation and what direction you want to navigate.
ContextScopeKind lets you specify the "direction" in the graph that you want the operation to navigate.
NOTE: If contextScopeKind = "self" then depth in ContextInfo is ignored.

«record»
Object::DeletedResource

+ customData: DataValue [0..n] (map) = EmptyMap
+ deletedTime: long
+ uri: string

notes
Record for data fields retained for deleted data objects (tombstones).
NOTE: The fields on DeletedResource are a subset of the fields on the Resource record and include the fields most likely to be retained for a deleted object plus customData (which the store may use to send any custom or additional information).

«Message»
GetResourcesResponse

+ resources: Resource [0..n] (array) = EmptyArray

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 4
MultiPart = True
SenderRole = store

notes
A store sends to a customer in response to the GetResources message; each GetResourcesResponse message contains an array of Resource records.
Discovery (Protocol 3) works based on the notion of the data model as a graph. For an explanation of this concept and related definitions, see Section 8.1.1.

«enumeration»
Object::ActiveStatusKind

Active
Inactive

notes
Enumeration of possible channel or growing data object statuses. Statuses are mapped from domain data objects, such as wellbores, channels, and growing data objects.

«Message»
GetResourcesEdgesResponse

+ edges: Edge [1..n] (array)

notes
If the customer sets the includeEdges flag to true in the GetResources message, the store returns one or more of these messages, which lists the edges in the graph, which represent the relationships between data objects (nodes). This message is returned in addition to the GetResourcesResponse message(s).
RECOMMENDATION:

1. First return resources (in the GetResourcesResponse message) in breadth-first search order.
2. Send edges AFTER sending resource records for both ends of an edge.

«record»
Object::Edge

+ customData: DataValue [0..*] (map) = EmptyMap
+ relationshipKind: RelationshipKind
+ sourceUri: string
+ targetUri: string

notes
Record that contains the information to define an edge between 2 nodes in a graph data model.

Figure 17: Discovery: message schemas

8.3.1 Message: GetResources

A Customer sends one or more of these messages to a store to discover data objects in the store. The response to this message is a GetResourcesResponse message, which contains an array of Resource records and, in some cases, ***GetResourcesEdgesResponse*** messages (Section 8.3.3).

Discovery (Protocol 3) works based on the notion of the data model as a graph. For an explanation of this concept and related definitions, see Section **8.1.1**.

Discovery proceeds in three steps:

1) An initial candidate set of nodes and edges is discovered based on the *uri* and *depth* fields specified in ContextInfo and the *scope* field.

2) This set is optionally expanded to include the secondary edges and nodes for the initial candidate nodes (based on other flags also specified in ContextInfo).

3) Nodes with types not specified in the *dataObjectTypes* field (also in ContextInfo) are removed from the set. Edges not connected to a node in the final set are removed.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
context	As defined in the ContextInfo record, which includes the URI of the dataspace or data object to begin the discovery, what specific types of data objects are of interest, and how many "levels" of relationships in the model to discover, among others. The URI MUST be a canonical Energistics data object or dataspace URI; for more information, see **Appendix: Energistics Identifiers**.	ContextInfo	1	1
scope	Scope is specified in reference to the URI (which is entered in the *context* field). It indicates which direction in the graph that the operation should proceed (targets or sources) and whether or not to include the starting point (self). The enumerated values to choose from are specified in ContextScopeKind. For definitions of targets and sources, see Section **8.1.1**. **NOTE:** If scope = "self", then depth (in ContextInfo) is ignored.	ContextScopeKind	1	1
storeLastWriteFilter	Use this to optionally filter the discovery on a date when the data object was *last written in a particular store*. The store returns resources whose *storeLastWrite* date/time is GREATER than the date/time specified in this filter field. Purpose of this field is part of the behavior for eventual consistency between 2 stores. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	0	1
countObjects	If true, the store provides counts of sources and targets for each resource identified by Discovery. Default is false.	boolean	1	1
activeStatusFilter	Use this to optionally filter the discovery for data objects that are currently "active" or "inactive" as defined in ActiveStatusKind. This field is for data objects that have a notion of being active or inactive. Each ML defines which data objects this applies to and how it applies to them. Examples include WITSML channel data objects and growing data objects, which have a	ActiveStatusKind	0	1

Field Name	Description	Data Type	Min	Max
	GrowingStatus property (which maps to the ETP field *activeStatus* on ***Resource***), which may be • active = A channel or growing data object is actively producing data points. • inactive = A channel or growing object is offline or not currently producing data points. The store returns resources for data objects whose value for *activeStatus* matches the value specified in the *activeStatusFilter*.			
includeEdges	If true, the store returns "edges" (relationships between the nodes as defined in Edge) in GetResourcesEdgesResponse messages (in addition to the nodes (data objects) in the GetResourcesResponse messages). Default is false.	boolean	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Discovery",
     "name": "GetResources",
     "protocol": "3",
     "messageType": "1",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "context", "type": "Energistics.Etp.v12.Datatypes.Object.ContextInfo" },
         { "name": "scope", "type": "Energistics.Etp.v12.Datatypes.Object.ContextScopeKind" },
         { "name": "countObjects", "type": "boolean", "default": false },
         { "name": "storeLastWriteFilter", "type": ["null", "long"] },
         { "name": "activeStatusFilter", "type": ["null",
"Energistics.Etp.v12.Datatypes.Object.ActiveStatusKind"] },
         { "name": "includeEdges", "type": "boolean", "default": false }
     ]
}
```

8.3.2 Message: GetResourcesResponse

A store sends to a customer in response to the GetResources message; each ***GetResourcesResponse*** message contains an array of Resource records.

Discovery (Protocol 3) works based on the notion of the data model as a graph. For an explanation of this concept and related definitions, see Section **8.1.1**.

Message Type ID: 4

Correlation Id Usage: MUST be set to the *messageId* of the ***GetResources*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
resources	The list of Resource records the store is returning. The URI MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	Resource	0	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Discovery",
     "name": "GetResourcesResponse",
     "protocol": "3",
     "messageType": "4",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "resources",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.Object.Resource" }, "default": []
         }
     ]
}
```

8.3.3 Message: GetResourcesEdgesResponse

If the customer sets the *includeEdges* flag to true in the GetResources message, the store returns one or more of these messages, which lists the edges in the graph, which represent the relationships between data objects (nodes). This message is returned in addition to the GetResourcesResponse message(s).

RECOMMENDATION:

1. First return resources (in the ***GetResourcesResponse*** message) in breadth-first search order.
2. Send edges AFTER sending resource records for both ends of an edge.

Message Type ID: 7

Correlation Id Usage: MUST be set to the *messageId* of the ***GetResources*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
edges	An array of Edge records, each of which specifies the URI of the source node and target node that defines each edge. The URI MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	Edge	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Discovery",
     "name": "GetResourcesEdgesResponse",
     "protocol": "3",
     "messageType": "7",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "edges",
             "type": { "type": "array", "items": "Energistics.Etp.v12.Datatypes.Object.Edge" }
         }
     ]
}
```

8.3.4 Message: GetDeletedResources

A customer sends to a store to discover data objects that have been deleted (which are sometimes called "tombstones").

This message is provided to support the workflow for eventual consistency between 2 stores. For more information, see Appendix: Data Replication and Outage Recovery Workflows.

The response to this message is the GetDeletedResourcesResponse message.

Message Type ID: 5

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
dataspaceUri	The URI of the dataspace where the objects were deleted. The URI MUST be a canonical Energistics dataspace URI; for more information, see **Appendix: Energistics Identifiers**. **NOTE:** Tombstones for deleted objects most likely no longer have sufficient history to put them in a context/scope of the data model, so the discovery MUST be done at the dataspace level.	string	1	1
deleteTimeFilter	Optionally, specify a delete time. The store returns resources for objects whose delete times are greater than this value. Must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	0	1
dataObjectTypes	Optionally, filter for the types of data objects you want. The default is an empty array, which means ALL data types. The form of the list is an array of strings, where each value is a dataObjectType as described in Appendix: Energistics Identifiers and serialized as JSON. It is the semantic equivalent of a qualifiedEntityType in OData. They ARE case sensitive. **EXAMPLES:** "witsml20.Well", "witsml20.Wellbore", "prodml21.WellTest", "resqml20.obj_TectonicBoundaryFeature" "eml21.DataAssuranceRecord" To indicate that all data objects within a data schema version are supported, you can use a star (*) as a wildcard, **EXAMPLES:** "witsml20.*", "prodml21.*", "resqml20.*", So "witsml20.*" means all data objects defined by WITSML v2.0 data schemas.	string	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Discovery",
     "name": "GetDeletedResources",
     "protocol": "3",
     "messageType": "5",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "dataspaceUri", "type": "string" },
         { "name": "deleteTimeFilter", "type": ["null", "long"] },
         {
             "name": "dataObjectTypes",
             "type": { "type": "array", "items": "string" }, "default": []
         }
     ]
}
```

8.3.5 Message: GetDeletedResourcesResponse

A store sends to a customer as the response to the GetDeletedResources message.

It contains the array of deleted resources that the store can return. Or if the store has no deleted resources, or none that match any filtering criteria specified, it returns an empty array.

Message Type ID: 6

Correlation Id Usage: MUST be set to the *messageId* of the ***GetDeletedResources*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
deletedResources	An array of DeletedResource records, one for each tombstone. Or if the store has no tombstones or none that meet the specified criteria, an empty array. The URI MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	DeletedResource	0	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Discovery",
     "name": "GetDeletedResourcesResponse",
     "protocol": "3",
     "messageType": "6",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "deletedResources",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.Object.DeletedResource" }, "default": []
         }
     ]
}
```

9 Store (Protocol 4)

ProtocolID: 4

Defined Roles: store, customer

Use Store (Protocol 4) to get, put and delete ALL data objects defined by Energistics domain data models—which includes data objects that "contain" other data objects (such as channel sets and logs), channels, and "growing data objects" (such as WITSML trajectories, mud logs and others). **NOTE:** The ability to handle some operations on growing data objects in this protocol is new behavior for ETP v1.2.

For the Energistics' definition of data object, information on the kinds of data objects, and information on identification, see **Appendix: Energistics Identifiers** (Section **25.1**).

Other ETP sub-protocols that may be used with Store (Protocol 4):

- To subscribe to and receive notifications from the store about operations/events in the store resulting from operations using Store (Protocol 4), use StoreNotification (Protocol 5). That is, this chapter explains events that trigger notifications in StoreNotification (Protocol 5); however, **the store is only required to send notifications if the customer is subscribed to notifications for the appropriate context.** For more information on Protocol 5, see Chapter **10**.
- To UPDATE a growing data object—including the "header" or any of the parts—use GrowingObject (Protocol 6) (Chapter **11**).
- To query on fields in a data object for Store get operations, use StoreQuery (Protocol 14) (Chapter◦**16**).
- For information on streaming channel data or other operations specific to channels, see:
  - ChannelStreaming (Protocol 1), Chapter **6**
  - ChannelDataFrame (Protocol 2), Chapter **7**
  - ChannelSubscribe (Protocol 21), Chapter **19**
  - ChannelDataLoad (Protocol 22), Chapter **20**

This chapter includes main sections for:

- **Key Concepts** (Section **9.1**). Key ETP concepts that are important to understanding how this protocol is intended to work.
- **Required Behavior** (Section **9.2**), which includes:
  - **Message Sequence** (Section **9.2.1**). Description of the message sequence for main tasks, along with required behavior—including use of endpoint, data object, and protocol capabilities for preventing and protecting against aberrant behavior—possible errors, and related error codes.
  - **General Requirements** (Section **9.2.2**). Other functional requirements and required behavior (not covered in the message sequence), including use of additional endpoint, data object, and protocol capabilities, possible errors and related error codes.
  - **Capabilities** (Section **9.2.3**). The list of endpoint, data object, and protocol capabilities particularly relevant to this protocol, which includes links to required behavior (e.g., for "global" capabilities used throughout ETP).
- **Message Schemas** (Section **9.3**). Sample schemas of the messages defined in this protocol (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes documentation for each field in a schema.

9.1 Store: Key Concepts

This section contains key concepts that are important to understanding how Store (Protocol 4) works.

9.1.1 ETP uses 'Upsert' Semantics

Store (Protocol 4) uses 'Upsert' semantics for insert (create/add) and update operations. That is, a single ***PutDataObjects*** message is used for both operations.

- If the object already exists, it is completely replaced by the new version.
- If the object does not exist, it is created.
- **NOTE:** Exceptions to these behavior for specific object kinds (e.g., growing data objects) are noted in Section **9.2.2**.

Unlike the WITSML SOAP protocols, there are no partial updates of data objects; it is not possible to update individual fields in an object.

Though the same message is used for insert and update, notifications sent in StoreNotification (Protocol 5) have options for a store to designate if an operation was an insert (add) or an update operation (if the store is able to determine this). For more information, see Chapter **10.**

9.1.2 Handling Binary Large Objects (BLOBs) in ETP

Some messages in this protocol allow or require a data object to be sent with the message. If the size of the data object (bytes) is too large for the WebSocket message size (which for some WebSocket libraries can be quite small, e.g. 128 kb), you must subdivide the data object and send it in "chunks" (using the ***Chunk*** message). For information on how to handle these binary large objects (BLOBs), see Section **3.7.3.2**.

9.1.3 "Container" and "Contained" Data Objects

In Energistics standards, the concept of a "contained" data object refers to a data object (as defined in Section **25.1**) that is contained by another data object with a ByValue reference—and ONLY a ByValue reference—that is, relationships specified by an Energistics Data Object Reference (DOR) DO NOT result in container/contained objects.

EXAMPLE:

An Energistics data object MAY be included in one or more container data objects.

One of the best-known examples comes from WITSML where:

- One or more Channel data objects can be contained in one or more ChannelSet data objects. In this example, the Channels are the "contained" data objects and the Channel Set is the "container".
- One or more ChannelSet data objects can be contained in one or more Log data objects. In this example, the ChannelSets are the "contained" data objects and the Log is the "container".

  NOTE: Individual container/contained data objects are listed in the relevant ML's ETP implementation specification (which is a companion document to this ETP Specification). For example, Channel, Channel Set and other contained data objects defined in WITSML are listed in the *ETP v1.2 for WITSML v2.0 Implementation Specification*.

  For more details about the relationships between Channel, Channel Set and Log data objects, and this contained object concept, see http://docs.energistics.org/#WITSML/WITSML_TOPICS/WITSML-000-050-0-C-sv2000.html.

The inherent design of these container/contained data objects requires some additional handling for store operations and related notifications. ETP defines a data object capability, MaxContainedDataObjectCount, which allows an endpoint to limit the number of contained data objects in a container data object.

9.1.3.1 Joining and Unjoining

"Joining" is connecting a contained data object to a container. Conversely, "unjoining" is disconnecting a contained data object from a container. The details of these operations are specified in Section **9.2.2**, but a simple example here helps to show some key concepts.

When putting a container data object (which includes its contained data objects) into a store, those contained objects may already exist in the store OR they may be new objects that are being added to the store (with the adding or updating of the container). If the contained data objects already exist, they are joined to the container. If they DO NOT exist, then the store must add them, which is considered these 2 operations:

- Add (put, insert) the new contained data objects (which are standalone data objects in their own right) to the store.)
- Join the contained data objects to their container.

Because it can be 2 operations, it is possible to generate 2 notifications (in StoreNotification (Protocol 5), which is also explained below).

"Unjoining" happens when a contained data object is removed from a container. Unjoins happen in three main ways:

- Putting a container data object into a store may unjoin contained data objects. **EXAMPLE:** This can happen when the container itself previously existed in the store and the new copy of the container has fewer contained data objects. The contained data objects no longer in the container are unjoined from the container and, in some cases, may be pruned (see Section **9.1.3.2**).
- Deleting a container data object unjoins any data objects it contains.
- Deleting a data object that is contained in another data object unjoins the deleted data object from its container(s).

9.1.3.2 Pruning

Another important concept is the notion of "pruning," which is deletion of contained data objects when an operation to the container would result in "orphan" contained data objects (that is, the contained data object is no longer joined to any container at all).

ETP specifies a data object capability, OrphanedChildrenPrunedOnDelete, which allows an endpoint to specify for each type of data object, whether or not it allows pruning operations. In addition, Store (Protocol 4) request messages that might result in orphaned contained data objects have a field named *pruneContainedObjects* flag, which allows the customer to request that orphans be pruned. Both conditions (the capability and the flag) must be true for pruning to occur.

- For more information on capabilities for this protocol, see Section **9.2.3**.
- For related behavior for their use and all details of operations on container/contained data objects Section◦**9.2.2.**

9.2 Store: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.

- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**
- Customer must have the URIs of the data objects it's interested in; these URIs are typically found using Discovery (Protocol 3) (Chapter **8**). (They may also come "out of band" of ETP, for example, by email.)

9.2.1 Store: Message Sequences

This section explains the basic message sequence for main tasks to be done using this protocol and includes related key behaviors including usage of capabilities and some possible errors.

The General Requirements section provides additional requirements and rules for how this protocol works.

NOTES:

1. This chapter explains events (operations) in Store (Protocol 4) that trigger the store to send notifications, which the store sends using StoreNotification (Protocol 5). However, statements of **NOTIFICATION BEHAVIOR** are here in this chapter, in the context of the detailed explanation of the behavior that triggers the notification.
2. Notification behavior is described here using MUST. However, the store MUST ONLY send notifications IF AND ONLY IF there is a customer subscribed to notifications for an appropriate context (i.e., a context that includes the data object) and the store MUST ONLY send notifications to those customers that are subscribed to appropriate contexts.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section.

Store (Protocol 4): Basic Message-Response flow by ETP Role	
Message from customer	**Response Message from store**
GetDataObjects: Request to get data objects from a store.	***GetDataObjectsResponse*** (multipart): Lists the data objects the store could return. ***Chunk*** (optional, multipart): If the data object is too large to fit into the response message (exceeds the WebSocket message size), it MUST be subdivided into a set of Chunk messages.
PutDataObjects (multipart): Request to add data objects to a store. Optionally, specify "prune" behavior for contained objects. ***Chunk*** (optional, multipart): If the data object is too large to fit into the request message (exceeds the WebSocket message size), it MUST be subdivided into a set of Chunk messages	***PutDataObjectsResponse*** (multipart): List of the data objects the store successfully put. For container data objects, includes additional information.
DeleteDataObjects: Request to delete data objects from a store. Optionally, specify "prune" behavior for contained objects.	***DeleteDataObjectsResponse*** (multipart): List of the data objects the store successfully deleted. For container data objects, includes additional information.

9.2.1.1 To get data objects from a store:

1. The customer MUST send the store the ***GetDataObjects*** message (Section **9.3.1**), which contains a map whose values are the URIs of the data objects that the customer wants to retrieve.

2. For the URIs it successfully returns data objects for, the store MUST send one or more ***GetDataObjectsResponse*** map response messages (Section **9.3.6**) where the map values are ***DataObject*** records with the data object URIs and data.
    a. For more information on how map response messages work, see Section **3.7.3**.
    b. The store MUST return all data objects that it can that meet the criteria of the request.
        i. For definition of data object, see Section **Appendix: Energistics Identifiers**.
    c. The store MUST observe limits specified by its own and the customer's values for the MaxDataObjectSize capability. For more information about how this capability works and required behavior, see Section **3.3.2.4**.
    d. If a data object is too large to fit in a WebSocket message, the store MAY subdivide the object and send it in "chunks" using the ***Chunk*** message. For more information on how to use ***Chunk*** messages, see Section **3.7.3.2**.
    e. For more information on how ***GetDataObjects*** works for container/contained data objects, see Section **9.2.2**, Row **15**.
        i. For definitions of container/contained objects, see Section **9.1.3**.
    f. For more information on how ***GetDataObjects*** works for channel data objects, see Section **9.2.2**, Row **16**.
    g. For more information on how ***GetDataObjects*** works for growing data objects, see Section **9.2.2**, Row **17**.
3. For the URIs it does NOT successfully return data objects for, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors, such as ENOT_FOUND (11).
    a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.

9.2.1.2 To put data objects in a store:

1. The customer MUST send the ***PutDataObjects*** message (Section **9.3.2**), which contains a map of ***DataObject*** records (Section **23.34.5**) each of which has the URI and data, one record for each data object that the customer wants to put in the request.
    a. Store (Protocol 4) uses "upsert" semantics, where update and insert use the same message—***PutDataObjects***, which means a put operation MUST always be a complete replacement of the data object(s).
    b. The customer MUST observe limits specified the store's values for the MaxDataObjectSize capability. For more information about how this capability works and required behavior, see Section **3.3.2.4**.
    c. If a data object is too large to fit in the WebSocket message, the customer MUST subdivide it and send it in "chunks" using the ***Chunk*** message (Section **9.3.7**). For more information, see Section **3.7.3.2**.
    d. A ***PutDataObjects*** message is designated as "multipart" because it may require use of ***Chunk*** messages. A request MUST have only 1 ***PutDataObjects*** message, followed by zero or more ***Chunk*** messages.
    e. For more information on how ***PutDataObjects*** works for container/contained data objects, see Section **9.2.2**, Row **19**.
    f. For more information on how ***PutDataObjects*** works for channels and channel sets, see Section **9.2.2**, Row **20**.
    g. For more information on how ***PutDataObjects*** works for growing data objects, see Section **9.2.2**,

Row **21**.

2. For the data objects it successfully puts (add to/replace in the store), the store MUST send one or more ***PutDataObjectsResponse*** map response messages (Section **9.3.3**).
    a. For more information on how map response messages work, see Section **3.7.3**.
    b. For data objects that already exist in the store, the put operation MUST ALWAYS be a complete replacement of the object.
        i. The store MUST also update the *storeLastWrite* field, which is only on the ***Resource*** for the data object (not on the data object itself). For more information about the *storeLastWrite* field, see Section **3.12.5.2**.
        ii. The store MUST NOT set or change any elements on the data object that are not "store managed". The *Creation* or *LastUpdate* elements on data objects are NOT "store managed". The store MUST NOT set or change the value of these elements.
    c. For data objects that do not yet exist in the store, the store MUST add them.
        i. The store MUST also update the *storeCreated and the storeLastWrite* fields, which are only on the ***Resource*** for the data object (not the data object itself).
            1. The store MUST NOT set or change any elements on the data object that are not "store managed". The *Creation* or *LastUpdate* elements on data objects are NOT "store managed". The store MUST NOT set or change the value of these elements.
        ii. If the data object being added is a Channel data object, growing data object, or other data object that can be "active", the store MUST also set its *activeStatus* flag to "inactive" (the default when a new data object is added to a store).
        iii. For additional information for growing data objects and container/contained data objects, see Section **9.2.2**.
    d. A store MAY schema-validate an object, but it is NOT REQUIRED to do so.
    e. **NOTIFICATION BEHAVIOR:** The store MUST send an ***ObjectChanged*** notification message with a type (objectChangeKind) of "insert" or "update".
        i. A store MUST send a notification for only the most recent effective state of a data object. So if multiple insert or update changes happened to a data object since the most recent insert or update notifications were sent for the data object, the store MAY send only one notification. If the object was inserted since the most recent insert or update notification was sent, the store MUST send an insert notification with the timestamp of the most recent insert or update change. Otherwise, the store MUST send an update notification with the timestamp of the most recent update.
        ii. Notifications are sent in StoreNotification (Protocol 5). For more information on rules for populating/sending notifications and why notification behavior is specified here, see Section **9.2.2**.
    f. The store MUST observe limits specified by its values for the MaxDataObjectSize capability. For more information about how this capability works and required behavior, see Section **3.3.2.4**.
    g. If the ***PutDataObjects*** message includes container objects, the ***PutDataObjectsResponse*** message MUST contain additional information as specified in the ***PutResponse*** record (Section **23.34.9**) in the message.
        i. For more information on how ***PutDataObjects*** works for container/contained data objects, see Section **9.2.2**.
    h. For more information on how ***PutDataObjects*** works for growing data objects, see Section **9.2.2**.
3. For the data objects it does NOT successfully put, the store MUST send one or more map

ProtocolException messages where values in the *errors* field (a map) are appropriate errors, such as EREQUEST_DENIED (6).

   a. For more information on how ***ProtocolException*** messages work with a plural messages, see Section **3.7.3**.

   b. The store MAY schema validate a data object but is not required to.

      i. A store MAY reject any document that is not schema valid and send error EINVALID_OBJECT (14).

9.2.1.3 To delete one or more data objects from a store:

1. The customer MUST send the ***DeleteDataObjects*** message (Section **9.3.4**), which contains a map whose values are the URIs for data objects to be deleted.

   a. For more information on how ***DeleteDataObjects*** works for container/contained data objects, see Section **9.2.2**.

2. For the URIs it successfully deletes data objects for, the store:

   a. MUST delete the data object(s).

      i. A store MUST retain a "tombstone" for each deleted data object for its value for the ChangeRetentionPeriod capability, which MUST be greater than or equal to the minimum value stated in this specification (see Section **9.2.3**).

         1. The "tombstone" and its content are defined by the ***DeletedResource*** schema (see Section **23.34.12**), which MUST include the data object's canonical URI, qualified type, and the time it was deleted.

         2. Tombstones MUST be retained by a store endpoint for at least the ChangeRetentionPeriod as long as there is at least one session connected to it. It is STRONGLY recommended to always retain tombstones for the ChangeRetentionPeriod.

         3. If a store is unable to retain tombstones for the full ChangeRetentionPeriod (e.g., because the store application restarted and it has no persistent storage for tombstones), the store MUST advise customers of the earliest timestamp tombstones that are available in the *earliestRetainedChangeTime* field in either ***OpenSession*** or ***RequestSession***.

         4. **NOTE:** For growing and channel data objects, a store MUST also delete any relevant change annotations. For more information about change annotations, see Chapter **11 GrowingObject (Protocol 6)** and Chapter◦**19◦ChannelSubscribe (Protocol 21)**.

      ii. Tombstones are used in the workflow for eventual consistency; for more information, see **Appendix: Data Replication and Outage Recovery Workflows**.

      iii. For growing data objects, the store MUST also delete the parts of the growing data object.

   b. MUST perform this **NOTIFICATION BEHAVIOR** (StoreNotification (Protocol 5):

      i. After a delete operation, the store MUST send an ***ObjectDeleted*** notification.

      ii. A delete is an atomic operation; the store MUST perform the delete operation and then send notifications.

      iii. A store MUST send a notification for only the most recent effective state of a data object. So if notifications are queued, and the object is subsequently deleted, the store MAY discard any previous notifications.

   c. MUST respond with one or more ***DeletedDataObjectsResponse*** map response messages. (Section **9.3.5**), which lists the URIs for each of the data objects that was successfully deleted.

      i. For more information on how map response messages work, see Section **3.7.3**.

3. For the URIs it does NOT successfully delete data objects for, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors, such

as ENOT_FOUND (11).

a. For more information on how ***ProtocolException*** messages work with a plural messages, see Section **3.7.3**.

9.2.2 Store: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) some rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter 5. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3**. a. For the list of global capabilities and related behavior, see Section **3.3.2**. 3. Section **9.2.3** identifies the capabilities most relevant to this ETP sub-protocol. If one or more of the defined capabilities is presented by an endpoint, the other endpoint in the ETP session MUST accept it (them) and process the value, and apply them to the behavior as specified in this document. a. Additional details for how to use the capabilities are included below in this table and in Section **9.2.1 Store: Message Sequence.**

Row#	Requirement	Behavior
3.	Message Sequence See Section **9.2.1.**	1. The Message Sequence section above (Section **9.2.1**) describes requirements for the main tasks listed there and also defines required behavior.
4.	Plural messages (which includes maps)	1. This protocol uses plural messages. For detailed rules on handling plural messages (including ProtocolException handling), see Section **3.7.3**.
5.	Requirements for use of PWLS	Practical Well Log Standard (PWLS) is an industry standard stewarded by Energistics. It provides an industry-agreed list of logging tool classes and a hierarchy of measurement properties and applies all known mnemonics to them. For more information, see Section **3.12.7**. 1. If an ETP store supports the WITSML Channel data object, then it MUST support PropertyKind data objects (which are an implementation of PWLS). 2. Endpoints MUST be able to discover property kind data objects (to determine available property kinds) and use the returned property kinds in relevant Discovery, Store and Query operations.
6.	For data objects that exceed an endpoint's WebSocket message size, use the ***Chunk*** message.	1. Some messages in this protocol allow or require a data object to be sent with the message. If the size of the data object (bytes) is too large for the WebSocket message size (which for some WebSocket libraries can be quite small, e.g. 128 kb), an endpoint MAY subdivide the data object and send it in "chunks" using the ***Chunk*** message defined in this protocol. For information on how to handle these binary large objects (BLOBs), see Section **3.7.3.2**. 2. **NOTE:** Use of Chunk messages DOES NOT address an endpoint's MaxDataObjectSize limit. 3. The specific messages in this protocol that may use ***Chunk*** messages are: a. ***GetDataObjectsResponse*** b. ***PutDataObjects***
7.	Notifications	1. This chapter explains events (operations) in Store (Protocol 4) that trigger the store to send notifications, which the store sends using StoreNotification (Protocol 5). However, statements of **NOTIFICATION BEHAVIOR** are here in this chapter, in the context of the detailed explanation of the behavior that triggers the notification. 2. Notification behavior is described here using MUST. However, the store MUST ONLY send notifications IF AND ONLY IF there is a customer subscribed to notifications for an appropriate context (i.e., a context that includes the data object) and the store MUST ONLY send notifications to those customers that are subscribed to appropriate contexts. a. For more information on data object notifications, see Chapter **10 StoreNotification (Protocol 5)**. b. For information on notifications for parts in growing data objects, see Chapter◦**12 GrowingObjectNotification (Protocol 7)**.
8.	**Store Behavior:** Updates to *storeCreated* and *storeLastWrite* fields.	1. Each ***Resource*** in ETP has these two fields: *storeCreated* and *storeLastWrite*. a. These fields appear ONLY on the ***Resource*** NOT on the data object and are used in workflows for eventual consistency between 2 stores. b. For more information about these fields, see Section **3.12.5.1** and their definitions/required format in ***Resource*** (see Section **23.34.11**). 2. For operations in Store (Protocol 4) that ADD a new data object (e.g. ***PutDataObjects***), the store MUST do both of these: a. Set the *storeCreated* field to the time that the data object was added in the store. b. Set the *storeLastWrite* to the same time as *storeCreated*. 3. For operations to data objects that may occur in another protocol that change any data for the data object (e.g., GrowingObject (Protocol 6), which may result in changes to the growing data object header or its parts, or ChannelSubscribe (Protocol 21) where data may be appended to a channel), the store MUST update the *storeLastWrite* field with the time of the change in the store. a. Currently other protocols that trigger updates to these fields include: i. GrowingObject (Protocol 6); see Chapter **11**. ii. ChannelStreaming (Protocol 1); see Chapter **6**. iii. ChannelDataLoad (Protocol 22); see **20**.

Row#	Requirement	Behavior
9.	**Store behavior for data objects that can be "active":** Setting the *activeStatus* field	1. In ETP, channel data objects, growing data objects and other data objects that can be "active" have a field named *activeStatus*. For information about this field and required behavior for setting it to "inactive" related to the ActiveTimeoutPeriod capability, see Section **3.3.2.1**. 2. If a data object that can be "active" has an *activeStatus* of "inactive" and relevant updates are made to data objects in the store, the store MUST set the *activeStatus* to "active". For channel data objects, the relevant changes are adding or changing data points. For growing data objects, the relevant changes are adding or changing parts. For other data objects, see the relevant ML ETP implementation guide. a. For Channel data objects, data points are added or changed using ChannelStreaming (Protocol 1) (see Chapter **6**), or ChannelDataLoad (Protocol 22) (see Chapter **20**). b. For growing data objects, parts are added or changed using GrowingObject (Protocol 6). For more information, see Chapter **11**. c. **NOTIFICATION BEHAVIOR:** When a data object's *activeStatus* field changes, a store MUST send an ***ObjectActiveStatusChanged*** notification message. For more information see Chapter **10 StoreNotification (Protocol 5)**.
10.	**Store Behavior:** Store managed elements and attributes on data objects	Some elements and attributes on Energistics data objects are managed by an ETP store. Examples of these are the index range elements on channels and growing data objects and the GrowingStatus element on growing data objects. The individual MLs define which elements and attributes are store managed. Observe these rules for store managed elements and attributes: **1.** When a customer requests a data object, the store MUST populate these elements and attributes with the correct and current values before returning the data object to the customer. **2.** When a customer puts a data object into a store, the store MUST ignore any values the customer may have provided for these elements and attributes. If the customer provides values for these elements and attributes, the store MUST NOT treat it as an error.
11.	**Store Behavior:** Immutable elements and attributes	Some elements and attributes on Energistics data objects are immutable. That is, the values for these are set when the data object is created, and the values cannot be changed after that. Examples of these are a data object's UUID and the unit of measure for channel data. Observe these rules for immutable elements and attributes: 1. When the customer creates the data object, the store MUST use the values provided by the customer for these elements and attributes. 2. If a customer attempts to update an existing data object and provides different values for immutable elements or attributes, the store MUST reject the update and send error EREQUEST_DENIED (6). 3. If a customer needs to change the values of any immutable elements or attributes, the customer MUST first delete the data object and then recreate it with the correct values.
12.	**Container/contained data objects:** General rules (including optional pruning behavior)	**1. Relevant definitions:** a. Of container and contained objects and related concepts, see Section **9.1.3**. b. Of protocol and data object capabilities, see Section **9.2.3.** 2. A customer MUST limit in a put request the count of data objects contained in each container data object to a store's value for the MaxContainedDataObjectCount data object capability for that specific container data object type. a. For any request that exceeds the store's limit, the store MUST deny the request and send error ELIMIT_EXCEEDED (12). b. In situations where there are multiple nested levels of contained data objects, more than one limit may apply. In the example in Paragraph **7** above, the value for MaxContainedDataObjectCount may be different for logs and channel sets. 3. An endpoint MAY specify on which types of container data objects it allows "pruning operations" to occur by setting the OrphanedChildrenPrunedOnDelete data object capability to true (see Section **9.2.3**).

Row#	Requirement	Behavior
		4. The ***PutDataObjects*** and ***DeleteDataObjects*** messages have a *pruneContainedObjects* Boolean flag, which allows the store to cleanup "orphan" contained objects following an operation on a container. a. For more information on ***PutDataObjects*** operations for container/contained data objects, see Rows **19.** b. For more information on ***DeleteDataObjects*** operations for container/contained data objects, see Rows **25.** 5. For successful pruning operations to occur on a specific data object type, both of these conditions MUST be true: a. The OrphanedChildrenPrunedOnDelete data object capability MUST be set to true. b. The *pruneContainedObjects* Boolean flag on the request message MUST be set to true. 6. A prune operation is a "soft delete" for "garbage collection". The store will only delete those objects that qualify for pruning. a. There may be reasons (e.g., security) unknown to the customers, that a customer cannot see all contained objects or that all contained objects are not deleted. 7. A prune operation is applied to ALL levels of ByValue relationships in the container being acted upon. **EXAMPLE:** A log contains channel sets ByValue, and a channel set contains channels ByValue. If both conditions in Paragraph **5** are true for both logs and channel sets, for an operation on a log, the store MUST cleanup both orphaned channel sets and channels. a. Putting or deleting a data object is an atomic operation. Any pruning caused by a put or delete MUST be included in the atomic put or delete operation. b. The OrphanedChildrenPrunedOnDelete data object capability may be different for different objects. c. The *pruneContainedObjects* is applied to ALL data objects listed in a given request message. If a customer has some data objects it wants "pruned" and others it does not, then the customer MUST send separate requests for each case. 8. If a customer sends a message with the *pruneContainedObjects* flag set to true for a data object type whose OrphanedChildrenPrunedOnDelete data object capability is false, the store MUST reject the entire operation and send error ENOCASCADE_DELETE (4003).
13.	**Get data objects:** General rules	1. For the general requirements and the message sequence for getting data objects from a store, see Section **9.2.1.1.**
14.	**Index metadata:** General rules for channels, channel sets and growing data objects.	1. A growing data object's index metadata MUST be consistent: a. All parts MUST have the same index unit and the same vertical datum. b. The index units and vertical datums in the growing data header MUST match the parts. 2. A channel data object's index metadata MUST be consistent: a. The index units and vertical datums MUST match the channel's index metadata. 3. A channel set data object's index metadata MUST be consistent: a. The index units and vertical datums MUST match the channel set's index metadata. b. The channel set's index metadata MUST match the relevant index metadata in the channels it contains. 4. When sending messages, both the store AND the customer MUST ensure that all index metadata and data derived from index metadata are consistent in all fields in the message, including in XML or JSON object data or part data. a. **EXAMPLE:** The *uom* and *depthDatum* in an ***IndexInterval*** record MUST be consistent with the data object's index metadata. b. **EXAMPLE:** Data object elements related to index values in growing data object headers (e.g., MdMn and MdMx on a WITSML 2.0 Trajectory) and parts (e.g., Md on a WITSML 2.0 TrajectoryStation) MUST be consistent with each other AND the data object's index metadata.

Row#	Requirement	Behavior
		5. A store MUST reject requests with inconsistent index metadata with an appropriate error such as EINVALID_OBJECT (14) or EINVALID_ARGUMENT (5).
15.	**Get data objects:** Additional rules for container/contained objects	1. You MUST follow the general rules for get operations in Section **9.2.1.1** and the general rules for container/contained objects in Row **12**, and the additional requirements listed in this row. 2. A store MUST return contained objects in a container object, as defined by the container data object schema. a. For more information about specific kinds of container objects, consult the relevant ML documentation and companion ETP implementation specification.
16.	**Get data objects:** Additional rules for channels and channel sets	1. You MUST observe the rules in previous row for container/contained objects and these additional rules for ChannelSets and Channels. 2. For a ChannelSet data object, the store MUST return the Channels but NOT the data for each channel. 3. For a Channel data object, the store MUST NOT return the data for the channel.
17.	**Get data objects:** Additional rules for growing data objects	1. You MUST follow the rules specified in Section **9.2.1.1** with these additional requirements for growing data objects. 2. The store MUST return the full growing data object, including its parts, as defined by the growing data object schema. **(NOTE:** The parts of growing data objects are not themselves Energistics data objects. As such, Store (Protocol 4) does not operate directly on parts. Store (Protocol 4) only handles parts of growing data objects when they are included within the body of the growing data object. To operate directly on parts, use GrowingObject (Protocol 6).) a. The store must observe limits specified by its own and the customer's values for the MaxPartSize capability. For more information about how this capability works and required behavior, see Section **3.3.2.5**. b. For more information about specific growing data objects, consult the relevant ML documentation and companion ETP implementation specification. 3. When returning a growing data object, any store-managed elements or attributes in the growing data object header that are populated with information from the growing data object's index metadata MUST be populated consistently with the index metadata. a. **EXAMPLE:** The MdMn and MdMx elements on a WITSML 2.0 Trajectory MUST have the same unit and depth datum as the Md elements on the trajectory's stations.
18.	**Put (insert or update) data objects:** General rules	1. For the general requirements and message sequence for creating/inserting or updating data objects, see Section **9.2.1.2**.
19.	**Put/update data objects:** Additional rules for container/contained data objects	1. You MUST follow the general rules for put operations in Section **9.2.1.2** and the general rules for container/contained objects in Row **12**, and the additional requirements listed in this row. a. If the customer wants to prune orphaned contained data objects, it MUST set the *pruneContainedObjects* field in ***PutDataObjects*** message to true. For all details on how prune operations work, see Row **12** above. b. Rule **2** in Section **9.2.1.2** MUST be applied to the <u>container</u> object only—NOT to the contained objects. **EXAMPLE:** If the customer request is to put a ChannelSet with 6 Channels, the store MUST replace ONLY the ChannelSet (NOT each of the Channels in the set). c. The notification behavior in Rule 2.**e** MUST be applied for the <u>container</u> object only. NOTE: Additional items below in this row explain additional notifications that MAY be sent for contained objects. 2. A customer MUST limit in a put request, the count of data objects contained in each container data object to a store's value for MaxContainedDataObjectCount data object capability for that specific container data object type. a. For any request that exceeds the store's limit, the store MUST deny the request and send error ELIMIT_EXCEEDED (12).

Row#	Requirement	Behavior
		For the <u>contained</u> objects in a put container operation, a store MUST observe these rules: 3. If a contained object already exists in the store and was not previously contained in the container data object: a. The store MUST link (join) it to the container ("link" as appropriate for the underlying store). b. The store MUST NOT update any data elements (including *LastChanged*) on the contained objects; the store MUST ignore any changes that may have been sent for the contained object. c. **NOTIFICATION BEHAVIOR** (StoreNotification (Protocol 5) (reminder: Row **7**): The store MUST send an ***ObjectChanged*** notification with an *objectChangeKind* of "joined". 4. If a contained object already exists in the store and is already contained in the container: a. The store MUST NOT update any data fields (including the *lastChanged* field) on the contained objects; the store MUST ignore any changes that may have been sent for the contained object. 5. If the contained object DOES NOT exist in the store: a. The store MUST add the object and MUST link it to the container. b. **NOTIFICATION BEHAVIOR** (StoreNotification (Protocol 5) (reminder: Row **7**): The store MUST send 2 ***ObjectChanged*** notifications: one for "insert" of the new contained data object and one for "joined" to its container. 6. A put operation may have an "implied delete". **EXAMPLE:** If an existing channel set originally had 6 channels, and the channel set is put again with only 5 channels, then the channel that was not included in the latest put is removed from the channel set. The store MUST observe this behavior: a. **NOTIFICATION BEHAVIOR** (StoreNotification (Protocol 5) (reminder: Row **7**): It MUST send an ***ObjectChanged*** notification with a type of "unjoined". b. If *pruneContainedObjects* is set to true in the ***PutDataObjects*** message, the store MUST follow prune behavior as specified in Row **12**. i. **NOTIFICATION BEHAVIOR** (StoreNotification (Protocol 5): If the contained object was pruned, the store MUST send an ***ObjectDeleted*** notification. ii. If the contained data object was not pruned, it is skipped (not included in the response). c. The ***PutDataObjectsResponse*** message contains a ***PutResponse*** record which contains additional information for operations on the contained objects in the container. For more information, see Section **23.34.9**. 7. If the contained object is itself a container object, the store MUST repeat Steps 3 through 6 (implied delete) on the data objects contained within it. That is, apply the put container operation recursively to nested container objects such as Log and Channel Set. 8. At completion of the operation, all contained objects in the ***PutDataObjects*** message MUST be linked or unlinked (depending on specifics of the operation as described in the steps above in this row) to the specified container. 9. Possible errors: a. The entire put operation MUST succeed. If any part of the put operation fails, the store MUST do the following: i. Send error EREQUEST_DENIED (06) (for more information about using ***ProtocolException*** messages with maps, see Section **3.7.3**.) ii. Roll back all parts of the operation (**EXAMPLE:** If putting a ChannelSet with 100 Channels, and one Channel cannot be put, then the entire operation MUST be rolled back.) **RECOMMENDED:** The ***ProtocolException*** message include information that indicates the object(s) in the set that failed. b. If the *pruneContainedObjects* flag on the ***PutDataObjects*** message is set to true, but the OrphanedChildrenPrunedOnDelete capability for the data object is false, the store MUST send error ENOCASCADE_DELETE (4003).
20.	**Put/update data objects:** Additional rules for channels and channel sets	1. You MUST observe the rules in previous row for container/contained objects and these additional rules for ChannelSets and Channels 2. If the data object being put is a channel or channel set, and the request exceeds the store's value for MaxSecondaryIndexCount data object capability, the store

Row#	Requirement	Behavior
		MUST deny the request with ELIMIT_EXCEEDED (12). a. If a store allows write/put operations, it MUST support at least 1 secondary index. 3. When putting a channel or channel set, the index range elements MUST be consistent with the channel or channel set's index metadata. a. The store MUST reject any channel or channel set with inconsistent index range elements send error EINVALID_OBJECT (14). For the definition of compatible index metadata, see Section **7.1.1**. 4. When putting a channel set, the customer MUST ensure that the channel set's index metadata is compatible with the index metadata for all channels in the channel set. a. The store MUST reject any channel set with incompatible index metadata and send error EINVALID_OBJECT (14). For the definition of compatible index metadata, see Section **7.1.1**.
21.	**Put/update data objects:** Additional rules for growing data objects	1. You MUST follow the rules specified in Section **9.2.1.2** with these additional requirements for growing data objects. 2. A customer MAY use the ***PutDataObjects*** message to put (insert, add) a NEW growing data object (new = one that does not exist in the store) including its parts, in a single operation (which is new behavior in ETP v1.2). a. Store (Protocol 4) only handles growing data object parts when the parts are themselves data objects or when creating a new growing data object and parts are included within the body of the growing data object. To operate directly on parts, use GrowingObject (Protocol 6). b. The customer must observe limits specified by the store's value for the MaxPartSize capability. For more information about how this capability works and required behavior, see Section **3.3.2.5**. c. Alternatively, an endpoint MAY use GrowingObject (Protocol 6) to add a new growing data object and its parts, but use of that protocol requires a couple of messages/operations (see Chapter **11**). d. When creating a growing data object that includes parts, the customer MUST ensure that ALL parts have consistent index metadata for the elements on the parts that are used as the part index values or index range values. The parts MUST also be consistent with any index metadata related to the parts included in the growing object header (**EXAMPLE:** MdMn and MdMx on a WITSML 2.0 Trajectory). i. The store MUST reject any growing data object that has parts with inconsistent index metadata or parts that have index metadata that is inconsistent with the header and send error EINVALID_OBJECT (14). 3. For successful put operations, a store MUST do the following in response: a. Add the new growing data object(s). b. If parts are included and the parts themselves are NOT data objects, add the parts to the growing data object. c. If parts are included and the parts themselves ARE also data objects: i. Create any parts that do not exist. ii. Update any previously existing parts with the new content. iii. Link all parts to the growing data object. d. Set the growing data object's *activeStatus* flag to "inactive" (default when a new growing data object or channel is added to a store). e. Send the ***PutDataObjectsResponse*** message, which lists the growing data objects that were successfully added. f. **NOTIFICATION BEHAVIOR:** (reminder: Row **7**) in StoreNotification (Protocol 5), the store MUST send an ***ObjectChanged*** notification message for the growing data object. i. If any parts that are themselves also data objects previously existed and were updated, the store MUST also send ***ObjectChanged*** and notifications for these parts. ii. If any previously parts that are themselves data objects are also contained in other growing data objects, the store MUST also send

Row#	Requirement	Behavior
		"joined" ***ObjectChanged*** notifications AND ***PartsChanged*** notifications for these parts. 4. All UPDATES to existing growing data objects and their parts MUST be done using GrowingObject (Protocol 6). a. If a customer tries to update an existing growing data object using the ***PutDataObjects*** message, the store MUST send error EUPDATEGROWINGOBJECT_DENIED (23).
22.	**Put/update data objects:** Additional rules for data objects that are BOTH growing data objects AND container data objects	1. For data objects that are both growing data objects AND container data objects (such as the WITSML v2.0 InterpretedGeology object), the additional rules for both growing data objects and container data objects apply, but the growing data object rules take precedence. a. When creating a new growing data object that contains previously existing parts, the previously existing parts MUST be both linked AND updated. i. **NOTIFICATION BEHAVIOR:** (reminder: Row **7**) in StoreNotification (Protocol 5), for previously existing parts that are linked, the store MUST send an ***ObjectChanged*** notification message, with an ***ObjectChangeKind*** of "joined". b. When creating a new growing data object, a customer MUST also limit the count of parts in the growing data object to the store's relevant value for MaxContainedDataObjectCount. c. **EXAMPLE:** Putting a new InterpretedGeology data object that contains a combination of new and existing InterpretedGeologyInterval contained data objects/parts WILL SUCCEED. i. The new InterpretedGeology data object will be created in the store; any new InterpretedGeologyInterval data objects will be created in the store; any previously existing InterpretedGeologyInterval data objects that are included in the new InterpretedGeology data object will be updated with the new content; the InterpretedGeology data object will be linked with all intervals it contains; and an ***ObjectChanged*** notification message, with an ***ObjectChangeKind*** of "joined" will be sent for the previously existing intervals. d. **EXAMPLE:** Putting an existing InterpretedGeology data object with a different set of InterpretedGeologyInterval contained data objects/parts WILL FAIL because Store (Protocol 4) does not support updates to growing data objects. i. Because the put fails, no changes will be made to the InterpretedGeologyInterval data objects/parts contained in the InterpretedGeology interval and no linking or unlinking will happen.
23.	**Put/update data objects:** Additional rules for data objects that are BOTH growing data object parts AND contained data objects	1. Observe these rules for data objects that are both growing data object parts AND contained data objects (such as the WITSML v2.0 InterpretedGeologyInterval object): a. You MUST follow the general rules for put operations in Section **9.2.1.2**. b. **NOTIFICATION BEHAVIOR:** If the data objects are included in any growing data objects, in GrowingObjectNotification (Protocol 7), the store MUST send a ***PartsChanged*** notification message.
24.	**Delete data objects:** General rules (including growing data objects)	1. For the general requirements and message sequence for deleting one or more data objects, see Section **9.2.1.3**.
25.	**Delete data Objects:** Additional rules for container and contained data objects	1. You MUST follow the general rules for delete operations in Section **9.2.1.3** and the general rules container/contained objects in Row **12**, and the additional requirements listed in this row. a. Step **2** in Section **9.2.1.3** applies to the <u>container</u> object only (not the contained objects). For <u>contained</u> objects, observe these rules for a delete container data object operation: 2. If a customer wants to prune orphan contained data objects, it MUST set the *pruneContainedObjects* flag to true in the ***DeleteDataObjects*** message. a. The store MUST delete orphan contained objects as described in Row **12.** **NOTIFICATION BEHAVIOR** (StoreNotification (Protocol 5) (reminder: Row

Row#	Requirement	Behavior
		7): The store MUST send an ***ObjectDeleted*** notification for each deleted container data object and for pruned contained data objects. 3. The ***DeleteDataObjectsResponse*** message MUST also list any contained data objects that were pruned/deleted (in addition to the container objects that were in the delete request). 4. To remove contained objects from a container (but NOT delete the data object), a customer MUST send a ***PutDataObjects*** message that lists only those contained objects that must be included in the set. For more information, see Row **19**. For contained objects, observe these rules for a delete contained data object operation: 5. The store MUST unlink (unjoin) it to from any container data object it is contained in. 6. The store MUST update *storeLastWrite* on any container data object it is contained in. 7. **NOTIFICATION BEHAVIOR** (StoreNotification (Protocol 5) (reminder: Row **7**): The store MUST send an ***ObjectChanged*** notification with an *objectChangeKind* of "unjoined".
26.	**Delete data objects:** Additional rules for data objects that are BOTH growing data objects AND container data objects	1. For data objects that are both growing data objects AND container data objects (such as the WITSML v2.0 InterpretedGeology object), these additional rules for container data objects apply: a. When deleting a growing data object, the parts will only be deleted if pruning is requested and supported for the growing data object type AND the parts are not also included in other data objects (i.e., deleting the growing data object will orphan the parts). b. **EXAMPLE:** Deleting an InterpretedGeology data object that contains InterpretedGeologyIntervals with *pruneContainedObjects* = false will NOT delete the intervals: i. The InterpretedGeology data object will be deleted. ii. The InterpretedGeologyInterval data objects it contained will still exist in the store. c. **EXAMPLE:** Deleting an InterpretedGeology data object that contains two InterpretedGeologyIntervals, interval A and interval B, where interval A is contained in another InterpretedGeology data object and interval B is ONLY contained in the InterpretedGeology data object that is being deleted, with *pruneContainedObjects* = true will ONLY delete the orphaned interval: i. The InterpretedGeology data object will be deleted. ii. Interval A will NOT be deleted (because it is still contained in another data object). iii. Interval B WILL be deleted (because it is orphaned).
27.	**Delete data objects:** Additional rules for data objects that are BOTH growing data object parts AND contained data objects	1. Observe these rules for data objects that are both growing data object parts AND contained data objects (such as the WITSML v2.0 InterpretedGeologyInterval object): a. You MUST follow the general rules for delete operations in Section **9.2.1.3.** b. **NOTIFICATION BEHAVIOR:** If the data objects are included in any growing data objects, in GrowingObjectNotification (Protocol 7), the store MUST send a ***PartsDeleted*** notification message.

9.2.3 Store: Capabilities

The table below lists key capabilities for this protocol. The protocol capabilities are all defined here. For this protocol, two particularly crucial endpoint capabilities are defined here.

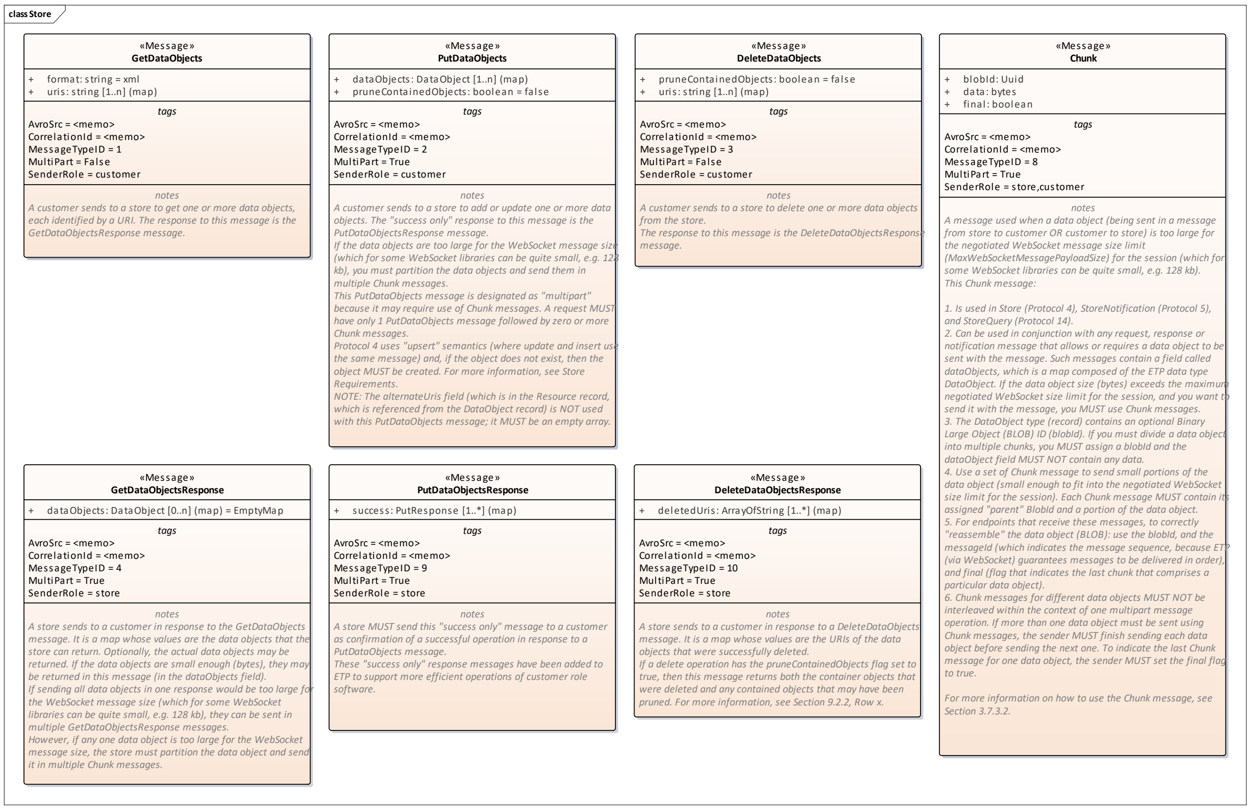
- For protocol-specific behavior relating to using these capabilities in this protocol, see Sections **9.2.1,◦Store: Message Sequence** and **9.2.2**, **Store: General Requirements**.
- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section **3.3**.

Store (Protocol 4): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of all endpoint capabilities, see Section 3.3.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section 3.3.2. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section 4.3, **How a Client Establishes a WebSocket Connection to an ETP Server**.			
ChangeRetentionPeriod: The minimum time period in seconds that a store retains the canonical URI of a deleted data object and any change annotations for channels and growing data objects. **RECOMMENDATION:** This period should be as long as is feasible in an implementation. When the period is shorter, the risk is that additional data will need to be transmitted to recover from outages, leading to higher initial load on sessions.	Long	Seconds **Value units:** <number of seconds>	**Default:** 86,400 **MIN:** 86,400
MaxPartSize: The maximum size in bytes of each data object part allowed in a standalone message or a complete multipart message. Size in bytes is the total size in bytes of the uncompressed string representation of the data object part in the format in which it is sent or received. Applies to get and put operations of growing data objects.	Long	byte <number of bytes>	Min: 10,000 bytes
Data Object Capabilities (For definitions of all data object capabilities, see Section 3.3.4)			
SupportsGet, SupportsPut and **SupportsDelete** For definitions and usage rules for each of these data object capabilities, see Section 3.3.4.			
ActiveTimeoutPeriod: (This is also an endpoint capability.) The minimum time period in seconds that a store keeps the active status (*activeStatus* field in ETP) for a data object as "active" after the most recent update causing the data object's active status to be set to true. For growing data objects, this is any change to its parts. For channels, this is any change to its data points. This capability can be set for an endpoint and/or for a data object. A data object-specific value overrides an endpoint-specific value.	Long	second <number of seconds>	**Default:** 3,600 **MIN:** 60 seconds
MaxContainedDataObjectCount: The maximum count of contained data objects allowed in a single instance of the data object type that the capability applies to. **EXAMPLE:** If this capability is set to 2000 for a ChannelSet, then the ChannelSet may contain a maximum of 2000 Channels.	Long	Count <count of objects>	**MIN:** Should be specified per domain
OrphanedChildrenPrunedOnDelete: For a container data object type (i.e., a data object type that may contain other data objects ByValue), this capability indicates whether contained data objects	Boolean	N/A	**Default:** false

Store (Protocol 4): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
that are orphaned as a result of an operation on its container data object may be deleted (pruned). NOTES: 1. Both delete or put operations of a container data object may result in contained data objects being orphaned. 2. For successful pruning operations to occur on a specific data object type, both of these conditions MUST be true: a. This capability MUST be set to true. b. The *pruneContainedObjects* Boolean flag on the request message MUST be set to true. **EXAMPLE:** If this capability is set to true for ChannelSet, and on a ***DeleteDataObjects*** message for a ChannelSet the *pruneContainedObjects* Boolean flag is set to true, and (after the container is deleted) a Channel in that ChannelSet belongs to no other ChannelSets, then that "orphaned" Channel is also deleted.			
MaxSecondaryIndexCount: The maximum count of secondary indexes allowed in a single instance of the data object type that the capability applies to, which may be Channel or ChannelSet.	long	Count <count of secondary indexes>	**MIN:** 1 **Default:** 1
Protocol Capabilities			
MaxDataObjectSize: (This is also an endpoint capability and a data object.) The maximum size in bytes of a data object allowed in a complete multipart message. Size in bytes is the size in bytes of the uncompressed string representation of the data object in the format in which it is sent or received. This capability can be set for an endpoint, a protocol, and/or a data object. If set for all three, here is how they generally work: • An object-specific value overrides an endpoint-specific value. • A protocol-specific value can further lower (but NOT raise) the limit for the protocol. **EXAMPLE:** A store may wish to generally support sending and receiving any data object that is one megabyte or less with the exceptions of Wells that are 100 kilobytes or less and Attachments that are 5 megabytes or less. A store may further wish to limit the size of any data object sent as part of a notification in StoreNotification (Protocol 5) to 256 kilobytes.	long	byte <number of bytes>	**MIN:** 100,000 bytes

9.3 Store: Message Schemas

This section provides a figure that displays all messages defined in Store (Protocol 4). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.



«Message»
GetDataObjects

+ format: string = xml
+ uris: string [1..n] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to get one or more data objects, each identified by a URI. The response to this message is the GetDataObjectsResponse message.

«Message»
PutDataObjects

+ dataObjects: DataObject [1..n] (map)
+ pruneContainedObjects: boolean = false

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 2
MultiPart = True
SenderRole = customer

notes
A customer sends to a store to add or update one or more data objects. The "success only" response to this message is the PutDataObjectsResponse message.
If the data objects are too large for the WebSocket message size (which for some WebSocket libraries can be quite small, e.g. 128 kb), you must partition the data objects and send them in multiple Chunk messages.
This PutDataObjects message is designated as "multipart" because it may require use of Chunk messages. A request MUST have only 1 PutDataObjects message followed by zero or more Chunk messages.
Protocol 4 uses "upsert" semantics (where update and insert use the same message) and, if the object does not exist, then the object MUST be created. For more information, see Store Requirements.
NOTE: The alternateUris field (which is in the Resource record, which is referenced from the DataObject record) is NOT used with this PutDataObjects message; it MUST be an empty array.

«Message»
DeleteDataObjects

+ pruneContainedObjects: boolean = false
+ uris: string [1..n] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 3
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to delete one or more data objects from the store.
The response to this message is the DeleteDataObjectsResponse message.

«Message»
Chunk

+ blobId: Uuid
+ data: bytes
+ final: boolean

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 8
MultiPart = True
SenderRole = store,customer

notes
A message used when a data object (being sent in a message from store to customer OR customer to store) is too large for the negotiated WebSocket message size limit (MaxWebSocketMessagePayloadSize) for the session (which for some WebSocket libraries can be quite small, e.g. 128 kb).
This Chunk message:

1. Is used in Store (Protocol 4), StoreNotification (Protocol 5), and StoreQuery (Protocol 14).
2. Can be used in conjunction with any request, response or notification message that allows or requires a data object to be sent with the message. Such messages contain a field called dataObjects, which is a map composed of the ETP data type DataObject. If the data object size (bytes) exceeds the maximum negotiated WebSocket size limit for the session, and you want to send it with the message, you MUST use Chunk messages.
3. The DataObject type (record) contains an optional Binary Large Object (BLOB) ID (blobId). If you must divide a data object into multiple chunks, you MUST assign a blobId and the dataObject field MUST NOT contain any data.
4. Use a set of Chunk message to send small portions of the data object (small enough to fit into the negotiated WebSocket size limit for the session). Each Chunk message MUST contain its assigned "parent" BlobId and a portion of the data object.
5. For endpoints that receive these messages, to correctly "reassemble" the data object (BLOB): use the blobId, and the messageId (which indicates the message sequence, because ETP (via WebSocket) guarantees messages to be delivered in order), and final (flag that indicates the last chunk that comprises a particular data object).
6. Chunk messages for different data objects MUST NOT be interleaved within the context of one multipart message operation. If more than one data object must be sent using Chunk messages, the sender MUST finish sending each data object before sending the next one. To indicate the last Chunk message for one data object, the sender MUST set the final flag to true.

For more information on how to use the Chunk message, see Section 3.7.3.2.

«Message»
GetDataObjectsResponse

+ dataObjects: DataObject [0..n] (map) = EmptyMap

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 4
MultiPart = True
SenderRole = store

notes
A store sends to a customer in response to the GetDataObjects message. It is a map whose values are the data objects that the store can return. Optionally, the actual data objects may be returned. If the data objects are small enough (bytes), they may be returned in this message (in the dataObjects field).
If sending all data objects in one response would be too large for the WebSocket message size (which for some WebSocket libraries can be quite small, e.g. 128 kb), they can be sent in multiple GetDataObjectsResponse messages.
However, if any one data object is too large for the WebSocket message size, the store must partition the data object and send it in multiple Chunk messages.

«Message»
PutDataObjectsResponse

+ success: PutResponse [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 9
MultiPart = True
SenderRole = store

notes
A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutDataObjects message.
These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

«Message»
DeleteDataObjectsResponse

+ deletedUris: ArrayOfString [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 10
MultiPart = True
SenderRole = store

notes
A store sends to a customer in response to a DeleteDataObjects message. It is a map whose values are the URIs of the data objects that were successfully deleted.
If a delete operation has the pruneContainedObjects flag set to true, then this message returns both the container objects that were deleted and any contained objects that may have been pruned. For more information, see Section 9.2.2, Row x.

Figure 18: Store: message schemas

9.3.1 Message: GetDataObjects

A customer sends to a store to get one or more data objects, each identified by a URI. The response to this message is the GetDataObjectsResponse message.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uris	General ETP map where each value MUST be the URI of a data object to be retrieved. If both endpoints support alternate URIs for the session, these MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	string	1	n
format	Specifies the format (e.g., XML or JSON) in which you want to receive data for the requested data objects. This MUST be a format that was negotiated when establishing the session.	string	1	1

Field Name	Description	Data Type	Min	Max
	Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.			

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Store",
     "name": "GetDataObjects",
     "protocol": "4",
     "messageType": "1",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "uris",
             "type": { "type": "map", "values": "string" }
         },
         { "name": "format", "type": "string", "default": "xml" }
     ]
}
```

9.3.2 Message: PutDataObjects

A customer sends to a store to add or update one or more data objects. The "success only" response to this message is the PutDataObjectsResponse message.

If the data objects are too large for the WebSocket message size (which for some WebSocket libraries can be quite small, e.g. 128 kb), you must partition the data objects and send them in multiple Chunk messages.

This ***PutDataObjects*** message is designated as "multipart" because it may require use of ***Chunk*** messages. A request MUST have only 1 ***PutDataObjects*** message followed by zero or more ***Chunk*** messages.

Protocol 4 uses "upsert" semantics (where update and insert use the same message) and, if the object does not exist, then the object MUST be created. For more information, see Store Requirements.

NOTE: The *alternateUris* field (which is in the Resource record, which is referenced from the DataObject record) is NOT used with this ***PutDataObjects*** message; it MUST be an empty array.

Message Type ID: 2

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: True

Sent by: customer

Field Name	Description	Data Type	Min	Max
dataObjects	General ETP map of DataObject records, each of which contains the data for each data object in the request, including each one's URI. The URIs in the ***Resource*** records MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.	DataObject	1	n
pruneContainedObjects	For this field to work as described, the OrphanedChildrenPrunedOnDelete data object capability for the type of data object MUST be	boolean	1	1

Field Name	Description	Data Type	Min	Max
	true. For more information, see Section **9.2.2**, Row **12**. Boolean. Default = false. • If true, the store deletes any data objects contained ByValue (that is, data objects contained in the data object identified by the URI, such as channels in a channel set) that are NOT contained by any other data objects. This prune will carry down ALL ByValue relationships of an object. **EXAMPLE:** If the request is to put a channel set, and pruneContainedObjects is true, the store will delete any channels that are not contained in any other channel sets after the operation is complete (e.g., if you update/replace a channel set that effectively deletes a channel from the set) . In other words, the store deletes any "orphan" channels. • If false, only the data objects (identified by the URIs in the ***DataObject*** record) are affected/updated; any contained data objects are not affected. **NOTE:** This flag is applied to ALL data objects listed in a given ***PutDataObjects*** request message. If a customer has some data objects it wants "pruned" and others it does not, then the customer MUST send separate requests for each case.			

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Store",
     "name": "PutDataObjects",
     "protocol": "4",
     "messageType": "2",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "dataObjects",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.Object.DataObject" }
         },
         { "name": "pruneContainedObjects", "type": "boolean", "default": false }
     ]
}
```

9.3.3 Message: PutDataObjectsResponse

A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutDataObjects message.

These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

Message Type ID: 9

Correlation Id Usage: MUST be set to the *messageId* of the ***PutDataObjects*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
success	• For non-container data objects, the map value MUST an empty ***PutResponse*** record (Section **23.34.9**), which has all arrays set to empty arrays. • For contained data objects, the map value MUST be a ***PutResponse*** record with the arrays populated appropriately. • For more information about container/contained data objects, see Section **9.2.2**, Row **19**.	PutResponse	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Store",
     "name": "PutDataObjectsResponse",
     "protocol": "4",
     "messageType": "9",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "success",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.Object.PutResponse" }
         }
     ]
}
```

9.3.4 Message: DeleteDataObjects

A customer sends to a store to delete one or more data objects from the store.

The response to this message is the DeleteDataObjectsResponse message.

Message Type ID: 3

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uris	General ETP map where each value MUST be the URI of a data object to be deleted. These MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.	string	1	n
pruneContainedObjects	Boolean. Default = false. • If true, the store deletes any data objects contained ByValue (that is, data objects contained in the data object identified by the URI, such as channel sets in log and channels in a channel set) that are NOT contained by any other data objects. It does this for ALL ByValue relationships in the data	boolean	1	1

	object. EXAMPLE: If the request is to delete a log, and pruneContainedObjects is true, the store will delete any channel sets that are not contained in any other logs, AND any channels that are not contained in any other channel sets. In other words, the store deletes any "orphan" channel sets and channels. • If false, only the data objects (identified by the URIs) are deleted; any contained data objects are not deleted. **NOTE:** This flag is applied to ALL data objects listed in a given DeleteDataObjects request message. If a customer has some data objects it wants "pruned" and others it does not, then the customer MUST send separate requests for each case.			

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Store",
     "name": "DeleteDataObjects",
     "protocol": "4",
     "messageType": "3",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "uris",
             "type": { "type": "map", "values": "string" }
         },
         { "name": "pruneContainedObjects", "type": "boolean", "default": false }
     ]
}
```

9.3.5 Message: DeleteDataObjectsResponse

A store sends to a customer in response to a DeleteDataObjects message. It is a map whose values are the URIs of the data objects that were successfully deleted.

If a delete operation has the *pruneContainedObjects* flag set to true, then this message returns both the container objects that were deleted and any contained objects that may have been pruned. For more information, see Section **9.2.2**, Row **12**.

Message Type ID: 10

Correlation Id Usage: MUST be set to the *messageId* of the ***DeleteDataObjects*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
deletedUris	A general ETP map where each value is an array of URIs containing the URI of a data object from the request that was successfully deleted and the URIs of any data objects that were pruned when deleting the object from the request.	ArrayOfString	1	*

Field Name	Description	Data Type	Min	Max
	These MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.			

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Store",
     "name": "DeleteDataObjectsResponse",
     "protocol": "4",
     "messageType": "10",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "deletedUris",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.ArrayOfString" }
         }
     ]
}
```

9.3.6 Message: GetDataObjectsResponse

A store sends to a customer in response to the GetDataObjects message. It is a map whose values are the data objects that the store can return. Optionally, the actual data objects may be returned. If the data objects are small enough (bytes), they may be returned in this message (in the *dataObjects* field).

If sending all data objects in one response would be too large for the WebSocket message size (which for some WebSocket libraries can be quite small, e.g. 128 kb), they can be sent in multiple ***GetDataObjectsResponse*** messages.

However, if any one data object is too large for the WebSocket message size, the store must partition the data object and send it in multiple Chunk messages.

Message Type ID: 4

Correlation Id Usage: MUST be set to the *messageId* of the ***GetDataObjects*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
dataObjects	General ETP map of DataObject records, one each for the data objects to that the store could return. The URIs in the ***Resource*** records MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.	DataObject	0	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Store",
     "name": "GetDataObjectsResponse",
     "protocol": "4",
     "messageType": "4",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "dataObjects",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.Object.DataObject" }, "default": {}
         }
     ]
}
```

9.3.7 Message: Chunk

A message used when a data object (being sent in a message from store to customer OR customer to store) is too large for the negotiated WebSocket message size limit (MaxWebSocketMessagePayloadSize) for the session (which for some WebSocket libraries can be quite small, e.g. 128 kb).

This ***Chunk*** message:

1. Is used in Store (Protocol 4), StoreNotification (Protocol 5), and StoreQuery (Protocol 14).
2. Can be used in conjunction with any request, response or notification message that allows or requires a data object to be sent with the message. Such messages contain a field called *dataObjects*, which is a map composed of the ETP data type DataObject. If the data object size (bytes) exceeds the maximum negotiated WebSocket size limit for the session, and you want to send it with the message, you MUST use ***Chunk*** messages.
3. The ***DataObject*** type (record) contains an optional Binary Large Object (BLOB) ID (*blobId*). If you must divide a data object into multiple chunks, you MUST assign a *blobId* and the *dataObject* field MUST NOT contain any data.
4. Use a set of ***Chunk*** message to send small portions of the data object (small enough to fit into the negotiated WebSocket size limit for the session). Each ***Chunk*** message MUST contain its assigned "parent" BlobId and a portion of the data object.
5. For endpoints that receive these messages, to correctly "reassemble" the data object (BLOB): use the *blobId*, and the *messageId* (which indicates the message sequence, because ETP (via WebSocket) guarantees messages to be delivered in order), and *final* (flag that indicates the last chunk that comprises a particular data object).
6. ***Chunk*** messages for different data objects MUST NOT be interleaved within the context of one multipart message operation. If more than one data object must be sent using ***Chunk*** messages, the sender MUST finish sending each data object before sending the next one. To indicate the last ***Chunk*** message for one data object, the sender MUST set the *final* flag to true.

For more information on how to use the ***Chunk*** message, see Section **3.7.3.2**.

Message Type ID: 8

Correlation Id Usage: MUST be set to the *messageId* of the ***GetDataObjects*** or ***PutDataObjects*** message that resulted in the assignment of a *blobId* and this ***Chunk*** message being created.

Multi-part: True

Sent by: store,customer

Field Name	Description	Data Type	Min	Max
blobId	The BLOB ID assigned by an endpoint when a data object being sent in a request, response or notification message must be subdivided into multiple chunks. Each ***Chunk*** message that comprises a BLOB must contain the *blobId* of its "parent" BLOB. The *blobId*:	Uuid	1	1

Field Name	Description	Data Type	Min	Max
	• is entered in the DataObject record referenced in the *dataObjects* field of the request or response message. • Must be of type ***Uuid*** (Section **23.6**).			
data	The data that comprises a chunk (portion) of the data object/BLOB.	bytes	1	1
final	Flag to indicate that this the final message of a set of ***Chunk*** messages that comprise one particular data object.	boolean	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Store",
     "name": "Chunk",
     "protocol": "4",
     "messageType": "8",
     "senderRole": "store,customer",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         { "name": "blobId", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         { "name": "data", "type": "bytes" },
         { "name": "final", "type": "boolean" }
     ]
}
```

10 StoreNotification (Protocol 5)

ProtocolID: 5

Defined Roles: store, customer

StoreNotification (Protocol 5) allows store customers to subscribe to and receive notifications of changes to data objects in the store, in an event-driven manner, from events (operations) that occur in Store (Protocol 4). Customers can choose to receive notifications with the changed data object OR only notifications of change—then based on the change, can decide whether or not to get the full data object.

Customers subscribe to changes within a given context (defined, in part, by a URI) in the store (**EXAMPLE:** A context might be all changes that occur in a specific well). The store provides notifications to the customer—only while the session is valid—of additions, changes, and deletions in the specified context. Additionally, this protocol contains a message for so-called "unsolicited" subscriptions (subscriptions that a store may automatically create for a customer) to support new workflows.

NOTE: Notification messages are a "fire and forget" operation. They are a reliable way for a store to inform a customer that data has changed, which is useful for typical customer applications, such as visualizations, calculations and data synchronization tools. However, notification messages are not a reliable way for the store to ensure the customer successfully receives and persists the changed data. If a data store needs to ensure that another data store is eventually consistent with it, the preferred workflow is for the data store to instead act as a store customer using the push workflow to deliver data to the other data store as described in **Appendix: Data Replication and Outage Recovery Workflows** (Section **26.4**).

Other ETP sub-protocols that may be used with StoreNotification (Protocol 5):

- The events that trigger notifications in this protocol happen in Store (Protocol 4). Some of the details of operations that trigger notifications are explained in Chapter **9**.
  - **NOTE:** Use of the ***PutGrowingDataObjectsHeader*** message in GrowingObject (Protocol 6) creates or updates the header information of a data object, so operations using that message trigger notifications in this protocol, StoreNotification (Protocol 5).
- To receive notifications for changes to the parts of one growing data object, ETP has similar protocols: GrowingObject (Protocol 6) where the event/operations occur and GrowingObjectNotification (Protocol 7), where customers can subscribe to receive notifications about operations on/to the parts within the context of one growing data object. For information on operations and notifications related to parts of a growing data object, see Chapters **11** and **12**.

  For data objects that are both growing data objects AND container data objects (i.e., where the parts are themselves also data objects), other operations in GrowingObject (Protocol 6) will also trigger StoreNotification (Protocol 5) messages.

This chapter includes main sections for:

- Key ETP concepts that are important to understanding how this protocol is intended to work (see Section **10.1**).
- Required behavior, which includes:
  - Description of the message sequence for main tasks, along with required behavior, use of capabilities, and possible errors (see Section **10.2.1**).
  - Other functional requirements (not covered in the message sequence) including use of additional endpoint, data object, and protocol capabilities for preventing and protecting against aberrant behavior (see Section **10.2.2**).
  - Definitions of the endpoint, data object, and protocol capabilities used in this protocol (see Section **10.2.3**).

- Sample schemas of the messages defined in this protocol (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes documentation for each field in a schema (see Section **10.3**).

10.1 StoreNotification: Key Concepts

This section explains concepts that are important to understanding how StoreNotification (Protocol 5) works.

10.1.1 Definitions

This section defines terms for this protocol.

Term	**Definition**
Subscription	We're all familiar with the concept of a video or audio streaming subscription or a magazine subscription, which is the action of making or agreeing to make an advance payment in order to receive or participate in something. In the context of ETP, a subscription is an agreement to receive notifications of events or operations, e.g., adds, deletes or updates of data objects that happen in Store (Protocol 4). Subscriptions are created and notifications sent using StoreNotification (Protocol 5). **NOTE:** Subscriptions work similarly for parts in growing data objects. That is, subscriptions are created and notifications about changes to parts in a growing data object are sent in GrowingObjectNotification (Protocol 7) for changes that happen as result of actions in GrowingObject (Protocol 6). Subscriptions can be established in these main ways: 1. A customer can create one or more subscriptions using the ***SubscribeNotifications*** message. 2. A store can automatically create a subscription for a customer. This is referred to as an “unsolicited subscription”. See the row below.
Unsolicited subscription	Subscriptions created by the store on behalf of the customer, usually based on business agreements or other information exchanged out of band of an ETP session. **EXAMPLE:** In some newer workflows, operators want to automatically create subscriptions for contracted data providers, based on business agreements (contracts) executed outside of ETP. When a contracted data provider connects to the operator's data store, the data provider will automatically be subscribed to notifications for an appropriate context, e.g., a well, wellbore, etc. as agreed in a contract. When the data provider connects to the operator's system, it automatically receives ***UnsolicitedStoreNotifications*** messages. For more information about these workflows in the drilling domain, see the *WITSML v2.0 for ETP v1.2 Implementation Specification*.

10.1.2 Data Model as Graph

The messages in StoreNotification (Protocol 5) have been developed to work with data models as graphs. When understood and used properly, this graph approach allows customers to specify precisely and in a single request the desired set of objects to monitor for notifications, thereby reducing traffic on the wire.

- For general definition of a graph, how it works, and key concepts and how they are used as inputs, see Section **8.1.1**.
- For the details of how to create a subscription to receive notifications, see Section **10.2.1.1**.

10.1.3 Handling Binary Large Objects (BLOBs) in ETP

Some messages in this protocol allow or require a data object to be sent with the message. If the size of the data object (bytes) is too large for the WebSocket message size (which for some WebSocket libraries can be quite small, e.g. 128 kb), you must subdivide the data object and send it in "chunks" (using the Chunk.avsc message). For information on how to handle these binary large objects (BLOBs), see Section **3.7.3.2**.

10.2 StoreNotification: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.
- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5**.
- Customer must have the details of the data objects or subscription contexts it’s interested in; these details are typically found using Discovery (Protocol 3) (Chapter **8**) but may also come out of band of ETP (e.g., in an email).

10.2.1 StoreNotification: Message Sequences

This section explains the basic message sequence for main tasks to be done using this protocol and includes related key behaviors and possible errors. The following General Requirements section provides additional requirements and rules for how this protocol works.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section.

StoreNotification (Protocol 5): Basic Message-Response flow by ETP Role	
Message from customer	**Response Message from store**
SubscribeNotifications: A request to create a subscription for notifications.	***SubscribeNotificationsResponse*** (multipart): Reply listing the subscriptions that were successfully created.
	UnsolicitedStoreNotifications: Automatically sent to a customer when it connects to a store where a subscription was created for the customer based on out-of-band business knowledge (e.g., a contract).
	Notification messages sent by the store for established subscriptions (see details of each message in Section **10.3**): ***ObjectChanged*** (multipart) ***ObjectDeleted*** ***ObjectActiveStatusChanged*** ***ObjectAccessRevoked***

StoreNotification (Protocol 5): Basic Message-Response flow by ETP Role	
Message from customer	**Response Message from store**
	Chunk (multipart): If the data object is too large to be included in the ObjectChanged message, use to subdivide and send the data object in smaller "chunks".
UnsubscribeNotifications: A request to cancel/stop a subscription (either a requested or unsolicited one).	***SubscriptionEnded***: Response to an unsubscribe request OR notice from the store that it has canceled a subscription.

The main tasks in this protocol are subscribing to the appropriate objects or contexts (sets of related objects) in a store to receive the desired notifications and canceling/stopping those subscriptions. Once a subscription has been created, a store MUST send appropriate notifications based on events in Store (Protocol 4) and put header operations in GrowingObject (Protocol 6).

10.2.1.1 To subscribe to notifications (i.e., create a subscription):

1. A customer MUST send a store a ***SubscribeNotifications*** message (Section **10.3.6**).
    a. This message is a map of subscription requests. The details of each subscription request are specified in the ***SubscriptionInfo*** record, each of which uses a ***ContextInfo*** record to specify details of the data objects of interest in the store (Section **23.34.15**).
    b. ***SubscriptionInfo*** contains a lot of important information where the customer specifies details of the notification subscription it wants to create, but some key fields worth noting here are:
        i. *requestUuid*, which assigns a UUID to uniquely identify each subscription.
        ii. *includeObjectData*, a Boolean flag the customer uses to request that data objects be included with notification messages.
    c. A customer MUST limit the total count of subscriptions in a session to the store's value for the MaxSubscriptionSessionCount protocol capability.
        i. The Store MUST deny requests that exceed this limit by sending error ELIMIT_EXCEEDED◦(12).
2. For the requests it successfully creates subscriptions for, the store MUST respond one or more ***SubscribeNotificationsResponse*** map response messages (Section **10.3.7**), which list the successful subscriptions that the store has created.
    a. For more information on how map response messages work, see Section **3.7.3**.
    b. The store MUST then send notifications for the subscriptions identified in this response message (according to criteria specified in the ***SubscribeNotifications*** message) and according to any rules stated in this specification. 10.2.2, Row
    c. For details about general requirements for when to send specific notifications, see Section **10.2.2**.
3. For the requests it does NOT successfully create subscriptions for, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors, such as ENOT_FOUND (11) if a request URI could not be resolved.
    a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.
4. **NOTE:** A store can also create "unsolicited" notification subscriptions on behalf of a customer. For more information, see Section **10.2.2** (Row **7**).
5. If a customer sends a ***ProtocolException*** message in response to an ***ObjectChanged***, ***ObjectDeleted***, ***ObjectActiveStatusChanged***, or ***ObjectAccessRevoked*** message, the store MAY

attempt to take corrective action but the store MUST NOT terminate the associated subscriptions.

10.2.1.2 To unsubscribe to notifications (i.e., cancel a subscription):

1. A customer sends a store an ***UnsubscribeNotifications*** message (Section **10.3.1**).
   a. This message must identify the subscription to be cancelled by its request UUID, which the customer assigned to the subscription when it was requested or may have been assigned by an ***UnsolicitedStoreNotifications*** message (Section **10.3.9**).
2. If the store successfully cancels the subscription, the store MUST respond with a ***SubscriptionEnded*** message (Section **10.3.5**)**,** which holds the request UUID of the subscription that was successfully stopped.
   a. The store MUST stop sending any further notifications that were specified in the subscription that has now been ended. It's possible that the customer COULD receive a few additional notifications that were in process before the subscription was stopped.
   b. After sending ***SubscriptionEnded***, the store MUST NOT send any notifications for the subscription.
3. If the store does NOT successfully cancel the subscription, it MUST send a ***ProtocolException*** message with an appropriate error code (e.g., if the request UUID could not be found by the store send ENOT_FOUND (11)).
4. The store MAY also end a subscription without receiving a customer request. If the store does so, it MUST notify the customer by sending a ***SubscriptionEnded*** message (Section◦**10.3.5**). **EXAMPLE:**◦This happens if the subscription's context URI refers to a data object that has been deleted.
5. When a customer has canceled a subscription, the store MUST NOT restart it, even if the subscription was created by the store on behalf of the customer with ***UnsolicitedStoreNotifications***.
   a. If the customer wants to restart the subscription, it MUST instead set up a new subscription by sending ***SubscribeNotifications*** as described in Section **10.2.1.1** using a NEW *requestUuid*.

10.2.2 StoreNotification: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) some rows with additional requirements for specific types of operations.

Row#	Requirement	Description
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**.

		4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter **5**. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3**. a. For the list of global capabilities and related behavior, see Section **3.3.2**. 3. Section **10.2.3** identifies the capabilities most relevant to this ETP sub-protocol. Additional details for how to use the protocol capabilities are included below in this table and in Section **10.2.1 StoreNotification: Message Sequence**.
3.	**Message sequence** See Section **10.2.1**.	1. The Message Sequence section above (Section **10.2.1**) describes requirements for the main tasks listed there and also defines required behavior.
4.	**Plural messages** (which includes maps)	1. This protocol uses plural messages. For detailed rules on handling plural messages (including ***ProtocolException*** handling), see Section **3.7.3**.
5.	For data objects that exceed an endpoint's WebSocket message size, use the ***Chunk*** message.	1. Some messages in this protocol allow or require a data object to be sent with the message. If the size of the data object (bytes) is too large for the WebSocket message size (which for some WebSocket libraries can be quite small, e.g. 128 kb), an endpoint MAY subdivide the data object and send it in "chunks" using the ***Chunk*** message defined in this protocol. For information on how to handle these binary large objects (BLOBs), see Section **3.7.3.2**. 2. **NOTE:** Use of Chunk messages DOES NOT address an endpoint's MaxDataObjectSize limit. 3. The specific messages in this protocol that may use ***Chunk*** messages are: a. ***ObjectChanged***
6.	Customers must be able to receive and consume data objects.	1. All customer role applications MUST implement support for receiving and consuming notifications that include the data objects (that is, all data for the object in a format (e.g., XML or JSON) negotiated when establishing the session).
7.	Unsolicited subscriptions	1. The store may automatically configure unsolicited subscriptions to include the data objects (i.e., the *includeObjectData* on the unsolicited ***SubscriptionInfo*** record may be true). If the customer application does not want the data, it can do one of the following: a. Unsubscribe and stop receiving the notifications. b. Simply ignore the data payloads and get the data manually. c. Unsubscribe from the unsolicited subscription and then explicitly create the subscription (see Section **10.2.1.1**) and set *includeObjectData* to false.
8.	All behaviors defined in this table assume that a valid customer	1. We are aiming to state these requirements and behaviors as clearly and concisely as possible. All required behaviors ("MUST" statements) described in the rows below assume:

	subscription for the correct context has been created.	a. A valid subscription has been created as described in Section◦**10.2.1.1**. b. References to "data object(s)" means "data object(s) within the context specified in the subscription". c. **EXAMPLE:** Below in this table where it states "When a store performs a ***PutDataObjects*** operation, it MUST send an ***ObjectChanged*** message"; this means, if the customer has a subscription whose scope and context includes the data object that was put, then the store must send the ***ObjectChanged*** message to the subscribed customer. 2. A valid subscription is one where all of the following conditions are met: a. ***SubscriptionInfo***.*context* is a valid: i. ***ContextInfo***.*uri* references a data object or dataspace that exists and is available in the store (i.e., the store will return it if requested using Store (Protocol 4) or Dataspaces (Protocol 24)). ii. ***ContextInfo***.*dataObjectTypes* is empty or only includes data object types negotiated when establishing the session. b. ***SubscriptionInfo***.*requestUuid* is not already in use by another subscription. c. ***SubscriptionInfo***.*format* is a format negotiated when establishing the session.
9.	Notifications are for operations that happen in Store (Protocol 4) and put header operations in GrowingObject (Protocol 6).	1. The notifications sent in this protocol are based on operations that happen in Store (Protocol 4). As such, detailed behaviors that trigger notifications are described in Chapter 9 (see Sections **9.2.1** and **9.2.2**) an indicated with text "NOTIFICATION BEHAVIOR". a. **RECOMMENDATION:** For complete understanding of notification behavior, use both Chapters 9 and 10. 2. Additionally, operations to a growing data object "header" and growing data object parts that are themselves data objects in GrowingObject (Protocol 6) may trigger a notification in StoreNotification (Protocol 5); these operations add and update growing data object headers and add, update, link, unlink and delete growing data object parts that are data objects. As such, the notification requirements for these operations are the same as for changes to data objects as described in Store (Protocol 4) (Chapter **9**). a. For more information about growing data object operations and notifications, see Chapter **11**.
10.	No session survivability for subscriptions	1. If the ETP session is closed or the connection drops, then the store MUST cancel notification subscriptions for the dropped customer endpoint. 2. On reconnect, the customer MUST re-create subscriptions (as explained in Section **10.2.1.1**). a. For information on resuming operations after a disconnect, see **Appendix: Data Replication and Outage Recovery Workflows**.
11.	Order of notifications	1. For a given data object, the store MUST send notifications in the same order that operations are performed in the store. a. The intent of this rule is that objects are always "correct" (schema compliant), and never left in an inconsistent state. The rule applies primarily to contained data objects and growing data objects. b. In general, global ordering of notifications is NOT required. However, there are some situations where the order of notifications affecting multiple objects is important and must be preserved.
12.	Objects covered by more than one subscription	A customer can create multiple subscriptions on a store. It is possible that the same data object is included in more than one subscription. 1. In this case, the store MUST send one notification per relevant subscription. **EXAMPLE:** If a customer has subscribed to two different scope/contexts that include the same data object, then the customer will receive at least 2 notifications, one for each subscription. a. Each notification message includes the *requestUuid* that uniquely identifies each subscription (so a customer can determine which subscription resulted in each notification message).
13.	**Sending notifications:** general requirements	1. REMINDER: Row **8** 2. Notification messages are those whose name begins with the word "Object". Each message's definition/description provides general information for when each endpoint role (store or customer) must send each message.

		(**EXAMPLE:** The store MUST send an ***ObjectDeleted*** message, when it deletes an object.) a. Other rows in this table state additional requirements for specific operations and requirements for notifications. b. Section **9.2.2** (for Store (Protocol 4) also specifies some NOTIFICATION BEHAVIOR in the context of the detailed store operation(s) that trigger one or more notifications. **RECOMMENDATION:** Work with this chapter and Chapter 9 together. 3. A store MUST send all appropriate notifications, including ***ObjectChanged*** and ***ObjectDeleted***, even if the change was not through an ETP store operation. 4. A store MUST send notifications within its value for ChangePropagationPeriod endpoint capability, which MUST be less than or equal to the maximum value stated in this specification (see Section **3.3.2.2**). 5. If in the subscription request (***SubscribeNotifications*** message) the *includeObjectData* field was set to true, the store MUST send the object data with the notification (for all notifications that include the *dataObject* field, which is included on the ***ObjectChange*** record). a. For all data objects, the store must observe limits specified by its own and the customer's values for the MaxDataObjectSize capability. For more information about how this capability works and required behavior, see Section **3.3.2.4**. b. For growing data objects, the store MUST observe limits specified by its own and the other customer's values for the MaxPartSize capability. For more information about how this capability works and required behavior, see Section **3.3.2.5**.
14.	**Putting (inserting) and updating data objects:** Additional notification requirements	1. **REMINDER:** Row **8**. 2. When a store completes a ***PutDataObjects*** operation (in Store (Protocol 4)), it MUST send an ***ObjectChanged*** message. a. Because ETP uses upsert semantics, this message includes information about the type of change, which is specified on the ***ObjectChange*** record, which references the *ObjectChangeKind* enumeration. i. If the store inserted (added) a new data object, then it MUST set *ObjectChangeKind* to "insert". ii. If the store updated (replaced) an existing data object, then it MUST set *ObjectChangeKind* to "update". iii. If the change was caused by an ETP store operation, the store MUST differentiate between insert and update. iv. If the change was NOT caused by an ETP store operation and the store cannot determine if the operation was an insert or update, it MUST set *ObjectChangeKind* to "insert". **NOTE:** "insert" was chosen because it is the "pessimistic" choice. That is, customers using the replication workflow will assume the affected data objects and any associated bulk data (channel data, growing object parts, data arrays) have been completely replaced. While this may cause customers to query more data than is necessary when the operation is actually an update, using "insert" and the pessimistic assumptions that go with it are necessary in some edge cases to achieve eventual consistency between data stores. 3. If the store adds or updates a data object using ***PutGrowingDataObjectsHeader*** in GrowingObject (Protocol 6), it MUST perform the same actions as specified in Step **2** (above in this table row). 4. For additional requirements for container and contained data objects, see Row **18 below** in this table.
15.	**Deleting data objects:** additional notification requirements	1. **REMINDER:** Row **8** 2. When a data object is deleted, the store MUST send an ***ObjectDeleted*** message and it MUST NOT send any additional notification messages for the deleted object. a. A delete is an atomic operation; the store MUST perform the delete operation and then send notifications.

		b. A store MUST send notifications for only the most recent effective state of a data object. So if notifications are queued for a data object, and that data object is subsequently deleted, the store MAY discard any previous notifications. 3. If the data object being deleted is the primary data object of a subscription, the store MUST also do the following: a. MAY send any relevant notifications that may have already been queued (i.e., for other data objects in the subscription). b. MUST stop any subscriptions for the deleted object by sending the ***SubscriptionEnded*** message. c. After sending the ***SubscriptionEnded*** message, MUST NOT send any further notifications for the subscription.
16.	**Data objects that can be "active":** changes to activeStatus field	1. **REMINDER:** Row **8** 2. Growing data objects, channel data objects, and other data objects that can be "active" in ETP have a field named *activeStatus*, which may have a value of "inactive" or "active". a. For information about this field and required behavior for setting it to "inactive" related to the ActiveTimeoutPeriod capability, see Section **3.3.2.1**. b. Behavior that causes the field to be set to "active" are described in the protocols in which they occur and summarized in Section **9.2.2**, Row **9**. 3. **NOTIFICATION BEHAVIOR:** When a data object's *activeStatus* field changes, a store MUST send an ***ObjectActiveStatusChanged*** notification message.
17.	Entitlement changes to data objects	**REMINDER:** Row **8** Many stores grant entitlements (access to data) at the well, wellbore or log level. This means: even if a customer-user is subscribed to the correct context, it cannot receive notification of the new object (e.g., well or wellbore) until the user is granted permission. In this situation, the store MUST do the following: 1. When the customer is granted access to a data object, the store MUST send the ***ObjectChanged*** notification message with an *ObjectChangeKind* of authorized. Conversely, a customer-user may initially be given access to a data object, only to have it later revoked. In this situation, the store MUST do the following: 1. When the customer's access to a data object is revoked, the store MUST send the ***ObjectAccessRevoked*** notification message.
18.	**Container/contained data objects:** Additional notification requirements	1. REMINDER: Row **8** 2. For definitions of container and contained objects and related concepts, see Section **9.1.3**. 3. ETP specifies 2 additional values for *ObjectChangeKind* for operations on container data objects; these notifications pertain to the contained objects which may be added (joined) to a container or removed (unjoined) from a container. a. For details on the behavior that triggers notifications and the notifications to send, see Section **9.2.2**; for put operations see Row **19**, for delete operations see Row **25**.
19.	Data objects entering or leaving subscription context/scope	1. **REMINDER**: Row **8** 2. Certain events in a store may cause new data objects to enter a subscription's scope/context, and other events in a store may cause data objects to leave. **EXAMPLES:** a. If a subscription's scope/context is a wellbore data object and all data objects associated with the wellbore: new data objects related to the wellbore will enter the subscription's scope/context. b. Adding a relationship between an existing data object and the wellbore will also bring the data object into the subscription's scope/context. c. Deleting data objects associated with the wellbore will remove them from the subscription's scope/context. d. Similarly, removing a relationship between the wellbore and an existing data object will remove it from the subscription's scope/context. 3. When data objects enter the scope/context of a subscription, the store MUST

		send an ***ObjectChanged*** notification with *ObjectChangeKind* set to "joinedSubscription". 4. When data objects leave the scope/context of a subscription, the store MUST send an ***ObjectChanged*** notification with *ObjectChangeKind* set to "unjoinedSubscription".
20.	Ending subscriptions	1. **REMINDER**: Row **8** 2. A store MUST end a customer's subscription to store notifications when any of these events occur: a. The customer cancels the subscription by sending an ***UnsubscribeNotifications*** message. b. The primary data object for the subscription (i.e., the data object identified by the URI in the *context* field of the subscription's ***SubscriptionInfo*** record) is deleted. c. The customer loses access to the primary data object for the subscription. 3. When ending a subscription: a. The store MUST send the ***SubscriptionEnded*** message either as a response to a customer's ***UnsubscribeNotifications*** request or as a notification. i. The store MUST include a human readable reason why the subscription was ended in the ***UnsubscribeNotifications*** message. 4. When a store ends a subscription in response to a customer's ***UnsubscribeNotifications*** request, the store MAY discard any queued notifications for the subscription. 5. When a store end's a subscription WITHOUT a customer request: a. If the subscription's primary data object was deleted and it was in the subscription's scope (i.e., scope was self, sourcesOrSelf or targetsOrSelf), the store MUST first send an ***ObjectDeleted*** message for the primary data object before it sends the ***SubscriptionEnded*** message. The store MAY discard any other queued notifications for the subscription. b. If the customer lost access to the subscription's primary data object and it was in the subscription's scope, the store MUST send an ***ObjectAccessRevoked*** message for the primary data object before it sends the ***SubscriptionEnded*** message. The store MAY discard any other queued notifications for the subscription. 6. After it sends the ***SubscriptionEnded*** message, the store MUST NOT send any further notifications for the subscription. 7. After a subscription has ended, the store MUST NOT restart it, even if the subscription was created by the store on behalf of the customer with the ***UnsolicitedStoreNotifications*** message.

10.2.3 StoreNotification: Capabilities

The table below lists key capabilities for this protocol. The protocol capabilities are all defined here. For this protocol, one particularly crucial endpoint capability is defined here.

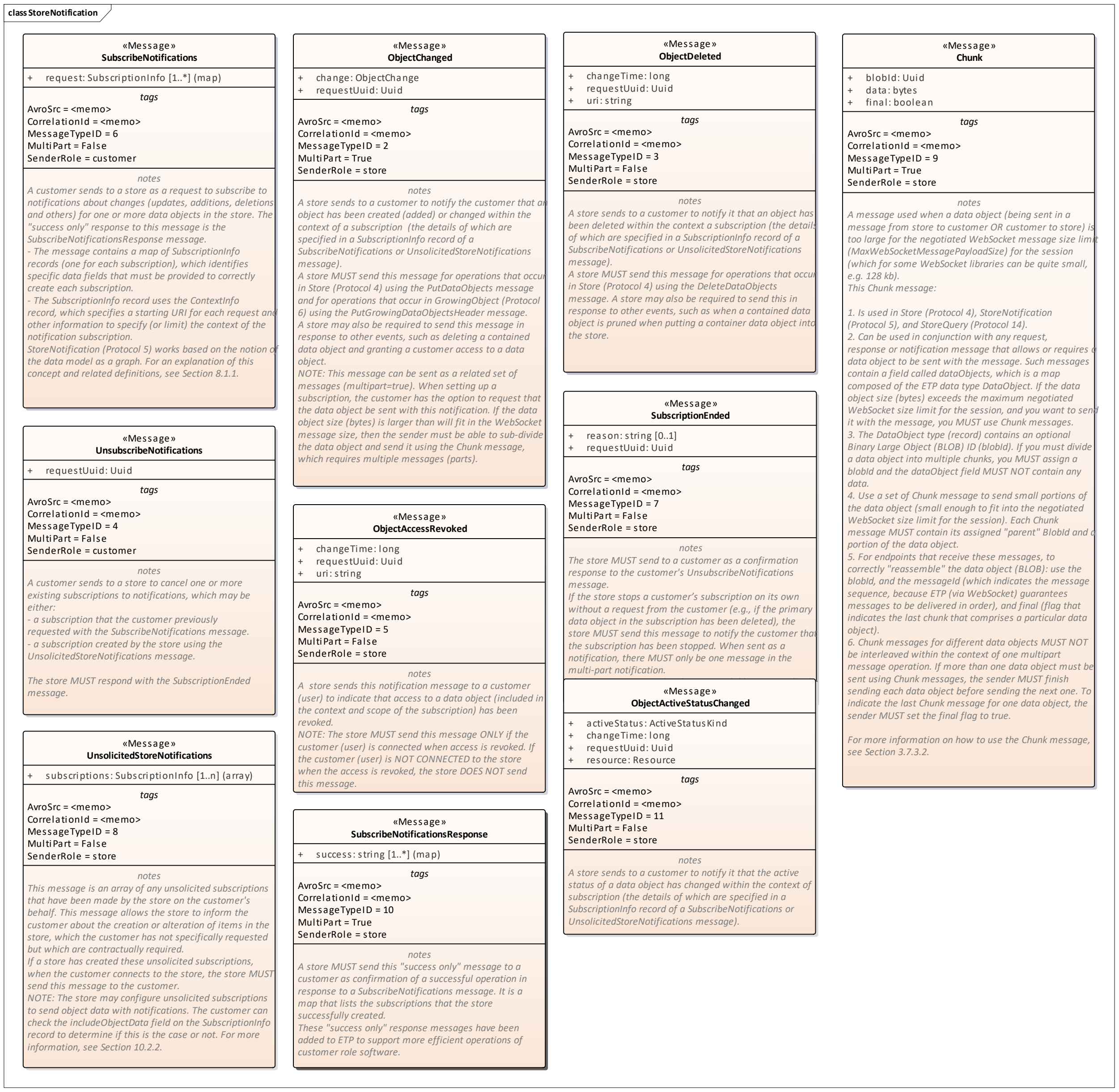
- For protocol-specific behavior relating to using these capabilities in this protocol, see Section **10.2.2**, **StoreNotification: General Requirements**.
- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section **3.3**.

StoreNotification (Protocol 5): Capabilities			
Name: Description	**Type**	**Units** **Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of each endpoint capability, see Section **3.3**.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section **3.3.2**. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP** Server.			
MaxPartSize: The maximum size in bytes of each data object part allowed in a standalone message or a complete multipart message. Size in bytes is the total size in bytes of the uncompressed string representation of the data object part in the format in which it is sent or received.	long	byte <number of bytes>	Min: 10,000 bytes
Data Object Capabilities			
ActiveTimeoutPeriod: (This is also an endpoint capability.) The minimum time period in seconds that a store keeps the active status (*activeStatus* field in ETP) for a data object as “active” after the most recent update causing the data object’s active status to be set to true. For growing data objects, this is any change to its parts. For channels, this is any change to its data points. This capability can be set for an endpoint and/or for a data object. A data object-specific value overrides an endpoint-specific value.	long	second <number of seconds>	**Default:** 3,600 **MIN:** 60 seconds
MaxContainedDataObjectCount: The maximum count of contained data objects allowed in a single instance of the data object type that the capability applies to. **EXAMPLE:** If this capability is set to 2000 for a ChannelSet, then the ChannelSet may contain a maximum of 2000 Channels**.**	long	Count <count of objects>	**MIN:** Should be specified per domain
SupportsGet For definitions and usage rules for this data object capability, see Section **3.3.4**.			
Protocol Capabilities			
MaxDataObjectSize: (This is also an endpoint capability and a data object.) The maximum size in bytes of a data object allowed in a complete multipart message. Size in bytes is the size in bytes of the uncompressed string representation of the data object in the format in which it is sent or received. This capability can be set for an endpoint, a protocol, and/or a data object. If set for all three, here is how they generally work: • An object-specific value overrides an endpoint-specific value. • A protocol-specific value can further lower (but NOT raise) the limit for the protocol.	long	byte <number of bytes>	**MIN:** 100,000 bytes

StoreNotification (Protocol 5): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
EXAMPLE: A store may wish to generally support sending and receiving any data object that is one megabyte or less with the exceptions of Wells that are 100 kilobytes or less and Attachments that are 5 megabytes or less. A store may further wish to limit the size of any data object sent as part of a notification in StoreNotification (Protocol 5) to 256 kilobytes.			
MaxResponseCount: The maximum total count of responses allowed in a complete multipart message response to a single request.	long	count <count of responses>	**MIN:** 10,000
MaxSubscriptionSessionCount: The maximum total count of concurrent subscriptions allowed in a session. The limit applies separately for each protocol with the capability. **EXAMPLE:** Different values can be specified for StoreNotification (Protocol 5) and GrowingObjectNotification (Protocol 7).	long	count <count of subscriptions>	**MIN:** 100

10.3 StoreNotification: Message Schemas

This section provides a figure that displays all messages defined in StoreNotification (Protocol 5). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.



«Message»
SubscribeNotifications

+ request: SubscriptionInfo [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 6
MultiPart = False
SenderRole = customer

notes
A customer sends to a store as a request to subscribe to notifications about changes (updates, additions, deletions and others) for one or more data objects in the store. The "success only" response to this message is the SubscribeNotificationsResponse message.
- The message contains a map of SubscriptionInfo records (one for each subscription), which identifies specific data fields that must be provided to correctly create each subscription.
- The SubscriptionInfo record uses the ContextInfo record, which specifies a starting URI for each request and other information to specify (or limit) the context of the notification subscription.
StoreNotification (Protocol 5) works based on the notion of the data model as a graph. For an explanation of this concept and related definitions, see Section 8.1.1.

«Message»
UnsubscribeNotifications

+ requestUuid: Uuid

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 4
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to cancel one or more existing subscriptions to notifications, which may be either:
- a subscription that the customer previously requested with the SubscribeNotifications message.
- a subscription created by the store using the UnsolicitedStoreNotifications message.

The store MUST respond with the SubscriptionEnded message.

«Message»
UnsolicitedStoreNotifications

+ subscriptions: SubscriptionInfo [1..n] (array)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 8
MultiPart = False
SenderRole = store

notes
This message is an array of any unsolicited subscriptions that have been made by the store on the customer's behalf. This message allows the store to inform the customer about the creation or alteration of items in the store, which the customer has not specifically requested but which are contractually required.
If a store has created these unsolicited subscriptions, when the customer connects to the store, the store MUST send this message to the customer.
NOTE: The store may configure unsolicited subscriptions to send object data with notifications. The customer can check the includeObjectData field on the SubscriptionInfo record to determine if this is the case or not. For more information, see Section 10.2.2.

«Message»
ObjectChanged

+ change: ObjectChange
+ requestUuid: Uuid

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 2
MultiPart = True
SenderRole = store

notes
A store sends to a customer to notify the customer that an object has been created (added) or changed within the context of a subscription (the details of which are specified in a SubscriptionInfo record of a SubscribeNotifications or UnsolicitedStoreNotifications message).
A store MUST send this message for operations that occur in Store (Protocol 4) using the PutDataObjects message and for operations that occur in GrowingObject (Protocol 6) using the PutGrowingDataObjectsHeader message.
A store may also be required to send this message in response to other events, such as deleting a contained data object and granting a customer access to a data object.
NOTE: This message can be sent as a related set of messages (multipart=true). When setting up a subscription, the customer has the option to request that the data object be sent with this notification. If the data object size (bytes) is larger than will fit in the WebSocket message size, then the sender must be able to sub-divide the data object and send it using the Chunk message, which requires multiple messages (parts).

«Message»
ObjectAccessRevoked

+ changeTime: long
+ requestUuid: Uuid
+ uri: string

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 5
MultiPart = False
SenderRole = store

notes
A store sends this notification message to a customer (user) to indicate that access to a data object (included in the context and scope of the subscription) has been revoked.
NOTE: The store MUST send this message ONLY if the customer (user) is connected when access is revoked. If the customer (user) is NOT CONNECTED to the store when the access is revoked, the store DOES NOT send this message.

«Message»
SubscribeNotificationsResponse

+ success: string [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 10
MultiPart = True
SenderRole = store

notes
A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a SubscribeNotifications message. It is a map that lists the subscriptions that the store successfully created.
These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

«Message»
ObjectDeleted

+ changeTime: long
+ requestUuid: Uuid
+ uri: string

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 3
MultiPart = False
SenderRole = store

notes
A store sends to a customer to notify it that an object has been deleted within the context a subscription (the details of which are specified in a SubscriptionInfo record of a SubscribeNotifications or UnsolicitedStoreNotifications message).
A store MUST send this message for operations that occur in Store (Protocol 4) using the DeleteDataObjects message. A store may also be required to send this in response to other events, such as when a contained data object is pruned when putting a container data object into the store.

«Message»
SubscriptionEnded

+ reason: string [0..1]
+ requestUuid: Uuid

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 7
MultiPart = False
SenderRole = store

notes
The store MUST send to a customer as a confirmation response to the customer's UnsubscribeNotifications message.
If the store stops a customer's subscription on its own without a request from the customer (e.g., if the primary data object in the subscription has been deleted), the store MUST send this message to notify the customer that the subscription has been stopped. When sent as a notification, there MUST only be one message in the multi-part notification.

«Message»
ObjectActiveStatusChanged

+ activeStatus: ActiveStatusKind
+ changeTime: long
+ requestUuid: Uuid
+ resource: Resource

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 11
MultiPart = False
SenderRole = store

notes
A store sends to a customer to notify it that the active status of a data object has changed within the context of subscription (the details of which are specified in a SubscriptionInfo record of a SubscribeNotifications or UnsolicitedStoreNotifications message).

«Message»
Chunk

+ blobId: Uuid
+ data: bytes
+ final: boolean

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 9
MultiPart = True
SenderRole = store

notes
A message used when a data object (being sent in a message from store to customer OR customer to store) is too large for the negotiated WebSocket message size limit (MaxWebSocketMessagePayloadSize) for the session (which for some WebSocket libraries can be quite small, e.g. 128 kb).
This Chunk message:

1. Is used in Store (Protocol 4), StoreNotification (Protocol 5), and StoreQuery (Protocol 14).
2. Can be used in conjunction with any request, response or notification message that allows or requires a data object to be sent with the message. Such messages contain a field called dataObjects, which is a map composed of the ETP data type DataObject. If the data object size (bytes) exceeds the maximum negotiated WebSocket size limit for the session, and you want to send it with the message, you MUST use Chunk messages.
3. The DataObject type (record) contains an optional Binary Large Object (BLOB) ID (blobId). If you must divide a data object into multiple chunks, you MUST assign a blobId and the dataObject field MUST NOT contain any data.
4. Use a set of Chunk message to send small portions of the data object (small enough to fit into the negotiated WebSocket size limit for the session). Each Chunk message MUST contain its assigned "parent" BlobId and a portion of the data object.
5. For endpoints that receive these messages, to correctly "reassemble" the data object (BLOB): use the blobId, and the messageId (which indicates the message sequence, because ETP (via WebSocket) guarantees messages to be delivered in order), and final (flag that indicates the last chunk that comprises a particular data object).
6. Chunk messages for different data objects MUST NOT be interleaved within the context of one multipart message operation. If more than one data object must be sent using Chunk messages, the sender MUST finish sending each data object before sending the next one. To indicate the last Chunk message for one data object, the sender MUST set the final flag to true.

For more information on how to use the Chunk message, see Section 3.7.3.2.

Figure 19: StoreNotification: message schemas

10.3.1 Message: UnsubscribeNotifications

A customer sends to a store to cancel one or more existing subscriptions to notifications, which may be either:

- a subscription that the customer previously requested with the SubscribeNotifications message.
- a subscription created by the store using the UnsolicitedStoreNotifications message.

The store MUST respond with the ***SubscriptionEnded*** message (Section **10.3.5**).

Message Type ID: 4

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.)

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
requestUuid	The UUID of the subscription that is being canceled. Each subscription was assigned a UUID by the customer requesting it, when the subscription was created (in the SubscriptionInfo record) or was assigned in an UnsolicitedStoreNotifications message. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.StoreNotification",
     "name": "UnsubscribeNotifications",
     "protocol": "5",
     "messageType": "4",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" }
     ]
}
```

10.3.2 Message: ObjectChanged

A store sends to a customer to notify the customer that an object has been created (added) or changed within the context of a subscription (the details of which are specified in a SubscriptionInfo record of a SubscribeNotifications or UnsolicitedStoreNotifications message).

A store MUST send this message for operations that occur in Store (Protocol 4) using the ***PutDataObjects*** message and for operations that occur in GrowingObject (Protocol 6) using the ***PutGrowingDataObjectsHeader*** message.

A store may also be required to send this message in response to other events, such as deleting a contained data object and granting a customer access to a data object.

NOTE: This message can be sent as a related set of messages (multipart=true). When setting up a subscription, the customer has the option to request that the data object be sent with this notification. If the data object size (bytes) is larger than will fit in the WebSocket message size, then the sender must be able to sub-divide the data object and send it using the ***Chunk*** message, which requires multiple messages (parts).

Message Type ID: 2

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
change	The information that describes the change to the data object or identifies a new data object that has	ObjectChange	1	1

Field Name	Description	Data Type	Min	Max
	been added. For details of information that must be sent, see ObjectChange record. The URI in the ***DataObject*** record's ***Resource*** record MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.			
requestUuid	The UUID of the subscription request that resulted in this notification message being sent. The UUID was assigned by the customer when the subscription was requested and created (in the SubscriptionInfo record) or by an UnsolicitedStoreNotifications message. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.StoreNotification",
     "name": "ObjectChanged",
     "protocol": "5",
     "messageType": "2",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         { "name": "change", "type": "Energistics.Etp.v12.Datatypes.Object.ObjectChange" },
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" }
     ]
}
```

10.3.3 Message: ObjectDeleted

A store sends to a customer to notify it that an object has been deleted within the context a subscription (the details of which are specified in a SubscriptionInfo record of a SubscribeNotifications or UnsolicitedStoreNotifications message).

A store MUST send this message for operations that occur in Store (Protocol 4) using the ***DeleteDataObjects*** message. A store may also be required to send this in response to other events, such as when a contained data object is pruned when putting a container data object into the store.

Message Type ID: 3

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
uri	The URI of the data object that was deleted. This MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
changeTime	The time the change occurred in the store. This is the value from the *deletedTime* field on the DeletedResource record. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1

Field Name	Description	Data Type	Min	Max
requestUuid	The UUID of the subscription request that resulted in this notification message being sent. The UUID was assigned by the customer when the subscription was requested and created (in the SubscriptionInfo record) or by an UnsolicitedStoreNotifications message. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.StoreNotification",
     "name": "ObjectDeleted",
     "protocol": "5",
     "messageType": "3",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "changeTime", "type": "long" },
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" }
     ]
}
```

10.3.4 Message: ObjectAccessRevoked

A store sends this notification message to a customer (user) to indicate that access to a data object (included in the context and scope of the subscription) has been revoked.

NOTE: The store MUST send this message ONLY if the customer (user) is connected when access is revoked. If the customer (user) is NOT CONNECTED to the store when the access is revoked, the store DOES NOT send this message.

Message Type ID: 5

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
uri	The URI of the data object for which access was revoked. This MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
changeTime	The time the change occurred in the store. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
requestUuid	The UUID of the subscription request that resulted in this notification message being sent. The UUID was assigned by the customer when the subscription was requested and created (in the SubscriptionInfo record) or by an UnsolicitedStoreNotifications message.	Uuid	1	1

Field Name	Description	Data Type	Min	Max
	Must be of type ***Uuid*** (Section **23.6**).			

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.StoreNotification",
     "name": "ObjectAccessRevoked",
     "protocol": "5",
     "messageType": "5",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "changeTime", "type": "long" },
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" }
     ]
}
```

10.3.5 Message: SubscriptionEnded

The store MUST send to a customer as a confirmation response to the customer's UnsubscribeNotifications message.

If the store stops a customer's subscription on its own without a request from the customer (e.g., if the primary data object in the subscription has been deleted), the store MUST send this message to notify the customer that the subscription has been stopped. When sent as a notification, there MUST only be one message in the multi-part notification.

The store MUST provide a human readable reason why the subscription was stopped.

Message Type ID: 7

Correlation Id Usage: When sent as a response: MUST be set to the *messageId* of the ***UnsubscribeNotifications*** message that this message is a response to. When sent as a notification: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
reason	A reason why the subscriptions have been stopped.	String	1	1
requestUuid	The UUID of the subscription the store is ending. These UUIDs were assigned by the customer when the subscription was requested (in the SubscriptionInfo record) or by an UnsolicitedStoreNotifications message. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.StoreNotification",
     "name": "SubscriptionEnded",
     "protocol": "5",
     "messageType": "7",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "reason", "type": "string" },
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" }
     ]
}
```

10.3.6 Message: SubscribeNotifications

A customer sends to a store as a request to subscribe to notifications about changes (updates, additions, deletions and others) for one or more data objects in the store. The "success only" response to this message is the SubscribeNotificationsResponse message.

- The message contains a map of SuscriptionInfo records (one for each subscription), which identifies specific data fields that must be provided to correctly create each subscription.
- The SubscriptionInfo record uses the ContextInfo record, which specifies a starting URI for each request and other information to specify (or limit) the context of the notification subscription.

StoreNotification (Protocol 5) works based on the notion of the data model as a graph. For an explanation of this concept and related definitions, see Section **8.1.1**.

Message Type ID: 6

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
request	General ETP map of subscription requests; the details of each request is specified in a SuscriptionInfo record and includes information such as the context and scope of the request and the request UUID that initiated the subscription. If both endpoints support alternate URIs for the session, the URIs in the ***ContextInfo*** records MAY be alternate data object or dataspace URIs. Otherwise, they MUST be canonical Energistics data object or dataspace URIs. For more information, see **Appendix: Energistics Identifiers**. If alternate URIs are used, the store MUST resolve them to canonical URIs and treat the subscription as a subscription to the canonical URI.	SubscriptionInfo	1	*

Avro Source

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Protocol.StoreNotification",
    "name": "SubscribeNotifications",
    "protocol": "5",
    "messageType": "6",
    "senderRole": "customer",
    "protocolRoles": "store,customer",
    "multipartFlag": false,

    "fields":
    [
        {
            "name": "request",
            "type": { "type": "map", "values": 
"Energistics.Etp.v12.Datatypes.Object.SubscriptionInfo" }
        }
    ]
}
```

10.3.7 Message: SubscribeNotificationsResponse

A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a SubscribeNotifications message. It is a map that lists the subscriptions that the store successfully created.

These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

Message Type ID: 10

Correlation Id Usage: MUST be set to the *messageId* of the ***SubscribeNotifications*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
success	• The presence of the map key represents success. • The associated map string value SHOULD be empty because its content is ignored.	string	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.StoreNotification",
     "name": "SubscribeNotificationsResponse",
     "protocol": "5",
     "messageType": "10",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "success",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

10.3.8 Message: ObjectActiveStatusChanged

A store sends to a customer to notify it that the active status of a data object has changed within the context of subscription (the details of which are specified in a SubscriptionInfo record of a SubscribeNotifications or UnsolicitedStoreNotifications message).

Message Type ID: 11

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
activeStatus	Indicates the updated status, which may be "active" or "inactive", as defined in ActiveStatusKind. Statuses are mapped from domain data object such as wellbores, channels, and growing data objects. For the detailed mapping, see the *WITSMLv2.0 for ETP v1.2 Implementation Specification*.	ActiveStatusKind	1	1
changeTime	The time the change occurred in the store. This is the value from *storeLastWrite* field (for more information see Resource). It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
resource	The information about the data object as specified in the Resource record. The URI in the ***Resource*** record MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	Resource	1	1
requestUuid	The UUID of the subscription request that resulted in this notification message being sent. The UUID that was assigned by the customer when the subscription was requested and created (in the SubscriptionInfo record) or by an UnsolicitedStoreNotifications message. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.StoreNotification",
     "name": "ObjectActiveStatusChanged",
     "protocol": "5",
     "messageType": "11",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "activeStatus", "type":
"Energistics.Etp.v12.Datatypes.Object.ActiveStatusKind" },
         { "name": "changeTime", "type": "long" },
         { "name": "resource", "type": "Energistics.Etp.v12.Datatypes.Object.Resource" },
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" }
     ]
}
```

10.3.9 Message: UnsolicitedStoreNotifications

This message is an array of any unsolicited subscriptions that have been made by the store on the customer's behalf. This message allows the store to inform the customer about the creation or alteration of items in the store, which the customer has not specifically requested but which are contractually required.

If a store has created these unsolicited subscriptions, when the customer connects to the store, the store automatically sends it to the customer.

NOTE: The store may configure unsolicited subscriptions to send object data with notifications. The customer can check the *includeObjectData* field on the ***SubscriptionInfo*** record to determine if this is the case or not. For more information, see Section **10.2.2**.

Message Type ID: 8

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
subscriptions	An array of SubscriptionInfo records, each of which identifies the details of an unsolicited subscription. Each record includes information such the context and scope of the subscription, and the request UUID that initiated a subscription. The URI in the ***ContextInfo*** record MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	SubscriptionInfo	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.StoreNotification",
     "name": "UnsolicitedStoreNotifications",
     "protocol": "5",
     "messageType": "8",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "subscriptions",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.Object.SubscriptionInfo" }
         }
     ]
}
```

10.3.10 Message: Chunk

A message used when a data object (being sent in a message from store to customer OR customer to store) is too large for the negotiated WebSocket message size limit (MaxWebSocketMessagePayloadSize) for the session (which for some WebSocket libraries can be quite small, e.g. 128 kb).

This ***Chunk*** message:

1. Is used in Store (Protocol 4), StoreNotification (Protocol 5), and StoreQuery (Protocol 14).
2. Can be used in conjunction with any request, response or notification message that allows or requires a data object to be sent with the message. Such messages contain a field called *dataObjects*, which is a map composed of the ETP data type DataObject. If the data object size (bytes) exceeds the maximum negotiated WebSocket size limit for the session, and you want to send it with the message, you MUST use ***Chunk*** messages.
3. The ***DataObject*** type (record) contains an optional Binary Large Object (BLOB) ID (*blobId*). If you must divide a data object into multiple chunks, you MUST assign a *blobId* and the *dataObject* field MUST NOT contain any data.

4. Use a set of ***Chunk*** message to send small portions of the data object (small enough to fit into the negotiated WebSocket size limit for the session). Each ***Chunk*** message MUST contain its assigned "parent" BlobId and a portion of the data object.
5. For endpoints that receive these messages, to correctly "reassemble" the data object (BLOB): use the *blobId*, and the *messageId* (which indicates the message sequence, because ETP (via WebSocket) guarantees messages to be delivered in order), and *final* (flag that indicates the last chunk that comprises a particular data object).
6. ***Chunk*** messages for different data objects MUST NOT be interleaved within the context of one multipart message operation. If more than one data object must be sent using ***Chunk*** messages, the sender MUST finish sending each data object before sending the next one. To indicate the last ***Chunk*** message for one data object, the sender MUST set the *final* flag to true.

For more information on how to use the ***Chunk*** message, see Section **3.7.3.2**.

Message Type ID: 9

Correlation Id Usage: MUST be set to the *messageId* of the ***ObjectChanged*** message that resulted in this Chunk message being created.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
blobId	The BLOB ID assigned by an endpoint when a data object being sent in a request, response, or notification message must be subdivided into multiple chunks. Each ***Chunk*** message that comprises a BLOB must contain the *blobId* of its "parent" BLOB. The *blobId*: • is entered in the DataObject record referenced in the *dataObjects* field of the request, response, or notification message. • must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1
data	The data that comprises a chunk (portion) of the data object/BLOB.	bytes	1	1
final	Flag to indicate that this is the final message of a set of ***Chunk*** messages that comprise one particular data object.	boolean	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.StoreNotification",
     "name": "Chunk",
     "protocol": "5",
     "messageType": "9",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         { "name": "blobId", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         { "name": "data", "type": "bytes" },
         { "name": "final", "type": "boolean" }
     ]
}
```

11 GrowingObject (Protocol 6)

ProtocolID: 6

Defined Roles: store, customer

GrowingObject (Protocol 6) allows customer applications to operate independently on the two main elements that comprise a growing data object: its "header" (or parent) data object and its set of parts, which are index-based (i.e., either time or depth). (For a definition of growing data object, see Section **11.1.1**.)

The ability to operate separately on the header and set of parts supports use cases and workflows that minimize traffic on the wire. For example, an end-user of a customer application can get a list of only headers to review, and then determine which headers they want to get some or all of the parts for.

GrowingObject (Protocol 6) defines messages that allow a customer to work with growing data object headers, individual parts, or a range of parts, independently of one another. The combination of these messages lets customers:

- Edit existing growing data objects, by editing the header, the set of parts, or both.
  - To edit growing data objects a customer MUST use GrowingObject (Protocol 6) but some operations on growing data objects MAY be done with Store (Protocol 4) (see below on this page).
- Add new growing data objects.
- Request metadata about the parts in growing data objects.
- Do a full range of operations on the set of parts: including get, put, delete and do similar operations on a specified range of parts.
- Determine what intervals of growing data objects have changed while disconnected, which helps guide "catch up" operations and minimizes the likelihood of having to get "all data" again.

NOTE: All of these operations work the same for all growing data objects, regardless of their current design. That is, all growing data objects (e.g., in WITSML v2.0) are NOT currently designed identically, but all are handled with Protocol 6.

Each Energistics domain standard defines its growing data objects; for the list of growing data objects, see an ML's ETP implementation specification.

Other ETP sub-protocols that may be used with GrowingObject (Protocol 6):

- To subscribe to notifications of changes to growing data object parts that occur in Protocol 6, use GrowingObjectNotification (Protocol 7) (Chapter **12**). (These two protocols work together similarly as Store (Protocol 4) and StoreNotification (Protocol 5).)
  - **NOTE:** Use of the ***PutGrowingDataObjectsHeader*** message in this protocol primarily triggers notifications in StoreNotification (Protocol 5)—not Protocol 7. This difference is because this message actually creates or updates the growing data object (i.e., the header, which is also called the parent growing data object), not parts.
- Store (Protocol 4) allows some operations on a "complete" growing data object (complete = the growing data object "header" and all its parts). It is possible to add (insert, but NOT update), get or delete the "complete" growing data object. See Chapter **9**.
- To query the parts of a growing data object, see GrowingObjectQuery (Protocol 16) (Chapter **17**).

NOTE: Beginning with WITSML v2.0, Logs are no longer categorized as growing data objects (they were in WITSML v1.4.1.1) but are explicitly defined using the Channel, ChannelSet and Log data objects. To edit channel data, you MUST use protocols specifically designed for channels (see ChannelSubscribe (Protocol 21), Chapter **19** and ChannelDataLoad (Protocol 22), Chapter **20**).

This chapter includes main sections for:

- Key ETP concepts that are important to understanding how this protocol is intended to work (see Section **11.1**).
- Required behavior, which includes:
  - Description of the message sequence for main tasks, along with required behavior, use of capabilities, and possible errors (see Section **11.2.1**).
  - Other functional requirements (not covered in the message sequence) including use of additional endpoint, data object, and protocol capabilities for preventing and protecting against aberrant behavior (see Section◦**11.2.2**).
  - Definitions of the endpoint, data object, and protocol capabilities used in this protocol (see Section **11.2.2.2**).
- Sample schemas of the messages defined in this protocol (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes documentation for each field in a schema (see Section **11.3**).

11.1 GrowingObject: Key Concepts

This section defines key concepts that are important to understand for using this protocol.

11.1.1 What is a Growing Data Object and how is it Handled in ETP?

For the definition of an Energistics data object, see Section **25.1** (in **Appendix: Energistics Identifiers**).

A "growing data object" refers to a data object that gets added to (or grows) over time, it is characterized by:

- A relatively large number of occurrences of recurring data that is indexed to time or depth.
- Additional occurrences of recurring data that are inserted (or updated) over time.

By design, an Energistics' growing data object has:

- a "header" portion, which is also called the "parent" data object, which contains the identifying and shared information that is less likely to change over time.
- its set of parts, which are the indexed-based set that gets added to and edited over time, which is how the object "grows".

These data objects typically exist in the drilling domain and are defined in WITSML, such as trajectories (grows as new trajectory stations are added) and "mud logs" (now called wellbore geology). "Growing" is in contrast to "static" data objects, which are called this because they typically change only when people, processes, and/or software change them.

Each Energistics domain standard defines its growing data objects; for the list of growing data objects, see an ML's ETP implementation specification.

NOTE: Beginning with WITSML v2.0, Logs are no longer categorized as growing data objects (they were in WITSML v1.4.1.1) but are explicitly defined using the Channel, ChannelSet and Log data objects. To edit channel data, you MUST use protocols specifically designed for channels (see ChannelSubscribe (Protocol 21, Chapter **19** and ChannelDataLoad (Protocol 22), Chapter **20**).

11.1.2 Most Actions are on the "Parts" in the Context of One "Parent" Data Object

GrowingObject (Protocol 6) has 3 main kinds of messages, one for each kind of data the protocol operates on: parts, ranges of parts, and headers. Each message name contains the word "parts", "range" or "header" depending on the type of data it was designed to handle.

Key message types and related facts include:

- Most "part" and "range" messages are operations for the parts or ranges of parts in one growing data object. (**EXCEPTION: *GetPartsMetadata*** returns metadata for a list of growing data objects, not just one data object.) That is, each part is sent in the context of one "parent" data object and involves sending/receiving one or more object fragments in a format (e.g., XML or JSON) that comprise the growing parts of the data object. The parent data object is always referenced by its URI.

  - Each individual part in a growing data object is identified by a UID that must be unique within the context of the parent data object. NOTE: The application that first creates a growing data object assigns its UUID (for more information see Section **25.2**); the application that first creates parts of a growing data object assigns part UIDs.
    NOTE: Some parts are also data objects themselves. These parts have both a UID and a UUID. GrowingObject (Protocol 6) references these parts by their UID, NOT their UUID.
  - A range of parts is specified with an *indexInterval*, which is defined in the relevant messages in this document.

- "header" messages are get or put operations for one or more growing data object(s), each one identified by its URI.
  - Put header messages MAY include parts when first adding a growing data object to a store.
  - Put header messages MUST NOT include parts when updating an existing growing data object header in a store.
  - Get header messages do NOT return parts.
  - As stated above, if an application creates a growing data object, that application must assign the growing data object's UUID.
  - If any parts are included when creating the growing data object, the application must also assign UIDs to the parts.

11.1.3 An Update Operation on a Range of Parts is an Atomic Operation

This protocol defines a message named ***ReplacePartsByRange***, which allows a customer to specify a range of parts to be deleted and (optionally) replaced with another specified set of parts.

This operation is an atomic operation, which means the entire request either succeeds or fails. Operational details are described below in this chapter.

11.1.4 Change Annotations

A change annotation is an ETP data structure (***ChangeAnnotation*** record; see Section **23.34.18**) that describes a range of data that has changed (historical data changes) in a store. ETP stores use ***ChangeAnnotation*** records to track changes to channel data and parts in growing data objects. A ***ChangeAnnotation*** includes the inclusive range of data affected by the change and the timestamp associated with the change.

IMPORTANT: When data is appended to a channel or growing data object, no ***ChangeAnnotation*** is created.

When a customer reconnects to a store, it can request change annotations to help understand what has changed while it was disconnected and determine necessary actions based on the information.

For more information on how change annotations are used, see **Appendix: Data Replication and Outage Recovery Workflows**.

This section includes these sub-sections:

- **11.1.4.1 Definitions for ChangeAnnotation-Related Behavior**
- **11.1.4.2 Overview of How Change Annotations Work**

For information on how to create and manage ***ChangeAnnotation*** records, see these sections (which are in the General Requirements section, Section **11.2.2**):

- **11.2.2.2 Rules for Creating Change Annotations for Channel Data Objects**
- **11.2.2.3 Rules for Creating Change Annotations for Growing Data Objects**
- **11.2.2.4 Rules for Merging Change Annotations**

11.1.4.1 Definitions for ChangeAnnotation-Related Behavior

The following definitions are used to explain store behavior for creating and maintaining ***ChangeAnnotation*** records in response to changes to channel data and growing data object parts.

IMPORTANT: Pay careful attention to these definitions. They were deliberately and carefully chosen to allow optimized store behavior.

Term	Definition
Adjacent	• Two ranges are adjacent when the end index of one is equal to the start index of the other. • Two ***ChangeAnnotations*** records are adjacent if the ranges defined by their *interval* fields are adjacent. **IMPORTANT:** Even though ***ChangeAnnotation*** records may be adjacent, store customers MUST consider the entire interval in a ***ChangeAnnotation*** to be affected by the change, including the end index. When ***ChangeAnnotation*** records are adjacent, store customers MUST consider the *changeTime* for channel data points or non-interval parts (e.g., WITSML TrajectoryStations) at the index value shared by both ***ChangeAnnotation*** records to be the most recent *changeTime* of the two records.
Append	An append is when new data points or parts are added to the “end” of a channel or growing data object such that: 1. No added data point or part overlaps the existing data range. 2. For increasing data, all added data points and parts have a primary index value or start index value that is greater than or equal to the end index of the existing data range. 3. For decreasing data, all added data points and parts have a primary index value or start index value that is less than or equal to the end index of the existing data range.
Covering	A range covers an index value if the index value is: • For increasing data, greater than or equal to the range’s start index and less than or equal to the range’s end index. • For decreasing data, less than or equal to the range’s start index and greater than or equal to range’s end index. A range covers another range if it covers both the start and end index of the other range.
Decreasing data	Data for a channel or growing data object is decreasing if the *direction* field on the ***IndexMetadataRecord*** for the primary index is set to “Decreasing”. With decreasing data, the end index is less than or equal to the start index for all data ranges. This includes the data range for the channel or growing data object. This also includes the *interval* field on any ***ChangeAnnotation*** record for the channel or growing data object.
Increasing Data	Data for a channel or growing data object is increasing if the *direction* field on the ***IndexMetadataRecord*** for the primary index is set to “Increasing”. With increasing data, the end index is greater than or equal to the start index for all data ranges. This includes the data range for the channel or growing data object. This also includes the *interval* field on any ***ChangeAnnotation*** record for the channel or growing data object.
Inside	An index value is inside a range if: • For increasing data, strictly greater than the range’s start index and strictly less than the range’s end index. • For decreasing data, strictly less than the range’s start index and strictly greater than the range’s end index. If range A’s start index and end index are both inside range B, then range A is inside range B.

Term	Definition
Overlapping	Range A and Range B overlap when they are NOT adjacent and any index value is in both Range A and Range B. That is, when any of the following are true: 1. Range A's start index is the same as range B's start index. 2. Range A's end index is the same as range B's end index. 3. Range A's start index or end index are inside Range B. 4. Range B's start index or end index are inside Range A. Two ***ChangeAnnotations*** records overlap if the ranges defined by their *interval* fields overlap. **NOTE:** Adjacent ranges and adjacent ***ChangeAnnotation*** records do NOT overlap each other.
Prepend	A prepend is when new data points or parts are added to the "start" of a channel or growing data object such that: 1. No added data point or part overlaps the existing data range. 2. For increasing data, all added data points and parts have a primary index value or end index value that is less than or equal to the start index of the existing data range. 3. For decreasing data, all added data points and parts have a primary index value or end index value that is greater than or equal to the start index of the existing data range.

11.1.4.2 Overview of How Change Annotations Work

Stores must persist only one ***ChangeAnnotation*** for any range of data. ***ChangeAnnotation*** records MAY be adjacent, but they MUST NOT overlap.

Stores must merge any overlapping ***ChangeAnnotation*** records. When this happens, the store must use the most recent change time for any annotations that are merged together. A store may also choose to combine non-overlapping intervals together, which may simplify the bookkeeping needed for the store at the expense of having customers potentially request additional data. Additionally, a store may age out or remove ***ChangeAnnotation*** records with a *changeTime* that is older than the store's ChangeRetentionPeriod.

When an ETP customer connects to a store, it should request ***ChangeAnnotation*** records for any channel or growing data objects that it wants to inspect for changes to previously known data. A customer must assume any data covered by the range of a ***ChangeAnnotation*** interval to be affected by the change at the time in the annotation, even if some of the data was unaffected by the change.

11.2 GrowingObject: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.
- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**
- The customer has the URIs for the growing data objects of interest, which may found using Discovery (Protocol 3) or may come out of band of ETP (e.g., in an email).

11.2.1 GrowingObject: Message Sequences

This section explains the basic message sequence for main tasks to be done using this protocol and includes related key behaviors. The following General Requirements section provides additional requirements and rules for how this protocol works.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section.

GrowingObject (Protocol 6): Basic Message-Response flow by ETP Role	
Message sent by customer	**Response Message from store**
GetPartsMetadata: Request for parts metadata for a list of growing data objects.	***GetPartsMetadataResponse*** (multipart): The list of growing data objects and parts metadata for each that the store could return.
GetGrowingDataObjectsHeader: Request for header information only for a list of growing data objects.	***GetGrowingDataObjectsHeaderResponse*** (multipart): The list of growing data object header information that the store could return.
PutGrowingDataObjectsHeader: Request to add or update the header information for a list of growing data objects; each object is identified by a URI and includes the header data being inserted or updated.	***PutGrowingDataObjectsHeaderResponse*** (multipart): The list of growing data object headers that the store successfully inserted or updated.
GetParts: Request for the list of parts in one parent growing data object, each part identified by a UID.	***GetPartsResponse*** (multipart): The list of parts and data for each that the store could return.
PutParts: Request to add or update one or more parts in one parent growing data object; each part is identified by its UID and includes the new part data to be inserted/updated in the store.	***PutPartsResponse*** (multipart): The list of parts that the store successfully inserted or updated.
DeleteParts: Request to delete one or more parts in one parent growing data object, each part identified by a UID.	***DeletePartsResponse*** (multipart): The list of parts that the store successfully deleted.
GetPartsByRange: Request to retrieve a range of parts as specified by the start and end index of an interval.	***GetPartsByRangeResponse*** (multipart): The list of parts and data for each that were in the specified interval.
ReplacePartsByRange (multipart): Request to delete a range of parts as specified by the start and end index of an interval and (optionally) specify a set of parts to replace the deleted parts.	***ReplacePartsByRangeResponse***: Empty message (no data fields) that indicate the operation completed.
GetChangeAnnotations: A request for changes to the parts of a specified list of growing data objects since a specific time.	***GetChangeAnnotationsResponse:*** (multipart): The list of changed intervals, per the request.

11.2.1.1 To get parts metadata for one or more growing data objects:

1. The customer MUST send the store the ***GetPartsMetadata*** message (Section **11.3.1**), which contains a map whose values MUST each be the URI of a growing data object that the customer wants to get parts metadata for.
2. For the growing data objects that the store successfully returns parts metadata for, it MUST send one or more ***GetPartsMetadataResponse*** map response messages (Section **11.3.2**), which contains a map whose values are ***PartsMetadataInfo*** records (Section **23.34.17**).

   a. For more information on how map response messages work, see Section **3.7.3**.

3. For the URIs it does NOT successfully return parts metadata for, the store MUST send one or more map ***ProtocolException*** messages, where values in the *errors* field (a map) are appropriate errors, such as ENOT_FOUND (11).
   a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.

11.2.1.2 To get the headers for one or more growing data objects:

1. The customer MUST send the store the ***GetGrowingDataObjectsHeader*** message (Section **11.3.3**), which contains a map whose values MUST be the URI of a growing data object that the customer wants to get header information for.
2. For the URIs it successfully returns growing data object header information for, the store MUST send one or more ***GetGrowingDataObjectsHeaderResponse*** map response messages (Section **11.3.4**) where the map values are ***DataObject*** records (Section **23.34.5**) with the growing data object URIs and header data.
   a. For more information on how map response messages work, see Section **3.7.3**.
3. For the URIs it does NOT successfully return growing data object header information for, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors, such as ENOT_FOUND (11).
   a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.

11.2.1.3 To get the parts for one growing data object:

1. The customer MUST send the store the ***GetParts*** message (Section **11.3.1**), which contains the URI of the parent growing data object and a map whose values MUST be the part UIDs for each part the customer wants to get.
2. For the UIDs it successfully returns parts for, the store MUST send one or more ***GetParts*** map response messages (Section **11.3.2**) where the map values are ***ObjectPart*** records with the part UIDs and data.
   a. For more information on how map response messages work, see Section **3.7.3**.
   b. **Order of parts returned.** The store is expected to respect the order specified by an Energistics domain standard (e.g., WITSML). For more information, see the ML's ETP implementation specification.
      i. The part order SHOULD be stable. For example, if there are two trajectory stations with the same measured depth, the store should return these in a consistent order across all requests. **RECOMMENDATION:** Use the same order as in GrowingObjectQuery (Protocol 16) for the ***FindPartsResponse*** message. For more information, see Section **14.1.2.1**.
3. For the UIDs it does NOT successfully return parts for, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors, such as ENOT_FOUND (11).
   a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.

11.2.1.4 To get a range of parts (interval) for one growing data object:

1. The customer MUST send the store the ***GetPartsByRange*** message (Section **11.3.11**), which contains the URI of the parent growing data object, the index interval (which specifies the range of interest), and a flag to *includeOverlappingIntervals*.
   a. For more information on how overlapping intervals work, see Section **11.2.2.1**.

2. If the store successfully returns parts from the request interval, it MUST send one or more ***GetPartsByRangeResponse*** messages (Section **11.3.12**), each of which contains an array of UIDs and data for each part that the store could return.
    a. The store MUST limit the total count of parts returned to the customer's value for MaxResponseCount protocol capability.
    b. The customer MAY notify the store of responses that exceed this limit by sending error ERESPONSECOUNT_EXCEEDED (30).
    c. If a store's value for MaxResponseCount protocol capability is smaller than a customer's value, a store MAY further limit the total count of parts to its value.
    d. If a store is unable to return all parts to a request due to exceeding the lower of the customer's or the store's value for MaxResponseCount protocol capability, the Store MUST terminate the multipart response by sending error ERESPONSECOUNT_EXCEEDED (30).
        i. A store MUST NOT send ERESPONSECOUNT_EXCEEDED until it has sent MaxResponseCount parts.
3. If the store has no parts in the request interval, it MUST send a ***GetPartsByRangeResponse*** message with the FIN bit set and the *parts* field set to an empty array.
4. If the store does NOT successfully return parts or a ***GetPartsByRangeResponse*** with an empty *parts* array, it MUST send a non-map ***ProtocolException*** message with an appropriate error, such as EREQUEST_DENIED (6).

11.2.1.5 To add or update the headers for one or more growing data objects:

1. The customer MUST send the store the ***PutGrowingDataObjectsHeader*** message (Section **11.3.7**), which contains a map whose values MUST be the URIs of the growing data objects that the customer wants to add (insert) or update and the data for each.
    a. **REMINDER:** ETP uses "upsert" semantics, so all put operations are a complete replace of any existing data. For more information, see Section **9.1.1**.
    b. A customer MUST honor the store's MaxDataObjectSize capability. For more information, see Section **3.3.2.4**.
    c. When adding a new growing data object, the growing data object MAY include parts. When updating an existing growing data object, the growing data object MUST NOT include parts. For additional details on required behavior when adding parts, see Section **11.2.1.6**.
    d. When a growing data object includes parts, the customer MUST honor the store's MaxPartSize capability. For more information, see Section **3.3.2.5**.
2. For growing data object headers it successfully puts (add to/replace in the store), the store MUST send one or more ***PutGrowingDataObjectsHeaderResponse*** map response messages (Section **11.3.8**).
    a. For more information on how map response messages work, see Section **3.7.3**.
    b. The store MUST send this message AFTER it performs these operations:
        i. If the growing data object does not exist in the store, the store MUST add it. If the growing data object includes parts, the store MUST follow the same rules defined for Store (Protocol 4) when creating a growing data object that includes parts as described in Section **9.2.2**, Row◦**21**. If the parts are themselves also data objects, the store MUST also follow the rules described in Section **9.2.2**, Row **22**.
        ii. If the growing data object does exist in the store and the customer included parts in the update, the store MUST reject the update and send error EUPDATEGROWINGOBJECT_DENIED (23).

   iii. If the growing data object does exist in the store, the store MUST replace the entire existing header with the information the customer provided in the ***PutGrowingDataObjectsHeader*** message.

   iv. Store-managed fields on the ***Resource*** only (*storeCreated* and *storeLastWrite*) MUST be updated for these operations; for more information, see Section **11.2.2**, Row **8**.

   c. Successful put header operations MAY trigger notifications in StoreNotification (Protocol 5) (because putting a header = inserting or updating a data object). For more information, see Section **10.2.2**, Row **9**.

   d. **NOTIFICATION BEHAVIOR:** When a put header operation succeeds and includes parts, the store MUST send ***PartsChanged*** notifications as described in Section **11.2.1.6** for the added or updated parts.

3. For growing data object headers the store does NOT successfully put, it must send a ***ProtocolException*** message with *errors* field (map) whose values MUST be the URIs of the growing data objects from the request that could not be added and an appropriate error code for each, for example, EREQUEST_DENIED (6).

   a. For more information about use of ***ProtocolException*** messages with plural messages, see Section **3.7.3**.

4. After adding a growing data object header, a customer can use the ***PutParts, DeleteParts*** and ***ReplacePartsByRange*** messages to add and edit a growing data object's parts.

11.2.1.6 To add or update one or more parts for one growing data object:

1. The customer MUST send the store the ***PutParts*** message (Section **11.3.5**), which contains the URI of the parent growing data object and a map whose values MUST be the UIDs and data for each part that the customer wants to add (insert) or update.

   a. ***PutParts*** represents a set of distinct add or update operations. It does not explicitly operate on a range of data. To operate on a range of data, use ***ReplacePartsByRange***.

   b. **REMINDER:** ETP uses "upsert" semantics, so all put operations are a complete replace of any existing data. For more information, see Section **9.1.1**.

2. For the parts it successfully puts (add to/replace in the store), the store MUST send one or more ***PutPartsResponse*** map response messages (Section **11.3.6**).

   a. For more information on how map response messages work, see Section **3.7.3**.

   b. The store MUST send this message AFTER it performs these operations:

      i. If the parts do not exist in the store, the store MUST add them.

         1. If the parts are themselves also data objects, adding new parts MUST NOT exceed the store's value for MaxContainedDataObjectCount data object capability for the parent growing data object type. For each part that would exceed this limit, the store MUST NOT add the part. The store MUST instead send ELIMIT_EXCEEDED (12).

      ii. If the parts do exist, the store MUST replace them with the information the customer provided in the ***PutParts*** message.

      iii. For BOTH i and ii, the store MUST do the following:

         1. If the parts are themselves also data objects, the store MUST also follow these rules for the parts:

            a. The rules for putting data objects into a store defined in Section **9.2.1.2**.

            b. The store MUST link any parts not previously in the growing data object to the growing data object.

c. The additional rules for putting parts that are data objects into a store defined in Section **9.2.2**, Row **23**.

2. Update the *storeLastWrite* field on the growing data object's ***Resource***. For more information, see Section **11.2.2**, Row **8**.
3. Update the *activeStatus* field on the growing data object. For more information, see Section **11.2.2**, Row **9**.
4. Create appropriate ***ChangeAnnotation*** records. For more information, see Section **11.2.2.3**.

3. For the parts it does NOT successfully put, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors, such as EREQUEST_DENIED (6).
    a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.
4. **NOTIFICATION BEHAVIOR:** The store MUST send a ***PartsChanged*** notification message with a type (objectChangeKind) of "insert" or "update".
    a. If the parts are themselves also data objects, for any parts that were newly linked to the growing data objects, the store MUST send an ***ObjectChanged*** notification with *ObjectChangeKind* set to “joined”.
    b. When a ***PutParts*** message both inserts and updates parts, 2 ***PartsChanged*** notifications must be sent: one for the inserted parts and one for the updated parts.
    c. A store MUST send a notification for only the most recent effective state of a part. So if multiple insert or update changes to a part since the notifications were sent for the part, the store MAY send only one notification.
        i. If the part is in a range that will be included in a ***ReplacePartsByRange*** message, ***PartsChanged*** MUST NOT be sent. Instead, the part MUST be included in the ***PartsReplacedByRange*** message.
        ii. If the part will NOT be included in a ***ReplacePartsByRange*** and it was inserted since the most recent insert or update notification was sent, the store MUST send an insert notification with the timestamp of the most recent insert or update change.
        iii. Otherwise, the store MUST send an update notification with the timestamp of the most recent update.
    d. Notifications are sent in GrowingObjectNotification (Protocol 7). For more information on rules for populating/sending notifications and why notification behavior is specified here, see Section **11.2.2**, Row **5**.
    e. When the parts in a ***PutParts*** message are themselves also data objects, the store MUST also send ***ObjectChanged*** notification messages in StoreNotification (Protocol 5) as described in Section **9.2.1.2** and Section **9.2.2**.

11.2.1.7 To delete one or more parts from one growing data object:

1. The customer MUST send the store the ***DeleteParts*** message (Section **11.3.9**), which contains the URI of the parent growing data object and the map whose values MUST be the part UIDs that the customer wants to delete.
    a. When the parts in a ***DeleteParts*** message are themselves data objects, the store MUST also treat ***DeleteParts*** as a request to delete (NOT prune or unjoin) the data objects.
2. For the parts it successfully deletes, the store MUST send one or more ***DeletePartsResponse*** map response messages (Section **11.3.10**).

   a. For more information on how map response messages work, see Section **3.7.3**.

   b. The store MUST send this message AFTER it performs these operations:

      i. Update the *storeLastWrite* field on the growing data object's ***Resource***. For more information, see Section **11.2.2**, Row **8**.

      ii. Update the *activeStatus* field on the growing data object. For more information, see Section **11.2.2**, Row **9**.

      iii. Create appropriate ***ChangeAnnotation*** records. For more information, see Section **11.2.2.3**.

      iv. If the parts are themselves also data objects, the store MUST also follow these rules for the parts:

         1. The rules for deleting data objects from a store defined in Section **9.2.1.3**.

         2. The additional rules for deleting contained data objects defined in Section **9.2.2**, Row **25**.

         3. The additional rules for deleting parts that are data objects from a store defined in Section **9.2.2**, Row **27**.

3. For the parts it does NOT successfully delete, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors, such as ENOT_FOUND (11) or EREQUEST_DENIED (6).

   a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.

4. **NOTIFICATION BEHAVIOR:** The store MUST send a ***PartsDeleted*** notification message.

   a. A store MUST send a notification for only the most recent effective state of a part. So if notifications are queued, and the part is subsequently deleted, the store MAY discard any previous notifications.

   b. If the part is in a range that will be included in a ***ReplacePartsByRange*** message, ***PartsChanged*** MUST NOT be sent. Instead, the part MUST be included in the ***PartsReplacedByRange*** message.

   c. Notifications are sent in GrowingObjectNotification (Protocol 7). For more information on rules for populating/sending notifications and why notification behavior is specified here, see Section **11.2.2**.

   d. When the parts in a ***DeleteParts*** message are themselves also data objects, the store MUST also send ***ObjectDeleted*** notification messages in StoreNotification (Protocol 5) as described in Section **9.2.1.3** and Section **9.2.2**.

11.2.1.8 To delete a range of parts (interval) and (optionally) replace it with another range of parts:

1. The customer MUST send the store the ***ReplacePartsByRange*** message (Section **11.3.15**), which contains these fields: *uri,* which MUST be the URI of the parent growing data object from which the parts are to be deleted; *deleteInterval*, which MUST specify he index interval for the range of parts to be deleted; *parts*, which is an array that MUST identify the UIDs and data for each part that is to be added (i.e., the new parts that will replace the parts that have been deleted); and the *includeOverlappingIntervals* flag.

   a. The number of parts deleted DOES NOT have to equal the number of parts added.

   b. If the *parts* field is left empty, then the message is a delete request for the interval specified in *deleteInterval.*

   c. For information on how overlapping intervals work, see Section **11.2.2.1**.

   d. When the parts deleted by a ***ReplacePartsByRange*** message are themselves data objects, the store MUST also treat ***ReplacePartsByRange*** as a request to delete (NOT prune or unjoin) the data objects.

2. ***ReplacePartsByRange*** is an atomic operation: the entire request either succeeds or fails.
    a. The store MUST delete the range of parts specified in *deleteInterval* and replace it with the parts specified in *parts*.
        i. If the parts are themselves also data objects, the store MUST also follow these rules for the deleted parts:
            1. The rules for deleting data objects from a store defined in Section **9.2.1.3**.
            2. The additional rules for deleting contained data objects defined in Section **9.2.2**, Row **25**.
            3. The additional rules for deleting parts that are data objects from a store defined in Section **9.2.2**, Row **27**.
        ii. If the parts are themselves also data objects, the store MUST also follow these rules for the replacement parts:
            1. The rules for putting data objects into a store defined in Section **9.2.1.2**.
            2. The store MUST link any parts not previously in the growing data object to the growing data object.
            3. The additional rules for parts that are data objects into the store defined in Section **9.2.2**, Row **23**.
        iii. If it completes these operations successfully, it MUST send a ***ReplacePartsByRangeResponse*** message (Section **11.3.16**), which is a "success only" message indicating that the store has successfully completed the entire operation as requested.
    b. If any replacement part is NOT covered by the *deleteInterval*, the store MUST fail the operation and send EINVALID_OPERATION (32). **NOTE:** *includeOverlappingIntervals* DOES NOT allow replacement parts to overlap the *deleteInterval*. They MUST always be covered by the *deleteInterval*.
    c. If the parts are themselves data objects and adding the replacement parts would exceed the store's value for MaxContainedDataObjectCount data object capability for the parent growing data object type, the store MUST fail the operation and send ELIMIT_EXCEEDED (12).
    d. After deleting the range of parts specified in *deleteInterval*, if any replacement parts would have the same UID as a part still in the growing data object, the store MUST fail the operation and send EINVALID_OPERATION (32). That is, replacement parts MUST ONLY replace parts that are deleted by the message.
    e. If the operation fails, the store MUST:
        i. Rollback the entire request. That is, the store MUST be in the state it was in before receiving the ***ReplacePartsByRange*** message.
        ii. Send a non-map ***ProtocolException*** message with an appropriate error code such as EREQUEST_DENIED (6).
3. **NOTIFICATION BEHAVIOR:** The store MUST send a ***PartsReplacedByRange*** notification message.
    a. A store MUST send a notification for only the most recent effective state of a part. So if notifications are queued:
        i. If the parts affected by a ***ReplacePartsByRange*** message were PREVIOUSLY affected by ***PutParts*** or ***DeleteParts*** messages before ***PartsReplacedByRange*** is sent, the store MAY discard the previous notifications and only send ***PartsReplacedByRange***.

ii. If the parts affected by a ***ReplacePartsByRange*** message were LATER affected by other ***PutParts*** or ***DeleteParts*** messages before ***PartsReplacedByRange*** is sent, the store MAY discard the later notifications and only send ***PartsReplacedByRange*** with *changeTime* set to the most recent change covered by range included in the message.

iii. If a range is affected by more than one ***ReplacePartsByRange*** message before ***PartsReplacedByRange*** is sent, a store MAY choose to only send one ***PartsReplacedByRange*** message that covers the combined range of all relevant ***ReplacePartsByRange*** messages with *changeTime* set to the most recent relevant timestamp.

iv. When combining multiple notifications into a single ***PartsReplacedByRange*** message, the store MUST set *includeOverlappingIntervals* to false, set the *deletedInterval* to the smallest range that covers all affected parts, and include as replacement parts any existing parts covered by the message's *deletedInterval*.

b. Notifications are sent in GrowingObjectNotification (Protocol 7). For more information on rules for populating/sending notifications and why notification behavior is specified here, see Section **11.2.2**.

c. When the parts deleted by a ***ReplacePartsByRange*** message are themselves also data objects, the store MUST also send ***ObjectDeleted*** notification messages in StoreNotification (Protocol 5) as described in Section **9.2.1.3** and Section **9.2.2**.

11.2.1.9 To determine what has changed in a store after a disconnect (using ChangeAnnotations):

This process can be used by a customer anytime it first connects to a store and wants to determine latest changes on the parts/interval in growing data objects of interest (it must have the growing data object's URI). (There are similar processes for channel and other data objects; for more information about related workflows, see **Appendix: Data Replication and Outage Recovery Workflows**.)

ETP has no session survivability. So if a session is interrupted (e.g., a satellite connection drops), using this process makes it easier for a customer to determine what has changed while disconnected, get any changed data it requires, and resume operations that were in process when the dropped session happened—with a reduced likelihood of NOT having to "resend all data from the beginning" (i.e., all data from before the session dropped).

1. The customer MUST reconnect (as described in Chapter **5**) and MAY want to get parts metadata using the process described in Section **11.2.1.1**.

2. To determine what has changed while disconnected, the customer MUST send the store a ***GetChangeAnnotations*** message (Section **11.3.17**).

   a. This message contains a map whose values MUST be the URIs of growing data objects to get change annotations for. In the message, the customer MUST also enter a "changes since" time (that is, the customer wants all changes since this time, which should be based on the time the customer was last sure it received data from the store) and indicate if it wants all change annotations or only the latest change annotation for each growing data object.

      i. The "changes since" time (*sinceChangeTime* field) MUST BE equal to or more recent than the store's ChangeRetentionPeriod endpoint capability.

3. For URIs it successfully returns change annotations for, the store MUST respond with one or more ***GetChangeAnnotationsResponse*** map response messages (Section **11.3.18**).

   a. For more information on how map response messages work, see Section **3.7.3**.

   b. The map values in each message are ***ChangeResponseInfo*** records (Section **23.34.19**), which contains a time stamp for when the response was sent and the ***ChangeAnnotation*** records (Section **23.34.18**) for a growing data object.

i. Each ***ChangeAnnotation*** record contains a timestamp for when the change occurred in the store and the interval of the growing data object that changed. **(NOTE:** Change annotations keep track ONLY of the interval that changed, NOT the actual data that changed).

c. For information about how the store tracks and manages these change annotations, see Section◦**11.2.2**, Row **15**).

4. For the URIs it does NOT successfully return change annotations for, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors.

   a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.

5. Based on information in the ***GetChangeAnnotationsResponse*** message, the customer MAY:

   a. Use the ***GetPartsByRange*** message to retrieve intervals of interest that have changed (as described in Section **11.2.1.4**).

   b. Re-establish growing data object or growing data object parts notification subscriptions that were in place when a session was disconnected, (see Sections **10.2.1.1** and **12.2.1.1**, respectively).

11.2.2 GrowingObject: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) additional rows with additional requirements for specific types of operations.

Row#	Requirement	Description
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter **5**. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session.

Row#	Requirement	Description
		i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3**. a. For the list of global capabilities and related behavior, see Section **3.3.2**. 3. Section **11.2.2.2** identifies the capabilities most relevant to this ETP sub-protocol. Additional details for how to use the protocol capabilities are included below in this table and in Section **11.2.1 GrowingObject: Message Sequence**. 4. Endpoint Capability MaxPartSize must be honored for most requests and response in this protocol. For the general behavior that must be applied, see Section **3.3.2.5**.
3.	Message Sequence See Section **11.2.1**.	1. The Message Sequence section above (Section **11.2.1**) describes requirements for the main tasks listed there and also defines required behavior.
4.	Plural messages (which includes maps)	1. This protocol uses plural message. For detailed rules on handling plural messages (including handling of ***ProtocolException*** messages), see Section **3.7.3**.
5.	Notifications	1. This chapter explains events (operations) in GrowingObject (Protocol 6) that trigger the store to send notifications, which the store sends using StoreNotification (Protocol 5) and/or GrowingObjectNotification (Protocol 6). However, statements of **NOTIFICATION BEHAVIOR** are here in this chapter, in the context of the detailed explanation of the behavior that triggers the notification. 2. Notification behavior is described here using MUST. However, the store MUST ONLY send notifications IF AND ONLY IF there is a customer subscribed to notifications for an appropriate context (i.e., a context that includes the data object) and the store MUST ONLY send notifications to those customers that are subscribed to appropriate contexts. a. For more information on data object notifications, see Chapter **10 StoreNotification (Protocol 5)**. b. For information on notifications for parts in growing data objects, see Chapter◦**12 GrowingObjectNotification (Protocol 7)**.
6.	Growing data object operations that may be performed using Store (Protocol 4)	1. To perform all operations listed in this row, a customer MUST use messages in Store (Protocol 4); for more information, see Chapter **9**. a. **To add (insert) a new growing data object and its parts in one operation**, a customer MUST use a ***PutDataObjects*** message. See Section **9.2.1.2**. i. A customer MAY add a growing data object using GrowingObject (Protocol 6) by first adding the growing data object header and then adding the parts. For more information, see Sections **11.2.1.5** and **11.2.1.6**. b. **To get a growing data object and its parts in one operation**, a customer MUST use a ***GetDataObjects*** message. See Section **9.2.1.1**. c. **To delete a growing data object**, a customer MUST use a ***DeleteDataObjects*** message. See Section **9.2.1.3**. i. A growing data object CANNOT be deleted using GrowingObject (Protocol 6), only Store (Protocol 4).
7.	Growing data object operations that MUST be performed using GrowingObject (Protocol 6)	1. To perform all operations listed in this row, a customer MUST use messages in GrowingObject (Protocol 6): a. All "updates" to growing data objects, for header and parts information. b. All operations (additions, edits, deletes) on parts only in the context of one growing data object. c. For the list of all tasks that can be done in this protocol and how they work, see Section **11.2.1**.

Row#	Requirement	Description
8.	**Store Behavior:** Updates to *storeCreated* and *storeLastWrite* fields.	1. Each ***Resource*** in ETP has these two fields: *storeCreated* and *storeLastWrite*. a. These fields appear ONLY on the ***Resource*** NOT on the data object and are used in workflows for eventual consistency between 2 stores. b. For more information about these fields, see Section **3.12.5.2** and their definitions/required format in ***Resource*** (see Section **23.34.11**). 2. For operations in GrowingObject (Protocol 6) that ADD a new data object (e.g., ***PutGrowingDataObjectsHeader***), the store MUST do both of these: a. Set the *storeCreated* field to the time that the header was added in the store. b. Set the *storeLastWrite* to the same time as *storeCreated*. 3. For operations in GrowingObject (Protocol 6) that result in ANY CHANGE to the growing data object—header or its parts—the store MUST update the *storeLastWrite* field with the time of the change in the store.
9.	**Store Behavior:** Updates to *activeStatus* field	1. The ***Resource*** (Section **23.34.11**) associated with each data object in ETP has an *activeStatus* field. a. This field appears ONLY on the ***Resource*** NOT on the data object. There MAY be an equivalent element on the data object. The mapping between *activeStatus* and the data object element is defined by the relevant ML implementation guide. b. For growing data objects, this field may have a value of "active" or "inactive". c. For information about this field and behavior related to setting it to "inactive" related to the ActiveTimeoutPeriod capability, see Section **3.3.2.1**. 2. If a growing data object's *activeStatus* has a value of "inactive" and messages in this ETP sub-protocol begin operations that change the growing data object's parts data (e.g., appends data with a ***PutParts*** message or replaces a range with a ***ReplacePartsByRange*** message), the store MUST do the following: a. Set the growing data object's *activeStatus* to "active". b. Reset the timer for the ActiveTimeoutPeriod capability. c. **NOTIFICATION BEHAVIOR:** Send an ***ObjectActiveStatusChanged*** notification message for the growing data object in StoreNotification (Protocol 5). For more information, see Section **10.2.2**, Row **16**. 3. If a growing data object is added/inserted (e.g., in Store (Protocol 4) or with ***PutGrowingDataObjectsHeader***) the store MUST set *activeStatus* to "inactive" (the default).
10.	**Store Behavior:** Immutable elements and attributes	Some elements and attributes on Energistics growing data object headers and parts are immutable. That is, the values for these elements and attributes are set when the growing data object header or part is created, and the values cannot be changed after that. Examples of these are a data object's UUID, a part's UID, and the unit of measure for the index value of a part. Observe these rules for immutable elements and attributes: 1. When the customer creates the growing data object header or part, the store MUST use the values provided by the customer for these elements and attributes. 2. If a customer attempts to update an existing growing data object header or part and provides different values for immutable elements or attributes, the store MUST reject the update and send error EREQUEST_DENIED (6). 3. If a customer needs to change the values of any immutable elements or attributes on a growing data object header, the customer MUST first delete the entire growing data object and then recreate it with the correct values, re-adding parts as required. 4. If a customer needs to change the values of any immutable elements or attributes on a growing data object part, the customer MUST first delete the part and then recreate it with the correct values.
11.	All URIs used in this protocol must resolve to a growing data object	1. All URIs specified in messages in this protocol MUST resolve to a data object that is a growing data object; if the URI does not resolve to a growing data object, the store MUST send error ENOTGROWINGOBJECT (6001). a. Data objects that are growing data objects are identified in the relevant ML's ETP implementation specification. 2. For the messages listed below, the URI specified MUST resolve to a single growing data object (the parent growing data object for the specified parts or ranges); if it does not, send error ENOTGROWINGOBJECT (6001) GetParts | GetPartsResponse PutParts | PutPartsResponse

Row#	Requirement	Description
		DeleteParts / DeletePartsResponse GetPartsByRange / GetPartsByRangeResponse ReplacePartsByRange / ReplacePartsByRangeResponse
12.	Indexes for growing data objects	1. Indexes for growing data objects MUST be only "DateTime" or "MeasuredDepth". **NOTE: *ChannelIndexKind*** enumerates the type of indexes for both growing data objects and channel data objects. ETP specifies additional index kinds that may be used for channel indexes only.
13.	Index Metadata	1. A growing data object's index metadata MUST be consistent: a. All parts MUST have the same index unit and the same vertical datum. b. The index units and vertical datums in the growing data header MUST match the parts. c. The index units and vertical datums MUST match the index unit and vertical datum in the ***PartsMetadataInfo*** record. 2. For growing data objects that do not explicitly store the index metadata in the growing data object header: a. The index metadata for the growing data object is derived from the first part in the growing data object as defined in the relevant ML implementation guide. b. When a growing data object does not yet have any parts, the ***PartsMetadataInfo*** record MUST be populated as follows: i. *indexKind*, *direction*, and, optionally, *indexPropertyKindUri* MUST be set to the correct value based on the type of data object (e.g., MeasuredDepth and increasing for Trajectories). ii. *startIndex* and *endIndex* in *interval* MUST be null. iii. *uom*, *depthDatum*, and *both uom* and *datum* in *interval* MUST all be empty strings. iv. *name* MUST be set. 3. When sending messages, both the store AND the customer MUST ensure that all index metadata and data derived from index metadata are consistent in all fields in the message, including in XML or JSON object data or part data. a. **EXAMPLE:** The *uom* and *depthDatum* in an ***IndexInterval*** record MUST be consistent with the data object's index metadata. b. **EXAMPLE:** Data object elements related to index values in growing data object headers (e.g., MdMn and MdMx on a WITSML 2.0 Trajectory) and parts (e.g., Md on a WITSML 2.0 TrajectoryStation) MUST be consistent with each other AND the data object's index metadata. c. A store MUST reject requests with inconsistent index metadata with an appropriate error such as EINVALID_OBJECT (14) or EINVALID_ARGUMENT (5).
14.	Range operations/messages	1. Messages with the words "ByRange" in their name perform operations on the range of parts included in a specified index interval. No individual parts are listed; therefore, no UIDs are used. 2. For information on how overlapping range operations work, see Section **11.2.2.1**.
15.	**Store Behavior:** Creating and managing change annotations	1. For a definition of change annotations and related terms, see Section **11.1.4**. 2. For the requirements on how to create and manage change annotations for growing data objects, See Sections **11.2.2.3**, **11.2.2.4**, and 11.2.2.5 3. A store MUST track annotations globally, NOT per user, customer, or endpoint. a. There is NO requirement for the store to remember the data that changed, ONLY the interval where a change occurred and the time. 4. A store MUST create annotations (***ChangeAnnotation*** records) for the following operations: a. When a range of parts data is replace or deleted: i. The change annotation must reflect the full range of the delete operation. 5. Managing change intervals and annotations: a. A store MUST retain ***ChangeAnnotation*** records for its ChangeRetentionPeriod endpoint capability.

Row#	Requirement	Description
		b. For change intervals that overlap, a store MUST combine the annotations into 1 change interval/change annotation and set the *changeTime* to the *changeTime* of the most recent of those annotations that were combined. c. A store MAY combine annotations over time as it sees fit. This behavior is recommended: i. Small changes near each other (indexes/intervals), SHOULD be bundled together into a single change interval. ii. Bundling SHOULD occur in a manner that promotes efficient data transfer and honors load limits.

11.2.2.1 Overlapping Interval Range Operations

As an alternative to listing each individual part to be operated on, ETP provides functionality to specify range or interval (which is defined by a start and end index) and the operation is performed on all parts that are within that interval. This functionality is included in messages that have the words "ByRange" in their name.

However, some growing data objects (e.g., wellbore geology (previously known as mud log) have "parts" that represent a range, as opposed to a single point. These "range parts" are identified by having start/end or top/base type elements for their relevant indices. The issue with having a range part (vs. a single point) is how to handle "ByRange" operations when only a portion of the range part overlaps the specified interval of interest (i.e., the request interval).

When specifying operations with intervals, it is possible to specify whether "range parts" that partially overlap the indices of the request interval are included or not in the operation by using the *includeOverlappingIntervals* flag.

11.2.2.1.1 EXAMPLE

A growing data object has these 3 "range parts":

- Range Part 1: 1,000 to 2,000 ft
- Range Part 2: 2,000 to 3,000 ft
- Range Part 3: 3,000 to 4,000 ft

A "ByRange" request specifies an interval (request interval) of 1,500 to 3,500 ft.

- If the *includeOverlappingIntervals* flag is **true**, all 3 range parts are included in the operation (because a portion or each range part overlaps the request interval).
  - **EXAMPLE**: In the ***ReplacePartsByRange*** message, if *includeOverlappingIntervals* flag is **true**, the store will delete any range part that overlaps the *deleteInterval*, so all 3 range parts.
- If the *includeOverlappingIntervals* flag is **false**, only Range Part 2 is included in the operation (where the minimum and maximum points that define the range part are wholly contained in the request interval).
  - **EXAMPLE:** In the ***ReplacePartsByRange*** message, if *includeOverlappingIntervals* flag is **false**, the store will delete only range part that are completely contained in the *deleteInterval*, so only Range Part 2.

11.2.2.1.2 Logic for how IncludeOverlappingIntervals Works

This section explains the general logic for how the *includeOverlappingIntervals* flag works. The StartIndex/EndIndex represents the request interval and mdTop/mdBase represents the range part in the growing data object.

NOTE: Operations/checks are inclusive of the startIndex and endIndex parameters (>=, <=).

includeOverlappingIntervals: true (default behavior)

Definition: "Range parts" are affected where ANY part of their interval overlaps with the request interval. There are 4 cases of how an object may or may not overlap, or be contained within the request interval. The table below shows the logic that addresses these cases.

A. Range part falls completely within the request interval
B. mdTop is inside the request interval, but mdBase is outside
C. mdTop is outside the request interval, but mdBase is inside
D. both mdTop and mdBase are outside the request interval, but the range part SPANS the request interval.

Logic	*Case(s) Addressed*
(mdtop >= startIndex && mdTop <= endIndex)	*A & B*
|| (mdBase >= startIndex && mdBase <= endIndex)	*A & C*
|| (mdtop <= startIndex && mdBase >= endIndex)	*D*

NOTE: && indicates **AND** Operation, **||** indicates **OR** operation.

includeOverlappingIntervals: false

Range parts are affected only where their interval is wholly contained within the request interval. Any partially overlapping range parts are ignored. The following logic applies to all ByRange operations (get, put, and delete):

mdTop >= startIndex && mdTop <= endIndex

&& mdBase >= startIndex && mdBase <= endIndex

11.2.2.2 Rules for Creating Change Annotations for Channel Data Objects

Figure 20 illustrates some common types of changes to channel data and how a store may or must create or update ***ChangeAnnotation*** records in response to them.

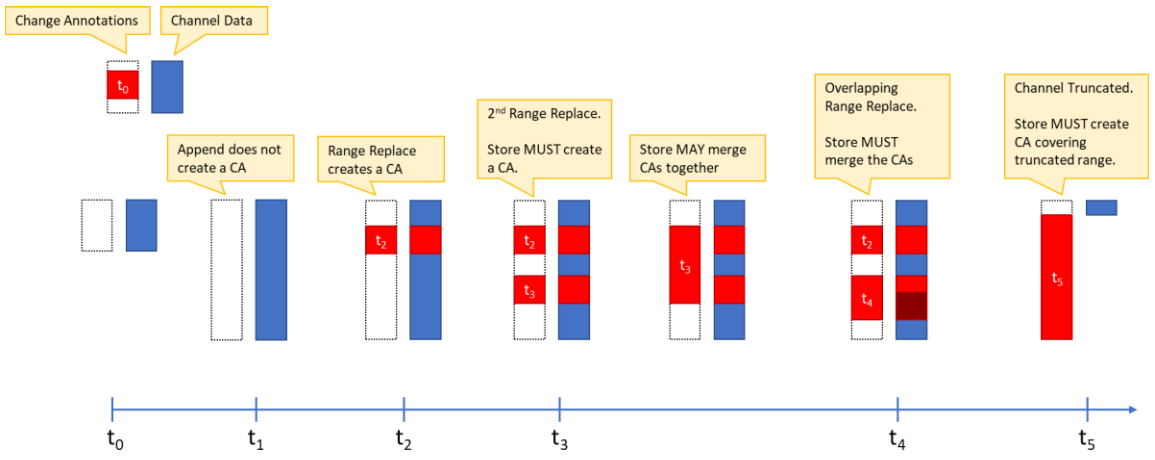


Figure 20: Example showing how change annotations (CA) work over time for channels. Blue box = channel data, white box with red label = change annotation. The size of the white box is intended to show that the annotation is for the entire corresponding channel (blue box).

The table below describes how stores create ***ChangeAnnotation*** records in different scenarios for channel data objects.

IMPORTANT: ***ChangeAnnotation*** records MUST be created based on the type of change that happens and NOT solely based on the ETP message used. For example, ***ReplaceRange*** in ChannelDataLoad (Protocol 22) may replace data at the start, in the middle, or at the end of a channel's data range.

IMPORTANT: The table explains how to create ***ChangeAnnotation*** records in response to customer requests. Whenever new records overlap each other or existing records, the store MUST merge the overlapping records together. In addition, the store MAY merge non-overlapping records. For rules governing merging ***ChangeAnnotation*** records, see Section **11.2.2.4**.

CHANNEL Data Objects: Scenarios for how Stores Create ChangeAnnotations				
Change Type	**Primary Indexes Affected?**	**Annotation Created?**	**Annotation Range**	**Annotation Timestamp**
Data Appended: new data appended; existing data unaffected	For increasing data, end index increases. For decreasing data, end index decreases.	No		
Data Prepended: new data prepended; existing data unaffected	For increasing data, start index decreases. For decreasing data, start index increases.	Yes (required)	Range between new and old start index.	Time when store prepended data.
Range Deleted Covering End Index: existing data removed; no data added or changed	For increasing data, end index decreases. For "decreasing" data, end index increases.	Yes (required)	Range between new and old end index.	Time when store deleted data.
Range Deleted Covering Start Index: existing data removed; no data added or changed	For increasing data, start index increases. For decreasing data, start index decreases.	Yes (required)	Range between new and old start index.	Time when store deleted data.
Range Deleted Inside Existing Data Range: existing data removed; no data added or changed	No.	Yes (required)	*changedInterval* from request.	Time when store deleted data.
All Data Deleted: existing data removed; no data added or changed	Start and end indexes become null.	Yes (required)	Range between old start and end index.	Time when store deleted data.
Range Replaced Covering End Index: existing data removed; replacement data added	End index may increase or decrease.	Yes (required)	Smallest range covering: a) range between new and old end index, and b) added data range.	Time when store replaced data.
Range Replaced Covering Start Index: existing data removed;	Start index may increase or decrease.	Yes (required)	Smallest range covering:	Time when store replaced data.

CHANNEL Data Objects: Scenarios for how Stores Create ChangeAnnotations				
Change Type	**Primary Indexes Affected?**	**Annotation Created?**	**Annotation Range**	**Annotation Timestamp**
replacement data added			a) range between new and old start index, and b) added data range.	
Range Replaced Inside Existing Data Range: existing data removed; replacement data added	No.	Yes (required)	*changedInterval* from request.	Time when store replaced range.
All Data Replaced: existing data removed; replacement data added	Both start and end indexes may increase or decrease.	Yes (required)	Range between old start and end index.	

11.2.2.3 *Rules for Creating Change Annotations for Growing Data Objects*

Figure 21 illustrates some common types of changes to growing data object parts and how a store may or must create or update ChangeAnnotation records in response to them.

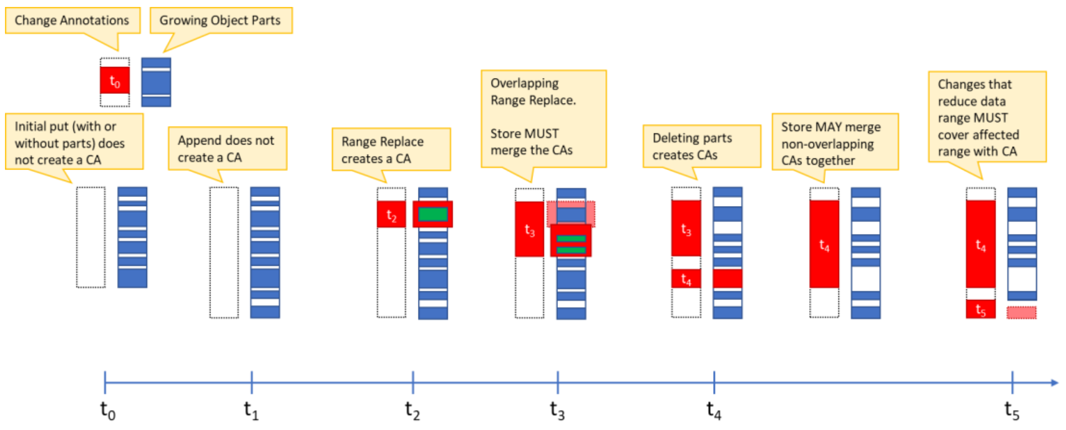

Change Annotations
Growing Object Parts
t_0
Initial put (with or without parts) does not create a CA
Append does not create a CA
Range Replace creates a CA
Overlapping Range Replace. Store MUST merge the CAs
Deleting parts creates CAs
Store MAY merge non-overlapping CAs together
Changes that reduce data range MUST cover affected range with CA
t_2
t_3
t_3
t_4
t_4
t_4
t_5
t_0
t_1
t_2
t_3
t_4
t_5

Figure 21: Example showing how change annotations (CA) work over time for growing data objects. Blue boxes = growing data object parts, white box with red label = change annotations. The size of the white box is intended to show that change annotations could potentially exist anywhere in the full range of data covered by the parts, and the red boxes show the actual ranges of data covered by change annotations.

The table below describes how stores create ***ChangeAnnotation*** records in different scenarios for growing data objects.

IMPORTANT: ***ChangeAnnotation*** records MUST be created based on the type of change that happens and NOT solely based on the ETP message used. For example, ***ReplacePartsByRange*** in GrowingObject (Protocol 6) may replace parts at the start, in the middle or at the end of a growing data object's data range. The ***PutParts*** and ***DeleteParts*** messages may cause multiple types of changes that result in multiple ***ChangeAnnotation*** records being created.

IMPORTANT: The table explains how to create ***ChangeAnnotation*** records in response to customer requests. Whenever new records overlap each other or existing records, the store MUST merge the overlapping records together. In addition, the store MAY merge non-overlapping records. For rules governing merging ***ChangeAnnotation*** records, see Section **11.2.2.4**.

GROWING Data Objects: Scenarios for how Stores Create ChangeAnnotations				
Change Type	**Indexes Affected?**	**Annotation Created?**	**Annotation Range**	**Annotation Timestamp**
Part(s) Appended: new part(s) appended; existing parts unaffected.	End index increases.	No		
Part(s) Prepended: new part(s) prepended; existing parts unaffected.	Start index decreases.	Yes (required)	Range between new and old start index.	Time when store prepended parts.
Part(s) Added Covering End Index: new part(s) added; existing parts unaffected.	End index may increase.	Yes (required)	Smallest range covering start and end index of each added part.	Time when store added parts.
Part(s) Added Covering Start Index: new part(s) added; existing parts unaffected.	Start index may decrease.	Yes (required)	Smallest range covering start and end index of each added part.	Time when store added parts.
Part(s) Added Inside Existing Data Range: new part(s) added; existing parts unaffected.	No.	Yes (required, one per part)	Range of each added part.	Time when store added each part.
Part(s) Updated: existing part(s) updated; no parts added or deleted.	Both start and end indexes may increase or decrease	Yes (required, one per part)	Range of each updated part.	Time when store updated each part.
Part(s) Deleted Covering End Index: existing part(s) deleted; no parts added or updated.	End index may decrease.	Yes (required)	Smallest range covering: a) range between new and old end index, and b) start and end index of each deleted part.	Time when store deleted parts.
Part(s) Deleted Covering Start Index: existing part(s) deleted; no parts added or updated.	Start index may increase.	Yes (required)	Smallest range covering: a) range between new and old start index, and b) start and end index of each deleted part.	Time when store deleted parts.
Part(s) Deleted Inside Existing Data Range: existing part(s) deleted;	No.	Yes (required, one per part)	Range of each deleted part.	Time when store deleted each part.

GROWING Data Objects: Scenarios for how Stores Create ChangeAnnotations				
Change Type	**Indexes Affected?**	**Annotation Created?**	**Annotation Range**	**Annotation Timestamp**
no parts added or updated.				
Range Deleted Covering End Index: existing parts removed; no parts added or changed.	End index decreases.	Yes (required)	Smallest range covering: a) start index of *deleteInterval* from request, b) range between new and old end index, and c) start and end index of each deleted part.	Time when store deleted data.
Range Deleted Covering Start Index: existing parts removed; no parts added or changed.	Start index increases.	Yes (required)	Smallest range covering: a) end index of *deleteInterval* from request, b) range between new and old start index, and c) start and end index of all deleted parts.	Time when store deleted data.
Range Deleted Inside Existing Data Range: existing parts removed; no parts added or changed.	No.	Yes (required)	Smallest range covering: a) *deleteInterval* from request, and b) start and end index of each deleted part.	Time when store deleted data.
All Parts Deleted: existing parts removed; no parts added or changed.	Start and end indexes become null.	Yes (required)	Range between old start and end index.	Time when store deleted data.
Range Replaced Covering End Index: existing parts removed; replacement parts added.	End index may increase or decrease.	Yes (required)	Smallest range covering: a) start index of *deleteInterval* from request, b) range between new and old end index, c) start and end index of each added part, and d) start and end index of each deleted part.	Time when store replaced data.
Range Replaced Covering Start Index: existing parts removed;	Start index may increase or decrease.	Yes (required)	Smallest range covering:	Time when store replaced data.

GROWING Data Objects: Scenarios for how Stores Create ChangeAnnotations				
Change Type	**Indexes Affected?**	**Annotation Created?**	**Annotation Range**	**Annotation Timestamp**
replacement parts added.			a) end index of *deleteInterval* from request, b) range between new and old start index, c) start and end index of each added part, and d) start and end index of each deleted part.	
Range Replaced Inside Existing Data Range: existing parts removed; replacement parts added.	No.	Yes (required)	Range that was deleted.	Time when store replaced range.
All Parts Replaced: existing parts removed; replacement parts added.	Both start and end indexes may increase or decrease	Yes (required)	Smallest range covering: a) Range between new and old start index, and b) start and end index of each added part.	

11.2.2.4 Rules for Merging Change Annotations

The table below explains how a store may or must merge ***ChangeAnnotation*** records in different scenarios.

- When creating new records, stores MUST apply the required rules in this table for overlapping ***ChangeAnnotation*** records.
- At any time (including when creating new records), stores MAY apply the optional rules in this table for merging non-overlapping intervals.

Rules for Merging *ChangeAnnotation* Records			
Scenario	**Annotations Merged?**	**Merged Range**	**Merged Timestamp**
Overlapping Annotations	Yes (required)	Smallest range covering the range of each merged annotation.	Most recent timestamp of merged annotations.
Adjacent Annotations	Merging is optional	Smallest range covering the range of each merged annotation.	Most recent timestamp of merged annotations.
Other Annotations	Merging is optional	Smallest range covering the range of each merged annotation.	Most recent timestamp of merged annotations.

11.2.2.5 Additional Rules for Change Annotations

The following additional rules also apply to ***ChangeAnnotation*** records:

1. A store MUST create and merge ***ChangeAnnotation*** records even if the data was not changed through an ETP store operation.
2. A store MUST retain any ***ChangeAnnotation*** records that are created for its value for the ChangeRetentionPeriod capability, which MUST be greater than or equal to the minimum value stated in this specification (see Section **9.2.3**).
    a. ***ChangeAnnotation*** records MUST be retained by a store endpoint for at least the ChangeRetentionPeriod as long as there is at least one session connected to it. It is STRONGLY recommended to always retain ***ChangeAnnotation*** records for the ChangeRetentionPeriod.
    b. If a store is unable to retain ***ChangeAnnotation*** records for the full ChangeRetentionPeriod (e.g., because the store application restarted and it has no persistent storage for tombstones), the store MUST advise customers of the earliest timestamp ***ChangeAnnotation*** records are available in the *earliestRetainedChangeTime* field in either ***OpenSession*** or ***RequestSession***.
3. Not every store will be able to accurately track *changeTime* for ***ChangeAnnotation*** records over long periods of time. For example, they may lose track of this information if the store application is restarted. **The minimum requirements to enable eventual consistency workflows are that:**
    a. *changeTime* on a ***ChangeAnnotation*** MUST ALWAYS be equal to or more recent than the change to that data interval.
    b. *changeTime* on any ***ChangeAnnotation*** record for a data object MUST ALWAYS be equal to or less recent than the data object's *storeLastWrite* and equal to or more recent than the data object's *storeCreated*. This includes when the change was not done through an ETP store operation.
    c. When the change happens **through an ETP store operation**, the store MUST set *changeTime* to the actual change time.
    d. If a store loses track of which ***ChangeAnnotation*** records are associated with a data object, the store MUST do all of the following:
        i. Delete all ***ChangeAnnotation*** records associated with the data object.
        ii. Set the data object's *storeCreated* and, if required, *storeLastWrite* to a time after the most recent data change.
        iii. Send any appropriate notifications in response to these changes.

11.2.3 GrowingObject: Capabilities

The table below lists key capabilities for this protocol. The protocol capabilities are all defined here. For this protocol, two particularly crucial endpoint capabilities are defined here.

- For protocol-specific behavior related to using these capabilities in this protocol, see Sections **11.2.1** and **11.2.2**.
- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section **3.3**.

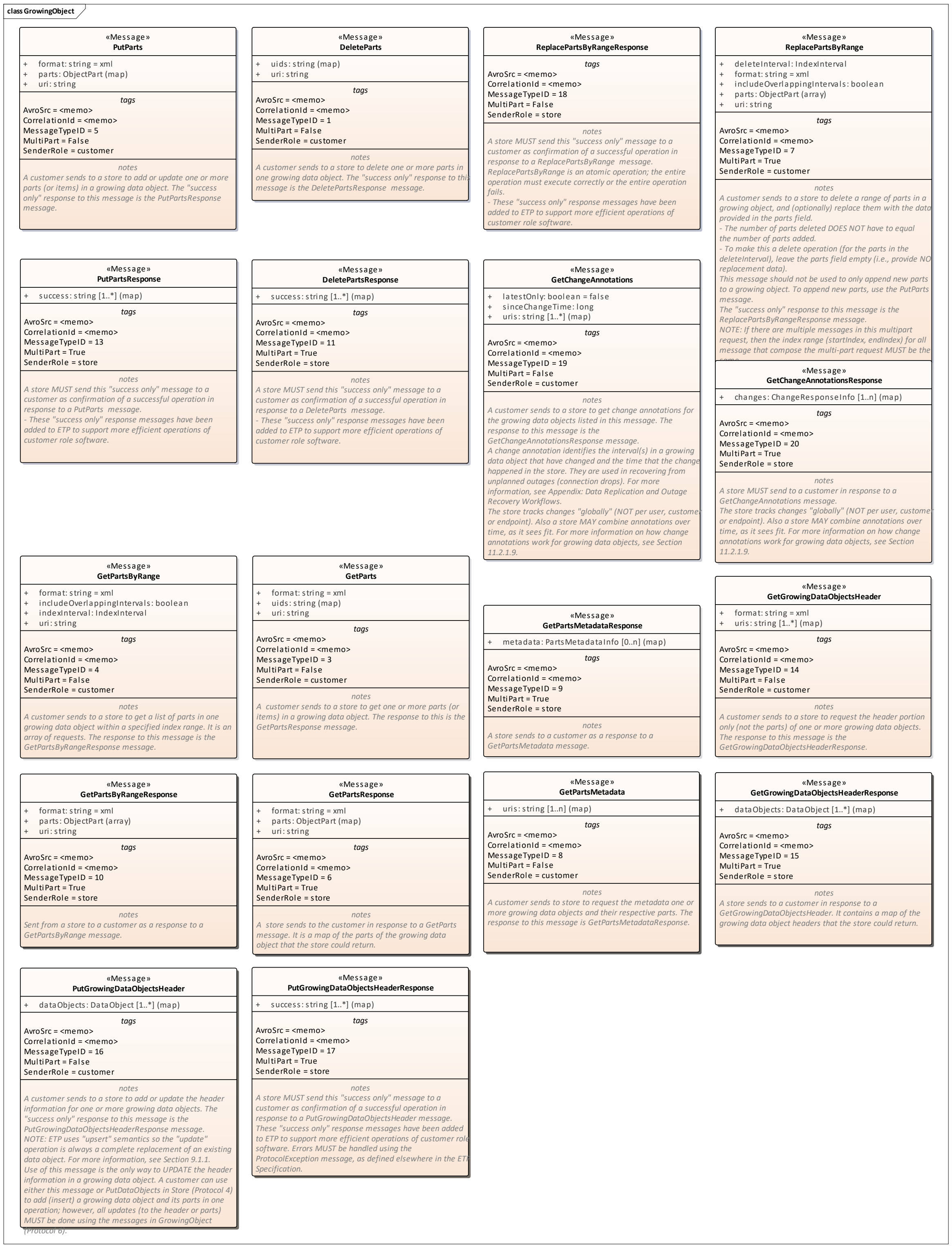
GrowingObject (Protocol 6): Capabilities			
Name: Description	**Type**	**Units** **Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of each endpoint capability, see Section **3.3**.)			

GrowingObject (Protocol 6): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section 3.3.2. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section 4.3, How a Client Establishes a WebSocket Connection to an ETP Server.			
ChangeRetentionPeriod: The minimum time period in seconds that a store retains the canonical URI of a deleted data object and any change annotations for channels and growing data objects. **RECOMMENDATION:** This period should be as long as is feasible in an implementation. When the period is shorter, the risk is that additional data will need to be transmitted to recover from outages, leading to higher initial load on sessions.	long	Seconds **Value units:** <number of seconds>	**Default:** 86,400 **MIN:** 86,400
MaxPartSize: The maximum size in bytes of each data object part allowed in a standalone message or a complete multipart message. Size in bytes is the total size in bytes of the uncompressed string representation of the data object part in the format in which it is sent or received.	long	byte <number of bytes>	Min: 10,000 bytes
Data Object Capabilities (For definitions of each data object capability, see Section 3.3.4.)			
ActiveTimeoutPeriod: (This is also an endpoint capability.) The minimum time period in seconds that a store keeps the GrowingStatus for a growing data object or channel "active" after the last new part or data point resulting in a change to the data object's end index was added to the data object. This capability can be set for an endpoint and/or for a data object. A data object-specific value overrides an endpoint-specific value.	long	second <number of seconds>	**Default:** 3,600 **MIN:** 60 seconds
MaxDataObjectSize: (This is also a protocol and endpoint capability.) The maximum size in bytes of a data object allowed in a complete multipart message. Size in bytes is the size in bytes of the uncompressed string representation of the data object in the format in which it is sent or received. This capability can be set for an endpoint, a protocol, and/or a data object. If set for all three, here is how they generally work: • An object-specific value overrides an endpoint-specific value. • A protocol-specific value can further lower (but NOT raise) the limit for the protocol. **EXAMPLE:** A store may wish to generally support sending and receiving any data object that is one megabyte or less with the exceptions of Wells that are 100 kilobytes or less and Attachments that are 5 megabytes or less. A store may further wish to limit the size of any data object sent as part of a notification in StoreNotification (Protocol 5) to 256 kilobytes.	long	bytes <number of bytes>	**MIN:** 100,000 bytes
SupportsGet, SupportsPut			

GrowingObject (Protocol 6): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
For definitions and usage rules for each of these data object capabilities, see Section **3.3.4**.			
Protocol Capabilities			
MaxResponseCount: The maximum total count of responses allowed in a complete multipart message response to a single request.	long	count <count of responses>	**MIN:** 10,000

11.3 GrowingObject: Message Schemas

This section provides a figure that displays all messages defined in GrowingObject (Protocol 6). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.



class GrowingObject

«Message»
PutParts
+ format: string = xml
+ parts: ObjectPart (map)
+ uri: string

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 5
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to add or update one or more parts (or items) in a growing data object. The "success only" response to this message is the PutPartsResponse message.

«Message»
DeleteParts
+ uids: string (map)
+ uri: string

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to delete one or more parts in one growing data object. The "success only" response to this message is the DeletePartsResponse message.

«Message»
ReplacePartsByRangeResponse

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 18
MultiPart = False
SenderRole = store

notes
A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a ReplacePartsByRange message.
ReplacePartsByRange is an atomic operation; the entire operation must execute correctly or the entire operation fails.
- These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

«Message»
ReplacePartsByRange
+ deleteInterval: IndexInterval
+ format: string = xml
+ includeOverlappingIntervals: boolean
+ parts: ObjectPart (array)
+ uri: string

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 7
MultiPart = True
SenderRole = customer

notes
A customer sends to a store to delete a range of parts in a growing object, and (optionally) replace them with the data provided in the parts field.
- The number of parts deleted DOES NOT have to equal the number of parts added.
- To make this a delete operation (for the parts in the deleteInterval), leave the parts field empty (i.e., provide NO replacement data).
This message should not be used to only append new parts to a growing object. To append new parts, use the PutParts message.
The "success only" response to this message is the ReplacePartsByRangeResponse message.
NOTE: If there are multiple messages in this multipart request, then the index range (startIndex, endIndex) for all message that compose the multi-part request MUST be the

«Message»
PutPartsResponse
+ success: string [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 13
MultiPart = True
SenderRole = store

notes
A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutParts message.
- These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

«Message»
DeletePartsResponse
+ success: string [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 11
MultiPart = True
SenderRole = store

notes
A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a DeleteParts message.
- These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

«Message»
GetChangeAnnotations
+ latestOnly: boolean = false
+ sinceChangeTime: long
+ uris: string [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 19
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to get change annotations for the growing data objects listed in this message. The response to this message is the GetChangeAnnotationsResponse message.
A change annotation identifies the interval(s) in a growing data object that have changed and the time that the change happened in the store. They are used in recovering from unplanned outages (connection drops). For more information, see Appendix: Data Replication and Outage Recovery Workflows.
The store tracks changes "globally" (NOT per user, customer or endpoint). Also a store MAY combine annotations over time, as it sees fit. For more information on how change annotations work for growing data objects, see Section 11.2.1.9.

«Message»
GetChangeAnnotationsResponse
+ changes: ChangeResponseInfo [1..n] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 20
MultiPart = True
SenderRole = store

notes
A store MUST send to a customer in response to a GetChangeAnnotations message.
The store tracks changes "globally" (NOT per user, customer or endpoint). Also a store MAY combine annotations over time, as it sees fit. For more information on how change annotations work for growing data objects, see Section 11.2.1.9.

«Message»
GetPartsByRange
+ format: string = xml
+ includeOverlappingIntervals: boolean
+ indexInterval: IndexInterval
+ uri: string

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 4
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to get a list of parts in one growing data object within a specified index range. It is an array of requests. The response to this message is the GetPartsByRangeResponse message.

«Message»
GetParts
+ format: string = xml
+ uids: string (map)
+ uri: string

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 3
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to get one or more parts (or items) in a growing data object. The response to this is the GetPartsResponse message.

«Message»
GetPartsMetadataResponse
+ metadata: PartsMetadataInfo [0..n] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 9
MultiPart = True
SenderRole = store

notes
A store sends to a customer as a response to a GetPartsMetadata message.

«Message»
GetGrowingDataObjectsHeader
+ format: string = xml
+ uris: string [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 14
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to request the header portion only (not the parts) of one or more growing data objects. The response to this message is the GetGrowingDataObjectsHeaderResponse.

«Message»
GetPartsByRangeResponse
+ format: string = xml
+ parts: ObjectPart (array)
+ uri: string

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 10
MultiPart = True
SenderRole = store

notes
Sent from a store to a customer as a response to a GetPartsByRange message.

«Message»
GetPartsResponse
+ format: string = xml
+ parts: ObjectPart (map)
+ uri: string

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 6
MultiPart = True
SenderRole = store

notes
A store sends to the customer in response to a GetParts message. It is a map of the parts of the growing data object that the store could return.

«Message»
GetPartsMetadata
+ uris: string [1..n] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 8
MultiPart = False
SenderRole = customer

notes
A customer sends to store to request the metadata one or more growing data objects and their respective parts. The response to this message is GetPartsMetadataResponse.

«Message»
GetGrowingDataObjectsHeaderResponse
+ dataObjects: DataObject [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 15
MultiPart = True
SenderRole = store

notes
A store sends to a customer in response to a GetGrowingDataObjectsHeader. It contains a map of the growing data object headers that the store could return.

«Message»
PutGrowingDataObjectsHeader
+ dataObjects: DataObject [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 16
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to add or update the header information for one or more growing data objects. The "success only" response to this message is the PutGrowingDataObjectsHeaderResponse message.
NOTE: ETP uses "upsert" semantics so the "update" operation is always a complete replacement of an existing data object. For more information, see Section 9.1.1.
Use of this message is the only way to UPDATE the header information in a growing data object. A customer can use either this message or PutDataObjects in Store (Protocol 4) to add (insert) a growing data object and its parts in one operation; however, all updates (to the header or parts) MUST be done using the messages in GrowingObject (Protocol 6).

«Message»
PutGrowingDataObjectsHeaderResponse
+ success: string [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 17
MultiPart = True
SenderRole = store

notes
A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutGrowingDataObjectsHeader message. These "success only" response messages have been added to ETP to support more efficient operations of customer role software. Errors MUST be handled using the ProtocolException message, as defined elsewhere in the ETP Specification.

Figure 22: GrowingObject: message schemas

11.3.1 Message: GetParts

A customer sends to a store to get one or more parts (or items) in a growing data object. The response to this is the GetPartsResponse message.

Message Type ID: 3

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uri	MUST be the URI of the "parent" growing data object to which the parts belong. For example: in WITSML, a Trajectory is a growing data object and each TrajectoryStation is a part. You MUST specify the URI of the Trajectory object and the UID for each TrajectoryStation (part) you want. If both endpoints support alternate URIs for the session, this MAY be an alternate data object URI. Otherwise, this MUST be a canonical Energistics data object URI. For more information, see **Appendix: Energistics Identifiers**.	string	1	1
format	Specifies the format (e.g., XML or JSON) in which you want to receive data for the requested parts. This MUST be a format that was negotiated when establishing the session. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.	string	1	1
uids	General ETP map where each value MUST be the UID of a part in the parent growing data object.	string	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "GetParts",
     "protocol": "6",
     "messageType": "3",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "format", "type": "string", "default": "xml" },
         {
             "name": "uids",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

11.3.2 Message: GetPartsResponse

A store sends to the customer in response to a GetParts message. It is a map of the parts of the growing data object that the store could return.

Message Type ID: 6

Correlation Id Usage: MUST be set to the *messageId* of the ***GetParts*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
uri	The URI of the "parent" growing data object. For example: in WITSML, a Trajectory is a growing data object and each TrajectoryStation is a part. This MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
format	Specifies the format (e.g., XML or JSON) of the data for the parts being sent in this message. This MUST match the format in the GetParts request. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.	string	1	1
parts	General ETP map of ObjectPart records, one for each part the store could return, which each contains the UID for a part and its associated data.	ObjectPart	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "GetPartsResponse",
     "protocol": "6",
     "messageType": "6",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "format", "type": "string", "default": "xml" },
         {
             "name": "parts",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.Object.ObjectPart" }
         }
     ]
}
```

11.3.3 Message: GetGrowingDataObjectsHeader

A customer sends to a store to request the header portion only (not the parts) of one or more growing data objects. The response to this message is the GetGrowingDataObjectsHeaderResponse.

Message Type ID: 14

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uris	General ETP map where each value MUST be the URI for each growing data object "header" to be retrieved.	string	1	*

Field Name	Description	Data Type	Min	Max
	If both endpoints support alternate URIs for the session, this MAY be an alternate data object URI. Otherwise, this MUST be a canonical Energistics data object URI. For more information, see **Appendix: Energistics Identifiers**.			
format	Specifies the format (e.g., XML or JSON) in which you want to receive data for the requested growing data object headers. This MUST be a format that was negotiated when establishing the session. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.	string	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "GetGrowingDataObjectsHeader",
     "protocol": "6",
     "messageType": "14",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "uris",
             "type": { "type": "map", "values": "string" }
         },
         { "name": "format", "type": "string", "default": "xml" }
     ]
}
```

11.3.4 Message: GetGrowingDataObjectsHeaderResponse

A store sends to a customer in response to a GetGrowingDataObjectsHeader. It contains a map of the growing data object headers that the store could return.

Message Type ID: 15

Correlation Id Usage: MUST be set to the *messageId* of the ***GetGrowingDataObjectsHeader*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
dataObjects	General ETP map of DataObject records, one each for each growing data object "header" the store could return.	DataObject	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "GetGrowingDataObjectsHeaderResponse",
     "protocol": "6",
     "messageType": "15",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "dataObjects",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.Object.DataObject" }
         }
     ]
}
```

11.3.5 Message: PutParts

A customer sends to a store to add or update one or more parts (or items) in a growing data object. The "success only" response to this message is the PutPartsResponse message.

Message Type ID: 5

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uri	MUST be the URI of the "parent" growing data object where you want to add or update parts. For example: in WITSML, a Trajectory is a growing data object and each TrajectoryStation is a part. You MUST specify the URI of the Trajectory object and the UID for each TrajectoryStation (part) that you want to put. If both endpoints support alternate URIs for the session, this MAY be an alternate data object URI. Otherwise, this MUST be a canonical Energistics data object URI. For more information, see **Appendix: Energistics Identifiers**.	string	1	1
parts	General ETP map of ObjectPart records, one for each part being put. It contains the UID for each part and the data being put. Each part MUST be identified by a UID, which MUST be unique within the growing data object.	ObjectPart	1	1
format	Specifies the format (e.g., XML or JSON) of the data for the parts being sent in this message. This MUST be a format that was negotiated when establishing the session. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.	string	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "PutParts",
     "protocol": "6",
     "messageType": "5",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "format", "type": "string", "default": "xml" },
         {
             "name": "parts",
             "type": { "type": "map", "values": 
"Energistics.Etp.v12.Datatypes.Object.ObjectPart" }
         }
     ]
}
```

11.3.6 Message: PutPartsResponse

A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutParts message.

These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

Message Type ID: 13

Correlation Id Usage: MUST be set to the *messageId* of the ***PutParts*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
success	• The presence of the map key represents success. • The associated map string value SHOULD be empty because its content is ignored.	string	1	*

Avro Source

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
    "name": "PutPartsResponse",
    "protocol": "6",
    "messageType": "13",
    "senderRole": "store",
    "protocolRoles": "store,customer",
    "multipartFlag": true,
    "fields":
    [
        {
            "name": "success",
            "type": { "type": "map", "values": "string" }
        }
    ]
}
```

11.3.7 Message: PutGrowingDataObjectsHeader

A customer sends to a store to add or update the header information for one or more growing data objects. The "success only" response to this message is the PutGrowingDataObjectsHeaderResponse message.

NOTE: ETP uses "upsert" semantics so the "update" operation is always a complete replacement of an existing data object. For more information, see Section **9.1.1**.

Use of this message is the only way to UPDATE the header information in a growing data object. A customer can use either this message or ***PutDataObjects*** in Store (Protocol 4) to add (insert) a growing data object and its parts in one operation; however, all updates (to the header or parts) MUST be done using the messages in GrowingObject (Protocol 6).

Message Type ID: 16

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
dataObjects	General ETP map of the growing data object "headers" to be added/updated; it is a map of DataObject records, one for each growing data object. NOTE: DataObject encapsulates the Resource, which is where the URI for each growing data object MUST be entered. • If the growing data object is to be created, the ***DataObject*** record MAY include both the header and parts. • If the growing data object exists, the ***DataObject*** record MUST NOT include parts. The URIs in the ***Resource*** records MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.	DataObject	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "PutGrowingDataObjectsHeader",
     "protocol": "6",
     "messageType": "16",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "dataObjects",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.Object.DataObject" }
         }
     ]
}
```

11.3.8 Message: PutGrowingDataObjectsHeaderResponse

A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutGrowingDataObjectsHeader message.

These "success only" response messages have been added to ETP to support more efficient operations of customer role software. Errors MUST be handled using the ProtocolException message, as defined elsewhere in the ETP Specification.

Message Type ID: 17

Correlation Id Usage: MUST be set to the *messageId* of the ***PutGrowingDataObjectsHeader*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
success	• The presence of the map key represents success. • The associated map string value SHOULD be empty because its content is ignored.	string	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "PutGrowingDataObjectsHeaderResponse",
     "protocol": "6",
     "messageType": "17",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "success",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

11.3.9 Message: DeleteParts

A customer sends to a store to delete one or more parts in one growing data object. The "success only" response to this message is the DeletePartsResponse message.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uri	MUST be the URI of the "parent" growing data object from which you want to delete parts. For example: in WITSML, a Trajectory is a growing data object and each TrajectoryStation is a part. You MUST specify the URI of the Trajectory object and the UID for each TrajectoryStation (part) that you want to delete. If both endpoints support alternate URIs for the session, these MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	string	1	1
uids	General ETP map whose values must be the UID of the parts that are being deleted in the parent growing data object.	string	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "DeleteParts",
     "protocol": "6",
     "messageType": "1",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "uri", "type": "string" },
         {
             "name": "uids",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

11.3.10 Message: DeletePartsResponse

A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a DeleteParts message.

These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

Message Type ID: 11

Correlation Id Usage: MUST be set to the *messageId* of the ***DeleteParts*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
success	• The presence of the map key represents success. • The associated map string value SHOULD be empty because its content is ignored.	string	1	*

Avro Source

```
{
      "type": "record",
      "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
      "name": "DeletePartsResponse",
      "protocol": "6",
      "messageType": "11",
      "senderRole": "store",
      "protocolRoles": "store,customer",
      "multipartFlag": true,
      "fields":
      [
          {
              "name": "success",
              "type": { "type": "map", "values": "string" }
          }
      ]
}
```

11.3.11 Message: GetPartsByRange

A customer sends to a store to get a list of parts in one growing data object within a specified index range. It is an array of requests. The response to this message is the GetPartsByRangeResponse message.

Message Type ID: 4

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uri	MUST be the URI for the parent growing data object for the parts being requested. For example: in WITSML, a Trajectory is a growing data object and each TrajectoryStation is a part. If both endpoints support alternate URIs for the session, these MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	string	1	1
indexInterval	The index interval as defined in IndexInterval for the list of parts you want to get: ● If the StartIndex is specified as NULL, then the server MUST assume a value of negative infinity. ● If the endIndex is specified as NULL, then the server MUST assume a value of positive Infinity. ● The ending index for the get range MUST be NULL or >= startIndex or you MUST send error EINVALID_ARGUMENT (5).	IndexInterval	1	1
format	Specifies the format (e.g., XML or JSON) in which you want to receive data for the requested parts. This MUST be a format that was negotiated when establishing the session. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.	string	1	1
includeOverlappingIntervals	Some growing data objects (e.g., wellbore geology (previously known as mud log)) have "parts" that represent a range, as opposed to a single point. The issue with having a range part (vs. a single point) is how to handle "ByRange" operations when only a portion of the range part overlaps the specified interval of interest (request interval) (in this message the indexInterval). Use this flag to specify if the range operation is inclusive or exclusive for range parts that overlap the request interval. For more information on how overlapping ranges works, see Section **11.2.2.1**. ● If true, then any range part that overlaps the request interval is affected (included in the requested operation). ● If false, then only range parts that fall completely within the request interval are affected (included in the requested operation).	boolean	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "GetPartsByRange",
     "protocol": "6",
     "messageType": "4",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "format", "type": "string", "default": "xml" },
         { "name": "indexInterval", "type":
"Energistics.Etp.v12.Datatypes.Object.IndexInterval" },
         { "name": "includeOverlappingIntervals", "type": "boolean" }
     ]
}
```

11.3.12 Message: GetPartsByRangeResponse

Sent from a store to a customer as a response to a GetPartsByRange message.

Message Type ID: 10

Correlation Id Usage: MUST be set to the *messageId* of the ***GetPartsByRange*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
uri	MUST be the URI of the "parent" growing data object to which the parts belong. For example: in WITSML, a Trajectory is a growing data object and each TrajectoryStation is a part. You MUST specify the URI of the Trajectory object and the UID for each TrajectoryStation (part) you want. This MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
format	Specifies the format (e.g., XML or JSON) of the data for the parts being sent in this message. This MUST match the format in the GetPartsByRange request. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.	string	1	1
parts	An array of ObjectPart records, one for each part being returned in this response message. It contains the UID and data for each part.	ObjectPart	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "GetPartsByRangeResponse",
     "protocol": "6",
     "messageType": "10",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "format", "type": "string", "default": "xml" },
         {
             "name": "parts",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.Object.ObjectPart" }
         }
     ]
}
```

11.3.13 Message: GetPartsMetadata

A customer sends to store to request the metadata one or more growing data objects and their respective parts. The response to this message is GetPartsMetadataResponse.

Message Type ID: 8

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uris	General ETP map where each value MUST be the URI of each "parent" growing data object for which the customer wants to get parts metadata. If both endpoints support alternate URIs for the session, these MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	string	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "GetPartsMetadata",
     "protocol": "6",
     "messageType": "8",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "uris",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

11.3.14 Message: GetPartsMetadataResponse

A store sends to a customer as a response to a GetPartsMetadata message.

Message Type ID: 9

Correlation Id Usage: MUST be set to the *messageId* of the ***GetPartsMetadata*** that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
metadata	General ETP map of PartsMetadataInfo records, one for each growing data object that the store could successfully return information for.	PartsMetadataInfo	0	n

Field Name	Description	Data Type	Min	Max
	The URIs MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.			

Avro Source

```
{
      "type": "record",
      "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
      "name": "GetPartsMetadataResponse",
      "protocol": "6",
      "messageType": "9",
      "senderRole": "store",
      "protocolRoles": "store,customer",
      "multipartFlag": true,
      "fields":
      [
          {
              "name": "metadata",
              "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.Object.PartsMetadataInfo" }
          }
      ]
}
```

11.3.15 Message: ReplacePartsByRange

A customer sends to a store to delete a range of parts in a growing object, and (optionally) replace them with the data provided in the *parts* field.

- The number of parts deleted DOES NOT have to equal the number of parts added.
- To make this a delete operation (for the parts in the *deleteInterval*), leave the *parts* field empty (i.e., provide NO replacement data).

This message should not be used to only append new parts to a growing object. To append new parts, use the ***PutParts*** message.

The "success only" response to this message is the ReplacePartsByRangeResponse message.

NOTE: If this message is a multipart request, then the index range (startIndex, endIndex) for all message that compose the multipart request MUST be the same.

Message Type ID: 7

Correlation Id Usage: For the first message, the *correlationId* MUST be set to 0. If there are multiple messages in this multipart request, the *correlationId* of all successive messages that comprise the request MUST be set to the *messageId* of the first message in this request.

Multi-part: True

Sent by: customer

Field Name	Description	Data Type	Min	Max
uri	MUST be the URI of the "parent" growing data object to which the parts that are to be deleted and replaced belong. For example: in WITSML, a Trajectory is a growing data object and each TrajectoryStation is a part. You MUST specify the URI of the Trajectory object and the UID for each TrajectoryStation (part) you want. If both endpoints support alternate URIs for the session, these MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more	string	1	1

Field Name	Description	Data Type	Min	Max
	information, see **Appendix: Energistics Identifiers**.			
deleteInterval	The index interval for the range of parts to be deleted as defined in IndexInterval. This is NOT the index range of the parts (listed in the parts field) that will replace the deleted parts. • If the StartIndex is specified as NULL, then the server MUST assume a value of negative infinity. • If the endIndex is specified as NULL, then the server MUST assume a value of positive Infinity. • The ending index for the delete range MUST be NULL or >= startIndex or the store MUST send error EINVALID_ARGUMENT (5).	IndexInterval	1	1
includeOverlappingIntervals	Some growing data objects (e.g., wellbore geology (previously known as mud log)) have "parts" that represent a range, as opposed to a single point. The issue with having a range part (vs. a single point) is how to handle "ByRange" operations when only a portion of the range part overlaps the specified interval of interest (request interval) (in this message the deleteInterval). Use this flag to specify if the range operation is inclusive or exclusive for range parts that span the request interval. For more information on how overlapping ranges works, see Section **11.2.2.1**. • If true, then any range part that overlaps the delete interval is deleted. • If false, then only range parts that fall completely within the request interval are deleted. DEFAULT = true	boolean	1	1
format	Specifies the format (e.g., XML or JSON) of the data for the replacement parts being sent in this message. This MUST be a format that was negotiated when establishing the session. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.	string	1	1
parts	An array of ObjectPart records, each of which contains the UID and the data for each part that are being added (i.e., the parts that are replacing the parts being deleted as specified in the *deleteInterval* field). **NOTE:** If this list is left empty (i.e., the field is an empty array), then this message simply deletes the parts in the interval specified in the *deleteInterval* field.	ObjectPart	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "ReplacePartsByRange",
     "protocol": "6",
     "messageType": "7",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
```

```
     [
         { "name": "uri", "type": "string" },
         { "name": "deleteInterval", "type":
"Energistics.Etp.v12.Datatypes.Object.IndexInterval" },
         { "name": "includeOverlappingIntervals", "type": "boolean" },
         { "name": "format", "type": "string", "default": "xml" },
         {
             "name": "parts",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.Object.ObjectPart" }
         }
     ]
}
```

11.3.16 Message: ReplacePartsByRangeResponse

A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a ReplacePartsByRange message.

ReplacePartsByRange is an atomic operation; the entire operation must execute correctly or the entire operation fails.

These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

Message Type ID: 18

Correlation Id Usage: MUST be set to the messageId of the FIRST (or only) ***ReplacePartsByRange*** message in the multi-part request that this message is in response to.

Multi-part: False

Sent by: store

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "ReplacePartsByRangeResponse",
     "protocol": "6",
     "messageType": "18",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [

     ]
}
```

11.3.17 Message: GetChangeAnnotations

A customer sends to a store to get change annotations for the growing data objects listed in this message. The response to this message is the GetChangeAnnotationsResponse message.

A change annotation identifies the interval(s) in a growing data object that have changed and the time that the change happened in the store. They are used in recovering from unplanned outages (connection drops). For more information, see **Appendix: Data Replication and Outage Recovery Workflows**.

The store tracks changes "globally" (NOT per user, customer or endpoint). Also a store MAY combine annotations over time, as it sees fit. For more information on how change annotations work for growing data objects, see Section **11.2.1.9**.

Message Type ID: 19

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
sinceChangeTime	The time that the customer wants changes since, which should be based on the time the customer was last sure it received data from the store. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
uris	General ETP map where each value MUST be the URI of each growing data object for which the customer wants to get change annotations. If both endpoints support alternate URIs for the session, these MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	string	1	*
latestOnly	If true, it means get the latest (last) change annotation only for each of the growing data objects listed.	boolean	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "GetChangeAnnotations",
     "protocol": "6",
     "messageType": "19",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "sinceChangeTime", "type": "long" },
         {
             "name": "uris",
             "type": { "type": "map", "values": "string" }
         },
         { "name": "latestOnly", "type": "boolean", "default": false }
     ]
}
```

11.3.18 Message: GetChangeAnnotationsResponse

A store MUST send to a customer in response to a GetChangeAnnotations message.

The store tracks changes "globally" (NOT per user, customer or endpoint). Also a store MAY combine annotations over time, as it sees fit. For more information on how change annotations work for growing data objects, see Section **11.2.1.9**.

Message Type ID: 20

Correlation Id Usage: MUST be set to the *messageId* of the ***GetChangeAnnotations*** that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
changes	ETP general map where each value must be a ChangeResponseInfo record (which contain the change annotations (ChangeAnnotation records)) for each of the growing data objects it could return them for. The URIs used as map keys MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.	ChangeResponseInfo	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObject",
     "name": "GetChangeAnnotationsResponse",
     "protocol": "6",
     "messageType": "20",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "changes",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.Object.ChangeResponseInfo" }
         }
     ]
}
```

12 GrowingObjectNotification (Protocol 7)

ProtocolID: 7

Defined Roles: store, customer

GrowingObjectNotification (Protocol 7) allows store customers to subscribe to and receive notifications of changes to the parts of growing data objects in the store, in an event-driven manner, for events (or operations) that occur in GrowingObject (Protocol 6). That is a customer subscribes to one or more growing data objects using this protocol, and as parts are added, deleted, or changed (operations that happen using messages in GrowingObject (Protocol 6)), that behavior triggers the store to send notifications with Protocol 7. The store can also create "unsolicited" subscriptions to part notifications on a customer's behalf.

Other ETP sub-protocols that may be used with GrowingObjectNotification (Protocol 7):

- The events that trigger notifications in this protocol happen using GrowingObject (Protocol 6). For details of operations that trigger notifications, see Chapter **11**.
- To receive notifications for changes to data objects, ETP has similar protocols: Store (Protocol 4) where the event/operations occur and StoreNotification (Protocol 5), where customers can subscribe to receive notifications about operations on data objects in a specified context (e.g., all the changes that happen a well). For information on operations and notifications related to data objects, see Chapters **9** and **10**.

IMPORTANT: To subscribe to changes to a growing data object for changes other than to parts, a customer MUST subscribe to the growing data object in StoreNotification (Protocol 5). One operation in Protocol 6 (***PutGrowingDataObjectsHeader***) may cause StoreNotification (Protocol 5) to send an ***ObjectChange*** notification message (with *ObjectChangeKind* = insert) because ***PutGrowingDataObjectsHeader*** is adding a new data object. Details are explained below in this chapter and in Chapter **10**.

This chapter includes main sections for:

- Key ETP concepts that are important to understanding how this protocol is intended to work (see Section **12.1**).
- Required behavior, which includes:
  - Description of the message sequence for main tasks, along with required behavior and possible errors (see Section □).
  - Other functional requirements (not covered in the message sequence) including use of endpoint and protocol capabilities for preventing and protecting against aberrant behavior (see Section**12.2.2**).
  - Definitions of the endpoint and protocol capabilities used in this protocol (see Section **12.2.3**).
- Sample schemas of the messages defined in this protocol (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes documentation for each field in a schema (see Section **12.3**).

12.1 GrowingObjectNotification: Key Concepts

- For key concepts about growing data objects, see Section **11.1**.
- For key concepts about notification subscriptions and how they work in ETP, see Section **10.1.1**.

12.2 GrowingObjectNotification: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.
- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**
- The customer has the details of the growing data objects it's interested in; these details are typically found using Discovery (Protocol 3) (Chapter **8**) but may also come out of band of ETP (e.g., in an email).

12.2.1 GrowingObjectNotification: Message Sequences

This section explains the basic message sequence for main tasks to be done using this protocol and includes related key behaviors, usage of key capabilities, and possible errors. The following General Requirements section provides additional requirements and rules for how this protocol works.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section.

GrowingObjectNotification (Protocol 7): Basic Message-Response flow by ETP Role	
Message from customer	**Response Message from store**
SubscribePartNotifications: A request to create a subscription for notifications to parts in growing data objects.	**SubscribePartNotificationsResponse** (multipart)**:** Reply listing the subscriptions that were successfully created.
	UnsolicitedPartNotifications: Automatically sent to a customer when it connects to a store where a subscription was created for the customer based on out-of-band business knowledge (e.g., a contract).
	Notification messages sent by the store for established subscriptions (see details of each message see Section **12.3**): **PartsChanged** **PartsDeleted** **PartsReplacedByRange** (multipart)
UnsubscribePartNotification: A request to cancel/stop a parts subscription (either a requested or unsolicited one).	**PartSubscriptionEnded:** Response to an unsubscribe request OR notice from the store that it has canceled a subscription.

The main tasks in this protocol are subscribing to the appropriate growing data objects in a store to receive the desired notifications and canceling/stopping those subscriptions. Once a subscription has been created, a store MUST send appropriate notifications based on events in GrowingObject (Protocol 6).

12.2.1.1 To subscribe to notifications about parts in a growing data object (i.e., create a subscription):

1. A customer MUST send a store a ***SubscribePartNotifications*** message (Section **12.3.1**).
    a. This message is a map of subscription requests. The details of each subscription request is specified in the ***SubscriptionInfo*** record (Section **23.34.16**) each of which uses a ***ContextInfo*** record (see Section **23.34.15**).
    b. The ***SubscriptionInfo*** record contains a lot of important information where the customer specifies details of the part notification subscription it wants to create, but some key fields worth noting here are:
        i. *requestUuid*, which assigns a UUID to uniquely address each subscription, which can later be used to cancel a subscription.
        ii. *includeObjectData*, a Boolean flag the customer uses to request that added or updated data object parts be included with notification messages. By setting this field to true, a customer is essentially having growing data object parts streamed to it, as new parts are added to a store.
    a. A customer MUST limit the total count of subscriptions in a session to the store's value for the MaxSubscriptionSessionCount protocol capability.
        i. The Store MUST deny requests that exceed this limit by sending error ELIMIT_EXCEEDED◦(12).
2. For the requests it successfully creates subscriptions for, the store MUST respond with a one or more ***SubscribePartNotificationsResponse*** map response messages (Section **12.3.2**), which list the successful subscriptions that the store has created.
    a. For more information on how map response messages work, see Section **3.7.3**.
    b. The store MUST then send notification messages for the subscriptions identified in this response message (according to criteria specified in the ***SubscribePartNotifications*** message) and according to any rules stated in this specification.
    c. For details about general requirements for when to send specific notifications, see Section **12.2.2**.
3. For the requests it does NOT successfully create subscriptions for, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors, such as ENOT_FOUND (11) if a request URI could not be resolved.
    a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.
4. **NOTE:** A store can also create "unsolicited" part notification subscriptions on behalf of a customer. For more information, see Section **12.2.2** (Row **10**).
5. If a customer sends a ***ProtocolException*** message in response to a ***PartsChanged***, ***PartsDeleted***, or ***PartsReplacedByRange*** message, the store MAY attempt to take corrective action, but the store MUST NOT terminate the associated subscriptions.

12.2.1.2 To unsubscribe (cancel) notifications:

1. A customer MUST send a store an ***UnsubscribePartNotification*** message (Section **12.3.5**).
    a. This message must identify the subscription to be cancelled by its request UUID, which the customer assigned to the subscription when it was requested or may have been assigned by an ***UnsolicitedPartNotifications*** message (Section **12.3.8**).
2. If the store successfully cancels the subscription, the store MUST respond with a ***PartSubscriptionEnded*** message (Section **12.3.7**)**,** which holds the request UUID of the subscription that was successfully stopped

   a. The store MUST stop sending any further notifications that were specified in the subscription that has now been ended. It's possible that the customer COULD receive a few additional notifications that were in process/queued before the subscription was stopped.
   b. A store MUST NOT send any notifications for the subscription after sending ***PartSubscriptionEnded***.
3. If the store could not successfully cancel the subscription, it MUST send a ***ProtocolException*** message with an appropriate error code (e.g., if the request UUID could not be found by the store send ENOT_FOUND (11)).
4. The store MAY also end a subscription without receiving a customer request. If the store does so, it MUST notify the customer by sending a ***PartSubscriptionEnded*** message. **EXAMPLE:**◦This happens if the subscription's context URI refers to a growing data object that is deleted.
5. Once a customer has canceled a subscription, the store MUST NOT restart it, even if the subscription was created by the store on behalf of the customer with ***UnsolicitedPartNotifications*** messages.
   a. If the customer wants to restart the subscription, it MUST instead set up a new subscription by sending a ***SubscribePartNotifications*** message as described in Section 12.2.1.1, using a NEW *requestUuid*.

12.2.2 GrowingObjectNotification: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) some rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter 3. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ProtocolException messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI format, see **Appendix: Energistics Identifiers**. 3. For the complete list of error codes defined by ETP, see Chapter 24. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter 5. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).

Row#	Requirement	Behavior
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3**. a. For the list of global capabilities and related behavior, see Section **3.3.2**. 3. Section **12.2.3** identifies the capabilities most relevant to this ETP sub-protocol. Additional details for using these capabilities are included in relevant rows of this table of requirements and Section **12.2.1**.
3.	Message Sequence See Section **12.2.1**.	1. The Message Sequence section above (Section **12.2.1**) describes requirements for the main tasks listed there and also defines required behavior.
4.	Plural messages (which includes maps)	1. This protocol uses plural messages. For detailed rules on handling plural messages (including ***ProtocolException*** handling), see Section **3.7.3**.
5.	Customers must be able to receive and consume data object parts.	1. All customer role applications MUST implement support for receiving and consuming notifications that include the data object parts (that is, all data for the object parts in a format (e.g., XML or JSON) negotiated when establishing the session).
6.	All behaviors defined in this table assume that a valid customer subscription for the correct context has been created.	1. We are aiming to state these requirements and behaviors as clearly and concisely as possible. All behaviors described below assume: a. A valid subscription has been created as described in Section◦**12.2.1.1**. b. References to "parts" means "parts within the context specified in the subscription" which for this protocol must be one growing data object. 2. A valid parts subscription is one where all of the following conditions are met: a. ***SubscriptionInfo***.*context* is a valid: i. ***ContextInfo***.*uri* that references a growing data object that exists and is available in the store (i.e., the store will return it if requested using Store (Protocol 4)). ii. ***ContextInfo***.*dataObjectTypes* is empty. b. ***SubscriptionInfo***.*requestUuid* is not already in use by another subscription. c. ***SubscriptionInfo***.*format* is a format negotiated when establishing the session. 3. Consistent with the previous paragraph (2), for the store to create an unsolicited parts subscription, BOTH conditions MUST BE met. 4. **REMINDER:** To receive notification of other changes (other than add, update, or deleting of parts) to a growing data object, a customer MUST subscribe to changes to the growing data in StoreNotification (Protocol 5) (Chapter **10**).
7.	No Session Survivability	1. If the ETP session is closed or the connection drops, then the store MUST cancel notification subscriptions for the dropped customer endpoint. 2. On reconnect, the customer MUST re-create subscriptions (see Section **12.2.1.1**). 3. For information on resuming operations after a disconnect, see **Appendix: Data Replication and Outage Recovery Workflows**.
8.	Order of Notifications	1. For a given data object, the store MUST send notifications in the same order that operations are performed in the store. a. The intent of this rule is that objects are always "correct" (schema compliant), and never left in an inconsistent state. The rule applies primarily to contained data objects and growing data objects. b. In general, global ordering of notifications is NOT required. However, there are some situations where the order of notifications affecting multiple objects is important and must be preserved.
9.	Objects covered by more than one subscription	A customer can create multiple subscriptions on a store. It is possible that the same data object is include in more than one subscription. 1. In this case, the store MUST send one notification per relevant subscription.

Row#	Requirement	Behavior
		EXAMPLE: If a customer has subscribed to two different scope/contexts that include the same data object, then the customer will receive at least 2 notifications, one for each subscription. a. Each notification message includes the *requestUuid* that uniquely identifies each subscription (so a customer can determine which subscription resulted in each notification message).
10.	Unsolicited subscriptions	1. The store may automatically configure unsolicited part subscriptions to include the data object parts (i.e., the *includeObjectData* on the unsolicited ***SubscriptionInfo*** record may be true). If the customer application does not want the data, it can do one of the following: a. Unsubscribe and stop receiving the notifications. b. Simply ignore the data payloads and get the data manually. c. Unsubscribe from the unsolicited notification and then explicitly create the subscription (see Section **12.2.1.1**) and set *includeObjectData* to false.
11.	**Sending part notifications:** general requirements	1. **REMINDER**: Row **6** 2. Notification messages are those whose name begins with the word "Parts". Each message's definition/description provides general information for when the store must send each message. **EXAMPLE:** When a store deletes parts of a growing data object, the store MUST send a ***PartsDeleted*** message (to all subscribed customers. a. Other rows in this table state additional requirements for specific operations and requirements for notifications. 3. A store MUST send all appropriate notifications, including ***PartsChanged***, ***PartsDeleted***, and ***PartsReplacedByRange***, even if the change was not through an ETP store operation. 4. A store MUST send notifications within its value for ChangePropagationPeriod endpoint capability, which MUST be less than or equal to the maximum value stated in this specification (see Section **3.3.2.2**).
12.	**Putting (inserting/adding) and updating parts:** Additional notification requirements	1. **REMINDER:** Row **6** and Row **11**. 2. When a store completes a ***PutParts*** operation (in GrowingObject (Protocol 6), it MUST send a ***PartsChanged*** notification message. a. Because ETP uses upsert semantics, this message includes information about the type of change, which is specified by the *ObjectChangeKind* enumeration. i. If the store inserted (added) a new part, then it MUST set *ObjectChangeKind* to "insert". ii. If the store updated (replaced) an existing part, then it MUST set *ObjectChangeKind* to "update". iii. If the change was caused by an ETP store operation, the store MUST differentiate between insert and update. iv. If the change was NOT caused by an ETP store operation and the store cannot determine if the operation was an insert or update, it MUST set *ObjectChangeKind* to "insert". **NOTE:** "insert" was chosen because it is the "pessimistic" choice. That is, customers using the replication workflow will assume the affected parts have been completely replaced. While this may cause customers to query more data than is necessary when the operation is actually an update, using "insert" and the pessimistic assumptions that go with it are necessary in some edge cases achieve eventual consistency between data stores b. If a single ***PutParts*** operation resulted in parts being both inserted and updated, the store MUST send 2 notifications: one for inserted parts and one for updated parts. c. The MaxPartSize endpoint capability MUST be observed. See Row **15**.
13.	**Replacing parts by range:** Additional notification requirements	1. **REMINDER:** Row **6** and Row **11**

Row#	Requirement	Behavior
		2. When a range of parts in a store has been updated or deleted, such as when the store completes a ***ReplacePartsByRange*** operation (in GrowingObject (Protocol 6)), it MUST send a ***PartsReplacedByRange*** notification message. a. The MaxPartSize endpoint capability MUST be observed. See Row **15**.
14.	For notifications that exceed an endpoint's WebSocket message size, send smaller notifications	1. Some notifications in this protocol allow or require data object parts to be sent with the message. a. If including all required parts in the notification message causes it to exceed either endpoint's value for MaxWebSocketMessagePayloadSize endpoint capability, the store MUST first attempt to send the parts without data. That is, the *data* field in each ***ObjectPart*** record must be an empty array. b. If the approach described in Paragraph a. still exceeds the MaxWebSocketMessagePayloadSize, the store MUST break the notification into several, smaller messages (e.g., each with half of the parts) and send those as separate notifications.
15.	MaxPartSize capability	1. If on a subscription request (***SubscriptionInfo*** record) the *includeObjectData* field was true, the store MUST include the parts data in relevant notification messages. a. The store MUST limit the size of data object parts in the notifications to the lesser the store's and the customer's value for MaxPartSize endpoint capability. i. If any part would exceed this limit, the store MUST send the part without its data. That is, the *data* field in the ***ObjectPart*** record must be an empty array. b. If the part size exceeds this limit, the customer MAY notify the store by sending error EMAXSIZE_EXCEEDED (17).
16.	**Deleting parts:** additional requirements	1. **REMINDER:** Row **6** and Row **11**. 2. When one or more parts are deleted, the store MUST send a ***PartsDeleted*** message. a. A delete is an atomic operation; the store MUST perform the delete operation and then send notifications.
17.	Ending subscriptions	1. **REMINDER:** Row **6** and Row **11** 2. A store MUST end a customer's subscription to part notifications when: a. The customer cancels the subscription by sending an ***UnsubscribePartNotifications*** message. b. The parent growing data object for the subscription (i.e., the data object identified by the URI in the *context* field of the subscription's ***SubscriptionInfo*** record) is deleted. c. The customer loses access to the parent growing data object for the subscription. 3. When ending a subscription: a. The store MAY discard any queued part notifications for the subscription. b. The store MUST send a ***PartSubscriptionEnded*** message either as a response to a customer ***UnsubscribePartNotifications*** request or as a notification. c. The store MUST include a human readable reason why the subscription was ended in the ***PartSubscriptionEnded*** message. 4. After sending a ***PartSubscriptionEnded*** message, the store MUST NOT send any further part notifications for the subscription. 5. After a subscription has ended, the store MUST NOT restart it, even if the subscription was created by the store on behalf of the customer with the ***UnsolicitedPartNotifications*** message.
18.	Index Metadata	1. A growing data object's index metadata MUST be consistent: a. All parts MUST have the same index unit and the same vertical datum.

Row#	Requirement	Behavior
		b. The index units and vertical datums in the growing data header MUST match the parts. 2. When sending messages, both the store AND the customer MUST ensure that all index metadata and data derived from index metadata are consistent in all fields in the message, including in XML or JSON object data or part data. a. **EXAMPLE:** The *uom* and *depthDatum* in an ***IndexInterval*** record MUST be consistent with the channel's index metadata. b. **EXAMPLE:** Data object elements related to index values in growing data object headers (e.g., MdMn and MdMx on a WITSML 2.0 Trajectory) and parts (e.g., Md on a WITSML 2.0 TrajectoryStation) MUST be consistent with each other AND the data object's index metadata.

12.2.3 GrowingObjectNotification: Capabilities

The table below lists key capabilities for this protocol. The protocol capabilities are all defined here. For this protocol, one particularly crucial endpoint capability is defined here.

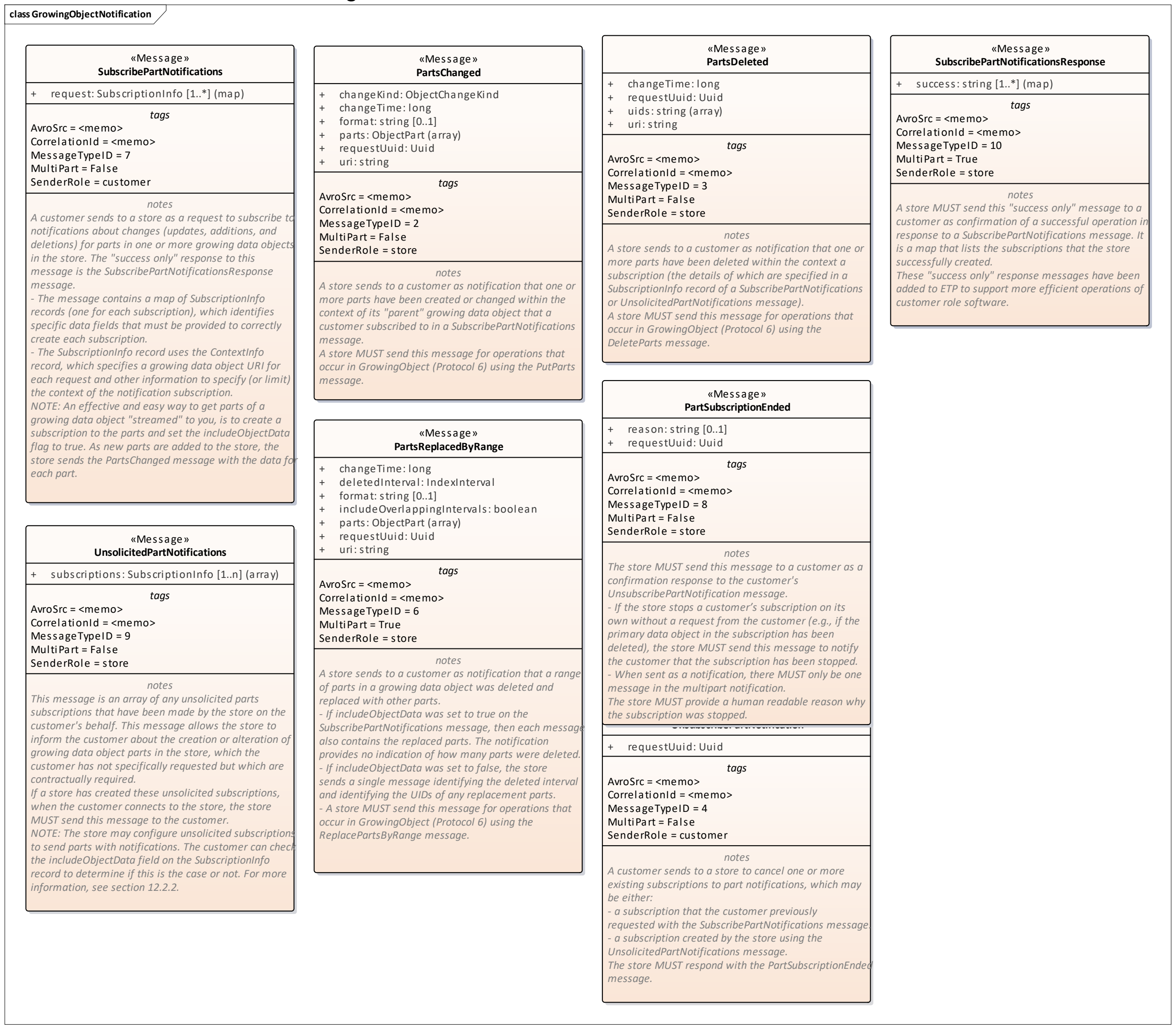
- For protocol-specific behavior relating to using these capabilities in this protocol, see Section **12.2.2**, **GrowingObjectNotification: General Requirements**.
- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section **3.3**.

GrowingObject (Protocol 6): Capabilities			
Name: Description	**Type**	**Units** **Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of each endpoint capability, see Section **3.3**.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section **3.3.2**. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP** Server.			
MaxPartSize: The maximum size in bytes of each data object part allowed in a standalone message or a complete multipart message. Size in bytes is the total size in bytes of the uncompressed string representation of the data object part in the format in which it is sent or received.	long	byte <number of bytes>	Min: 10,000 bytes
Data Object Capabilities (For definitions of each data object capability, see Section **3.3.4**.)			
Protocol Capabilities			
MaxSubscriptionSessionCount: The maximum total count of concurrent subscriptions allowed in a session. The limit applies separately for each protocol with the capability.	long	count <count of subscriptions>	**MIN:** 100

GrowingObject (Protocol 6): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
EXAMPLE: Different values can be specified for StoreNotification (Protocol 5) and GrowingObjectNotification (Protocol 7).			

12.3 GrowingObjectNotification: Message Schemas

This section provides a figure that displays all messages defined in GrowingObjectNotification (Protocol 7). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.



«Message»
SubscribePartNotifications

+ request: SubscriptionInfo [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 7
MultiPart = False
SenderRole = customer

notes
A customer sends to a store as a request to subscribe to notifications about changes (updates, additions, and deletions) for parts in one or more growing data objects in the store. The "success only" response to this message is the SubscribePartNotificationsResponse message.
- The message contains a map of SubscriptionInfo records (one for each subscription), which identifies specific data fields that must be provided to correctly create each subscription.
- The SubscriptionInfo record uses the ContextInfo record, which specifies a growing data object URI for each request and other information to specify (or limit) the context of the notification subscription.
NOTE: An effective and easy way to get parts of a growing data object "streamed" to you, is to create a subscription to the parts and set the includeObjectData flag to true. As new parts are added to the store, the store sends the PartsChanged message with the data for each part.

«Message»
UnsolicitedPartNotifications

+ subscriptions: SubscriptionInfo [1..n] (array)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 9
MultiPart = False
SenderRole = store

notes
This message is an array of any unsolicited parts subscriptions that have been made by the store on the customer's behalf. This message allows the store to inform the customer about the creation or alteration of growing data object parts in the store, which the customer has not specifically requested but which are contractually required.
If a store has created these unsolicited subscriptions, when the customer connects to the store, the store MUST send this message to the customer.
NOTE: The store may configure unsolicited subscriptions to send parts with notifications. The customer can check the includeObjectData field on the SubscriptionInfo record to determine if this is the case or not. For more information, see section 12.2.2.

«Message»
PartsChanged

+ changeKind: ObjectChangeKind
+ changeTime: long
+ format: string [0..1]
+ parts: ObjectPart (array)
+ requestUuid: Uuid
+ uri: string

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 2
MultiPart = False
SenderRole = store

notes
A store sends to a customer as notification that one or more parts have been created or changed within the context of its "parent" growing data object that a customer subscribed to in a SubscribePartNotifications message.
A store MUST send this message for operations that occur in GrowingObject (Protocol 6) using the PutParts message.

«Message»
PartsReplacedByRange

+ changeTime: long
+ deletedInterval: IndexInterval
+ format: string [0..1]
+ includeOverlappingIntervals: boolean
+ parts: ObjectPart (array)
+ requestUuid: Uuid
+ uri: string

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 6
MultiPart = True
SenderRole = store

notes
A store sends to a customer as notification that a range of parts in a growing data object was deleted and replaced with other parts.
- If includeObjectData was set to true on the SubscribePartNotifications message, then each message also contains the replaced parts. The notification provides no indication of how many parts were deleted.
- If includeObjectData was set to false, the store sends a single message identifying the deleted interval and identifying the UIDs of any replacement parts.
- A store MUST send this message for operations that occur in GrowingObject (Protocol 6) using the ReplacePartsByRange message.

«Message»
PartsDeleted

+ changeTime: long
+ requestUuid: Uuid
+ uids: string (array)
+ uri: string

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 3
MultiPart = False
SenderRole = store

notes
A store sends to a customer as notification that one or more parts have been deleted within the context a subscription (the details of which are specified in a SubscriptionInfo record of a SubscribePartNotifications or UnsolicitedPartNotifications message).
A store MUST send this message for operations that occur in GrowingObject (Protocol 6) using the DeleteParts message.

«Message»
PartSubscriptionEnded

+ reason: string [0..1]
+ requestUuid: Uuid

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 8
MultiPart = False
SenderRole = store

notes
The store MUST send this message to a customer as a confirmation response to the customer's UnsubscribePartNotification message.
- If the store stops a customer's subscription on its own without a request from the customer (e.g., if the primary data object in the subscription has been deleted), the store MUST send this message to notify the customer that the subscription has been stopped.
- When sent as a notification, there MUST only be one message in the multipart notification.
The store MUST provide a human readable reason why the subscription was stopped.

+ requestUuid: Uuid

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 4
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to cancel one or more existing subscriptions to part notifications, which may be either:
- a subscription that the customer previously requested with the SubscribePartNotifications message.
- a subscription created by the store using the UnsolicitedPartNotifications message.
The store MUST respond with the PartSubscriptionEnded message.

«Message»
SubscribePartNotificationsResponse

+ success: string [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 10
MultiPart = True
SenderRole = store

notes
A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a SubscribePartNotifications message. It is a map that lists the subscriptions that the store successfully created.
These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

Figure 23: GrowingObjectNotification: message schemas

12.3.1 Message: SubscribePartNotifications

A customer sends to a store as a request to subscribe to notifications about changes (updates, additions, and deletions) for parts in one or more growing data objects in the store.

The "success only" response to this message is the SubscribePartNotificationsResponse message.

- The message contains a map of SuscriptionInfo records (one for each subscription), which identifies specific data fields that must be provided to correctly create each subscription.
- The SubscriptionInfo record uses the ContextInfo record, which specifies a growing data object URI for each request and other information to specify (or limit) the context of the notification subscription.

NOTE: An effective and easy way to get parts of a growing data object "streamed" to you, is to create a subscription to the parts and set the *includeObjectData* flag to true. As new parts are added to the store, the store sends the ***PartsChanged*** message with the data for each part.

Message Type ID: 7

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
request	General ETP map of subscription requests; the details of each request is specified in a SuscriptionInfo record and includes information such as the URI of the subscription's growing data object and the request UUID that initiated the subscription. The *uri* field in the ***ContextInfo*** in the ***SubscriptionInfo*** MUST be set to the canonical URI for a growing data object. If both endpoints support alternate URIs for the session, the URIs MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**. If alternate URIs are used, the store MUST resolve them to canonical URIs and treat the subscription as a subscription to the canonical URI.	SubscriptionInfo	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObjectNotification",
     "name": "SubscribePartNotifications",
     "protocol": "7",
     "messageType": "7",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "request",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.Object.SubscriptionInfo" }
         }
     ]
}
```

12.3.2 Message: SubscribePartNotificationsResponse

A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a SubscribePartNotifications message. It is a map that lists the subscriptions that the store successfully created.

These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

Message Type ID: 10

Correlation Id Usage: MUST be set to the *messageId* of the ***SubscribePartNotifications*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
success	• The presence of the map key represents success. • The associated map string value SHOULD be empty because its content is ignored.	string	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObjectNotification",
     "name": "SubscribePartNotificationsResponse",
     "protocol": "7",
     "messageType": "10",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "success",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

12.3.3 Message: PartsChanged

A store sends to a customer as notification that one or more parts have been created or changed within the context a subscription (the details of which are specified in a SubscriptionInfo record of a SubscribePartNotifications or UnsolicitedPartNotifications message).

A store MUST send this message for operations that occur in GrowingObject (Protocol 6) using the ***PutParts*** message.

Message Type ID: 2

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
uri	The URI of the "parent" growing data object where the change occurred. For example: in WITSML, a Trajectory is a growing data object and each TrajectoryStation is a part.	string	1	1

Field Name	Description	Data Type	Min	Max
	This MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.			
requestUuid	The UUID of the subscription request that resulted in this notification message being sent. The UUID was assigned by the customer when the subscription was requested and created (in the SubscriptionInfo record) or by an UnsolicitedPartNotifications message. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1
changeKind	The information describing the change to the parts in the data object, which must be one of the enumerations specified in ObjectChangeKind.	ObjectChangeKind	1	1
changeTime	The time the data-change event occurred. This is not the time the event happened, but the time that the change occurred in the store database. This is the value from *storeLastWrite* field on the "parent" growing data object (for more information see Resource) and the ***ChangeAnnotation*** record created for the change. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
format	Specifies the format (e.g., XML or JSON) of the parts data being sent. This MUST match the format in the SubscriptionInfo record for the subscription.	string	0	1
parts	An array of the UIDs of the parts and optionally the data for each (as defined in the ObjectPart record) that have been added or updated in the growing data object (identified in the *uri* field above). The data for each part is only included if the customer set *includeObjectData* to true in the subscription request.	ObjectPart	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObjectNotification",
     "name": "PartsChanged",
     "protocol": "7",
     "messageType": "2",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         { "name": "changeKind", "type":
"Energistics.Etp.v12.Datatypes.Object.ObjectChangeKind" },
         { "name": "changeTime", "type": "long" },
         { "name": "format", "type": "string", "default": "" },
         {
             "name": "parts",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.Object.ObjectPart" }
         }
     ]
}
```

12.3.4 Message: PartsDeleted

A store sends to a customer as notification that one or more parts have been deleted within the context of a subscription (the details of which are specified in a SubscriptionInfo record of a SubscribePartNotifications or UnsolicitedPartNotifications message).

A store MUST send this message for operations that occur in GrowingObject (Protocol 6) using the ***DeleteParts*** message.

Message Type ID: 3

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
uri	The URI of the "parent" growing data object where the change occurred. For example: in WITSML, a Trajectory is a growing data object and each TrajectoryStation is a part. This MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
requestUuid	The UUID of the notification request (e.g., in SubscriptionInfo record of the SubscribePartNotification message) that resulted in this ***PartsDeleted*** message being sent. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1
uids	An array of parts (each identified by its UID) that have been deleted.	string	1	1
changeTime	The time the data-change event occurred. This is not the time the event happened, but the time that the change occurred in the store database. This is the value from *storeLastWrite* field on the "parent" growing data object (for more information see Resource) and the ***ChangeAnnotation*** record created for the change. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObjectNotification",
     "name": "PartsDeleted",
     "protocol": "7",
     "messageType": "3",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         { "name": "changeTime", "type": "long" },
         {
             "name": "uids",
             "type": { "type": "array", "items": "string" }
         }
     ]
}
```

12.3.5 Message: UnsubscribePartNotification

A customer sends to a store to cancel one or more existing subscriptions to part notifications, which may be either:

- a subscription that the customer previously requested with the SubscribePartNotifications message.
- a subscription created by the store using the UnsolicitedPartNotifications message.

The store MUST respond with the PartSubscriptionEnded message.

Message Type ID: 4

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
requestUuid	The UUID of the subscription that is being canceled. Each subscription was assigned a UUID by the customer requesting it, when the subscription was created (in the SubscriptionInfo record) or was assigned in an UnsolicitedPartNotifications message. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObjectNotification",
     "name": "UnsubscribePartNotification",
     "protocol": "7",
     "messageType": "4",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" }
     ]
}
```

12.3.6 Message: PartsReplacedByRange

A store sends to a customer as notification that a range of parts in a growing data object was deleted and replaced with other parts.

- If *includeObjectData* was set to true on the SubscribePartNotifications message, then each message also contains the replaced parts. The notification provides no indication of how many parts were deleted.
- If *includeObjectData* was set to false, the store sends a single message identifying the deleted interval and identifying the UIDs of any replacement parts.
- A store MUST send this message for operations that occur in GrowingObject (Protocol 6) using the ***ReplacePartsByRange*** message.

Message Type ID: 6

Correlation Id Usage: For the first message, MUST be set to 0. If there are multiple messages in this multipart request, the *correlationId* of all successive messages that comprise the request MUST be set to the *messageId* of the first message of the multipart request.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
uri	The URI of the "parent" growing data object from which the parts were deleted. For example: in WITSML, a Trajectory is a growing data object and each TrajectoryStation is a part. This MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
requestUuid	Each subscription was assigned a UUID by the customer requesting it, when the subscription was created (in the SubscriptionInfo record) or was assigned in an UnsolicitedPartNotifications message. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1
changeTime	The time the data-change event occurred. This is not the time the event happened, but the time that the change occurred in the store database. This is the value from *storeLastWrite* field on the "parent" growing data object (for more information see Resource) and the ***ChangeAnnotation*** record created for the change. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
deletedInterval	The index interval for the deleted range as specified in IndexInterval . This is NOT the index range of the parts that replaced the deleted parts.	IndexInterval	1	1
includeOverlappingIntervals	Specifies if the interval is inclusive or exclusive for objects that span the interval. For more information, see the Section **11.2.2.1**, which explains overlapping interval behavior. • If true, then any object with any part of it crossing the specified range is affected. • If false, then only objects that fall completely within the range are affected.	boolean	1	1
format	Specifies the format (e.g., XML or JSON) of the data for the replacement parts being sent in this message. This MUST match the format in the SubscriptionInfo record for the subscription. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.	string	0	1
parts	An array of ObjectPart records., which contain the UIDs of the parts that replaced the deleted If in the ***SubscribePartsNotification*** message, the customer: • set *includeObjectData* to true, then the array includes each part UID and its associated data. • set *includeObjectData* to false, then the array contains only the part UIDs.	ObjectPart	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObjectNotification",
     "name": "PartsReplacedByRange",
     "protocol": "7",
     "messageType": "6",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         { "name": "changeTime", "type": "long" },
         { "name": "deletedInterval", "type":
"Energistics.Etp.v12.Datatypes.Object.IndexInterval" },
         { "name": "includeOverlappingIntervals", "type": "boolean" },
         { "name": "format", "type": "string", "default": "" },
         {
             "name": "parts",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.Object.ObjectPart" }
         }
     ]
}
```

12.3.7 Message: PartSubscriptionEnded

The store MUST send this message to a customer as a confirmation response to the customer's UnsubscribePartNotification message.

If the store stops a customer's subscription on its own without a request from the customer (e.g., if the primary data object in the subscription has been deleted), the store MUST send this message to notify the customer that the subscription has been stopped.

When sent as a notification, there MUST only be one message in the multipart notification.

The store MUST provide a human readable reason why the subscription was stopped.

Message Type ID: 8

Correlation Id Usage: When sent as a response: MUST be set to the *messageId* of the ***UnsubscribePartNotifications*** message that this message is a response to. When sent as a notification: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
Reason	A reason why the subscriptions have been stopped.	string	1	1
requestUuid	The UUID of the subscription the store is ending. These UUIDs were assigned by the customer when the subscription was requested (in the SubscriptionInfo record) or by an UnsolicitedPartNotifications message. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObjectNotification",
     "name": "PartSubscriptionEnded",
     "protocol": "7",
     "messageType": "8",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "reason", "type": "string" },
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" }
     ]
}
```

12.3.8 Message: UnsolicitedPartNotifications

This message is an array of any unsolicited parts subscriptions that have been made by the store on the customer's behalf. This message allows the store to inform the customer about the creation or alteration of growing data object parts in the store, which the customer has not specifically requested but which are contractually required.

If a store has created these unsolicited subscriptions, when the customer connects to the store, the store MUST send this message to the customer.

NOTE: The store may configure unsolicited subscriptions to send parts with notifications. The customer can check the *includeObjectData* field on the ***SubscriptionInfo*** record to determine if this is the case or not. For more information, see section **12.2.2**.

Message Type ID: 9

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
subscriptions	An array of SubscriptionInfo records, each of which identifies the details of an unsolicited parts subscription. Each record includes information such as the URI of the subscription's growing data object, and the request UUID that initiated a subscription. The *uri* field in the ***ContextInfo*** record in the ***SubscriptionInfo*** record MUST be set to the canonical URI for a growing data object. The URI in the ***ContextInfo*** record MUST be a canonical Energistics data object URI for a growing data object; for more information, see **Appendix: Energistics Identifiers**.	SubscriptionInfo	1	n

Avro Source

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Protocol.GrowingObjectNotification",
    "name": "UnsolicitedPartNotifications",
    "protocol": "7",
    "messageType": "9",
    "senderRole": "store",
    "protocolRoles": "store,customer",
    "multipartFlag": false,

    "fields":
    [
        {
             "name": "subscriptions",
             "type": { "type": "array", "items": 
"Energistics.Etp.v12.Datatypes.Object.SubscriptionInfo" }
         }
     ]
}
```

13 DataArray (Protocol 9)

ProtocolID: 9

Defined Roles: store, customer

Use DataArray (Protocol 9) to transfer large, binary arrays of homogeneous data values. With Energistics domain standards, this data is often stored as an HDF5 file. However, this protocol can be used for any array data, even if HDF files are not required or used.

Energistics domain standards have typically store this type of data using HDF5. For example, RESQML uses HDF5 to store seismic, interpretation, and modeling data: PRODML-DAS uses HDF5 to store distributed acoustic sensing data. As such, the arrays that are transferred with the DataArray protocol are logical versions of the HDF5 data sets.

DataArray (Protocol 9):

- Supports any array of values of different types (bytes, integer, float, doubles, etc.). In Energistics data models, this array data is typically associated with a data object (that is, it is the binary array data for the data object).
- Imposes no limits on the dimensions of the array. Multi-dimensional arrays have no limits to the number of dimensions. However, a store may limit the size of a message and therefore the size of the arrays it can handle in a single message. For this reason, this protocol provides functionality to portion arrays into manageably sized sub-arrays for data transfer.
- Was designed to support transfer of the data typically stored in HDF5 files but also can be used to transfer this type of data when HDF5 files are not required or used.

This chapter includes main sections for:

- Key ETP concepts that are important to understanding how this protocol is intended to work (Section **13.1**).
- Required behavior, which includes:
  - Description of the message sequence for main tasks, along with required behavior and possible errors (Section **13.2.1**).
  - Other functional requirements (not covered in the message sequence) including use of endpoint and protocol capabilities for preventing and protecting against aberrant behavior (Section **13.2.2**).
  - Definitions of the endpoint and protocol capabilities used in this protocol (Section **13.2.3**).
- Sample schemas of the messages defined in this protocol (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes documentation for each field in a schema (Section **13.3**).

13.1 DataArray Key Concepts

This section explains key concepts related to this protocol.

13.1.1 HDF5 and EPC Files

The Energistics Common Technical Architecture (CTA) includes these technologies:

- **HDF5** is a data model, a set of open file formats, and libraries designed to store and organize large amounts of data for improved speed and efficiency of data processing.
- **Energistics Packaging Convention (EPC)** is a file packaging convention based on the Open Packaging Conventions (OPC), a widely used container-file technology that allows multiple types of files to be bundled together into a single package, which is built on the widely used ZIP file structure.

EPC provides a way to bundle together multiple files into a single file—referred to as an EPC file, which is actually a ZIP file tailored for Energistics use—and to identify the relationships among the contained files, for data transfer. If the file set includes large binary arrays, they are stored in one or more HDF5 files. In most cases, the HDF5 files are stored outside the EPC file. To accurately maintain all relationships, the package requires use of an external reference to the HDF5 file, which is called an EpcExternalPartReference.

DataArray (Protocol 9) has been designed to get and put data arrays in this context--and also to handle this type of array data when no files are required, for example, from endpoint to endpoint. For more information about these technologies and their use in the Energistics CTA, see Energistics Online: http://docs.energistics.org/#CTA/CTA_TOPICS/CTA-000-018-0-C-sv2100.html and http://docs.energistics.org/#ETP/ETP_TOPICS/ETP-000-000-titlepage.html.

13.2 DataArray: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.
- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**
- In most cases, when this text refers to providing a URI it is referring to the canonical Energistics URI. For more information, see Section **25.3.5**.

13.2.1 DataArray: Message Sequences

This section explains the basic message sequence for main tasks to be done using this protocol and includes related key behaviors and possible errors. The following General Requirements section provides additional requirements and rules for how this protocol works.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section (Section **13.3**).

DataArray (Protocol 9): Basic Message-Response flow by ETP Role	
Message sent by customer	**Response Message from store**
GetDataArrayMetadata**:** Request for metadata (dimensions) about one or more data arrays.	***GetDataArrayMetadataResponse*** (multipart): Response that provides the requested metadata.
GetDataArrays: Request for one or more data arrays.	***GetDataArraysResponse*** (multipart): Response message containing array data that the store could return.
GetDataSubarrays: Request for one or more sub-arrays (part of a larger array) of data.	***GetDataSubarraysResponse*** (multipart): Response message containing array data that the store could return.

DataArray (Protocol 9): Basic Message-Response flow by ETP Role	
Message sent by customer	**Response Message from store**
PutUninitializedDataArrays: Establishes the dimensions for an array in a store before sending the data for the array.	***PutUninitializedDataArraysResponse*** (multipart): Success only response that the lists the arrays that were initialized.
PutDataArrays: Request to put one or more data arrays.	***PutDataArraysResponse*** (multipart): Success only response that the lists the arrays that were added to the store.
PutDataSubarrays: Request to put one or more sub-arrays of data.	***PutDataSubarraysResponse*** (multipart): Success only response that the lists the sub-arrays that were added to the store.

13.2.1.1 To get one or more data arrays:

1. First a customer needs to determine the type and dimensions of the arrays it is interested in/wants to work with. To do this, the customer sends to the store, the ***GetDataArrayMetadata*** message (Section **13.3.11**), which is a map whose values are the identifiers of the data arrays of interest.
   a. The identifier is a ***DataArrayIdentifier*** record (Section **23.32.3**), which consists of a URI that identifies a resource that contains the array data and a path within the resource for the array data.
   b. **NOTE:** The resource is NOT a ***Resource*** record. The resource may or may not be an HDF file. The URI is not guaranteed to be an Energistics URI. For more information, see the relevant ML ETP implementation specification.
2. The store MUST respond with the ***GetDataArrayMetadataResponse*** message (Section **13.3.12**), which is a map whose values are the metadata (type (e.g., float, integer, string, etc.) indexes and dimensions) for the each requested array the store could respond for.
3. Based on the dimension data provided in the ***GetDataArrayMetadataResponse*** message, the customer determines:
   a. If the data array is small enough that it can be retrieved with one call, use the ***GetDataArrays*** message (Section **13.3.1**) to list the URIs of the data arrays it wants to retrieve.
      i. The store responds with the ***GetDataArraysResponse*** message (Section **13.3.2**) with the arrays it can return.
   b. If the data arrays are too large to get in one call, then the customer must retrieve the array in multiple calls, using ***GetDataSubarrays*** messages (Section **13.3.3**). Each call typically gets the different portions corresponding to a decomposition of the array. According to the size, this decomposition can be by columns, planes, sub-cubes, etc.
      i. The store responds with ***GetDataSubarraysResponse*** messages (Section **13.3.4**), to send the data for the arrays of interest in small sub-arrays that the endpoint can handle.
      ii. The customer then "assembles" the ***GetDataSubarraysResponse*** messages to process the complete array.
   c. For more information on the related protocol capability, MaxDataArraySize, see Section **13.2.2**, Row **5**.

13.2.1.2 To put one or more data arrays:

NOTE: ETP uses upsert semantics so the same message is used to add a new or update existing data arrays.

1. For a large array that will not fit in a single message, before a customer can put a data array in a store, it must specify the indexes and dimensions of the array. To do this, a customer MUST send to

the store the ***PutUninitializedDataArrays*** message (Section **13.3.9**).

   a. If the array is small enough, this step is not required.

2. The store MUST respond with a ***PutUninitializedDataArraysResponse*** message (Section **13.3.10**) indicating success of the operation.
3. Next the customer can add data to the array. Depending on the array size, the customer does one of the following:
   a. If the array is small enough, the customer sends a ***PutDataArrays*** message (Section **13.3.5**) which is a map of the arrays and the respective data for each.
      i. For requests that the store successfully completed, the store MUST respond with a ***PutDataArraysResponse*** message (Section **13.3.6**) indicating success of the operation.
   b. If the arrays are too large for one message, the customer MUST send multiple ***PutDataSubarrays*** messages (Section **13.3.7**).
      i. The store MUST process the various sub-array messages to fill the complete data array; for requests that it was able to complete, the store MUST respond with a ***PutDataSubarraysResponse*** message (Section **13.3.8**) indicating success of the operation.
   c. For more information on the related protocol capability, MaxDataArraySize, see Section **13.2.2**, Row **5**.

13.2.1.3 Transactions

Typically, arrays are not transferred in isolation because they correspond to the binary data associated to a data object. All array-put transfers corresponding to the same data object must be successful for the data object to be complete, so it is recommended to include transfer of the data object and its arrays in a transaction, using Transaction (Protocol 18); see Chapter **18**.

13.2.2 DataArray: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) additional rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and

Row#	Requirement	Behavior
		OpenSession messages in Core (Protocol 0). For more information, see Chapter **5**. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3**. a. For the list of global capabilities and related behavior, see Section **3.3.2**. 3. Section **13.2.3** identifies the capabilities most relevant to this ETP sub-protocol. Additional details for how to use the protocol capabilities are included below in this table and in Section **13.2.1 DataArray: Message Sequence**.
3.	Message Sequence See Section **13.2.1**.	1. The Message Sequence section above (Section **13.2.1**) describes requirements for the main tasks listed there and also defines required behavior.
4.	Plural Messages (which includes maps)	1. This protocol uses plural messages. For detailed rules on handling plural messages (including ***ProtocolException*** handling), see Section **3.7.3**.
5.	Endpoints MUST honor MaxDataArraySize protocol capability (for definition, see Section **13.2.3**).	1. For any get or put operations in this protocol where array data is being sent, each endpoint MUST NOT exceed the other's value for MaxDataArraySize protocol capability. 2. If an endpoint's value is exceeded, it MUST deny the request by sending error ELIMIT_EXCEEDED (12).
6.	Transaction (optional)	Because data arrays can be large and complex, operations associated with them can also be large and complex. Endpoints can optionally define a transaction using Transaction (Protocol 18). For more information, see Chapter **18**.
7.	**Store behavior:** updates to a data array's *storeCreated* and *storeLastWrite fields*	1. Similarly to a ***Resource***, data arrays have metadata fields named *storeCreated* and *storeLastWrite* fields, which are maintained by an ETP store, primarily to support replication workflows. For more information about these fields, see: a. Section **23.32.2**. b. Section **3.12.5.1**. 2. For operations in this protocol that ADD a new data array (e.g. ***PutDataArrays, PutUninitializedDataArrays***), the store MUST do both of these: a. Set the *storeCreated* field to the time that the array was added in the store. b. Set the *storeLastWrite* field to the same time as *storeCreated*. 3. For operations in this protocol that UPDATE a data array (e.g. ***PutDataArrays***, because ETP uses upsert semantics the same message is used to add or update the array), the store MUST update the *storeLastWrite* field with the time that the update happened in the store.
8.	Updates to data in a data array require an update to the data object(s) that reference that array.	1. When a data value in a data array is updated, each data object that references the array MUST also be updated.

Row#	Requirement	Behavior
		a. At a minimum, the *storeLastWrite* on the resource for the data object MUST be updated. Other changes may be required depending on the nature of the change.
9.	**Data Array Metadata:** Mapping of logical and transport array types	1. For operations in this protocol where data array metadata must be specified (see messages that use ***DataArrayMetadata*** record), these rules MUST be observed for populating the *logicalArrayType* and *transportArrayType* fields: a. Both fields are required and MUST be populated, each with a value from its respective enumeration. b. Values for *logicalArrayType* can only use specific values for *transportArrayType*. For the allowed mapping of values, see Section 13.2.2.1.

13.2.2.1 Allowed Mappings of Logical Array Types and Transport Array Types

In the table below, for each enumeration (specified in AnyLogicalArrayType) for the *logicalArrayType* field (left column), the right column specifies the allowed enumerations (specified in AnyArrayType) for the *transportArrayType* field.

The types in the left column are all the enumerations listed in the ***AnyLogicalArrayType*** record (see Section 23.1). These types have been specified based on signed/unsigned (U), bit size of the preferred sub-array dimension (8, 16, 32, 64 bits), and endianness (LE = little, BE = big).

The following usage rules apply:

1. Implementers decide which encoding is best for their data and particular implementation.
2. Type of "bytes" is a fixed-size encoding. So an 8-bit array uses 1 byte, 16 bit uses 2 bytes, 32 bit uses 4 bytes, and 64 bit uses 8 bytes.
3. Types "arrayOfLong" and "arrayOfInt" follow Avro encoding, which is variable length.

Logical Array Type (AnyLogicalArrayType)	**Allowed Transport Array Type (AnyArrayType)**
arrayOfBoolean	arrayOfBoolean
arrayOfInt8	bytes, arrayOfInt, arrayOfLong
arrayOfUInt8	bytes, arrayOfInt, arrayOfLong
arrayOfInt16LE	bytes, arrayOfInt, arrayOfLong
arrayOfInt32LE	bytes, arrayOfInt, arrayOfLong
arrayOfInt64LE	bytes, arrayOfLong
arrayOfUInt16LE	bytes, arrayOfInt, arrayOfLong
arrayOfUInt32LE	bytes, arrayOfInt, arrayOfLong **NOTE:** When arrayOfInt is used, reinterpret casts are required prior to Avro encoding.
arrayOfUInt64LE	bytes, arrayOfLong **NOTE:** When arrayOfLong is used, reinterpret casts are required prior to Avro encoding.
arrayOfFloat32LE	arrayOfFloat
arrayOfDouble64LE	arrayOfDouble
arrayOfInt16BE	bytes
arrayOfInt32BE	bytes

Logical Array Type (AnyLogicalArrayType)	Allowed Transport Array Type (AnyArrayType)
arrayOfInt64BE	bytes
arrayOfUInt16BE	bytes
arrayOfUInt32BE	bytes
arrayOfUInt64BE	bytes
arrayOfFloat32BE	bytes
arrayOfDouble64BE	bytes
arrayOfString	arrayOfString
arrayOfCustom	Not specified.

13.2.3 DataArray: Capabilities

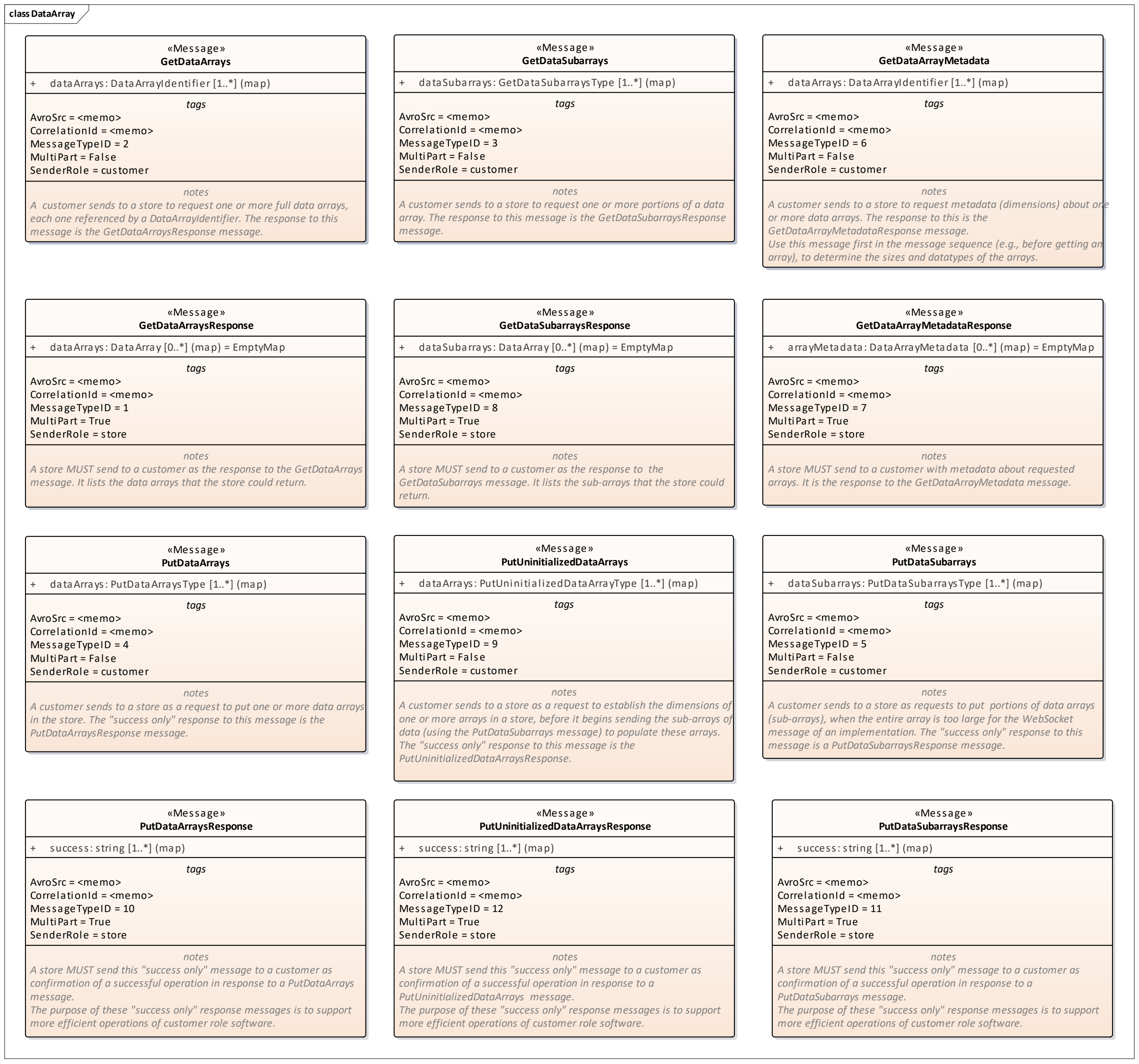
The table below lists key capabilities for this protocol. The protocol capabilities are all defined here.

- For protocol-specific behavior relating to using these capabilities in this protocol, see Section **13.2.2**, **DataArray: General Requirements**.
- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section **3.3**.

DataArray (Protocol 9): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of each endpoint capability, see Section **3.3**.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section **3.3.2**. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP** Server.			
Protocol Capabilities			
MaxDataArraySize: The maximum size in bytes of a data array allowed in a store. Size in bytes is the product of all array dimensions multiplied by the size in bytes of a single array element.	long	byte <number of bytes>	**MIN:**100,000

13.3 DataArray: Message Schemas

This section provides a figure that displays all messages defined in DataArray (Protocol 9). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.



«Message»
GetDataArrays
+ dataArrays: DataArrayIdentifier [1..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 2
MultiPart = False
SenderRole = customer
notes
A customer sends to a store to request one or more full data arrays, each one referenced by a DataArrayIdentifier. The response to this message is the GetDataArraysResponse message.

«Message»
GetDataSubarrays
+ dataSubarrays: GetDataSubarraysType [1..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 3
MultiPart = False
SenderRole = customer
notes
A customer sends to a store to request one or more portions of a data array. The response to this message is the GetDataSubarraysResponse message.

«Message»
GetDataArrayMetadata
+ dataArrays: DataArrayIdentifier [1..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 6
MultiPart = False
SenderRole = customer
notes
A customer sends to a store to request metadata (dimensions) about one or more data arrays. The response to this is the GetDataArrayMetadataResponse message.
Use this message first in the message sequence (e.g., before getting an array), to determine the sizes and datatypes of the arrays.

«Message»
GetDataArraysResponse
+ dataArrays: DataArray [0..*] (map) = EmptyMap
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1
MultiPart = True
SenderRole = store
notes
A store MUST send to a customer as the response to the GetDataArrays message. It lists the data arrays that the store could return.

«Message»
GetDataSubarraysResponse
+ dataSubarrays: DataArray [0..*] (map) = EmptyMap
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 8
MultiPart = True
SenderRole = store
notes
A store MUST send to a customer as the response to the GetDataSubarrays message. It lists the sub-arrays that the store could return.

«Message»
GetDataArrayMetadataResponse
+ arrayMetadata: DataArrayMetadata [0..*] (map) = EmptyMap
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 7
MultiPart = True
SenderRole = store
notes
A store MUST send to a customer with metadata about requested arrays. It is the response to the GetDataArrayMetadata message.

«Message»
PutDataArrays
+ dataArrays: PutDataArraysType [1..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 4
MultiPart = False
SenderRole = customer
notes
A customer sends to a store as a request to put one or more data arrays in the store. The "success only" response to this message is the PutDataArraysResponse message.

«Message»
PutUninitializedDataArrays
+ dataArrays: PutUninitializedDataArrayType [1..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 9
MultiPart = False
SenderRole = customer
notes
A customer sends to a store as a request to establish the dimensions of one or more arrays in a store, before it begins sending the sub-arrays of data (using the PutDataSubarrays message) to populate these arrays. The "success only" response to this message is the PutUninitializedDataArraysResponse.

«Message»
PutDataSubarrays
+ dataSubarrays: PutDataSubarraysType [1..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 5
MultiPart = False
SenderRole = customer
notes
A customer sends to a store as requests to put portions of data arrays (sub-arrays), when the entire array is too large for the WebSocket message of an implementation. The "success only" response to this message is a PutDataSubarraysResponse message.

«Message»
PutDataArraysResponse
+ success: string [1..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 10
MultiPart = True
SenderRole = store
notes
A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutDataArrays message.
The purpose of these "success only" response messages is to support more efficient operations of customer role software.

«Message»
PutUninitializedDataArraysResponse
+ success: string [1..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 12
MultiPart = True
SenderRole = store
notes
A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutUninitializedDataArrays message.
The purpose of these "success only" response messages is to support more efficient operations of customer role software.

«Message»
PutDataSubarraysResponse
+ success: string [1..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 11
MultiPart = True
SenderRole = store
notes
A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutDataSubarrays message.
The purpose of these "success only" response messages is to support more efficient operations of customer role software.

Figure 24: DataArray: message schemas

13.3.1 Message: GetDataArrays

A customer sends to a store to request one or more full data arrays, each one referenced by a DataArrayIdentifier. The response to this message is the GetDataArraysResponse message.

Message Type ID: 2

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
dataArrays	A map of DataArrayIdentifier records, one for each array being requested; each record identifies the URI (*uri* field) of the resource and the path in that resource to the specific array (*pathInResource* field). If both endpoints support alternate URIs for the session, the URIs MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	DataArrayIdentifier	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.DataArray",
     "name": "GetDataArrays",
     "protocol": "9",
     "messageType": "2",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "dataArrays",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.DataArrayIdentifier" }
         }
     ]
}
```

13.3.2 Message: GetDataArraysResponse

A store MUST send to a customer as the response to the GetDataArrays message. It lists the data arrays that the store could return.

Message Type ID: 1

Correlation Id Usage: MUST be set to the *messageId* of the ***GetDataArrays*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
dataArrays	A map of DataArray records, each of which is composed of the dimensions of an array and the data for the array.	DataArray	0	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.DataArray",
     "name": "GetDataArraysResponse",
     "protocol": "9",
     "messageType": "1",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
```

```
     [
         {
             "name": "dataArrays",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.DataArray" }, "default": {}
         }
     ]
}
```

13.3.3 Message: GetDataSubarrays

A customer sends to a store to request one or more portions of a data array. The response to this message is the GetDataSubarraysResponse message.

Message Type ID: 3

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
dataSubarrays	A map of GetDataSubarraysType records, each of which has the necessary fields to identify and locate the desired sub-arrays. If both endpoints support alternate URIs for the session, the URIs MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	GetDataSubarraysType	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.DataArray",
     "name": "GetDataSubarrays",
     "protocol": "9",
     "messageType": "3",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "dataSubarrays",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.GetDataSubarraysType" }
         }
     ]
}
```

13.3.4 Message: GetDataSubarraysResponse

A store MUST send to a customer as the response to the GetDataSubarrays message. It lists the sub-arrays that the store could return.

Message Type ID: 8

Correlation Id Usage: MUST be set to the *messageId* of the ***GetDataSubarrays*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
dataSubarrays	A map of DataArray records, each of which contains the dimensions and the data for one sub-array.	DataArray	0	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.DataArray",
     "name": "GetDataSubarraysResponse",
     "protocol": "9",
     "messageType": "8",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "dataSubarrays",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.DataArray" }, "default": {}
         }
     ]
}
```

13.3.5 Message: PutDataArrays

A customer sends to a store as a request to put one or more data arrays in the store. The "success only" response to this message is the PutDataArraysResponse message.

Message Type ID: 4

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
dataArrays	A map of PutDataArraysType records, each of which contains the identifier and data for each array in the request. If both endpoints support alternate URIs for the session, the URIs MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	PutDataArraysType	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.DataArray",
     "name": "PutDataArrays",
     "protocol": "9",
     "messageType": "4",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "dataArrays",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.PutDataArraysType" }
         }
     ]
}
```

13.3.6 Message: PutDataArraysResponse

A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutDataArrays message.

The purpose of these "success only" response messages is to support more efficient operations of customer role software.

Message Type ID: 10

Correlation Id Usage: MUST be set to the *messageId* of the ***PutDataArray***s message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
success	• The presence of the map key represents success. • The associated map string value SHOULD be empty because its content is ignored.	string	1	*

Avro Source

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Protocol.DataArray",
    "name": "PutDataArraysResponse",
    "protocol": "9",
    "messageType": "10",
    "senderRole": "store",
    "protocolRoles": "store,customer",
    "multipartFlag": true,
    "fields":
    [
        {
            "name": "success",
            "type": { "type": "map", "values": "string" }
        }
    ]
}
```

13.3.7 Message: PutDataSubarrays

A customer sends to a store as requests to put portions of data arrays (sub-arrays), when the entire array is too large for the WebSocket message of an implementation. The "success only" response to this message is a PutDataSubarraysResponse message.

Message Type ID: 5

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
dataSubarrays	A map of PutDataSubarraysType records, each of which contains the identifier and data for each sub-array in the request. If both endpoints support alternate URIs for the session, the URIs MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	PutDataSubarraysType	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.DataArray",
     "name": "PutDataSubarrays",
     "protocol": "9",
     "messageType": "5",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "dataSubarrays",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.PutDataSubarraysType" }
         }
     ]
}
```

13.3.8 Message: PutDataSubarraysResponse

A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutDataSubarrays message.

The purpose of these "success only" response messages is to support more efficient operations of customer role software.

Message Type ID: 11

Correlation Id Usage: MUST be set to the *messageId* of the ***PutDataSubarrays*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
success	• The presence of the map key represents success. • The associated map string value SHOULD be empty because its content is ignored.	string	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.DataArray",
     "name": "PutDataSubarraysResponse",
     "protocol": "9",
     "messageType": "11",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "success",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

13.3.9 Message: PutUninitializedDataArrays

A customer sends to a store as a request to establish the dimensions of one or more arrays in a store, before it begins sending the sub-arrays of data (using the PutDataSubarrays message) to populate these arrays.

The "success only" response to this message is the PutUninitializedDataArraysResponse.

Message Type ID: 9

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
dataArrays	A map of PutUninitializedDataArrayType records, each of which contains the identifier and data for metadata for each request. If both endpoints support alternate URIs for the session, the URIs MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	PutUninitializedDataArrayType	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.DataArray",
     "name": "PutUninitializedDataArrays",
     "protocol": "9",
     "messageType": "9",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "dataArrays",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.PutUninitializedDataArrayType" }
         }
     ]
}
```

13.3.10 Message: PutUninitializedDataArraysResponse

A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutUninitializedDataArrays message.

The purpose of these "success only" response messages is to support more efficient operations of customer role software.

Message Type ID: 12

Correlation Id Usage: MUST be set to the *messageId* of the ***PutUninitializedDataArrays*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
success	• The presence of the map key represents success. • The associated map string value SHOULD be empty because its content is ignored.	string	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.DataArray",
     "name": "PutUninitializedDataArraysResponse",
     "protocol": "9",
     "messageType": "12",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "success",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

13.3.11 Message: GetDataArrayMetadata

A customer sends to a store to request metadata (dimensions) about one or more data arrays. The response to this is the GetDataArrayMetadataResponse message.

Use this message first in the message sequence (e.g., before getting an array), to determine the sizes and datatypes of the arrays.

Message Type ID: 6

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
dataArrays	A map of DataArrayIdentifier records, one for each array being requested; each record identifies the URI (*uri* field) of the resource and the path in that resource to the specific array (*pathInResource* field). If both endpoints support alternate URIs for the session, the URIs MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	DataArrayIdentifier	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.DataArray",
     "name": "GetDataArrayMetadata",
     "protocol": "9",
     "messageType": "6",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "dataArrays",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.DataArrayIdentifier" }
         }
     ]
}
```

13.3.12 Message: GetDataArrayMetadataResponse

A store MUST send to a customer with metadata about requested arrays. It is the response to the GetDataArrayMetadata message.

Message Type ID: 7

Correlation Id Usage: MUST be set to the *messageId* of the ***GetDataArrayMetadata*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
arrayMetadata	A map of DataArrayMetadata records, one for each array that the store could return metadata for.	DataArrayMetadata	0	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.DataArray",
     "name": "GetDataArrayMetadataResponse",
     "protocol": "9",
     "messageType": "7",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "arrayMetadata",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.DataArrayMetadata" }, "default": {}
         }
     ]
}
```

14 Overview of Query Behavior

For query functionality, ETP uses query syntax based on parts of the Open Data Protocol (OData) query string syntax, specifically OData v4.0. ETP queries use a canonical Energistics data object query URI, which may include a query string. (For information about Energistics URI formats, see **Appendix: Energistics Identifiers**.)

OData is an OASIS standard (https://www.oasis-open.org/standards/#odatav4.0). This query syntax is very flexible and powerful, allowing complex queries.

OData was selected because it is a widely known (introduced in 2007) and maturing standard. Many client and server libraries are available for all major platforms.

This chapter provides general information that is applicable to all query sub-protocols published in the current version of ETP. The information in this chapter must be used with the information in the sub-protocol-specific query chapters, namely:

- DiscoveryQuery (Protocol 13); see Chapter **15**.
- StoreQuery (Protocol 14); see Chapter **16**.
- GrowingObjectQuery (Protocol 16); see Chapter **17**.

OData resources:

- **OASIS URL Conventions: http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part2-url-conventions.html**
- **OData.net libraries: http://www.odata.org/libraries/**
- **Link to Querystring features: http://linqtoquerystring.net/features.html**

How ETP query sub-protocols work:

- They operate by allowing filtering on specific values in data fields within a resource, data object or parts of a growing data object (separate query sub-protocols for each of these as listed above) and customer-side pagination, based on OData syntax.
- They can be seen as companion querying functionality in support of operations in other ETP sub-protocols. For example:
  - DiscoveryQuery (Protocol 13) has relevant query functionality in support of discovery operations (which are defined in Discovery (Protocol 3))
  - StoreQuery (Protocol 14) has relevant query functionality in support of store operations (Store (Protocol 4)
  - GrowingObjectQuery (Protocol 16) has relevant query functionality in support of operation on parts of a growing data object (GrowingObject (Protocol 6)).

  NOTE: Putting query functionality in separate ETP sub-protocols was a conscious design choice so that support of query functionality was optional. **REMINDER:** The ETP rule is if an application supports a protocol, it MUST support ALL functionality in that protocol. But you can choose which protocol(s) you choose to support.

14.1 Supported Query Options and Requirements

ETP query syntax supports only OData style filtering and client-side pagination. This section lists functional requirements and behaviors that all ETP-enabled applications that implement query protocols MUST support.

14.1.1 Filtering

The $filter clause allows you to create an expression filtering out objects based on the elements of the object. For example, the following statement:

```
eml:///witsml20.Channel?$filter=ChannelClass/Title eq 'Gamma'&$top=300
```

returns results where the Channel and the element *channelClass* has a *Title* with value ‘Gamma’.

Implementations MUST observe these rules for filtering:

Canonical data object query URIs ALWAYS include an object type:

1. The $filter query option filters data objects whose type that matches the data object type in the query URI.
    a. All canonical Energistics data object query URIs specify a data object type.
    b. To use $filter, the query URI MUST include a query string with the $filter operator.
    c. URIs containing filters for elements with abstract types are not supported.
2. Support these OData query options for $filter:
    a. Logical operators: eq, ne, gt, ge, lt, le, and, or, not
    b. Collection Operators: any
    c. Primitive Literals
    d. Precedence grouping with ( )
    e. String functions: startswith, endswith, contains, tolower, toupper

14.1.2 Pagination

Pagination is used to limit the number of elements returned from a query to prevent exceeding limits specified by the MaxResponseCount protocol capability, which results in error ERESPONSECOUNT_EXCEEDED (30) error. (**EXAMPLE:** If the customer's MaxResponseCount protocol capability is 1,000 and the store has 3,000 objects of interest, the customer can use pagination options to limit returned values to groups of 1,000.)

These pagination query options are supported:

- $top limits the number of items returned in the result
- $skip is used to indicate the starting index for the subset.

ETP implementations MUST observe these rules for pagination:

1. The values used for $skip and $top MUST be greater than zero.
2. $skip and $top CAN be used individually or in combination.
3. The customer MUST specify the pagination, and the server MUST handle the requested pagination. **EXAMPLE:** A query requesting the first 300 objects looks like this:

```
eml:///witsml20.Channel$top=300
```

4. For pagination to work, the store MUST implement a deterministic sort order and it MUST provide that

sort order in the query response.

   a. It is NOT part of the ETP Specification to specify how the server sorts data.

14.1.2.1 Server Sort Order Requirements

1. The server MUST provide the sort order attribute in the response message for each of the query protocols, using the *serverSortOrder* field.
   a. For multipart response messages, the *serverSortOrder* field is valid only in the first response message.
      i. In subsequent parts of the multipart response messages, the customer MUST ignore the field and the server MAY set it to an empty string.
2. The values in the *serverSortOrder* field is a comma-delimited list of:|
   odata path<space>direction tuples
   a. If direction is absent, **asc** (ascending) is assumed.

14.2 Unsupported Query Options

ETP does NOT officially support these OData query options. An individual implementation may use these options. (**NOTE:** This is not a complete list; if you don't see it listed under supported options above then it is not supported by ETP).

- $orderby – sorting
- $groupby – grouping
- $select – projection
- $expand – sub queries
- $count – result count
- $format – xml vs. json
- $search – free text
- All collection operator
- Arithmetic operators and functions, e.g. add, sub(tract)
- Other filter functions, e.g. date and time and geo
- Complex or collection literals

14.3 General Behavior for all ETP Query Sub-Protocols

This section explains the general message sequence and details of how the query syntax works for all ETP query sub-protocols. For additional details for a specific query sub-protocols, see the relevant chapter (as listed **above** in this chapter)

14.3.1 Message Sequence for All ETP Query Sub-Protocols

Each ETP query sub-protocol has the same two basic messages and MUST follow the same basic sequence, which is explained in this section.

NOTE: The protocol-specific chapters for each of the published ETP query sub-protocols may have additional requirement details. For any given query sub-protocol, an implementation MUST honor these rules and the rules specified in the Required Behavior section of the related protocol chapter.

The two messages defined for each query sub-protocol:

1. A "***Find****X*" message, where *X* is the appropriate "thing" for that protocol (i.e., For DiscoveryQuery (Protocol 13) it's ***FindResources***, for StoreQuery (Protocol 14) it is ***FindDataObjects***, for GrowingObjectQuery (Protocol 16) it is ***FindParts***).

2. A "***Find****X****Response***" message, where *X* is the same "thing" in the ***Find****X* message (i.e., For DiscoveryQuery (Protocol 13) it's ***FindResourcesResponse***, for StoreQuery (Protocol 14) it is ***FindDataObjectsResponse***, for GrowingObjectQuery (Protocol 16) it is ***FindPartsResponse***).

The main message sequence for all query sub-protocols:

1. To query a store, the customer MUST send the ***Find****X* message to the store, which contains a query URI, which may contain a query string.
   a. For ETP-allowed OData filtering options, see Section **14.1.1**.
   b. For ETP-allowed OData pagination options, see Section **14.1.2**.
   c. For other filtering options available on specific messages, see those messages in the query sub-protocol-specific chapter.
2. For the resources, data objects or parts the store can successfully return, the store MUST send one or more ***Find****X****Response*** messages.
   a. A store MUST limit the total count of responses to the customer's value for the MaxResponseCount protocol capability.
   b. If the store exceeds the customer's MaxResponseCount value, the customer MAY send error ERESPONSECOUNT_EXCEEDED (30). (**NOTE:** Pagination options can be used to avoid this error; see Section **14.1.2**.)
   c. If a store's MaxResponseCount value is less than the customer's MaxResponseCount value, the store MAY further limit the total count of responses (to its value).
   d. If a store cannot return all responses to a request because it would exceed the lower or the customer's or the store's value for MaxResponseCount, the store MUST terminate the multipart message with error ERESPONSECOUNT_EXCEEDED (30).
      i. A store MUST NOT send RESPONSECOUNT_EXCEEDED (30) until it has sent MaxResponseCount responses.
   e. For a query URI that targets a specific resource (i.e., an object qualified by its UUID): if that resource is not present, the store MUST send error ENOT_FOUND (11). **EXAMPLE:** If a query specifies in Well ABC, find the channels whose activeStatus = true, and Well ABC (by its URI) cannot be found, the store MUST send error ENOT_FOUND (11).
   f. For a URI in a query protocol that targets a collection (i.e., the wells in the store or the wellbores of a specific well) or uses a $filter on a collection: if the URI exists (including any objects qualified by their UUID in the URI) but the collection or result after evaluating the $filter parameter is an empty collection, the store MUST return the ***Find****X****Response*** message with an empty array.

For this ETP Query Sub-Protocol…	The store MUST respond with this message…
DiscoveryQuery (Protocol 13)	A ***FindResourcesResponse*** with an empty array
StoreQuery (Protocol 14)	A ***FindDataObjectsResponse*** with an empty array
GrowingObjectQuery (Protocol 16)	A ***FindPartsResponse*** with an empty array

14.3.2 Usage Rules for Query Syntax with ETP Query Sub-Protocols

ETP Implementations MUST observe these rules:

1. OData queries are case sensitive; if you want case insensitive, you MUST use tolower and toupper functions.
2. If a query includes query options that are NOT supported by the store, the store MUST send error ENOTSUPPORTED (7).

3. Query results are always bound by a user's permissions for access to any endpoint.
4. This ETP Specification provides syntax and transport information only. For specific details (e.g., which fields are queryable, sort order, etc.) on how to query objects for a specific data model (e.g., WITSML, RESQML or PRODML), see that ML's ETP implementation specification.
5. When a canonical Energistics data object query URI includes a query string, the query string uses OData query syntax, for example:

```
eml:///witsml20.Channel?$filter=ChannelClass/Title eq 'Gamma'
```

   a. You can use an instance of a data object as a filter in a query. **EXAMPLE:** In Well ABC give me all the channels with activeStatus = true.
   b. The mapping from ETP to OData entity sets is that an entity set is all data objects of a particular type (e.g., well). The types are defined by the underlying ML.
   c. OData property paths used in filters are constructed by concatenating the (nested) element names together for the (sub)element used as a filter. In the filter in the above example, ChannelClass refers to the ChannelClass element on a Channel, which is a Data Object Reference, and Title refers to the Title element within the Data Object Reference.
   d. When an element is ComplexTypeWithSimpleContent (i.e., it is a value with associated attributes), then the property path for the value is ElementName/_.
   EXAMPLE: To filter a list of WITSML 2.0 Channels by the value of the NominalHoleSize element, a filter could look like this: NominalHoleSize/_ eq 8.5.
6. Filters are evaluated for the current URI data object type only.
   a. Data object references are NOT resolved or expanded.

14.3.3 Use of PWLS in Queries

Practical Well Log Standard (PWLS) is an industry standard stewarded by Energistics. It provides an industry-agreed list of logging tool classes and a hierarchy of measurement properties and applies all known mnemonics to them. For more information, see Section **3.12.7**.

1. If an ETP store supports the WITSML Channel data object, then it MUST support PropertyKind data objects (which are an implementation of PWLS).
2. Endpoints MUST be able to discover property kind data objects (to determine available property kinds) and use the returned property kinds in relevant Discovery, Store and Query operations.

14.4 Query Examples

This section contains some query examples. NOTE: Often there is more than one way to do these queries. Also, the results of the query may vary depending on the protocol (i.e., DiscoveryQuery vs. StoreQuery).

- Find all channels logged by Schlumberger

```
eml:///witsml20.Channel?$filter=LoggginqCompanyName eq 'SLB'
```

DiscoveryQuery: multipart message each of which contains the ID (a URI, a UUID, etc.) of a channel

store query: multipart message each of which contains the channel header of a channel (i.e., it's the data)

- Find all channels for a wellbore

```
eml:///witsml20.Channel?$filter=Wellbore/Title eq 'NO 15/9-1'
```

Discovery query: (same as above)

StoreQuery: same as above

Returns: dependent on the Protocol you used.

- Find all channels where channel class is Gamma

```
eml:///witsml20.Channel?$filter=ChannelClass/Title eq 'Gamma'
```

- Find channel sets where channel class is Gamma and hole size is 8.5 inches

```
eml:///witsml20.ChannelSet?$filter=ChannelClass/Title eq 'Gamma' and
NominalHoleSize/_ eq 8.5 and NominalHoleSize/uom eq 'in'
```

NOTE: In NominalHoleSize/_ the **/**_ convention is the Energistics proposed JSON convention for a complex type with simple content.

- Find channel sets where channel class is Gamma and run number is 5

```
eml:///witsml20.ChannelSet?$filter=ChannelClass/Title eq 'Gamma' and RunNumber
eq '5'
```

- Find channel sets where channel class is Surface and run number is 5

```
eml:///witsml20.ChannelSet?$filter=ChannelClass/Title eq 'Surface' and
RunNumber eq '5'
```

- Find channel where mnemonic is ROP and hole size is 12.25

```
eml:///witsml20.Channel?$filter=Mnemonic eq 'ROP' and NominalHoleSize/_ eq
12.25
```

- Find all channel sets having a channel where mnemonic is BDEP

```
eml:///witsml20.ChannelSet?$filter=Channel/any(c:c/Mnemonic eq 'BDEP')
```

- Find all rig utilization objects where rig name is Songa Endurance

```
eml:///witsml20.RigUtilization?$filter=Rig/Title eq 'Songa Endurance'
```

- Find all logs having an extension name value where name is TestValue

```
eml:///witsml20.Log?$filter=ExtensionNameValue/any(e:e/Name eq 'TestValue')
```

- Find all data objects where a particular data assurance rule has not been met

```
eml:///eml21.DataAssuranceRecord?$filter=PolicyName eq 'My Policy' and
FailingRules/any(r:r/RuleName eq 'My Rule')
```

15 DiscoveryQuery (Protocol 13)

ProtocolID: 13

Defined Roles: store, customer

DiscoveryQuery (Protocol 13) includes a message for querying resources in a store or server. Individual queries return a list of one type of resource that meet criteria specified in the request.

- The main discovery behavior is defined in Discovery (Protocol 3) (Chapter **8**).
- For general concepts and basic functionality related to query behavior in ETP, see Chapter **14 Overview of Query Behavior**.

This chapter includes main sections for:

- Key ETP concepts that are important to understanding how this protocol is intended to work (see Section **15.1**).
- Required behavior, which includes:
  - Required behavior in addition to the requirements in Chapter **14** (see Section **15.2**)**.**
  - Definitions of the endpoint and protocol capabilities used in this protocol (see Section **15.2.3**).
- Sample schemas of the messages defined in this protocol, which are identical to the Avro schemas published with this version of ETP. However, only the schema content in this specification includes documentation for each field in a schema (see Section **15.3**).

15.1 DiscoveryQuery: Key Concepts

For key concept relate to how queries work in general in ETP, see Chapter **14**.

15.1.1 Data Model as Graph

The request message in DiscoveryQuery (Protocol 13) has been developed to work with data models as graphs. When understood and used properly, this graph approach allows customers to specify precisely and in a single request the desired set of objects to monitor for notifications, thereby reducing traffic on the wire.

- For general definition of a graph, how it works, and key concepts and how they are used as inputs, see Section **8.1.1**.

15.2 DiscoveryQuery: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.
- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**

15.2.1 DiscoveryQuery: Message Sequence

For the basic message sequence that applies to all query protocols, see Section **14.3.1**.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart.

Message sent by customer	Response Message from store
FindResources (Section **15.3.1**) Request to find all resources that match the specified query criteria.	**FindResourcesResponse** (multipart) (Section **15.3.2**) Responses the store could return in response to the query.

15.2.2 DiscoveryQuery: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) additional rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter **5**. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol.

Row#	Requirement	Behavior
		2. For an explanation of endpoint, data object, and protocol capabilities, see Section 3.3. a. For the list of global capabilities and related behavior, see Section 3.3.2. 3. Section 15.2.3 identifies the capabilities most relevant to this ETP sub-protocol.
3.	Plural Messages (which includes maps)	1. This protocol uses plural messages. For detailed rules on handling plural messages (including ***ProtocolException*** handling), see Section 3.7.3.
4.	Rules specified in Chapter **14 Overview of Query Behavior**	1. The general rules and requirements specified in Chapter 14 MUST be observed and used with the additional details specific to DiscoveryQuery (Protocol 13) (which are specified in the next row.)
5.	Rules specific to DiscoveryQuery (Protocol◦13)	1. In the ***FindResourcesResponse*** message, the store MUST return only resources ONLY for the data object type specified in the request (i.e., queries in this protocol are limited to one type of data object only). **EXAMPLE:** The following URI returns only channels that meet the filter criteria. eml:///witsml20.Channel?$filter=ChannelClass/Title eq 'Gamma' a. Discovery (Protocol 3) returns multiple types of data objects. Because of how the OData syntax works, the customer MUST specify only one type of data object in the ***FindResources*** request message. 2. Query protocols specify a pagination option, which make it possible to get results in groups and prevent exceeding MaxResponseCount limits. For more information, see Section 14.1.2. 3. This protocol DOES NOT support querying for parts in a growing data object; to do that you MUST use GrowingObjectQuery (Protocol 16) (Chapter 17). 4. When a store navigates the data object graph to return ***Resource*** records in response to a ***FindResources*** request, it MUST respect the navigation direction and the navigable edge types specified in the *scope* and *context* fields in the request. a. When the URI in the *context* includes a query URI with a specific data object followed by a data object type, the navigation direction and the navigable edge types also apply to the relationships between the data object and the data object type specified in the URI. **EXAMPLE:** If the query URI is eml:///witsml20.Well(34aa7e1d-adb6-486b-9100-65412100d24e)/witsml21.Wellbore and the navigation direction is targets and the navigable edge types is primary, then the query should navigate all Wellbores that are sources of primary relationships with witsml20.Well(34aa7e1d-adb6-486b-9100-65412100d24e) as a target. In this example, the query should NOT navigate secondary relationships or primary relationships where the Wellbore is the target of a relationship where the well is the source.

15.2.3 DiscoveryQuery: Capabilities

The table below lists key capabilities for this protocol. The protocol capabilities are all defined here.

- For information on how to use the protocol capability, see Section 14.3.1.
- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section 3.3.

DiscoveryQuery (Protocol 13): Capabilities			
Name: Description	**Type**	**Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions and usage rules, see Section 3.3.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section 3.3.2. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section 4.3, **How a Client Establishes a WebSocket Connection to an ETP** Server.			
Data Object Capabilities (See Section 3.3.4)			
ActiveTimeoutPeriod: (This is also an endpoint capability.) The minimum time period in seconds that a store keeps the GrowingStatus for a growing data object or channel "active" after the last new part or data point resulting in a change to the data object's end index was added to the data object. This capability can be set for an endpoint and/or for a data object. A data object-specific value overrides an endpoint-specific value.	long	second <number of seconds>	**Default:** 3,600 **MIN:** 60 seconds
Protocol Capabilities			
MaxResponseCount: The maximum total count of responses allowed in a complete multipart message response to a single request.	long	count **Value units:** <count of responses>	**MIN:** 10,000

15.3 DiscoveryQuery: Message Schemas

This section provides a figure that displays all messages defined in DiscoveryQuery (Protocol 13). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.

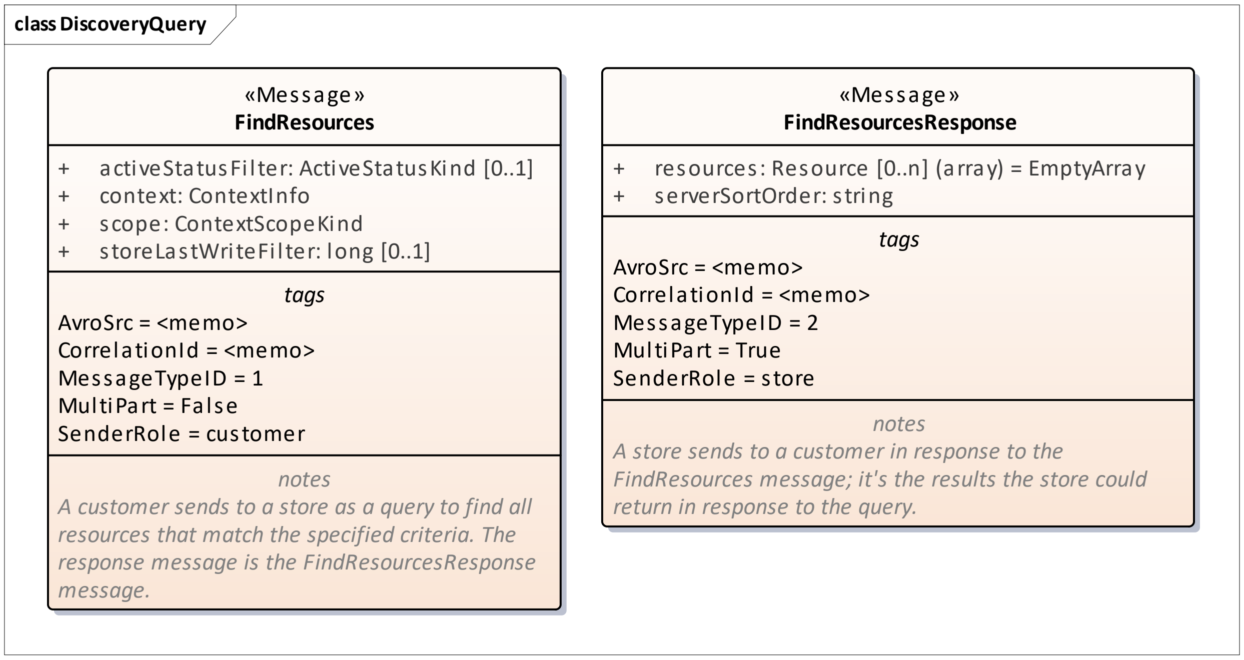

class DiscoveryQuery
«Message»
FindResources
+ activeStatusFilter: ActiveStatusKind [0..1]
+ context: ContextInfo
+ scope: ContextScopeKind
+ storeLastWriteFilter: long [0..1]
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1
MultiPart = False
SenderRole = customer
notes
A customer sends to a store as a query to find all resources that match the specified criteria. The response message is the FindResourcesResponse message.
«Message»
FindResourcesResponse
+ resources: Resource [0..n] (array) = EmptyArray
+ serverSortOrder: string
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 2
MultiPart = True
SenderRole = store
notes
A store sends to a customer in response to the FindResources message; it's the results the store could return in response to the query.

Figure 25: DiscoveryQuery: message schemas

15.3.1 Message: FindResources

A customer sends to a store as a query to find all resources that match the specified criteria. The response message is the FindResourcesResponse message.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
context	As defined in the ContextInfo record, which includes the URI of the data object query URI, what specific types of data objects are of interest, and how many "levels" of relationships in the model to discover, among others. It also includes the query parameters as specified in Chapter **14**. The URI MUST be a canonical Energistics data object query URI; for more information, see **Appendix: Energistics Identifiers**.	ContextInfo	1	1
scope	Scope is specified in reference to the URI (which is entered in the *context* field). It indicates which direction in the graph that the operation should proceed (targets or sources) and whether or not to include the starting point (self). The enumerated values to choose from are specified in ContextScopeKind. For definitions of targets and sources, see Section **8.1.1**.	ContextScopeKind	1	1

Field Name	Description	Data Type	Min	Max
storeLastWriteFilter	Use this to optionally filter the discovery on a date when the data object was *last written in a particular store*. The store returns resources whose storeLastWrite date/time is GREATER than the date/time specified in this filter field. Purpose of this field is part of the behavior for eventual consistency between 2 stores. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	0	1
activeStatusFilter	Use this to optionally filter the query for data objects that are currently "active" or "inactive" as defined in ActiveStatusKind. This field is for WITSML channels and growing data objects based on the value in the data object's GrowingStatus field, which may be: • active = A channel or growing data object is actively producing data points. • inactive = A channel or growing object is offline or not currently producing data points. The store returns resources for data objects whose GrowingStatus field matches the value specified in the activeStatusFilter.	ActiveStatusKind	0	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.DiscoveryQuery",
     "name": "FindResources",
     "protocol": "13",
     "messageType": "1",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "context", "type": "Energistics.Etp.v12.Datatypes.Object.ContextInfo" },
         { "name": "scope", "type": "Energistics.Etp.v12.Datatypes.Object.ContextScopeKind" },
         { "name": "storeLastWriteFilter", "type": ["null", "long"] },
         { "name": "activeStatusFilter", "type": ["null", 
"Energistics.Etp.v12.Datatypes.Object.ActiveStatusKind"] }
     ]
}
```

15.3.2 Message: FindResourcesResponse

A store sends to a consumer in response to the FindResources message; it's the results the store could return in response to the query.

Message Type ID: 2

Correlation Id Usage: MUST be set to the *messageId* of the ***FindResources*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
resources	The list of Resource records the store is returning.	Resource	0	n

Field Name	Description	Data Type	Min	Max
	The URIs MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.			
serverSortOrder	The deterministic sort order defined by the server.	string	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.DiscoveryQuery",
     "name": "FindResourcesResponse",
     "protocol": "13",
     "messageType": "2",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "resources",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.Object.Resource" }, "default": []
         },
         { "name": "serverSortOrder", "type": "string" }
     ]
}
```

16 StoreQuery (Protocol 14)

ProtocolID: 14

Defined Roles: store, customer

StoreQuery (Protocol 14) provides functionality for querying data objects in a store or server. This protocol can be used as alternative to Discovery (Protocol 3), to discover data objects based on specific filtering criteria of field within the data object. When the data object is found and the URI determined, it can then be used for operations in Store (Protocol 4).

- The main CRUD behavior for data objects is defined in Store (Protocol 4) (Chapter **9**).
- For general concepts and basic functionality related to query behavior in ETP, see Chapter **14 Overview of Query Behavior**.

This chapter includes main sections for:

- Key ETP concepts that are important to understanding how this protocol is intended to work (see Section **16.1**).
- Required behavior, which includes:
  - Required behavior in addition to the requirements in Chapter **14** (see Section **16.2**).
  - Definitions of the endpoint and protocol capabilities used in this protocol (see Section **16.2.3**).
- Sample schemas of the messages defined in this protocol, which are identical to the Avro schemas published with this version of ETP. However, only the schema content in this specification includes documentation for each field in a schema (see Section **16.3**).

16.1 StoreQuery: Key Concepts

For key concept relate to how queries work in general in ETP, see Chapter **14**.

16.1.1 Data Model as Graph

The request message in StoreQuery (Protocol 16) has been developed to work with data models as graphs. When understood and used properly, this graph approach allows customers to specify precisely and in a single request the desired set of objects to monitor for notifications, thereby reducing traffic on the wire.

- For general definition of a graph, how it works, and key concepts and how they are used as inputs, see Section **8.1.1**.

16.2 StoreQuery: Required Behavior

This section contains required behavior for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.
- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**

16.2.1 StoreQuery: Message Sequence

For the basic message sequence that applies to all query protocols, see Section **14.3.1**.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section.

Message sent by customer	Response Message from store
FindObjects (Section **16.3.1**) Request to find all data objects that match the specified criteria.	**FindObjectsResponse** (multipart) (Section **16.3.2**) The results the store could return in response to the query. **Chunk** (optional, multipart) (Section**16.3.3**) If the data object is too large to fit into the response message (exceeds the WebSocket message size), it MUST be subdivided into a set of ***Chunk*** messages.

16.2.2 StoreQuery: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) additional rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter **5**. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card.

Row#	Requirement	Behavior
		d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3**. a. For the list of global capabilities and related behavior, see Section **3.3.2**. 3. Section **16.2.3** identifies the capabilities most relevant to this ETP sub-protocol.
3.	Plural Messages (which includes maps)	1. This protocol uses plural messages. For detailed rules on handling plural messages (including ***ProtocolException*** handling), see Section **3.7.3**.
4.	Rules specified in Chapter **14 Overview of Query Behavior**	1. The general rules and requirements specified in Chapter 14 MUST be observed and used with the additional details specific to StoreQuery (Protocol 16) (which are specified in the next row.)
5.	Rules specific to StoreQuery (Protocol◦16)	1. In general, the results that are returned MUST follow the rules for Get operations in Store (Protocol 4). For details, see Section **9.2.2**. This includes behavior for: a. Oversized data objects and use the ***Chunk*** message (which is also defined in this protocol); for more information see Section **3.7.3.2**. b. Returning growing data objects and their parts. c. Returning container data objects and their contained data objects. d. All related capabilities behavior to these operations. e. **EXCEPTION:** Query protocols specify a pagination option, which make it possible to get results in groups and prevent exceeding MaxResponseCount limits. For more information, see Section **14.1.2**. 2. This protocol DOES NOT support querying for parts in a growing data object; to do that you MUST use GrowingObjectQuery (Protocol 16) (Chapter **17**).
6.	**Index Metadata:** General rules for channels, channel sets and growing data objects	1. A growing data object's index metadata MUST be consistent: a. All parts MUST have the same index unit and the same vertical datum. b. The index units and vertical datums in the growing data object header MUST match the parts. 2. A channel data object's index metadata MUST be consistent: a. The index units and vertical datums MUST match the channel's index metadata. 3. A channel set data object's index metadata MUST be consistent: a. The index units and vertical datums MUST match the channel set's index metadata. b. The channel set's index metadata MUST match the relevant index metadata in the channels it contains. 4. When sending messages, both the store AND the customer MUST ensure that all index metadata and data derived from index metadata are consistent in all fields in the message, including in XML or JSON object data or part data. a. **EXAMPLE:** The *uom* and *depthDatum* in an ***IndexInterval*** record MUST be consistent with the data object's index metadata. b. **EXAMPLE:** Data object elements related to index values in growing data object headers (e.g., MdMn and MdMx on a WITSML 2.0 Trajectory) and parts (e.g., Md on a WITSML 2.0 TrajectoryStation)

Row#	Requirement	Behavior
		MUST be consistent with each other AND the data object's index metadata.

16.2.3 StoreQuery: Capabilities

The table below lists key capabilities for this protocol. The protocol capabilities are all defined here. For this protocol, one particularly crucial endpoint capability is defined here.

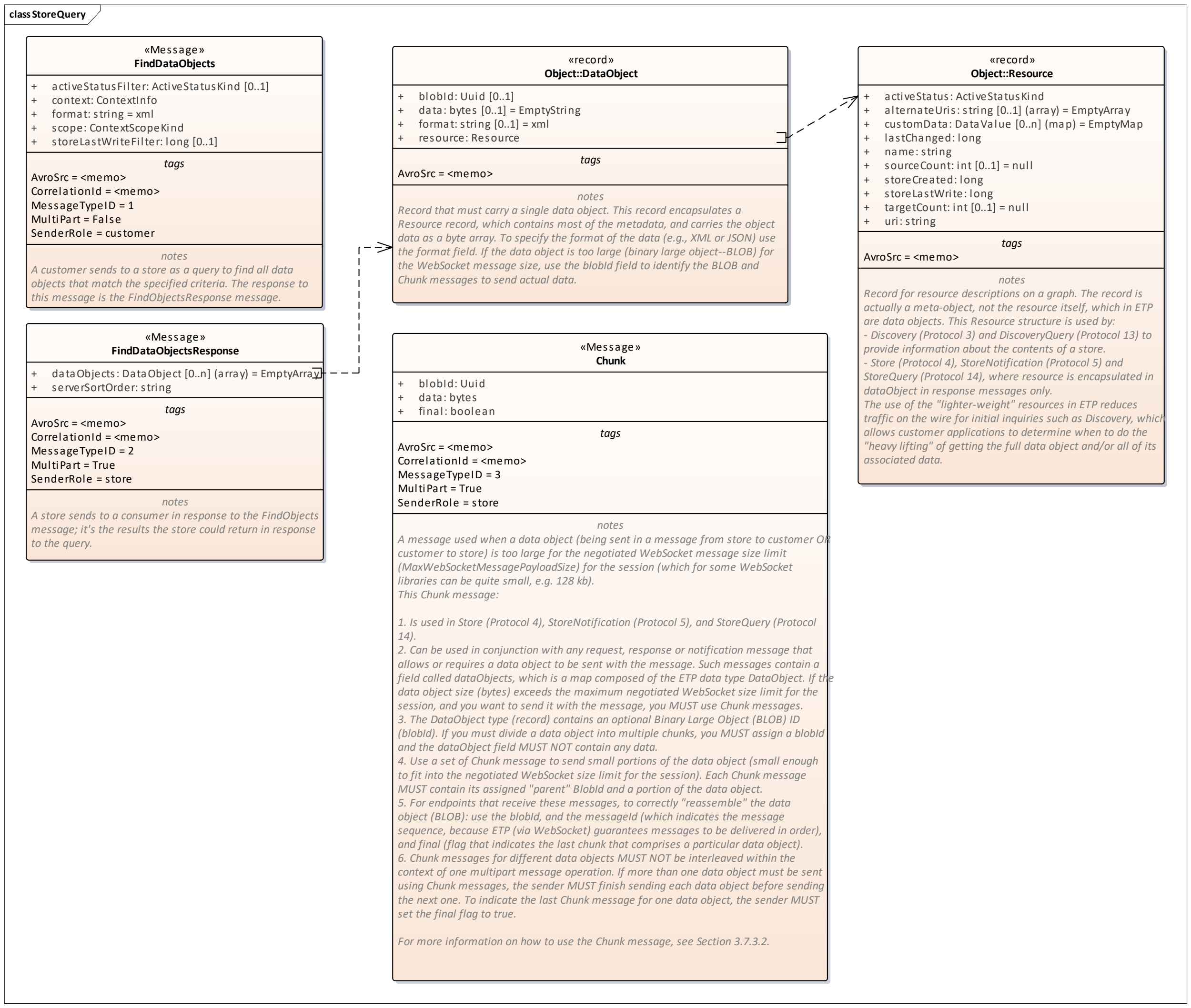
- For protocol-specific behavior relating to using these capabilities in this protocol, see◦Section◦**16.2.2,◦StoreQuery: General Requirements**.
- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section **3.3**.

StoreQuery (Protocol 14): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of each endpoint capability, see Section **3.3**.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section **3.3.2**. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP** Server.			
MaxPartSize: The maximum size in bytes of each data object part allowed in a standalone message or a complete multipart message. Size in bytes is the total size in bytes of the uncompressed string representation of the data object part in the format in which it is sent or received.	long	byte <number of bytes>	Min: 10,000 bytes
Data Object Capabilities			
SupportsGet For definitions and usage rules for this data object capability, see Section **3.3.4**.			
ActiveTimeoutPeriod: (This is also an endpoint capability.) The minimum time period in seconds that a store keeps the GrowingStatus for a growing data object or channel "active" after the last new part or data point resulting in a change to the data object's end index was added to the data object. This capability can be set for an endpoint and/or for a data object. A data object-specific value overrides an endpoint-specific value.	long	second <number of seconds>	**Default:** 3,600 **MIN:** 60 seconds
MaxContainedDataObjectCount: The maximum count of contained data objects allowed in a single instance of the data object type that the capability applies to.	long	Count <count of objects>	**MIN:** Should be specified per domain

StoreQuery (Protocol 14): Capabilities			
Name: Description	**Type**	**Units** **Value Units**	**Defaults and/or MIN/MAX**
EXAMPLE: If this capability is set to 2000 for a ChannelSet, then the ChannelSet may contain a maximum of 2000 Channels**.**			
Protocol Capabilities			
MaxDataObjectSize: (This is also an endpoint capability and a data object.) The maximum size in bytes of a data object allowed in a complete multipart message. Size in bytes is the size in bytes of the uncompressed string representation of the data object in the format in which it is sent or received. This capability can be set for an endpoint, a protocol, and/or a data object. If set for all three, here is how they generally work: • An object-specific value overrides an endpoint-specific value. • A protocol-specific value can further lower (but NOT raise) the limit for the protocol. **EXAMPLE:** A store may wish to generally support sending and receiving any data object that is one megabyte or less with the exceptions of Wells that are 100 kilobytes or less and Attachments that are 5 megabytes or less. A store may further wish to limit the size of any data object sent as part of a notification in StoreNotification (Protocol 5) to 256 kilobytes.	long	byte <number of bytes>	**MIN:** 100,000 bytes
MaxResponseCount: The maximum total count of responses allowed in a complete multipart message response to a single request.	long	count <count of responses>	**MIN:** 10,000

16.3 StoreQuery: Message Schemas

This section provides a figure that displays all messages defined in StoreQuery (Protocol 14). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.



A customer sends to a store as a query to find all data objects that match the specified criteria. The response to this message is the FindObjectsResponse message.

A store sends to a consumer in response to the FindObjects message; it's the results the store could return in response to the query.

Record that must carry a single data object. This record encapsulates a Resource record, which contains most of the metadata, and carries the object data as a byte array. To specify the format of the data (e.g., XML or JSON) use the format field. If the data object is too large (binary large object--BLOB) for the WebSocket message size, use the blobId field to identify the BLOB and Chunk messages to send actual data.

A message used when a data object (being sent in a message from store to customer OR customer to store) is too large for the negotiated WebSocket message size limit (MaxWebSocketMessagePayloadSize) for the session (which for some WebSocket libraries can be quite small, e.g. 128 kb).
This Chunk message:

1. Is used in Store (Protocol 4), StoreNotification (Protocol 5), and StoreQuery (Protocol 14).
2. Can be used in conjunction with any request, response or notification message that allows or requires a data object to be sent with the message. Such messages contain a field called dataObjects, which is a map composed of the ETP data type DataObject. If the data object size (bytes) exceeds the maximum negotiated WebSocket size limit for the session, and you want to send it with the message, you MUST use Chunk messages.
3. The DataObject type (record) contains an optional Binary Large Object (BLOB) ID (blobId). If you must divide a data object into multiple chunks, you MUST assign a blobId and the dataObject field MUST NOT contain any data.
4. Use a set of Chunk message to send small portions of the data object (small enough to fit into the negotiated WebSocket size limit for the session). Each Chunk message MUST contain its assigned "parent" BlobId and a portion of the data object.
5. For endpoints that receive these messages, to correctly "reassemble" the data object (BLOB): use the blobId, and the messageId (which indicates the message sequence, because ETP (via WebSocket) guarantees messages to be delivered in order), and final (flag that indicates the last chunk that comprises a particular data object).
6. Chunk messages for different data objects MUST NOT be interleaved within the context of one multipart message operation. If more than one data object must be sent using Chunk messages, the sender MUST finish sending each data object before sending the next one. To indicate the last Chunk message for one data object, the sender MUST set the final flag to true.

For more information on how to use the Chunk message, see Section 3.7.3.2.

Record for resource descriptions on a graph. The record is actually a meta-object, not the resource itself, which in ETP are data objects. This Resource structure is used by:
- Discovery (Protocol 3) and DiscoveryQuery (Protocol 13) to provide information about the contents of a store.
- Store (Protocol 4), StoreNotification (Protocol 5) and StoreQuery (Protocol 14), where resource is encapsulated in dataObject in response messages only.
The use of the "lighter-weight" resources in ETP reduces traffic on the wire for initial inquiries such as Discovery, which allows customer applications to determine when to do the "heavy lifting" of getting the full data object and/or all of its associated data.

Figure 26: StoreQuery: message schemas

16.3.1 Message: FindDataObjects

A customer sends to a store as a query to find all data objects that match the specified criteria. The response to this message is the FindObjectsResponse message.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
context	As defined in the ContextInfo record, which includes the data object query URI (for more information see Chapter **14**), what specific types of data objects are of	ContextInfo	1	1

Field Name	Description	Data Type	Min	Max
	interest, and how many "levels" of relationships in the model to discover. The URI MUST be a canonical Energistics data object query URI; for more information, see **Appendix: Energistics Identifiers**.			
scope	As defined in ContextScopeKind, which defines the extent of the query: For example: Are you interested only in the data object specified in the context URI (self)? Only sources of the data object? Only targets of data objects? Or some combination of these (sourcesOrSelf or targetsOrSelf)? For definitions of targets and sources, see Section **8.1.1**.	ContextScopeKind	1	1
storeLastWriteFilter	Use this to optionally filter the query on a date when the data object was *last written in a particular store*. The store returns resources whose storeLastWrite date/time is GREATER than the date/time specified in this filter field. Purpose of this field is part of the behavior for eventual consistency between 2 stores. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	0	1
activeStatusFilter	Use this to optionally filter the query for data objects that are currently "active" or "inactive" as defined in ActiveStatusKind. This field is for WITSML channels and growing data objects based on the value in the data object's GrowingStatus field, which may be: • active = A channel or growing data object is actively producing data points. • inactive = A channel or growing object is offline or not currently producing data points. The store returns resources for data objects whose GrowingStatus field matches the value specified in the activeStatusFilter.	ActiveStatusKind	0	1
format	Specifies the format (e.g., XML or JSON) in which you want to receive data for the returned data objects. This MUST be a format that was negotiated when establishing the session. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.	string	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.StoreQuery",
     "name": "FindDataObjects",
     "protocol": "14",
     "messageType": "1",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "context", "type": "Energistics.Etp.v12.Datatypes.Object.ContextInfo" },
         { "name": "scope", "type": "Energistics.Etp.v12.Datatypes.Object.ContextScopeKind" },
         { "name": "storeLastWriteFilter", "type": ["null", "long"] },
         { "name": "activeStatusFilter", "type": ["null", 
"Energistics.Etp.v12.Datatypes.Object.ActiveStatusKind"] },
         { "name": "format", "type": "string", "default": "xml" }
     ]
}
```

16.3.2 Message: FindDataObjectsResponse

A store sends to a consumer in response to the FindObjects message; it's the results the store could return in response to the query.

Message Type ID: 2

Correlation Id Usage: MUST be set to the *messageId* of the ***FindObjects*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
dataObjects	A list of data objects returned. For details on data sent for each data object listed, see DataObject record. The URIs in the ***Resource*** records MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.	DataObject	0	n
serverSortOrder	The deterministic sort order defined by the server.	string	1	1

Avro Source

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Protocol.StoreQuery",
    "name": "FindDataObjectsResponse",
    "protocol": "14",
    "messageType": "2",
    "senderRole": "store",
    "protocolRoles": "store,customer",
    "multipartFlag": true,
    "fields":
    [
        {
            "name": "dataObjects",
            "type": { "type": "array", "items": 
"Energistics.Etp.v12.Datatypes.Object.DataObject" }, "default": []
        },
        { "name": "serverSortOrder", "type": "string" }
    ]
}
```

16.3.3 Message: Chunk

A message used when a data object (being sent in a message from store to customer OR customer to store) is too large for the negotiated WebSocket message size limit (MaxWebSocketMessagePayloadSize) for the session (which for some WebSocket libraries can be quite small, e.g. 128 kb).

This ***Chunk*** message:

1. Is used in Store (Protocol 4), StoreNotification (Protocol 5), and StoreQuery (Protocol 14).
2. Can be used in conjunction with any request, response or notification message that allows or requires a data object to be sent with the message. Such messages contain a field called *dataObjects*, which

is a map composed of the ETP data type DataObject. If the data object size (bytes) exceeds the maximum negotiated WebSocket size limit for the session, and you want to send it with the message, you MUST use ***Chunk*** messages.

3. The ***DataObject*** type (record) contains an optional Binary Large Object (BLOB) ID (*blobId*). If you must divide a data object into multiple chunks, you MUST assign a *blobId* and the *dataObject* field MUST NOT contain any data.
4. Use a set of ***Chunk*** message to send small portions of the data object (small enough to fit into the negotiated WebSocket size limit for the session). Each ***Chunk*** message MUST contain its assigned "parent" BlobId and a portion of the data object.
5. For endpoints that receive these messages, to correctly "reassemble" the data object (BLOB): use the *blobId*, and the *messageId* (which indicates the message sequence, because ETP (via WebSocket) guarantees messages to be delivered in order), and *final* (flag that indicates the last chunk that comprises a particular data object).
6. ***Chunk*** messages for different data objects MUST NOT be interleaved within the context of one multipart message operation. If more than one data object must be sent using ***Chunk*** messages, the sender MUST finish sending each data object before sending the next one. To indicate the last ***Chunk*** message for one data object, the sender MUST set the *final* flag to true.

For more information on how to use the ***Chunk*** message, see Section **3.7.3.2**.

Message Type ID: 3

Correlation Id Usage: MUST be set to the *messageId* of the ***FindObjectsResponse*** message that resulted in the assignment of a *blobId* and this ***Chunk*** message being created.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
blobId	The BLOB ID assigned by an endpoint when a data object being sent in a request, response or notification message must be subdivided into multiple chunks. Each ***Chunk*** message that comprises a BLOB MUST contain the *blobId* of its "parent" BLOB. The *blobId*: • is entered in the DataObject record referenced in the *dataObjects* field of the request, response or notification message. • Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1
data	The data that comprises a chunk (portion) of the data object/BLOB.	bytes	1	1
final	Flag to indicate that this the final message of a set of ***Chunk*** messages that comprise one particular data object.	boolean	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.StoreQuery",
     "name": "Chunk",
     "protocol": "14",
     "messageType": "3",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         { "name": "blobId", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         { "name": "data", "type": "bytes" },
         { "name": "final", "type": "boolean" }
     ]
}
```

17 GrowingObjectQuery (Protocol 16)

ProtocolID: 16

Defined Roles: store, customer

GrowingObjectQuery (Protocol 16) includes functionality for querying parts of growing data objects in a store or server. The main CRUD behavior for parts of a growing data object is defined in GrowingObject (Protocol 6) (Chapter **11**).

For general concepts and basic functionality related to query behavior in ETP, see Chapter **14 Overview of Query Behavior**.

This chapter includes main sections for:

- Key ETP concepts that are important to understanding how this protocol is intended to work (see Section **17.1**).
- Required behavior, which includes:
  - Required behavior in addition to the requirements in Chapter **14** (Section **17.2**).
  - Definitions of the endpoint and protocol capabilities used in this protocol (Section **17.2.3**).
- Sample schemas of the messages defined in this protocol (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes documentation for each field in a schema (Section **17.3**).

17.1 GrowingObjectQuery: Key Concepts

- For key concepts related to growing data objects, including definition, design and how they work, see Section **11.1**.
- For key concept relate to how queries work in general in ETP, see Chapter **14**.

17.2 GrowingObjectQuery: Required Behavior

This section contains required behavior for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.
- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**

17.2.1 GrowingObjectQuery: Message Sequence

For the basic message sequence that applies to all query protocols, see Section **14.3.1**.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section.

Message sent by customer	Response Message from store
FindParts (Section **17.3.1**) Query request for parts of a growing data object that match the specified criteria.	**FindPartsResponse** (multipart) (Section **17.3.2**) The results the store could return in response to the query.

17.2.2 GrowingObjectQuery: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) additional rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter **5**. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3**. a. For the list of global capabilities and related behavior, see Section **3.3.2**.

Row#	Requirement	Behavior
		3. Section **17.2.3** identifies the capabilities most relevant to this ETP sub-protocol.
3.	Plural Messages (which includes maps)	1. This protocol uses plural messages. For detailed rules on handling plural messages (including ***ProtocolException*** handling), see Section **3.7.3**.
4.	Rules specified in Chapter **14 Overview of Query Behavior**	1. The general rules and requirements specified in Chapter 14 MUST be observed and used with the additional details specific to GrowingObjectQuery (Protocol 17) (which are specified in the next row.)
5.	Rules specific to GrowingObjectQuery (Protocol◦17)	1. The URI in the request message MUST be a canonical Energistics query that includes BOTH of these: a. A reference to a specific growing data object using the data object's qualified type and UUID. b. The qualified type of the data object's parts. c. **EXAMPLE:** To query a trajectory's stations, the URI could look like: eml:///witsml20.Trajectory(63b93219-e507-4934-a1b5-e7e550701934)/witsml20.TrajectoryStation?<query params> 2. The data object portion of the URI in the request message MUST resolve to a single growing data object; if not the store MUST send error ENOTGROWINGOBJECT (6001). 3. In general, the results that are returned MUST follow the rules for Get operations in GrowingObject (Protocol 6), including honoring limits specified by capabilities. For details, see Section **11.2.2.** a. **EXCEPTON:** Query protocols specify a pagination option, which make it possible to get results in groups and prevent exceeding MaxResponseCount limits. For more information, see Section **14.1.2**.
6.	Index Metadata	1. A growing data object's index metadata MUST be consistent: a. All parts MUST have the same index unit and the same vertical datum. b. The index units and vertical datums in the growing data header MUST match the parts. 2. When sending messages, both the store AND the customer MUST ensure that all index metadata and data derived from index metadata are consistent in all fields in the message, including in XML or JSON object data or part data. a. **EXAMPLE:** The *uom* and *depthDatum* in an ***IndexInterval*** record MUST be consistent with the channel's index metadata. b. **EXAMPLE:** Data object elements related to index values in growing data object headers (e.g., MdMn and MdMx on a WITSML 2.0 Trajectory) and parts (e.g., Md on a WITSML 2.0 TrajectoryStation) MUST be consistent with each other AND the data object's index metadata.

17.2.3 GrowingObjectQuery: Capabilities

The table below lists key capabilities for this protocol. The protocol capabilities are all defined here. For this protocol, one particularly crucial endpoint capability is defined here.

- For protocol-specific behavior relating to using these capabilities in this protocol, see◦Section◦**17.2.2,◦GrowingObjectQuery: General Requirements**.
- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section **3.3**.

GrowingObjectQuery (Protocol 16): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of each endpoint capability, see Section **3.3**.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section **3.3.2**. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP** Server.			
MaxPartSize: The maximum size in bytes of each data object part allowed in a standalone message or a complete multipart message. Size in bytes is the total size in bytes of the uncompressed string representation of the data object part in the format in which it is sent or received.	long	byte <number of bytes>	Min: 10,000 bytes
Data Object Capabilities (For definitions of each data object capability, see Section **3.3.4**.)			
ActiveTimeoutPeriod: (This is also an endpoint capability.) The minimum time period in seconds that a store keeps the GrowingStatus for a growing data object or channel "active" after the last new part or data point resulting in a change to the data object's end index was added to the data object. This capability can be set for an endpoint and/or for a data object. A data object-specific value overrides an endpoint-specific value.	long	second <number of seconds>	**Default:** 3,600 **MIN:** 60 seconds
SupportsGet For definitions and usage rules for each of these data object capabilities, see Section **3.3.4**.			
Protocol Capabilities			
MaxResponseCount: The maximum total count of responses allowed in a complete multipart message response to a single request.	long	count <count of responses>	**MIN:** 10,000

17.3 GrowingObjectQuery: Message Schemas

This section provides a figure that displays all messages defined in GrowingObjectQuery (Protocol 16). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.

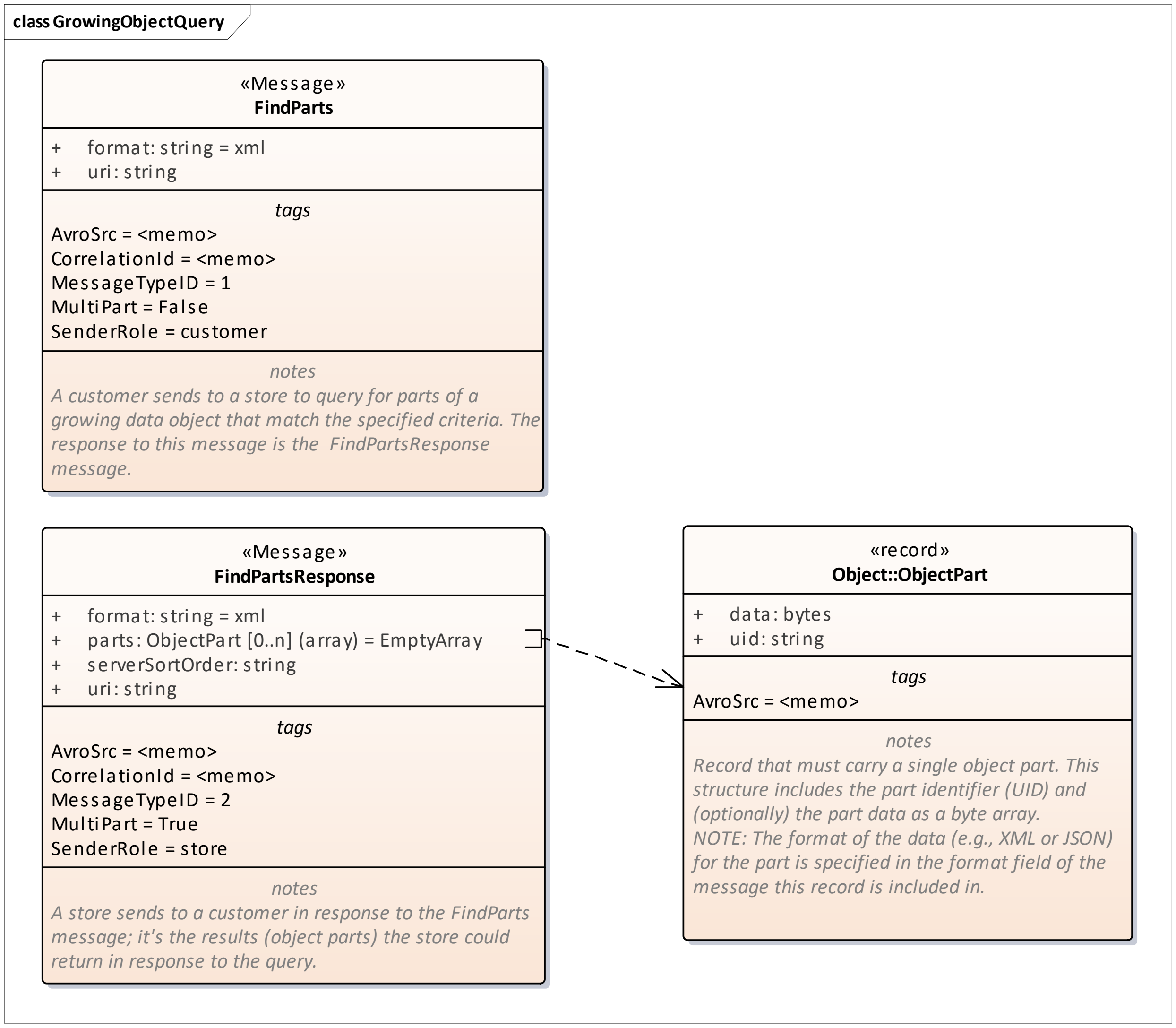

class GrowingObjectQuery
«Message»
FindParts
+ format: string = xml
+ uri: string
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1
MultiPart = False
SenderRole = customer
notes
A customer sends to a store to query for parts of a growing data object that match the specified criteria. The response to this message is the FindPartsResponse message.
«Message»
FindPartsResponse
+ format: string = xml
+ parts: ObjectPart [0..n] (array) = EmptyArray
+ serverSortOrder: string
+ uri: string
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 2
MultiPart = True
SenderRole = store
notes
A store sends to a customer in response to the FindParts message; it's the results (object parts) the store could return in response to the query.
«record»
Object::ObjectPart
+ data: bytes
+ uid: string
tags
AvroSrc = <memo>
notes
Record that must carry a single object part. This structure includes the part identifier (UID) and (optionally) the part data as a byte array.
NOTE: The format of the data (e.g., XML or JSON) for the part is specified in the format field of the message this record is included in.

Figure 27: GrowingObjectQuery: message schemas

17.3.1 Message: FindParts

A customer sends to a store to query for parts of a growing data object that match the specified criteria. The response to this message is the FindPartsResponse message.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uri	The data object query URI. The URI MUST identify both a specific growing data object AND the qualified type of the parts to query. For more information, see Chapter **14**. The URI MUST be a canonical Energistics data object query URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
format	Specifies the format (e.g., XML or JSON) in which you want to receive data for the return parts. This MUST be a format that was negotiated when establishing the session. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.	string	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObjectQuery",
     "name": "FindParts",
     "protocol": "16",
     "messageType": "1",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "format", "type": "string", "default": "xml" }
     ]
}
```

17.3.2 Message: FindPartsResponse

A store sends to a customer in response to the FindParts message; it's the results (object parts) the store could return in response to the query.

Message Type ID: 2

Correlation Id Usage: MUST be set to the *messageId* of the ***FindParts*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
uri	The URI that identifies the growing data object for which parts are being returned. The URI MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
serverSortOrder	The deterministic sort order defined by the server.	string	1	1
format	Specifies the format (e.g., XML or JSON) of the data for the parts being sent in this message. This MUST match the format in the FindParts request. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.	string	1	1

Field Name	Description	Data Type	Min	Max
parts	A list of the UIDs of the parts being returned in this response message and the data for each as defined in the ObjectPart record.	ObjectPart	0	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.GrowingObjectQuery",
     "name": "FindPartsResponse",
     "protocol": "16",
     "messageType": "2",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "serverSortOrder", "type": "string" },
         { "name": "format", "type": "string", "default": "xml" },
         {
             "name": "parts",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.Object.ObjectPart" }, "default": []
         }
     ]
}
```

18 Transaction (Protocol 18)

ProtocolID: 18

Defined Roles: store, customer

Transaction (Protocol 18) was developed to ensure store data consistency for applications that may have long, complex transactions (typically associated with earth modeling/RESQML). It establishes simple transaction semantics for ETP sessions. When implemented by a store, the Transaction protocol ensures that all "get" and "put" operations issued against a store, within the same transaction, refers to data in a consistent store state.

Even when a store supports Transaction (Protocol 18), the use of transactions is not required for all exchanges. However, when a customer application is pushing objects to a store and multiple ETP sub-protocols are required, use of a transaction is strongly recommended.

The messages in the Transaction protocol simply request that the store establish a "transaction", and the store returns a transaction UUID. When the customer determines its work is complete (which typically involves one or more of other ETP protocols), the customer informs the store it is finished by committing the transaction, which is identified by its UUID.

This protocol intentionally supports a single open transaction on a session. Additional messages in a later version may support multiple concurrent transactions, if the need arises. This protocol also has options to cancel (roll back) a transaction and indicate success/failure of a transaction.

The two use cases for which this protocol was developed are: 1) synchronization of servers and 2) movement of an earth model, both of which require moving multiple datatypes via multiple messages and maybe multiple protocols.

This chapter includes main sections for:

- Required behavior, which includes:
  - Description of the message sequence for main tasks, along with required behavior, ETP-defined capabilities, and possible errors (see Section **18.1.1**).
  - Other functional requirements (not covered in the message sequence) including use of ETP-defined endpoint and protocol capabilities for preventing and protecting against aberrant behavior (see Section **18.1.2**).
  - Definitions of the endpoint and protocol capabilities used in this protocol (see Section **18.1.3**).
- Sample schemas of the messages defined in this protocol, which are identical to the Avro schemas published with this version of ETP. However, only the schema content in this specification includes documentation for each field in a schema (see Section **18.2**).

18.1 Transaction: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.
- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**

18.1.1 Transaction: Message Sequence

This section explains the basic message sequence for main tasks to be done using this protocol and includes related key behaviors and possible errors; it assumes that an ETP session has been established using Core (Protocol 0) as described in Chapter **5**. The following General Requirements section provides additional requirements and rules for how this protocol works.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section (Section **18.2**).

Transaction (Protocol 18): Basic Message-Response flow by ETP Role	
Message from customer	**Response Message from store**
StartTransaction: A request to begin a transaction.	**StartTransactionResponse:** Response that the transaction was successfully started or failed.
RollbackTransaction: A request to cancel a transaction.	**RollbackTransactionResponse:** Response that the transaction rollback was successful or failed.
CommitTransaction: A request to commit a transaction.	**CommitTransactionResponse:** Response that the transaction was successfully committed or failed.

This section describes the basic sequence, related key behaviors, ETP-defined protocol capability usage and possible errors. By definition, Transaction (Protocol 18) involves work being done by other ETP protocols; for example, Store (Protocol 4) may be used to get or put a data object and DataArray (Protocol 9) may be used to get or put the related array data. The following Requirements section provides additional functional requirements and rules for how this protocol is intended to work.

18.1.1.1 To execute a transaction:

1. To initiate a transaction, the customer MUST send to the store the ***StartTransaction*** message (Section **18.2.1**).
   a. To increase efficiency, the customer MUST include the list of dataspaces that the transaction will cover.
      i. By default, the dataspace coverage includes only the default dataspace.
      ii. If the dataspace list is empty, the coverage extends to all dataspaces.
   b. Also, for server efficiency, it is recommended that a customer indicate if a transaction contains read-only Get requests.
   c. The customer MUST not exceed the store's value for MaxTransactionCount protocol capability. (**NOTE:** Currently ETP supports only 1 transaction.)
      i. If the customer attempts to start more than one transaction, the store MUST deny the request and send error EMAX_TRANSACTIONS_EXCEEDED (15).
2. The store MUST respond with the ***StartTransactionResponse*** message (Section **18.2.2**); the message contains:
   a. a UUID to identify and reference the transaction.
   b. a Boolean flag, to indicate the success or failure of initializing the transaction.
   c. the *failureReason* field, which must provide a brief reason the transaction was not started.

3. After receiving a successful ***StartTransactionResponse*** message, the customer MUST send a ***RollbackTransaction*** message (see Step 5 below) or ***CommitTransaction*** message (see Step 7 below) no later than the store's TransactionTimeoutPeriod protocol capability.
    a. If the customer exceeds this limit, the store MAY abort the transaction and send error ETIMED_OUT (26).
4. If the ***StartTransactionResponse*** message indicates success, the customer then begins the work it intends for this transaction, sending messages from the various protocols it needs to perform the work. **EXAMPLE:** It uses Store (Protocol 4) to put a data object and use DataArray (Protocol 9) to create and populate associated data arrays.
5. If a message/request fails for any reason, the customer can cancel the transaction by sending the ***RollbackTransaction*** message (Section **18.2.5)**.
6. On receipt of a ***RollbackTransaction*** message, the store MUST:
    a. Ignore the actions of all messages associated with the transaction. That is, the current transaction MUST NOT change the state of the store; the store MUST be in the same state as before the transaction began.
    b. Send a ***RollbackTransactionResponse*** message (Section **18.2.6**), which indicates the success/failure of the rollback and may provide an optional *failureReason*.
    c. If any data or requests are sent with the ***RollbackTransaction*** message, the store MUST ignore them.
7. If no rollback occurred and the customer has completed all requests/processes associated with the transaction, the customer MUST send to the store the ***CommitTransaction*** message (Section **18.2.3**).
8. When it receives the ***CommitTransaction*** message, the store MUST do ONE of the following:
    a. If the store IS NOT able to successfully deserialize the message OR the *transactionUuid* field does NOT identify a valid transaction, the store MUST send a non-map ***ProtocolException*** message with an appropriate error such as ENOT_FOUND (11).
        i. **NOTE:** For these types of general failures, the store MUST NOT send the ***CommitTransactionResponse*** message. In most cases, it is unlikely that the store would be able to send this message because it would be unable to determine the transaction.
    b. If the store IS able to successfully deserialize the message AND the *transactionUuid* field in the message identifies a valid and open transaction, the store MUST apply all requests/processes that the customer sent for the transaction and do ONE of the following:
        i. If the application of ALL requests/processes is successful, the store MUST send the customer the ***CommitTransactionResponse*** message (Section **18.2.4**), with the *successful* flag set to true (indicating the transaction was successful).
        ii. If the store cannot successfully apply ALL requests/processes that the customer sent for the transaction:
            1. The store MUST send the customer the ***CommitTransactionResponse*** message, with the *successful* flag set to false (indicating the transaction failed).The store MUST include a human readable *failureReason* that explains why or how the transaction failed.
            2. The store is expected to be in a state where all requests associated with the transaction have been ignored.
9. Before a customer ends a session, it SHOULD wait until it receives the ***CommitTransactionResponse*** message, to ensure the transaction is successful.

18.1.2 Transaction: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) additional rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter **5**. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3**. a. For the list of global capabilities and related behavior, see Section **3.3.2**. 3. Section**18.1.3** identifies the capabilities most relevant to this ETP sub-protocol. Additional details for how to use the protocol capabilities are included below in this table and in Section **18.1.1 Transaction: Message Sequence**.
3.	Message Sequence See Section **18.1.1**.	1. The Message Sequence section above (Section **18.1.1**) describes requirements for the main tasks listed there and also defines required behavior.

Row#	Requirement	Behavior
4.	Support for Transaction (Protocol 18)	1. To execute a transaction in ETP, both endpoints in a session MUST support this protocol. 2. Support for this protocol indicates support of transactions.
5.	Get transactions	1. For "get" transactions: the store MUST provide data corresponding to the same store state for the transaction data. If one or several data involved in the transaction is modified inside the store in the middle of a transaction, the server MUST return all data involved inside the transaction at a state either before or after those modifications.
6.	Put transactions	1. For "put" transactions: the store MUST NOT actually complete the store state modifications corresponding to the transaction until the entire transaction is complete.
7.	Conflicting updates	If a store operation on a data object in one session fails due to that data object being involved in a transaction in another session, the store MUST send error ETIMED_OUT (26) as the response to the failed operation.

18.1.3 Transaction: Capabilities

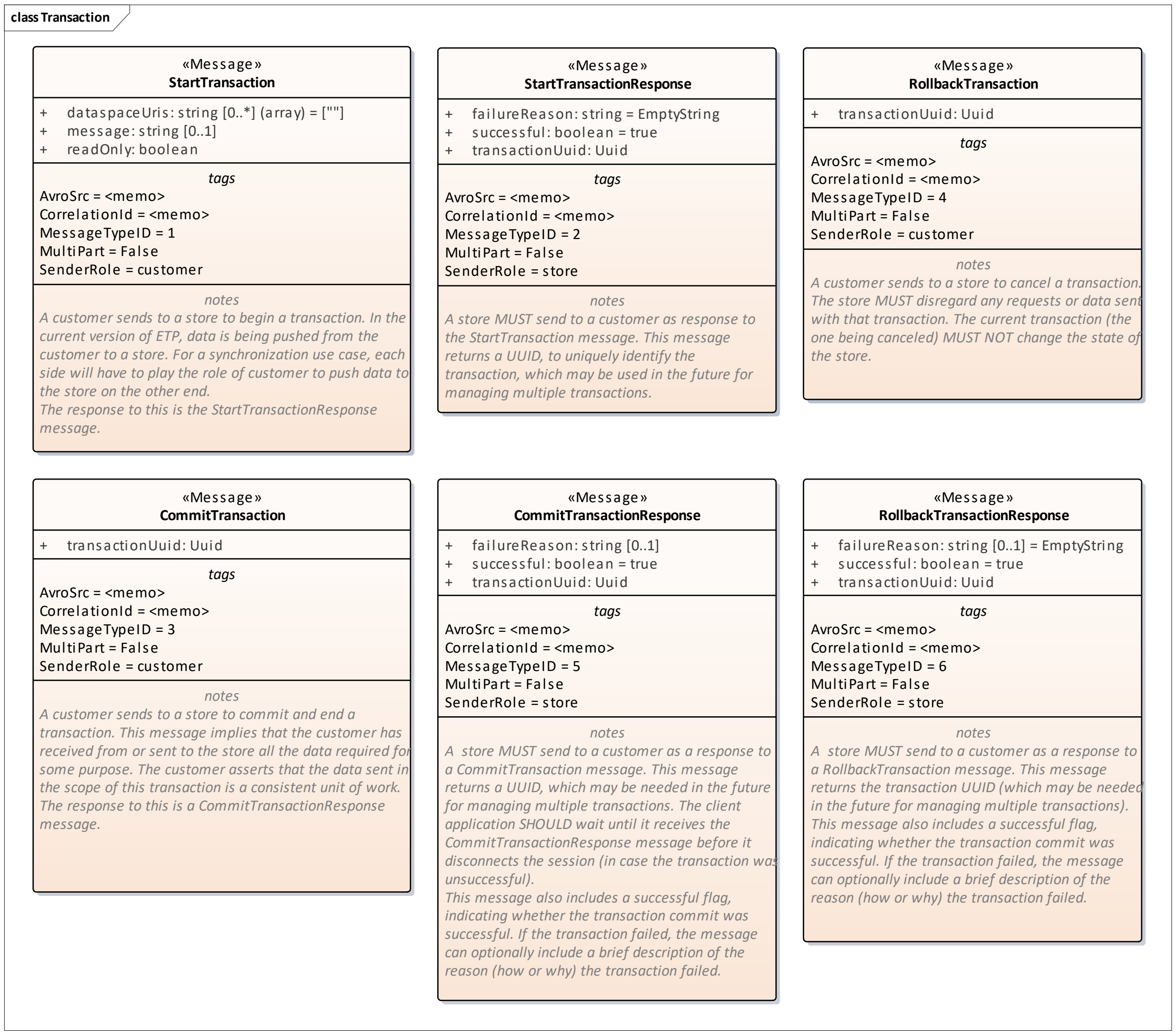
The table below lists key capabilities for this protocol. The protocol capabilities are all defined here.

- For protocol-specific behavior relating to using these capabilities in this protocol, see◦Section◦**18.1.2**,◦**Transaction: General Requirements**.
- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section **3.3**.

Transaction (Protocol 18): Capabilities			
Name: Description	**Type**	**Units** **Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of each endpoint capability, see Section **3.3**.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section **3.3.2**. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP** Server.			
Protocol Capabilities			
MaxTransactionCount: The maximum count of transactions allowed in parallel in a session.	long	count <count of transactions>	**MIN:** 1 Max: 1 Default: 1
TransactionTimeoutPeriod: The maximum time period in seconds allowed between receiving a ***StartTransactionResponse*** message and sending the corresponding ***CommitTransaction*** or ***RollbackTransaction*** request.	long	seconds <number of seconds>	**MIN:** 5

18.2 Transaction: Message Schemas

This section provides a figure that displays all messages defined in Transaction (Protocol 18). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.



«Message»
StartTransaction

+ dataspaceUris: string [0..*] (array) = [""]
+ message: string [0..1]
+ readOnly: boolean

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to begin a transaction. In the current version of ETP, data is being pushed from the customer to a store. For a synchronization use case, each side will have to play the role of customer to push data to the store on the other end.
The response to this is the StartTransactionResponse message.

«Message»
StartTransactionResponse

+ failureReason: string = EmptyString
+ successful: boolean = true
+ transactionUuid: Uuid

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 2
MultiPart = False
SenderRole = store

notes
A store MUST send to a customer as response to the StartTransaction message. This message returns a UUID, to uniquely identify the transaction, which may be used in the future for managing multiple transactions.

«Message»
RollbackTransaction

+ transactionUuid: Uuid

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 4
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to cancel a transaction. The store MUST disregard any requests or data sent with that transaction. The current transaction (the one being canceled) MUST NOT change the state of the store.

«Message»
CommitTransaction

+ transactionUuid: Uuid

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 3
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to commit and end a transaction. This message implies that the customer has received from or sent to the store all the data required for some purpose. The customer asserts that the data sent in the scope of this transaction is a consistent unit of work. The response to this is a CommitTransactionResponse message.

«Message»
CommitTransactionResponse

+ failureReason: string [0..1]
+ successful: boolean = true
+ transactionUuid: Uuid

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 5
MultiPart = False
SenderRole = store

notes
A store MUST send to a customer as a response to a CommitTransaction message. This message returns a UUID, which may be needed in the future for managing multiple transactions. The client application SHOULD wait until it receives the CommitTransactionResponse message before it disconnects the session (in case the transaction was unsuccessful).
This message also includes a successful flag, indicating whether the transaction commit was successful. If the transaction failed, the message can optionally include a brief description of the reason (how or why) the transaction failed.

«Message»
RollbackTransactionResponse

+ failureReason: string [0..1] = EmptyString
+ successful: boolean = true
+ transactionUuid: Uuid

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 6
MultiPart = False
SenderRole = store

notes
A store MUST send to a customer as a response to a RollbackTransaction message. This message returns the transaction UUID (which may be needed in the future for managing multiple transactions).
This message also includes a successful flag, indicating whether the transaction commit was successful. If the transaction failed, the message can optionally include a brief description of the reason (how or why) the transaction failed.

Figure 28: Transaction: message schemas

18.2.1 Message: StartTransaction

A customer sends to a store to begin a transaction. In the current version of ETP, data is being pushed from the customer to a store. For a synchronization use case, each side will have to play the role of customer to push data to the store on the other end.

The response to this is the StartTransactionResponse message.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
readOnly	Indicates that the request in the transaction is read-only (i.e., "get" messages).	boolean	1	1
message	Provides an optional message indicating the reason for the transaction.	string	0	1
dataspaceUris	Indicates the dataspaces involved in the transaction. • An empty STRING means the default dataspace. • An empty LIST means all dataspaces. If both endpoints support alternate URIs for the session, these MAY be alternate dataspace URIs. Otherwise, they MUST be canonical Energistics dataspace URIs. For more information, see **Appendix: Energistics Identifiers**.	string	0	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Transaction",
     "name": "StartTransaction",
     "protocol": "18",
     "messageType": "1",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "readOnly", "type": "boolean" },
         { "name": "message", "type": "string", "default": "" },
         {
             "name": "dataspaceUris",
             "type": { "type": "array", "items": "string" }, "default": [""]
         }
     ]
}
```

18.2.2 Message: StartTransactionResponse

A store MUST send to a customer as response to the StartTransaction message. This message returns a UUID, to uniquely identify the transaction, which may be used in the future for managing multiple transactions.

Message Type ID: 2

Correlation Id Usage: MUST be set to the *messageId* of the ***StartTransaction*** message that this message is a response to.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
transactionUuid	The UUID that the store assigns to the transaction. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1
successful	Boolean flag indicating that the store successfully started the transaction. Default = true (success)	boolean	1	1
failureReason	If the successful flag is "0" (false), provide a brief reason why the transaction failed to start.	string	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Transaction",
     "name": "StartTransactionResponse",
     "protocol": "18",
     "messageType": "2",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "transactionUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         { "name": "successful", "type": "boolean", "default": true },
         { "name": "failureReason", "type": "string", "default": "" }
     ]
}
```

18.2.3 Message: CommitTransaction

A customer sends to a store to commit and end a transaction. This message implies that the customer has received from or sent to the store all the data required for some purpose. The customer asserts that the data sent in the scope of this transaction is a consistent unit of work.

The response to this is a CommitTransactionResponse message.

Message Type ID: 3

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
transactionUuid	The UUID of the transaction (assigned by the store in the StartTransactionResponse message) that the customer wants to commit. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Transaction",
     "name": "CommitTransaction",
     "protocol": "18",
     "messageType": "3",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "transactionUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" }
     ]
}
```

18.2.4 Message: CommitTransactionResponse

A store MUST send to a customer as a response to a CommitTransaction message. This message returns a UUID, which may be needed in the future for managing multiple transactions. The client application SHOULD wait until it receives the CommitTransactionResponse message before it disconnects the session (in case the transaction was unsuccessful).

This message also includes a successful flag, indicating whether the transaction commit was successful. If the transaction failed, the message can optionally include a brief description of the reason (how or why) the transaction failed.

Message Type ID: 5

Correlation Id Usage: MUST be set to the *messageId* of the ***CommitTransaction*** message that this message is a response to.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
transactionUuid	The UUID (that the store assigns to the transaction) of the transaction that has been committed. Must be of type ***Uuid*** (Section 23.6).	Uuid	1	1
successful	Boolean flag indicating that the store successfully started the transaction. Default = true (success)	boolean	1	1
failureReason	An optional description from the store to the customer explaining why or how the transaction failed.	string	0	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Transaction",
     "name": "CommitTransactionResponse",
     "protocol": "18",
     "messageType": "5",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "transactionUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         { "name": "successful", "type": "boolean", "default": true },
         { "name": "failureReason", "type": "string", "default": "" }
     ]
}
```

18.2.5 Message: RollbackTransaction

A customer sends to a store to cancel a transaction. The store MUST disregard any requests or data sent with that transaction. The current transaction (the one being canceled) MUST NOT change the state of the store.

Message Type ID: 4

Correlation Id Usage: Because this message uses a UUID to identify the transaction, the *correlationId* is not used. It MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
transactionUuid	UUID of the transaction (assigned by the store in the StartTransactionResponse message) that is to be canceled/"rolled back". Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Transaction",
     "name": "RollbackTransaction",
     "protocol": "18",
     "messageType": "4",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "transactionUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" }
     ]
}
```

18.2.6 Message: RollbackTransactionResponse

A store MUST send to a customer as a response to a RollbackTransaction message. This message returns the transaction UUID (which may be needed in the future for managing multiple transactions).

This message also includes a successful flag, indicating whether the transaction commit was successful. If the transaction failed, the message can optionally include a brief description of the reason (how or why) the transaction failed.

Message Type ID: 6

Correlation Id Usage: MUST be set to the *messageId* of the ***RollbackTransaction*** message that this message is a response to.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
transactionUuid	The UUID (that the store assigns to the transaction) of the transaction that is being rolled back. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1
successful	A flag that indicates the success or failure of the transaction. Default = true (success)	boolean	1	1
failureReason	An optional description from the store to the customer explaining why or how the rollback failed.	string	0	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Transaction",
     "name": "RollbackTransactionResponse",
     "protocol": "18",
     "messageType": "6",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "transactionUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         { "name": "successful", "type": "boolean", "default": true },
         { "name": "failureReason", "type": "string", "default": "" }
     ]
}
```

19 ChannelSubscribe (Protocol 21)

ProtocolID: 21

Defined Roles: store, customer

ChannelSubscribe (Protocol 21) provides a publish/subscribe mechanism so that an endpoint with the ETP customer role can connect to an endpoint with the ETP store role, discover the store's active channels (using Discovery (Protocol 3)), and subscribe to specific channels to receive new data as soon as they become available (i.e., realtime streaming). In the context of the subscription, the store also sends data "edits" (replaced ranges and truncated channels) to previously sent data. A customer can also use this protocol to request a range of data.

This protocol also includes functionality for customers to reconnect after an unintended disconnect, and "catch up" on the changes during the disconnect—without having to re-stream an entire channel.

Some key points about this and related protocols:

- **Protocol 21** has the "get/read" behavior for channel data from a store and to "listen" for changes in channel data that require a notification (or data updates) to be sent while connected.
- **ChannelDataLoad (Protocol 22)** (see Chapter **20)** has the "put/write" behavior for channel data. Protocol 22 "pushes" data from the customer role endpoint to the store role endpoint.

Other protocols that "stream" channel data:

- **ChannelStreaming (Protocol 1)** (see Chapter **6**) is for very simple streaming of channel-oriented data, from "simple" producers (i.e., a sensor) to a consumer; it is designed to replace WITS data transfers. Protocol 1 allows a consumer to connect to a producer and receive whatever data the producer has (i.e., the consumer cannot discover available channels nor specify which channels it wants, etc. like in Protocol 21). **NOTE:** Beginning in ETP v1.2, Protocol 1 is used only for these so-called simple streamers (in previous versions of ETP, Protocol 1 included all channel streaming behavior).
- **ChannelDataFrame (Protocol 2)** (see Chapter **7**) allows a customer endpoint to get channel data from a store in a row-orientated 'frame' or 'table' of data. In oil and gas jargon, the general use case that Protocol 2 supports is typically referred to as getting a "historical log". (In ETP jargon you are actually getting a frame of data from a ChannelSet data object; for more information, see Section **7.1.1**).

Other ETP sub-protocols or information that may be used related for Channel data objects:

- **Store (Protocol 4)**. Because a Channel is an Energistics data object, to add, update or delete Channel data◦objects from a store, use Store (Protocol 4) (see Chapter **9**).
- **StoreNotification (Protocol 5)**. To subscribe to notifications about Channel data objects (e.g., channels added, deleted, status change, etc.), use StoreNotification (Protocol 5) (see Chapter **10**).
- For more information about high-level workflows for data replication and outage recovery, see **Appendix: Data Replication and Outage Recovery Workflows**.

NOTE: Energistics data models (e.g., WITSML) allow channels to be grouped into channel sets and logs. However, ETP channel streaming protocols handle individual channels; that is, whether or not the channel is part of a channel set or log is irrelevant to how it is handled in a channel streaming protocol. (For more information about channel sets, see Section **7.1.1**.)

This chapter includes main sections for:

- Key ETP concepts that are important to understanding how this protocol is intended to work (see Section **19.1**.
- Required behavior, which includes:

- – Description of the message sequence for main tasks, along with required behavior, use of capabilities, and possible errors (see Section **19.2.1**).
- – Other functional requirements (not covered in the message sequence) including use of endpoint, data object, and protocol capabilities for preventing and protecting against aberrant behavior (see Section **19.2.2**).
- – Definitions of the endpoint, data object, and protocol capabilities used in this protocol (see Section **19.2.3**).

- Sample schemas of the messages defined in this protocol (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes documentation for each field in a schema (see Section **19.3**).

19.1 ChannelSubscribe: Key Concepts

- For definitions and key concept related to channels and channel streaming, see Section **6.1**.
- For the definition of change annotations and how they work, see Section **11.1.4**.

19.2 Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.
- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**
- A customer has the Energistics URIs for each channel it is interested in.
  - – Most likely this list will be determined using Discovery (Protocol 3); see Chapter **8**. However, customers may also receive the URIs out of band of ETP. **NOTE:** If you are interested in a case such as subscribing to "all the channels in a particular wellbore" you MUST use Discovery (Protocol 3) to find all of those channels (and their respective URIs).
  - – For information about Energistics URIs, see **Appendix: Energistics Identifiers**.

19.2.1 ChannelSubscribe: Message Sequence

This section explains the basic message sequence for main tasks to be done using this protocol and includes related key behaviors including usage of capabilities and possible errors.

The following General Requirements section provides additional requirements and rules for how this protocol works (ones that don't fit neatly into a message sequence).

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section.

ChannelSubscribe (Protocol 21): Basic Message-Response flow by ETP Role	
Message from customer	**Response/Message from store**
GetChannelMetatdata: Request for metadata for a list of channels.	***GetChannelMetatdataResponse*** (multipart): Lists the metadata data for each channel in the request that it can respond to.
SubscribeChannels: Request to receive channel data messages (see list in next 3 rows) for specific channels.	***SubscribeChannelsResponse*** (multipart): Success-only response to confirm "subscriptions" that the store successfully created.
	ChannelData: Contains the data the store has for each channel; the store keeps sending these messages as new data becomes available for as long as the customer is connected and subscribed.
	ChannelsTruncated: Sent to notify a customer that the end indexes of channels that the customer is subscribed to have been reset and streaming may resume from the new end indexes; used to notify a customer that “index jump” errors have been corrected.
	RangeReplaced (multipart): Sent when a range of data has been replaced for channels that a customer is subscribed to. Includes the affected range and any replacement data.
GetRanges: A request for a specific range of data on one or more channels.	***GetRangesResponse*** (multipart): The data points within a specified range.
CancelGetRanges: A request to stop sending data for a previous GetRanges request.	***GetRangesResponse*** (multipart): With the FIN bit set. It may be an empty (no data) message.
GetChangeAnnotations: A request for changes to a specified list of channels since a specific time.	***GetChangeAnnotationsResponse*** (multipart): The list of channels and changed intervals, per the request.
UnsubscribeChannels: A request to cancel subscriptions (unsubscribe) to one or more channels and to discontinue streaming data for these channels.	***SubscriptionsStopped*** (multipart), which has 2 use cases: • Response to ***UnsubscribeChannels*** message. • Sent by a store as a notification (i.e., without a customer request) to stop previous subscriptions.

19.2.1.1 To do the initial setup to subscribe to channels and be streamed data as it is available:

1. The customer MUST send a store a ***GetChannelMetadata*** message (Section 19.3.1).
    a. The ***GetChannelMetadata*** message contains a map whose values MUST each be the URI of a channel that the customer wants to get channel metadata for.
        i. To find a particular set of channels and their respective URIs (e.g., all the channels in a particular wellbore) the customer MAY use Discovery (Protocol 3). The customer may also receive these URIs out of band of ETP.
    b. Before doing any other operations defined by other messages in this protocol, the customer MUST first get the metadata for each channel.

2. For the URIs that it successfully returns channel metadata for, the store MUST send one or more ***GetChannelMetadataResponse*** map response messages (Section **19.3.211.3.2**), which contains a map whose values are ***ChannelMetadataRecord*** records (Section **23.33.7**).
    a. For more information on how map response messages work, see Section **3.7.3**.
    b. ***ChannelMetadataRecord*** has the necessary contextual information (indexes, units of measure, etc.) that the customer needs to correctly interpret channel data.
    c. The store MUST assign the channel an integer identifier that is unique for the session in this protocol. This identifier is used instead of the channel URI to identify the channel in subsequent messages in this protocol for the session. This identifier is set in the *id* field in the ***ChannelMetadataRecord****.*
    RECOMMENDATION: Use the smallest available integer value for a new channel identifier.
    IMPORTANT: If the channel is deleted and recreated during a session, it MUST be assigned a new identifier.
3. For the URIs it does NOT return channel metadata for, the store MUST send one or more map ***ProtocolException*** messages, where values in the *errors* field (a map) are appropriate errors.
    a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.
    b. For requested channels that the store does not contain, send error ENOT_FOUND (11).
4. Based on the list in the ***GetChannelMetadataResponse*** message, the customer determines which channels it wants the store to stream data for.
    a. **NOTE:** For each channel, the customer compares its latest channel index with the latest channel index returned by the store.
    b. Before setting up subscriptions, a customer MAY want to do a ***GetRanges*** operation (see Section **19.2.1.3**) and then set up the subscription to receive future changes, when they occur. OR the customer can choose to set up a subscription and start streaming from a specified index as describe in this step below.
5. To subscribe to (i.e., to have the store stream data for) one or more of these channels, the customer MUST send to the store the ***SubscribeChannels*** message (Section **19.3.3**), which contains a map whose values MUST each be a ***ChannelSubscribeInfo*** record for a channel the customer wants to be streamed.
    a. A customer MUST limit the total count of channels concurrently open for streaming in the current ETP session to the store's value for MaxStreamingChannelsSessionCount protocol capability.
    b. For each channel that would exceed the store's limit, the store MUST NOT start streaming for the channel. The store MUST instead send ELIMIT_EXCEEDED (12).

    The ***ChannelSubscribeInfo*** record (Section **23.33.9**) contains these data fields (for each channel):

    c. The *channelId* (this is the *id* returned on the ***ChannelMetadataRecord***).
    d. One of the following fields MAY be populated:
        i. For *startIndex*, the customer MUST specify an index value that it wants the store to start streaming from. (The customer may determine this based on information in the ***GetChannelMetadataResponse*** message and the index that it currently has.) If *requestLatestIndexCount* is null AND *startIndex* is NOT null:
            1. **For increasing data:** If *startIndex* is greater than the channel's end index or, for decreasing data, less than the channel's end index, the store MUST deny that request with EINVALID_OPERATION (32).
            2. **Otherwise:** The store MUST start streaming from the first channel index that, for increasing data, is greater than or equal to the requested start index and, for decreasing

data, is less than or equal to the requested start index.

ii. For *requestLatestIndexCount*, the customer MUST specify a non-negative integer, *n,* which is the number of points back from "now" (the most recent data from the store) to start streaming from. If this property is provided, i.e., not null:

1. The store MUST ignore *startIndex*.
2. The store MUST stream the latest *n* values from the channel and MUST continue streaming per the subscription.
3. If *n* is negative, the store MUST deny that request and send error EINVALID_ARGUMENT (5).
4. If *n* is 0, the store MUST NOT send any existing data; the store MUST only stream new data.
5. If *n* is greater than the number of data points in the channel (which may be 0 for empty channels), the store MUST stream all available data points in the channel and MUST continue streaming per the subscription.

iii. If *requestLatestIndexCount* is null AND *startIndex* has a null value, then the store MUST NOT send any existing data. The store MUST only stream new data.

iv. If both *startIndex* AND *requestLatestIndexCount* are null, the store MUST start streaming as if *requestLatestIndexCount* is set to 0.

e. If the customer DOES NOT want to receive updates/historical changes (in addition to realtime streaming data) it MUST set the *dataChanges* flag to false. Historical changes are sent in ***RangeReplaced*** messages, which are explained below in this chapter.

6. In response to a customer's ***SubscribeChannels*** message, the store MUST do the following:

a. For the channels it successfully starts streaming for, the store MUST respond with a one or more ***SubscribeChannelsResponse*** map response messages (Section **19.3.4**), which list the channels the store has started streaming.

i. For more information on how map response messages work, see Section **3.7.3**.

b. For the channels it did NOT start streaming for, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors.

i. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.

ii. If a requested channel is already being streamed, send error EINVALID_STATE (8).

c. Start sending ***ChannelData*** messages (Section **19.3.5**), which contain the data for the channels that the customer subscribed to and the store can stream data for, beginning the customer's requested starting point (per Step **5**).

i. For information on index order of data in ***ChannelData*** messages, see Section **19.2.2**, Row **7**.

ii. If the customer has requested an "old" index (e.g., one the store initially produced 1 hour ago), the store MUST start streaming from that index (i.e., sending ***ChannelData*** messages from that index) and continue streaming. That is, the notions of "historical" or "new" are irrelevant: the store MUST simply start streaming from the index the customer specified and continue.

d. AND, if the *dataChanges* flag is set to true (default), start sending ***RangeReplaced*** messages (Section **19.3.7**) (as changes occur).

i. For information on how the ***RangeReplaced*** message/operations work, see Section **19.2.2**, Row **9**.

e. AND to correct "index jump errors" (a frequently occurring error when collecting data in oilfield operations), a store MUST send ***ChannelsTruncated*** messages (Section **19.3.6**).

i. For information on how the ***ChannelsTruncated*** message works, see Section **19.2.2**, Row **10**.

7. The store MUST continue sending ***ChannelData*** messages and ***ChannelsTruncated*** messages for the subscribed channels, for the life of the subscription, and if the customer requested data changes the store MUST also send ***RangeReplaced*** messages.
8. If a customer sends a ***ProtocolException*** message in response to a ***ChannelData***, ***ChannelsTruncated***, or ***RangeReplaced*** message, the store MAY attempt to take corrective action but the store MUST NOT terminate the associated channel subscriptions.
9. The store MAY need to stop sending data (e.g., a channel that a customer is subscribed to is deleted or permission for access to a channel is revoked).
   a. To stop sending data, the store MUST send the customer a ***SubscriptionsStopped*** message (Section **19.3.9**), which is a map of the channels being stopped. After sending ***SubscriptionsStopped*** for the channel, the store MUST NOT send any ***ChannelData***, ***RangeReplaced***, or ***ChannelsTruncated*** messages for a channel**.**

19.2.1.2 For the customer to "unsubscribe" from streaming data:

1. The customer MUST send the store the ***UnsubscribeChannels*** message (Section **19.3.8**), which is a map whose values MUST each be the channel ID of a channel the customer no longer wants to receive streaming data for.
2. For the channels it successfully stopped streaming for, the store MUST respond with a one or more ***SubscriptionsStopped*** map response messages (Section **19.3.9**), which list the IDs of the channels the store has stopped streaming.
   a. For more information on how map response messages work, see Section **3.7.3**.
   b. The store MUST NOT send any ***ChannelData***, ***RangeReplaced***, or ***ChannelsTruncated*** messages for a channel after sending ***SubscriptionsStopped*** for the channel**.**
3. For the channels it did NOT stop streaming for, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors.
   a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.
   b. If a requested channel is not being streamed, send error EINVALID_STATE (8).
4. After a customer has canceled a subscription, the store MUST NOT restart it.
   a. If the customer wants to restart the subscription, it MUST attempt to do so by sending a new ***SubscribeChannels*** message with the channel's *channelId* as described in Section **19.2.1.1**.

19.2.1.3 To request a range of data:

In addition to subscribing to a channel, a customer has the option of simply getting one or more ranges (intervals) of data from one or more channels. The customer can request multiple ranges for multiple channels. This request is a "get" operation, NOT a subscription.

Use cases supported by this request include: populating a graph or chart; or, when reconnecting after a dropped connection, using it to "catch up" on missed data.

1. The customer and store must exchange ***GetChannelMetadata*** and ***GetChannelMetadataResponse*** messages as describe in Section **19.2.1.1** (Steps 1 and 2).
2. The customer MUST send to the store a ***GetRanges*** message (Section **19.3.10**), which specifies the channels of interest and the index range or ranges (interval) for each channel.
   a. The ***GetRanges*** message contains an array of ***ChannelRangeInfo*** records, which specifies the

details for each range request.

i. All channels in the request MUST have a common index type, unit, direction and, for depth data, datum.

ii. A channel MUST NOT be included more than once in a single ***GetRanges*** message. That is, ***GetRanges*** may not be used to request multiple ranges from the same channel.

iii. Though this message may contain multiple range requests, from multiple channels, the responses are NOT aligned in any way (for example, like they are in ChannelDataFrame (Protocol 2)).

iv. For this operation, ETP supports requesting and filtering on secondary indexes. For more information, see Section **19.2.2**, Row **8**.

b. The customer MUST also assign a UUID to the request (*requestUuid*), which may be used later to cancel the range request.

c. A customer MUST limit the count of channels in a single range request to the store's value for the MaxRangeChannelCount protocol capability.

   i. If the customer request exceeds this limit, the store MUST deny requests that exceed this and send error ELIMIT_EXCEEDED (12).

3. The store MUST be able to return ALL data for ALL channels in the request or it MUST deny the entire request with an appropriate error code. **EXAMPLE:** If the request includes a channel ID for a channel that has been deleted, the entire request must be denied with EINVALID_CHANNELID (1002).

   a. Multiple ***ProtocolException*** messages MUST NOT be sent, and the map part of a ***ProtocolException*** message (i.e., the *errors* field) MUST NOT be used.

      i. This exception may occur after some data has already been sent, but a store SHOULD make a best effort to determine if there will be any errors as early as possible, preferably before sending any data.

   b. **NOTE:** Including channels with no data in the requested interval is NOT an error.

4. If the store successfully returns data from the request interval(s), it MUST send one or more ***GetRangesResponse*** messages (Section **19.3.11**).

   a. Each ***GetRangesResponse*** message is an array of ***DataItem*** records (see Section **23.33.5**), each of which contains the channel data for the request interval(s).

      i. A store MUST limit the count of ***DataItem*** records in the complete multipart range message to the customer's value for the MaxRangeDataItemCount protocol capability.

      ii. The customer MAY notify the store of responses that exceed this limit by sending error ERESPONSECOUNT_EXCEEDED (30).

      iii. A store MAY further limit the total count of ***DataItem*** records to its value for MaxRangeDataItemCount protocol capability, if it is smaller than the customer's value.

      iv. If sending additional ***DataItem*** records would exceed the limit, the store MUST terminate the response with ERESPONSECOUNT_EXCEEDED (30). The store MUST NOT terminate the response until it has sent MaxRangeDataItemCount ***DataItem*** records.

      v. For information on index order of data in ***GetRangesResponse*** messages, see Section **19.2.2**, Row **7**. It is up to the store how it sends data in the response message, e.g., whether it sends channel-based or row-based data in the response. **EXAMPLE:** If the customer asks for 50 channels, the store MAY send the entirety of each channel OR a "row" of the data available for all channels at one index. A store MAY also change how it is sending data, depending on the request and what is most "optimized" for that request and other requests it may be processing.

5. If the store has no data in any of the request intervals, it MUST send a ***GetRangesResponse*** message with the FIN bit set and the *data* field set to an empty array.
6. If the store does NOT successfully return data or a ***GetRangesResponse*** with an empty *data* array, it MUST send a non-map ***ProtocolException*** message with an appropriate error, such as EREQUEST_DENIED (6).
7. To cancel a ***GetRanges*** operation, the customer MUST send to the store a ***CancelGetRanges*** message (Section **19.3.12**), which identifies the UUID of the request to be stopped.
    a. If the store has not already finished responding to the request that is being canceled, the store MUST:
        i. Send a final ***GetRangesResponse*** message with the FIN bit set; this final message MAY be empty (no data).
        ii. Stop sending ***GetRangesResponse*** messages for the specified channels.

19.2.1.4 To reconnect and resume streaming when the session has been interrupted (using ChangeAnnotations):

This process can be used by a customer anytime it connects to a store; that is, when it first connects to a store and wants to determine the latest changes or after an unintended disconnect when it wants to determine data it may have missed.

ETP has no session survivability. If the session is interrupted (e.g., a satellite connection drops), using this process makes it easier for a customer to determine what has changed while disconnected, get any changed data it requires, and resume operations that were in process when the session dropped—and do that with the reduced likelihood of NOT having to "resend all data from the beginning" (i.e., all data from before the session dropped). For more information about related workflows, see **Appendix: Data Replication and Outage Recovery Workflows**.

1. The customer MUST reconnect to the store and re-create and re-authorize (if required) the ETP session (as described in Section **4.3** and **5.2.1.1**) and get channel metadata using the process described in Section **19.2.1.1** (Steps 1 and 2). (**REMINDER:** The ***GetChannelMetadataResponse*** message is where the store assigns channel IDs; these channel IDs are used in the messages for the remaining steps below).
2. The customer MUST "re-subscribe" to desired channels as described in Section **19.2.1.1** (Steps 3–5).
    a. For each channel, a customer should compare the end index it has for a channel with the end index it receives in the ***GetChannelMetadataResponse*** message to determine the index it wants the store to start streaming from.
    b. Recommended best practice is to re-subscribe to channels first (before getting change annotations), so you start receiving current data and related change notifications as soon as possible.
3. To determine what has changed while disconnected, the customer MUST send the store a ***GetChangeAnnotations*** message (Section **19.3.13**).
    a. This message contains a map whose values MUST each be a ***ChannelChangeRequestInfo*** record (Section **23.33.15**) where the customer specifies the list of channels and for each, the "changes since" time (that is, the customer wants all changes since this time, which should be based on the time the customer was last sure it received data from the store). In the message, the customer MUST also indicate if it wants all change annotations or only the latest change annotation for each channel.
        i. The "changes since" time (*sinceChangeTime* field) MUST BE equal to or more recent than the store's ChangeRetentionPeriod endpoint capability.
4. For ***ChannelChangeRequestInfo*** records it successfully returns change annotations for, the store MUST respond with one or more ***GetChangeAnnotationsResponse*** map response messages

(Section **19.3.14**).

   a. For more information on how map response messages work, see Section **3.7.3**.

   b. The map values in each message are ***ChangeResponseInfo*** records (Section **23.34.19**), which contains a time stamp for when the response was sent and the ***ChangeAnnotation*** records (Section **23.34.18**) for the channels in a ***ChannelChangeRequestInfo*** record.

      i. Each ***ChangeAnnotation*** record contains a timestamp for when the change occurred in the store and the interval of the channel that changed. (**NOTE:** Change annotations keep track ONLY of the interval that change, NOT the actual data that changed).

   c. For information about how the store tracks and manages these change annotations, see Section **19.2.2**, Row **12**.

5. For ***ChannelChangeRequestInfo*** records it does NOT successfully send change annotations for, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors.

   a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.

6. Based on information in the ***GetChangeAnnotationsResponse*** message, the customer MAY:

   a. Use the ***GetRanges*** message to retrieve intervals of interest that have changed (as described in Section **19.2.1.3**).

19.2.2 ChannelSubscribe: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) some rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter **5**. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session.

Row#	Requirement	Behavior
		c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol. **2.** For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3.** **a.** For the list of global capabilities and related behavior, see Section **3.3.2**. 3. Section **19.2.3** identifies the capabilities most relevant to this ETP sub-protocol. If one or more of the defined capabilities is presented by an endpoint, the other endpoint in the ETP session MUST accept it (them) and process the value, and apply them to the behavior as specified in this document. a. Additional details for using these capabilities are included in relevant rows of this table and Section **19.2.1 ChannelSubscribe: Message Sequence.**
3.	**Message Sequence for main tasks in this protocol:** See Section **19.2.1.**	1. The Message Sequence section above (Section **19.2.1**) describes requirements for the main tasks listed there and also defines required behavior.
4.	If a customer wants to be able to "reconnect and catch up on what happened while disconnected" after unintended outage, it MUST track this information during an ETP session	1. ETP supports workflows and provides mechanisms to help customers to more easily recover missed data (i.e., easier than "re-stream the entire channel") after reconnecting from an unintended outage. a. For more information, see **Appendix: Data Replication and Outage Recovery Workflows**. 2. If a customer wants to use these workflows, the customer role endpoint MUST do the following: a. When the customer first subscribes to a channel, it MUST get the most recent ***ChangeAnnotation*** record for each channel (if there are any). i. When reconnecting after an outage, the customer MUST get all ***ChangeAnnotation*** records. b. In the subscription request for each channel (specifically, in the ***ChannelSubscribeInfo*** record), it MUST set the *dataChanges* field to true, which means the store MUST send ***RangeReplaced*** messages with data changes. c. During the session, the customer MUST track the most recently received index value for each channel it is subscribed to. d. For more information about change annotations, see Row **12.**
5.	Plural messages (which includes maps)	1. This protocol uses plural messages, which includes maps. For detailed rules on handling plural messages (including ***ProtocolException*** handling), see Section **3.7.3**.
6.	To get notifications of changes to the channel itself (not the data in the channel) or new channels	1. A channel is a data object: as such adding, updating, and deleting channels is done using Store (Protocol 4). a. **NOTE:** "Updating" means updates to the channel data object itself—which DOES NOT include the data points in a channel, which is done using ChannelStreaming (Protocol 1) and ChannelDataLoad (Protocol 22). For more information, see Section **6.1.1**. 2. To receive notifications of changes, such as new channels added, changes in channel status (active or inactive), or a channel has been

Row#	Requirement	Behavior
		deleted, a customer MUST subscribe to changes to an appropriate context/scope using StoreNotification (Protocol 5) for changes that occur in the store. For more information, see: a. Section **9.2.2** (for Store (Protocol 4) b. Section **10.2.2** (for StoreNotification (Protocol 5) 3. Based on the information a customer receives from store notifications (e.g., a new channel was added) the customer can determine necessary actions that it may require in ChannelSubscribe (Protocol 21). **EXAMPLE**: When a customer receives notification that a new channel has been added, it may then subscribe to receive data for that new channel.
7.	**Store Behavior:** Data order for channel subscriptions and range responses	1. Streaming data points (in ***ChannelData*** messages) MUST be sent in primary index order for each channel, both within one message and across multiple messages. 2. Data points in ***GetRangesResponse*** and ***RangeReplaced*** messages MUST be sent in primary index order for each channel, both within a single message and across all messages within a multipart message (response or notification). 3. Primary index order is always as appropriate for the index direction of a channel (i.e., increasing or decreasing). 4. The index values for each data point are in the same order as their corresponding ***IndexMetadataRecord*** records in the corresponding channel's ***ChannelMetadataRecord*** record, and the primary index is always first. 5. The same primary index value MUST NOT appear more than once for the same channel in any ***ChannelData*** message UNLESS the channel data at that index was affected by a truncate operation during the session (i.e., a ***ChannelsTruncated*** message was received for the channel with a range that covered the primary index value). 6. The same primary index value MUST NOT appear more than once for the same channel in the same multipart ***GetRangesResponse*** or ***RangeReplaced*** message.
8.	**Secondary indexes in range operations**	1. For ***GetRanges*** operations, ETP provides support for additional filtering on secondary indexes/intervals. a. Support of secondary indexes is considered advanced functionality and is optional. 2. If an endpoint supports filtering on secondary indexes, it MUST set the SupportsSecondaryIndexFiltering protocol capability to true. a. If a store's SupportsSecondaryIndexFiltering protocol capability is false and a customer requests that data be filtered by secondary index values, then the Store MUST deny the request and send error ENOTSUPPORTED (7). b. ETP provides an optional field on the ***IndexMetadataRecord*** (Section **23.33.6**) named *filterable*, which allows a store to specify if a particular index can be filtered on in various request messages in some ETP sub-protocols. 3. Results with secondary indexes are highly variable depending on the specifics of the data and the indexes. **EXAMPLE:** Results based on secondary index filtering may result in no data values at some secondary indexes or multiple data values at some secondary indexes (e.g., a wireline tool where time is the primary index is time may result in multiple depth readings). 4. A customer specifies the secondary intervals that it wants to filter on in the ***ChannelRangeInfo*** record, which is used by the ***GetRanges*** message.
9.	Notifying that a range of data in a channel has been updated or deleted (When sending a ***RangeReplaced*** message to a customer)	1. The behavior described in this row assumes that a customer has subscribed to the channel as described in Section **19.2.1.1**. 2. When a range of data in a channel that a customer is subscribed to has been updated or deleted, a store MUST do the following: a. A store MUST send a ***RangeReplaced*** message with details about the change to the customer for the channel.

Row#	Requirement	Behavior
		i. If the customer set *dataChanges* = true when subscribing to the channel, then the store MUST send ***RangeReplaced*** to subscribed customers whether or not they have previously been sent the original data. ii. The *changedInterval* field MUST represent the full range of data affected by the change. b. If the channel data interval for the change includes the end index of the channel, the store MUST send a ***ChannelsTruncated*** message to notify the customer that the end index of the channel has been reset. i. The ***ChannelsTruncated*** message MUST reset the end index for each affected channel to its updated end index after the data changes were applied. ii. If the customer set *dataChanges* = true in the subscription, the store MUST send ***RangeReplaced*** first and send ***ChannelsTruncated*** second. The store MUST NOT send any other ChannelSubscribe (Protocol 21) messages affecting the channel between ***RangeReplaced*** and ***ChannelsTruncated***. iii. If the customer set *dataChanges* = false in the subscription, the store MUST ONLY send ***ChannelsTruncated***. It MUST NOT send ***RangeReplaced***. 3. A store MUST send all ***DataItem*** records for a particular change to a customer. a. A store MUST limit the count of ***DataItem*** records in a complete multipart range message to the customer's value for the MaxRangeDataItemCount protocol capability. b. The customer MAY notify the store of responses that exceed this limit by sending error ELIMIT_EXCEEDED (12). c. A store MAY further limit the total count of ***DataItem*** records to its value for MaxRangeDataItemCount protocol capability, if it is smaller than the customer's value. d. If a store is unable to send all ***DataItem*** records for a particular change in a single ***RangeReplaced*** multipart range message because doing so would exceed the limit, the store MUST split data for the change into more than one multipart ***RangeReplaced*** range message. When doing so, the store MUST ensure the *changedInterval* and replacement data are consistent in each, separate multipart message. **EXAMPLE:** MaxRangeDataItemCount is 1,000,000. The data for a change is 2,500,000 ***DataItem*** records. The store may divide this data into three separate multipart ***RangeReplaced*** messages: two with 1,000,000 ***DataItem*** records and one with 500,000 records. Each of the three multipart messages consists of an initial ***RangeReplaced*** message with *correlationId* set to 0 followed by zero or more additional ***RangeReplaced*** messages that correlate back to the first one in the multipart message. 4. ***RangeReplaced*** is a notification of an atomic 'delete and replace' operation. 5. The store MUST send ***RangeReplaced*** messages in primary index order. a. The same primary index value MUST NOT appear more than once for the same channel in a ***RangeReplaced*** multipart notification. 6. All data items in the *data* field of a ***RangeReplaced*** message MUST conform to these rules: a. The *channelIds* of each ***DataItem*** MUST match one of the channels listed in the *channelIds* field in the first message. i. The customer MAY ignore any ***DataItem*** where the channelId does not match. b. The *index* of each ***DataItem*** MUST be between (inclusive) of the *startIndex* and *endIndex* defined in the *changedInterval* field. i. The customer MAY ignore any data that falls outside of these bounds.

Row#	Requirement	Behavior
		7. To notify a customer of a range that was deleted without any replacement data, the store MUST send a ***RangeReplaced*** message and simply leave the *data* field empty (i.e., nothing to insert).
10.	Notifying that channel end indexes have been reset (***ChannelsTruncated*** message)	1. To notify a customer that the end indexes of channels have been reset (e.g., to correct for an "index jump" error) a store MUST send a ***ChannelsTruncated*** message, which contains an array of ***TruncateInfo*** records; each record contains the channel ID and its new end index. 2. On receipt of a ***ChannelsTruncated*** message, the customer MUST take note of the new end index of the channel. 3. When the store resumes sending ***ChannelData*** messages (for the channel whose index it corrected), the primary index for each new ***DataItem*** MUST be greater than (for increasing data) or less than (for decreasing data) the new end index in the ***ChannelsTruncated*** message.
11.	Ending subscriptions	1. A store MUST end a customer's channel subscription when: a. The customer cancels the subscription by sending an ***UnsubscribeChannels*** message. b. The channel for the subscription is deleted. c. The customer loses access to the channel. 2. When ending a subscription: a. The store MAY discard any queued data or notifications for the channel. b. The store MUST send a ***SubscriptionsStopped*** message either as a response to a customer's ***UnsubscribeChannels*** request or as a notification. c. The store MUST include a human readable reason why the subscription(s) were ended in the ***SubscriptionsStopped*** message. 3. After sending the ***SubscriptionsStopped*** message, the store MUST NOT send any further data or notifications for the channel until and unless the subscription is later restarted. 4. After a subscription has ended: a. A customer MAY request that the subscription be restarted by sending a new ***SubscribeChannels*** message with the channel's *channelId*. b. A store MUST NOT restart the subscription without a request from the customer.
12.	Detecting changes to channel data	1. For a definition of change annotations and related terms, see Section **11.1.4**. **2.** For the requirements on stores create and manage change annotations for channel data objects, see Sections **11.2.2.2**, **11.2.2.4**, and **11.2.2.5**. 3. For the main message sequence for reconnecting after an outage, see Section **19.2.1.4**.
13.	Index Metadata	1. A channel data object's index metadata MUST be consistent: a. The index units and vertical datums MUST match the channel's index metadata. 2. When sending messages, both the store AND the customer MUST ensure that all index metadata and data derived from index metadata are consistent in all fields in the message, including in XML or JSON object data or part data. a. **EXAMPLE:** The *uom* and *depthDatum* in an ***IndexInterval*** record MUST be consistent with the channel's index metadata. b. A store MUST reject requests with inconsistent index metadata with an appropriate error such as EINVALID_OBJECT (14) or EINVALID_ARGUMENT (5).

19.2.3 ChannelSubscribe: Capabilities

The table below lists key capabilities for this protocol. The protocol capabilities are all defined here. For this protocol, one particularly crucial endpoint capability is defined here.

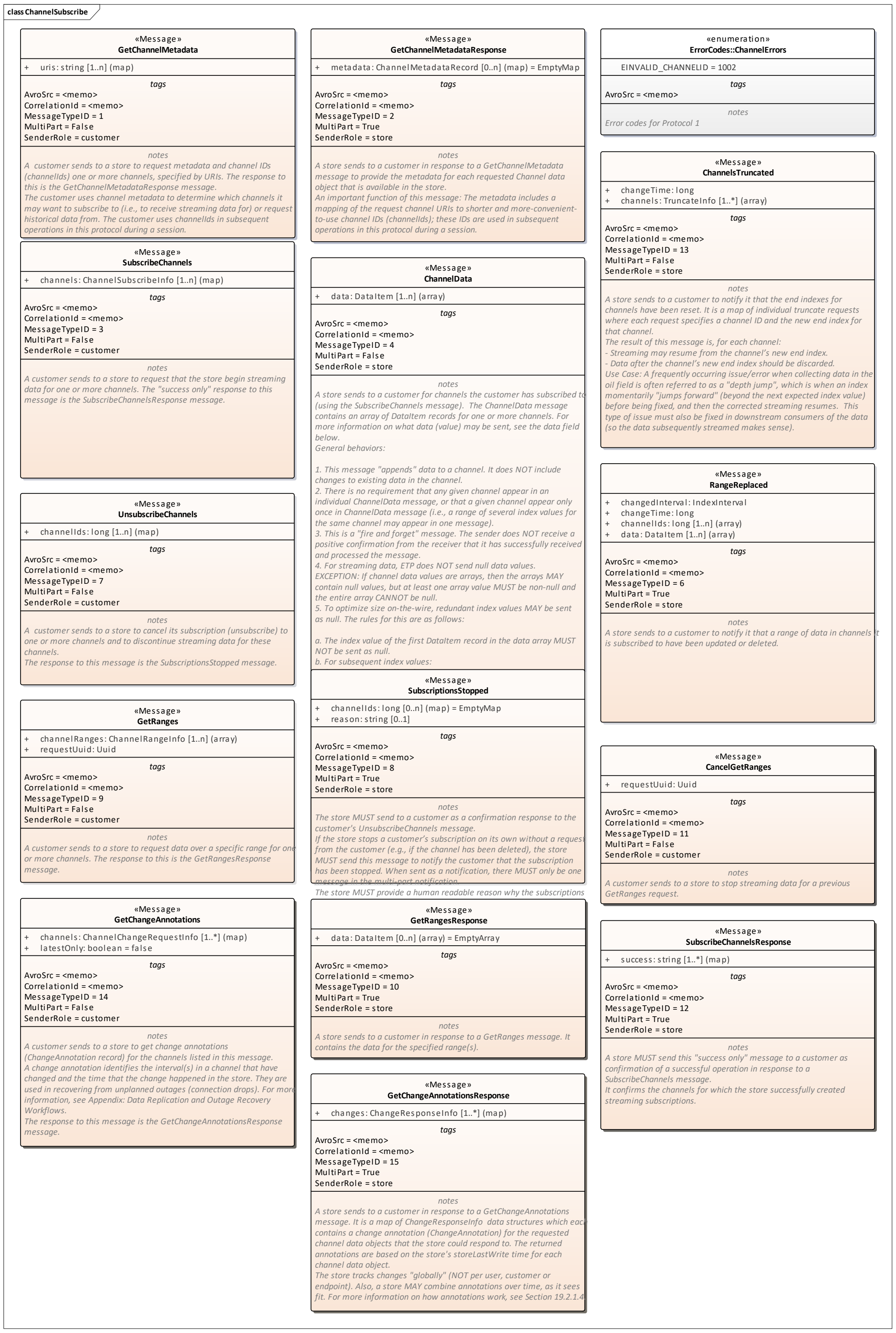
- For protocol-specific behavior relating to using these capabilities in this protocol, see◦Sections **19.2.1** and **19.2.2**.
- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section **3.3**.

ChannelSubscribe (Protocol 21): Capabilities			
Name: Description	**Type**	**Units** **Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of each endpoint capability, see Section **3.3**.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section **3.3.2**. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP Server**.			
ChangeRetentionPeriod: The minimum time period in seconds that a store retains the Canonical URI of a deleted data object and any change annotations for channels and growing data objects. **RECOMMENDATION:** This period should be as long as is feasible in an implementation. When the period is shorter, the risk is that additional data will need to be transmitted to recover from outages, leading to higher initial load on sessions.	long	second <number of seconds>	**Default:** 86,400 **MIN:** 86,400
Data Object Capabilities (For definitions of each data object capability, see Section **3.3.4**.)			
ActiveTimeoutPeriod: (This is also an endpoint capability.) The minimum time period in seconds that a store keeps the active status (*activeStatus* field in ETP) for a growing data object or channel "active" after the last new part or data point resulting in a change to the data object's end index was added to the data object. This capability can be set for an endpoint and/or for a data object. A data object-specific value overrides an endpoint-specific value.	long	second <number of seconds>	**Default:** 3,600 **MIN:** 60 seconds
Protocol Capabilities			
MaxIndexCount: The maximum index count value allowed for a channel streaming request.	long	count <count of indexes>	**Default:** 100 **MIN:** 1
MaxRangeChannelCount: The maximum count of channels allowed in a single range request.	long	count <count of channels>	**MIN:** Should be equivalent to MaxContained DataObjectCount for ChannelSet
MaxRangeDataItemCount: The maximum total count of DataItem records allowed in a complete multipart range message.	long	count <count of records>	**MIN:** 1,000,000
MaxStreamingChannelsSessionCount: The maximum total count of channels allowed to be concurrently open for streaming in a session. The limit applies separately for each protocol with the	long	count <count of channels>	**MIN:** 10,000

ChannelSubscribe (Protocol 21): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
capability. EXAMPLE: Different values can be specified for ChannelSubscribe (Protocol 21) and ChannelDataLoad (Protocol 22).			
SupportsSecondaryIndexFiltering: Indicates whether an endpoint supports filtering requested data by secondary index values. If the filtering can be technically supported by an endpoint, this capability should be true.	Boolean	N/A	N/A

19.3 ChannelSubscribe: Message Schemas

This section provides a figure that displays all messages defined in ChannelSubscribe (Protocol 21). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.



«Message»
GetChannelMetadata
+ uris: string [1..n] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to request metadata and channel IDs (channelIds) one or more channels, specified by URIs. The response to this is the GetChannelMetadataResponse message.
The customer uses channel metadata to determine which channels it may want to subscribe to (i.e., to receive streaming data for) or request historical data from. The customer uses channelIds in subsequent operations in this protocol during a session.

«Message»
SubscribeChannels
+ channels: ChannelSubscribeInfo [1..n] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 3
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to request that the store begin streaming data for one or more channels. The "success only" response to this message is the SubscribeChannelsResponse message.

«Message»
UnsubscribeChannels
+ channelIds: long [1..n] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 7
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to cancel its subscription (unsubscribe) to one or more channels and to discontinue streaming data for these channels.
The response to this message is the SubscriptionsStopped message.

«Message»
GetRanges
+ channelRanges: ChannelRangeInfo [1..n] (array)
+ requestUuid: Uuid

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 9
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to request data over a specific range for one or more channels. The response to this is the GetRangesResponse message.

«Message»
GetChangeAnnotations
+ channels: ChannelChangeRequestInfo [1..*] (map)
+ latestOnly: boolean = false

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 14
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to get change annotations (ChangeAnnotation record) for the channels listed in this message.
A change annotation identifies the interval(s) in a channel that have changed and the time that the change happened in the store. They are used in recovering from unplanned outages (connection drops). For more information, see Appendix: Data Replication and Outage Recovery Workflows.
The response to this message is the GetChangeAnnotationsResponse message.

«Message»
GetChannelMetadataResponse
+ metadata: ChannelMetadataRecord [0..n] (map) = EmptyMap

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 2
MultiPart = True
SenderRole = store

notes
A store sends to a customer in response to a GetChannelMetadata message to provide the metadata for each requested Channel data object that is available in the store.
An important function of this message: The metadata includes a mapping of the request channel URIs to shorter and more-convenient-to-use channel IDs (channelIds); these IDs are used in subsequent operations in this protocol during a session.

«Message»
ChannelData
+ data: DataItem [1..n] (array)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 4
MultiPart = False
SenderRole = store

notes
A store sends to a customer for channels the customer has subscribed to (using the SubscribeChannels message). The ChannelData message contains an array of DataItem records for one or more channels. For more information on what data (value) may be sent, see the data field below.
General behaviors:

1. This message "appends" data to a channel. It does NOT include changes to existing data in the channel.
2. There is no requirement that any given channel appear in an individual ChannelData message, or that a given channel appear only once in ChannelData message (i.e., a range of several index values for the same channel may appear in one message).
3. This is a "fire and forget" message. The sender does NOT receive a positive confirmation from the receiver that it has successfully received and processed the message.
4. For streaming data, ETP does NOT send null data values. EXCEPTION: If channel data values are arrays, then the arrays MAY contain null values, but at least one array value MUST be non-null and the entire array CANNOT be null.
5. To optimize size on-the-wire, redundant index values MAY be sent as null. The rules for this are as follows:

a. The index value of the first DataItem record in the data array MUST NOT be sent as null.
b. For subsequent index values:

«Message»
SubscriptionsStopped
+ channelIds: long [0..n] (map) = EmptyMap
+ reason: string [0..1]

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 8
MultiPart = True
SenderRole = store

notes
The store MUST send to a customer as a confirmation response to the customer's UnsubscribeChannels message.
If the store stops a customer's subscription on its own without a request from the customer (e.g., if the channel has been deleted), the store MUST send this message to notify the customer that the subscription has been stopped. When sent as a notification, there MUST only be one message in the multi-part notification.
The store MUST provide a human readable reason why the subscriptions

«Message»
GetRangesResponse
+ data: DataItem [0..n] (array) = EmptyArray

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 10
MultiPart = True
SenderRole = store

notes
A store sends to a customer in response to a GetRanges message. It contains the data for the specified range(s).

«Message»
GetChangeAnnotationsResponse
+ changes: ChangeResponseInfo [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 15
MultiPart = True
SenderRole = store

notes
A store sends to a customer in response to a GetChangeAnnotations message. It is a map of ChangeResponseInfo data structures which each contains a change annotation (ChangeAnnotation) for the requested channel data objects that the store could respond to. The returned annotations are based on the store's storeLastWrite time for each channel data object.
The store tracks changes "globally" (NOT per user, customer or endpoint). Also, a store MAY combine annotations over time, as it sees fit. For more information on how annotations work, see Section 19.2.1.4

«enumeration»
ErrorCodes::ChannelErrors
EINVALID_CHANNELID = 1002

tags
AvroSrc = <memo>

notes
Error codes for Protocol 1

«Message»
ChannelsTruncated
+ changeTime: long
+ channels: TruncateInfo [1..*] (array)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 13
MultiPart = False
SenderRole = store

notes
A store sends to a customer to notify it that the end indexes for channels have been reset. It is a map of individual truncate requests where each request specifies a channel ID and the new end index for that channel.
The result of this message is, for each channel:
- Streaming may resume from the channel's new end index.
- Data after the channel's new end index should be discarded.
Use Case: A frequently occurring issue/error when collecting data in the oil field is often referred to as a "depth jump", which is when an index momentarily "jumps forward" (beyond the next expected index value) before being fixed, and then the corrected streaming resumes. This type of issue must also be fixed in downstream consumers of the data (so the data subsequently streamed makes sense).

«Message»
RangeReplaced
+ changedInterval: IndexInterval
+ changeTime: long
+ channelIds: long [1..n] (array)
+ data: DataItem [1..n] (array)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 6
MultiPart = True
SenderRole = store

notes
A store sends to a customer to notify it that a range of data in channels it is subscribed to have been updated or deleted.

«Message»
CancelGetRanges
+ requestUuid: Uuid

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 11
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to stop streaming data for a previous GetRanges request.

«Message»
SubscribeChannelsResponse
+ success: string [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 12
MultiPart = True
SenderRole = store

notes
A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a SubscribeChannels message.
It confirms the channels for which the store successfully created streaming subscriptions.

Figure 29: ChannelSubscribe: message schemas

19.3.1 Message: GetChannelMetadata

A customer sends to a store to request metadata and channel IDs (*channelIds*) for one or more channels, specified by URIs. The response to this is the GetChannelMetadataResponse message.

The customer uses channel metadata to determine which channels it may want to subscribe to (i.e., to receive streaming data for). The customer uses *channelIds* in subsequent operations in this protocol during a session.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uris	A map whose values are the URIs for the channels that the customer wants information (metadata) about. The URIS MUST be URIs for channel data objects. If both endpoints support alternate URIs for the session, these MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	string	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
     "name": "GetChannelMetadata",
     "protocol": "21",
     "messageType": "1",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "uris",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

19.3.2 Message: GetChannelMetadataResponse

A store sends to a customer in response to a GetChannelMetadata message to provide the metadata for each requested Channel data object that is available in the store.

An important function of this message: The metadata includes a mapping of the request channel URIs to shorter and more-convenient-to-use channel IDs (*channelIds*); these IDs are used in subsequent operations in this protocol during a session.

Message Type ID: 2

Correlation Id Usage: MUST be set to the messageId of the ***GetChannelMetadata*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
metadata	A map of ChannelMetadataRecord records, which contains the metadata for each channel the store could successfully return. The URIs MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.	ChannelMetadataRecord	0	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
     "name": "GetChannelMetadataResponse",
     "protocol": "21",
     "messageType": "2",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "metadata",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.ChannelData.ChannelMetadataRecord" }, "default": {}
         }
     ]
}
```

19.3.3 Message: SubscribeChannels

A customer sends to a store to request that the store begin streaming data for one or more channels. The "success only" response to this message is the SubscribeChannelsResponse message.

Message Type ID: 3

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
channels	A map of ChannelSubscribeInfo records, one for each channel that the customer is requesting to subscribe to (i.e., have data streamed to it as soon as it is available). The startIndex value is a union of possible points to begin the stream (latest, *n* points back from now, etc.).	ChannelSubscribeInfo	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
     "name": "SubscribeChannels",
     "protocol": "21",
     "messageType": "3",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
```

```
     [
         {
             "name": "channels",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.ChannelData.ChannelSubscribeInfo" }
         }
     ]
}
```

19.3.4 Message: SubscribeChannelsResponse

A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a SubscribeChannels message. It confirms the channels for which the store successfully created streaming subscriptions.

Message Type ID: 12

Correlation Id Usage: MUST be set to the *messageId* of the ***SubscribeChannels*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
success	• The presence of the map key represents success. • The associated map string value SHOULD be empty because its content is ignored.	string	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
     "name": "SubscribeChannelsResponse",
     "protocol": "21",
     "messageType": "12",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "success",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

19.3.5 Message: ChannelData

A store sends to a customer, for channels the customer has subscribed to (using the SubscribeChannels message). The ***ChannelData*** message contains an array of DataItem records for one or more channels. For more information on what data (value) may be sent, see the *data* field below.

General behaviors:

1. This message "appends" data to a channel. It does NOT include changes to existing data in the channel.
2. There is no requirement that any given channel appear in an individual ***ChannelData*** message, or that a given channel appear only once in ***ChannelData*** message (i.e., a range of several index values for the same channel may appear in one message).

3. This is a "fire and forget" message. The sender does NOT receive a positive confirmation from the receiver that it has successfully received and processed the message.
4. For streaming data, ETP does NOT send null data values. **EXCEPTION:** If channel data values are arrays, then the arrays MAY contain null values, but at least one array value MUST be non-null and the entire array CANNOT be null.
5. The index values in each ***DataValue*** record are in the same order as their corresponding ***IndexMetadataRecord*** records in the corresponding channel's ***ChannelMetadataRecord*** record, and the primary index is always first.
6. To optimize size on-the-wire, redundant index values MAY be sent as null. The rules for this are as follows:

   a. The index value of the first ***DataItem*** record in the *data* array MUST NOT be sent as null.

   b. For subsequent index values:

      i. If an index value differs from the previous index value in the *data* array, the index value MUST NOT be sent as null.

      ii. If an index value is the same as the previous index value in the *data* array, the index value MAY be sent as null.

   c. **EXAMPLE:** These index values from adjacent ***DataItem*** records in the *data* array:

      [1.0, 1.0, 2.0, 3.0, 3.0]

      MAY be sent as:

      [1.0, null, 2.0, 3.0, null].

   d. When the ***DataItem*** records have both primary and secondary index values, these rules apply separately to each index.

   e. **EXAMPLE:** These primary and secondary index values from adjacent ***DataItem*** records in the *data* array:

      [[1.0, 10.0], [1.0, 11.0], [2.0, 11.0], [3.0, 11.0], [3.0, 12.0]]

      MAY be sent as:

      [[1.0, 10.0], [null, 11.0], [2.0, null], [3.0, null], [null, 12.0]].

   f. If ALL index values for a ***DataItem*** record are to be sent as null, the *indexes* field should be set to an empty array.

6. For more information about sending channel data, see Section **6.1.3**.

Message Type ID: 4

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
data	Contains the data points for channels, which is an array of DataItem records. Note that the value must be one of the types specified in ***DataValue*** (Section **23.30**)—which include options to send a single data value (of various types such as integers, longs, doubles, etc.) OR arrays of values. For more information, see Section **6.1.3**.	DataItem	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
     "name": "ChannelData",
     "protocol": "21",
     "messageType": "4",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "data",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.DataItem" }
         }
     ]
}
```

19.3.6 Message: ChannelsTruncated

A store sends to a customer to notify it that the end indexes for channels have been reset. It is a map of individual truncate requests where each request specifies a channel ID and the new end index for that channel.

The result of this message is, for each channel:

- Streaming may resume from the channel's new end index.
- Data after the channel's new end index should be discarded.

Use Case: A frequently occurring issue/error when collecting data in the oil field is often referred to as a "depth jump", which is when an index momentarily "jumps forward" (beyond the next expected index value) before being fixed, and then the corrected streaming resumes. This type of issue must also be fixed in downstream consumers of the data (so the data subsequently streamed makes sense).

Message Type ID: 13

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
channels	Contains a map of TruncateInfo structures, which each indicate the channel ID of a channel that was truncated and its new end index.	TruncateInfo	1	*
changeTime	The change time when the truncation/index update occurred in the store. It is a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
     "name": "ChannelsTruncated",
     "protocol": "21",
     "messageType": "13",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "channels",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.TruncateInfo" }
         },
         { "name": "changeTime", "type": "long" }
     ]
}
```

19.3.7 Message: RangeReplaced

A store sends to a customer to notify it that a range of data in channels it is subscribed to have been updated or deleted.

General behaviors and rules:

1. A complete, multi-part ***ReplaceRange*** message MUST include all data in the replacement range for all channels affected by the message, but there is no requirement that any given channel appear in an individual ***ReplaceRange*** message, or that a given channel appear only once in a ***ReplaceRange*** message (i.e., a range of several index values for the same channel may appear in one message).
2. It is recommended but NOT required to send data in row order rather than column order (i.e., send all data for all channels, one primary index value at a time rather than sending all data for each channel, one channel at a time).
3. For range replacement data, ETP does NOT send null data values. If there is no value for a channel for a particular primary index value in the replacement range, omit that primary index value for that channel from the notification. **EXCEPTION:** If channel data values are arrays, then the arrays MAY contain null values, but at least one array value MUST be non-null and the entire array CANNOT be null.
4. The index values in each ***DataValue*** record are in the same order as their corresponding ***IndexMetadataRecord*** records in the corresponding channel's ***ChannelMetadataRecord*** record, and the primary index is always first.
5. To optimize size on-the-wire, redundant index values MAY be sent as null. The rules for this are as follows:

    a. The index value of the first ***DataItem*** record in the *data* array MUST NOT be sent as null.

    b. For subsequent index values:

       i. If an index value differs from the previous index value in the *data* array, the index value MUST NOT be sent as null.

       ii. If an index value is the same as the previous index value in the *data* array, the index value MAY be sent as null.

    c. **EXAMPLE:** These index values from adjacent ***DataItem*** records in the *data* array:

       [1.0, 1.0, 2.0, 3.0, 3.0]

       MAY be sent as:

       [1.0, null, 2.0, 3.0, null].

    d. When the ***DataItem*** records have both primary and secondary index values, these rules apply separately to each index.

e. **EXAMPLE:** These primary and secondary index values from adjacent ***DataItem*** records in the *data* array:

[[1.0, 10.0], [1.0, 11.0], [2.0, 11.0], [3.0, 11.0], [3.0, 12.0]]

MAY be sent as:

[[1.0, 10.0], [null, 11.0], [2.0, null], [3.0, null], [null, 12.0]].

f. If ALL index values for a ***DataItem*** record are to be sent as null, the *indexes* field should be set to an empty array.

For more information about sending channel data, see Section **6.1.3**.

Message Type ID: 6

Correlation Id Usage: For the first message, MUST be set to 0. If there are multiple messages in this multipart request, the *correlationId* of all successive messages that comprise the request MUST be set to the *messageId* of the first message of the multipart request.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
changeTime	The change time when the range replace occurred in the store. It is a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
channelIds	The IDs of the channels that are being updated.	long	1	n
changedInterval	The indexes that define the interval that is changing as specified in IndexInterval.	IndexInterval	1	1
data	Contains the channel data as defined in DataItem that will replace the data defined by the *changedInterval* field. To delete an interval in a channel, leave this field blank. The interval identified in *changedInterval* is deleted (essentially replaced with nothing).	DataItem	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
     "name": "RangeReplaced",
     "protocol": "21",
     "messageType": "6",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         { "name": "changeTime", "type": "long" },
         {
             "name": "channelIds",
             "type": { "type": "array", "items": "long" }
         },
         { "name": "changedInterval", "type":
"Energistics.Etp.v12.Datatypes.Object.IndexInterval" },
         {
             "name": "data",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.DataItem" }
         }
```

```
    ]
}
```

19.3.8 Message: UnsubscribeChannels

A customer sends to a store to cancel its subscription (unsubscribe) to one or more channels and to discontinue streaming data for these channels.

The response to this message is the SubscriptionsStopped message.

Message Type ID: 7

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
channelIds	A map whose values are the channelIds to stop streaming.	long	1	n

Avro Source

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
    "name": "UnsubscribeChannels",
    "protocol": "21",
    "messageType": "7",
    "senderRole": "customer",
    "protocolRoles": "store,customer",
    "multipartFlag": false,

    "fields":
    [
        {
            "name": "channelIds",
            "type": { "type": "map", "values": "long" }
        }
    ]
}
```

19.3.9 Message: SubscriptionsStopped

The store MUST send to a customer as a confirmation response to the customer's UnsubscribeChannels message.

If the store stops a customer's subscription on its own without a request from the customer (e.g., if the channel has been deleted), the store MUST send this message to notify the customer that the subscription has been stopped. When sent as a notification, there MUST only be one message in the multipart notification.

The store MUST provide a human readable reason why the subscriptions were stopped.

Message Type ID: 8

Correlation Id Usage: When sent as a response: MUST be set to the *messageId* of the ***UnsubscribeChannels*** message that this message is a response to. When sent as a notification: MUST be ignored and SHOULD be set to 0.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
reason	A reason why the subscriptions have been stopped.	string	1	1
channelIds	A map whose values MUST be the channelIds of the channels that the store has stopped streaming.	long	0	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
     "name": "SubscriptionsStopped",
     "protocol": "21",
     "messageType": "8",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         { "name": "reason", "type": "string" },
         {
             "name": "channelIds",
             "type": { "type": "map", "values": "long" }, "default": {}
         }
     ]
}
```

19.3.10 Message: GetRanges

A customer sends to a store to request data over a specific range for one or more channels. The response to this is the GetRangesResponse message.

Message Type ID: 9

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
requestUuid	A UUID for this request. This MUST be a newly-generated UUID from the customer. This UUID can be used to cancel the request later. Must be of type Uuid.	Uuid	1	1
channelRanges	An array of data that specifies the ranges for which to request data and, for each range, the channels for which to get the data in that range as defined in the ChannelRangeInfo record.	ChannelRangeInfo	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
     "name": "GetRanges",
     "protocol": "21",
     "messageType": "9",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         {
             "name": "channelRanges",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.ChannelRangeInfo" }
         }
     ]
}
```

19.3.11 Message: GetRangesResponse

A store sends to a customer in response to a GetRanges message. It contains the data for the specified range(s).

General behaviors and rules:

1. A complete, multi-part ***GetRangesResponse*** message MUST include all data in the requested range for all channels in the request message, but there is no requirement that any given channel appear in an individual ***GetRangesResponse*** message, or that a given channel appear only once in a ***GetRangesResponse*** message (i.e., a range of several index values for the same channel may appear in one message).
2. It is recommended but NOT required to send data in row order rather than column order (i.e., send all data for all channels, one primary index value at a time rather than sending all data for each channel, one channel at a time).
3. For range response data, ETP does NOT send null data values. If there is no value for a channel for a particular primary index value in the requested range, omit that primary index value for that channel from the response. **EXCEPTION:** If channel data values are arrays, then the arrays MAY contain null values, but at least one array value MUST be non-null and the entire array CANNOT be null.
4. The index values in each ***DataValue*** record are in the same order as their corresponding ***IndexMetadataRecord*** records in the corresponding channel's ***ChannelMetadataRecord*** record, and the primary index is always first.
5. To optimize size on-the-wire, redundant index values MAY be sent as null. The rules for this are as follows:

   a. The index value of the first ***DataItem*** record in the *data* array MUST NOT be sent as null.

   b. For subsequent index values:

      i. If an index value differs from the previous index value in the *data* array, the index value MUST NOT be sent as null.

      ii. If an index value is the same as the previous index value in the *data* array, the index value MAY be sent as null.

   c. **EXAMPLE:** These index values from adjacent ***DataItem*** records in the *data* array:

      [1.0, 1.0, 2.0, 3.0, 3.0]

      MAY be sent as:

      [1.0, null, 2.0, 3.0, null].

   d. When the ***DataItem*** records have both primary and secondary index values, these rules apply separately to each index.

   e. **EXAMPLE:** These primary and secondary index values from adjacent ***DataItem*** records in the *data* array:

      [[1.0, 10.0], [1.0, 11.0], [2.0, 11.0], [3.0, 11.0], [3.0, 12.0]]

      MAY be sent as:

      [[1.0, 10.0], [null, 11.0], [2.0, null], [3.0, null], [null, 12.0]].

f. If ALL index values for a ***DataItem*** record are to be sent as null, the *indexes* field should be set to an empty array.

For more information about sending channel data, see Section **6.1.3**.

Message Type ID: 10

Correlation Id Usage: MUST be set to the *messageId* of the ***GetRanges*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
data	Contains an array of data items as defined in DataItem.	DataItem	0	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
     "name": "GetRangesResponse",
     "protocol": "21",
     "messageType": "10",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "data",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.DataItem" }, "default": []
         }
     ]
}
```

19.3.12 Message: CancelGetRanges

A customer sends to a store to stop streaming data for a previous GetRanges request.

Message Type ID: 11

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
requestUuid	The UUID of the request message that started streaming the range request that is now being canceled. Must be of type ***Uuid*** (Section **23.6**).	Uuid	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
     "name": "CancelGetRanges",
     "protocol": "21",
     "messageType": "11",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" }
     ]
}
```

19.3.13 Message: GetChangeAnnotations

A customer sends to a store to get change annotations (ChangeAnnotation record) for the channels listed in this message.

A change annotation identifies the interval(s) in a channel that have changed and the time that the change happened in the store. They are used in recovering from unplanned outages (connection drops). For more information, see **Appendix: Data Replication and Outage Recovery Workflows**.

The response to this message is the GetChangeAnnotationsResponse message.

Message Type ID: 14

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
channels	General ETP map of ChannelChangeRequestInfo records, which identify the channel and date from which changes are being requested.	ChannelChangeRequestInfo	1	*
latestOnly	If true, it means get the latest (last) change annotation only for each of the channels listed.	boolean	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
     "name": "GetChangeAnnotations",
     "protocol": "21",
     "messageType": "14",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "channels",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.ChannelData.ChannelChangeRequestInfo" }
         },
         { "name": "latestOnly", "type": "boolean", "default": false }
     ]
}
```

19.3.14 Message: GetChangeAnnotationsResponse

A store sends to a customer in response to a GetChangeAnnotations message. It is a map of ChangeResponseInfo data structures which each contains a change annotation (ChangeAnnotation) for the requested channel data objects that the store could respond to. The returned annotations are based on the store's *storeLastWrite* time for each channel data object.

The store tracks changes "globally" (NOT per user, customer or endpoint). Also, a store MAY combine annotations over time, as it sees fit. For more information on how annotations work, see Section **19.2.1.4**.

Message Type ID: 15

Correlation Id Usage: MUST be set to the *messageId* of the ***GetChangeAnnotations*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
changes	ETP general map of ChannelChangeResponseInfo records, one for each ChannelChangeRequestInfo the store can respond to, which lists the channels that have changed and the information for each as specified in the ChangeAnnotation record.	ChangeResponseInfo	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelSubscribe",
     "name": "GetChangeAnnotationsResponse",
     "protocol": "21",
     "messageType": "15",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "changes",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.Object.ChangeResponseInfo" }
         }
     ]
}
```

20 ChannelDataLoad (Protocol 22)

ProtocolID: 22

Defined Roles: store, customer

Use ChannelDataLoad (Protocol 22) to connect to an endpoint and push channel data to it. The ETP customer role controls the ETP session/behavior and pushes data to the ETP store role. Important use cases include rig acquisition workflow (where service companies doing data acquisition are required to deliver data to a rig aggregator) and any case in which you want to write data (e.g., load historical data) to a store.

With Protocol 22:

- The customer essentially treats a channel like a file, where the customer can add new data points to the channel, either at the "end" of the channel or update points anywhere in the channel. There are 2 main modes of loading data: 1) streaming real time and 2) historical updates (deletes, range replaces, and updates)
- The main expected workflow is exception based; that is, the customer first tries to open the channel on the store to write data to it. If the channel does not exist on the store, the customer can then use Store (Protocol 4) to add the channel to the store—and then use Protocol 22 to load data into it.
- A customer can "guarantee" data delivery to the store because it can confirm what data the store has, then stream data from that point forward.
- ETP DOES NOT support nor provide functionality to detect or recover from multi-master replication. When a customer endpoint is loading data to a store with ChannelDataLoad (Protocol 22), it is assumed to be the only entity (or agent) loading data to specific channels in the store at that time.

Some key points about this and related protocols:

- **Protocol 22** has the "put/write" behavior for channel data; it "pushes" data from the customer role endpoint to the store role endpoint.
- **ChannelSubscribe (Protocol 21)** (see Chapter **19**) has the "get/read" behavior for channel data from a store and to "listen" for changes in channel data that require a notification (or data updates) to be sent while connected.

Other ETP sub-protocols that "stream" channel data:

- **ChannelStreaming (Protocol 1)** (see Chapter **6**) is for very simple streaming of channel-oriented data, from "simple" producers (i.e., a sensor) to consumers; it is designed to replace WITS data transfers. Protocol 1 allows a consumer to connect to a producer and receive whatever data the producer has (i.e., the consumer cannot discover available channels nor specify which channels it wants, etc. like in Protocol 21). **NOTE:** Beginning in ETP v1.2, Protocol 1 is used only for these so-called simple streamers (in previous versions of ETP, Protocol 1 included all channel streaming behavior).
- **ChannelDataFrame (Protocol 2)** (see Chapter **7**) allows a customer endpoint to get channel data from a store in a row-orientated 'frame' or 'table' of data. In energy industry jargon, the general use case that Protocol 2 supports is typically referred to as getting a "historical log". (In ETP jargon you are actually getting a frame of data from a ChannelSet data object; for more information, see Section **7.1.1**).

Other ETP sub-protocols related to Channel data objects:

- **Store (Protocol 4)**. Because a Channel is an Energistics data object, to add or delete Channel data◦objects from a store, use Store (Protocol 4) (see Chapter **9**).
- **StoreNotification (Protocol 5)**. To subscribe to notifications about Channel data objects (e.g., channels added, deleted, status change, etc.), use StoreNotification (Protocol 5) (see Chapter **10**).

NOTE: Energistics data models (e.g., WITSML) allow channels to be grouped into channel sets and logs. However, ETP channel streaming protocols handle individual channels; that is, whether or not the channel is part of a channel set or log is irrelevant to how it is handled in a channel streaming protocol. (For more information about channel sets, see Section **7.1.1**.)

This chapter includes main sections for:

- Required behavior, which includes:
  - Description of the message sequence for main tasks, along with required behavior, related capabilities, and possible errors (see Section **20.2.1**).
  - Other functional requirements (not covered in the message sequence) including use of endpoint, data object, and protocol capabilities for preventing and protecting against aberrant behavior (see Section◦**20.2.2**).
  - Definitions of the endpoint, data object, and protocol capabilities used in this protocol (see Section **20.2.3**).
- Sample schemas of the messages defined in this protocol (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes documentation for each field in a schema (see Section **20.3**).

20.1 ChannelDataLoad: Key Concepts

- For definitions and key concept related to channels and channel streaming, see Section **6.1**.

20.2 ChannelDataLoad: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.
- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**
- A customer has the Energistics URIs for each channel it needs to push data to.
  - The set of channel data objects to be "pushed" comes from an external source, for example, a contract that states the channels for which data must be provided. If the store receiving the data supports Discovery (Protocol 3), it may be possible to discover the channels on the store (**EXAMPLE:** If the requirement is "all channels in Well XYZ", those channels could be discovered.)
  - For information about Energistics URIs, see **Appendix: Energistics Identifiers**.

20.2.1 ChannelDataLoad: Message Sequences

This section explains the basic message sequence for main tasks to be done using this protocol and includes related key behaviors and possible errors; it assumes that an ETP session has been established using Core (Protocol 0) as described in Chapter **5**. The following General Requirements section provides additional requirements and rules for how this protocol works.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section.

ChannelDataLoad (Protocol 22): Basic Message-Response flow by ETP Role	
Message (customer)	**Response Message (store)**
OpenChannels: Identifies the channels that the customer wants to load data for.	**OpenChannelsResponse** (multipart): Response indicating which channels the store can accept data for and the metadata for each.
ChannelData: Contains the data the customer has for each channel; the customer keeps sending these messages as new data becomes available.	N/A
ReplaceRange (multipart): Updates to channels (i.e., delete one range and replace with new data provided) or delete ranges for channels.	**ReplaceRangeResponse**: Response indicating the ranges in channels that were successfully replaced or deleted.
TruncateChannels: Sent to reset the end indexes of channels to allow streaming to resume from the new end indexes; used to correct "index jump" errors in previously sent data.	**TruncateChannelsResponse** (multipart): Response indicating which channels were successfully truncated (i.e., which end indexes were successfully updated).
CloseChannels: Informs the store that no more data will be sent for the listed channels in the current session.	**ChannelsClosed** (multipart) which has 2 use cases: • Optional response to the ***CloseChannels*** message. • Sent by a store as a notification (i.e., without a customer request) that it has stopped accepting data for some channels.

20.2.1.1 To do the initial setup and begin streaming data:

1. The customer MUST send the store the ***OpenChannels*** message (Section **20.3.1**), which is a map whose values are the URIs of the channels that the customer intends to send data for.
    a. A customer MUST limit the total count of channels concurrently open for streaming in the current ETP session to the store's value for MaxStreamingChannelsSessionCount protocol capability.
    b. If the customer's request exceeds this limit, the store MUST deny that request and send error ELIMIT_EXCEEDED (12).
2. The store MUST respond with the following:
    a. For the channels it successfully opens for receiving data, the store MUST respond with a one or more ***OpenChannelsResponse*** map response messages (Section **20.3.2**). This response message contains a map of ***OpenChannelInfo*** records, which contain the following data for each channel:
        i. *metadata* for each channel (as defined in ***ChannelMetadataRecord*** (Section **23.33.7**). The store MUST assign each channel an integer identifier that is unique for the session in this protocol. This identifier will be used instead of the channel URI to identify the channel in subsequent messages in this protocol for the session. This identifier is set in the *id* field in the ***ChannelMetadataRecord***.
        RECOMMENDATION: Use the smallest available integer value for a new channel identifier.
        IMPORTANT: If the channel is deleted and recreated during a session, it MUST be assigned a new identifier.

   ii. *preferRealtime*, flag to indicate preference to receive realtime data first/as priority, before historical data.

   iii. *dataChanges*, flag to indicate if it wants to receive historical data changes (which are sent with ***ReplaceRange*** messages).

  b. For the channels it does NOT successfully open for receiving data, the store MUST send one or more map ***ProtocolException*** messages, where values in the *errors* field (a map) are appropriate errors.

   i. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.

   ii. For the channels that do not currently exist on the store, error ENOT_FOUND (11).

   iii. For the channels that the customer does not have permission to access/write to, error EREQUEST_DENIED (6).

3. To send data to the store, the customer MAY do any of the following:

  a. To append new data for any channel that the store opened for receiving data, the customer MUST send ***ChannelData*** messages (Section **20.3.6**).

   i. New data MUST always be sent in primary index order and MUST always be an append (i.e., with primary index value greater than, for increasing data, or less than, for decreasing data, the channel's end index).

   ii. The customer MAY continue to send these messages as new data becomes available or until all new data for a channel is sent.

   iii. **NOTIFICATION BEHAVIOR:** When the customer streams new data to the store, the store MUST send ***ChannelData*** messages in ChannelSubscribe (Protocol 21).

   iv. To ensure the store's channel data remains in a consistent state, if the store is unable to successfully process all data received in a ***ChannelData*** message:

    1. If a store IS NOT able to parse the message and find the set of all *channelIds* included in the message (e.g., the message body could not be deserialized):

     a. The store MUST send a non-map ***ProtocolException*** message in response to the message.

     b. The store MUST close ALL channels currently open for receiving data in the session.

    2. If store IS able to parse the message and find the set of all *channelds* included in the message:

     a. For each *channelId* that represents a valid, open channel, the store MUST process ALL data for the channel, in primary index order, until it encounters an error for the channel.

     b. For the channels with errors (invalid or unable to process all data), or for all channels if an error prevents the store from processing any data in the message:

      i. The store MUST send a map ***ProtocolException*** message where the map keys are the string version of the *channelIds* of the affected channels and the values are an appropriate error for each channel.

       1. If data for a channel is NOT an append or is not in primary index order, the store MUST send error EINVALID_APPEND (31).

      ii. The store MUST close the channels that are valid and open for receiving data and send ***ChannelsClosed***.

  b. If the store indicated that it wants to receive data change, to update or delete an existing range of data in a channel, the customer MUST send ***ReplaceRange*** messages (see Section **20.3.7**).

   i. For successful range replaces, the store MUST respond with a ***ReplaceRangeResponse*** message.

   ii. For more information about how range replacement operations work, see Section **20.2.2**, Row◦**10.**

c. To correct "index jump" errors, the customer MUST send a ***TruncateChannels*** message (Section **20.3.4**).

   i. For channels that it successfully truncated, the store MUST respond with a ***TruncateChannelsResponse*** message (Section **20.3.5**).

   ii. For more information about how truncate channels operations work, see Section **20.2.2**, Row **9**.

d. For channels that are not found on the store, the customer MAY use Store (Protocol 4) to add the channels to the store.

   i. To begin streaming data to these newly added channels, the customer must send the ***OpenChannels*** message per Step **1 above**.

   ii. **NOTE:** This exception-based workflow (option to add channels that do not exist) is recommended to reduce load on the store, particularly on the reconnection process (if you've lost connection to a channel you were previously streaming to).

20.2.1.2 To "close" a channel:

1. If the customer will not stream any more data on a channel in the current session, it MAY send the ***CloseChannels*** message (see Section **20.3.3**) to "close" channels and indicate that no more data will be sent (this is best practice).

   a. Terminating a session closes any channels that haven't been closed.

   b. The store MUST treat a dropped connection as an implicit 'close' to any open channels and do any clean-up required.

2. For the channels it successfully closed, the store MUST respond with a one or more ***ChannelsClosed*** map response messages (Section **20.3.9**), which list the IDs of the channels the store has closed.

   a. For more information on how map response messages work, see Section **3.7.3**.

   b. The store MUST deny any ***ChannelData***, ***ReplaceRange***, or ***TruncateChannels*** messages received for a channel after sending ***ChannelsClosed*** for the channel and send EREQUEST_DENIED (6)**.**

3. For the channels it did NOT successfully close, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors.

   a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.

   b. If a requested channel is not open for receiving data, send error EINVALID_STATE (8).

20.2.2 ChannelDataLoad: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) some rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter **5**. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3**. a. For the list of global capabilities and related behavior, see Section **3.3.2.** 3. Section **20.2.3** identifies the capabilities most relevant to this ETP sub-protocol. Additional details for using these capabilities are included below in this table and in Section **20.2.1 ChannelDataLoad: Message Sequence**.
3.	Message Sequence See Section **20.2.1**.	1. The Message Sequence section above (Section **20.2.1**) describes requirements for the main tasks listed there and also defines required behavior.
4.	Plural messages (which includes maps)	**1.** This protocol uses ETP-wide message patterns including plural messages and multipart responses. For more information on behaviors related to these messages, see Section **3.7.3**.
5.	Notifications	1. This chapter explains events (operations) in ChannelDataLoad (Protocol 22) that trigger the store to send notifications, which the store sends using ChannelSubscribe (Protocol 21). However, statements of **NOTIFICATION**

Row#	Requirement	Behavior
		BEHAVIOR are here in this chapter, in the context of the detailed explanation of the behavior that triggers the notification. 2. Notification behavior is described here using MUST. However, the store MUST ONLY send notifications IF AND ONLY IF there is a customer subscribed to notifications for the affected channels and the store MUST ONLY send notifications to those customers that are subscribed to the affected channels. a. For more information on data object notifications, see Chapter **10 StoreNotification (Protocol 5)**. b. For information on notifications for channel data, see Chapter◦**19 ChannelSubscribe (Protocol 21)**.
6.	Data order for loading data	1. Streaming data points (in ***ChannelData*** messages) MUST be sent in primary index order for each channel, both within one message and across multiple messages. a. If data is not in primary index order for a channel, the store MUST send error EINVALID_APPEND (31) as the map entry for the channel as described in Section **20.2.1.1**. 2. Data points in ***ReplaceRange*** messages MUST be sent in primary index order for each channel, both within a single message and across all messages within a multi-part request. a. If data is not in primary index order for a channel, the store MUST fail the entire request and send error EINVALID_OPERATION (32). 3. Primary index order is always as appropriate for the index direction of a channel (i.e., increasing or decreasing). 4. The index values for each data point are in the same order as their corresponding ***IndexMetadataRecord*** records in the corresponding channel's ***ChannelMetadataRecord*** record, and the primary index is always first. 5. The same primary index value MUST NOT appear more than once for the same channel in any ***ChannelData*** message UNLESS the channel data at that index was affected by a truncate operation during the session (i.e., a ***TruncateChannels*** or ***ReplaceRange*** message was sent that reset the channel's end index to before the primary index value). 6. The same primary index value MUST NOT appear more than once for the same channel in the same multipart ***ReplaceRange*** message.
7.	**Store Behavior:** Update the active status field (*activeStatus*)	1. The ***Resource*** (Section **23.34.11**) associated with each data object in ETP has an *activeStatus* field. a. This field appears ONLY on the ***Resource*** NOT on the data object. There MAY be an equivalent element on the data object. The mapping between *activeStatus* and the data object element is defined by the relevant ML ETP implementation specification. b. For channel data objects, this field may have a value of "active" or "inactive". c. For information about this field and behavior related to setting it to "inactive" related to the ActiveTimeoutPeriod capability, see Section **3.3.2.1**. 2. If a channel data object's *activeStatus* has a value of "inactive" and messages in this ETP sub-protocol begin operations that change the channel's channel data (e.g., appends data with ***ChannelData*** message or replaces a range with a ***ReplaceRange*** message), the store MUST do the following: a. Set the channel data object's *activeStatus* to "active". b. Reset the timer for the ActiveTimeoutPeriod capability. c. **NOTIFICATION BEHAVIOR:** Send an ***ObjectActiveStatusChanged*** notification message for the channel data object in StoreNotification (Protocol 5). For more information, see Section **10.2.2**, Row **16**.
8.	**Store Behavior:** Update the *storeLastWrite* field	1. Each ***Resource*** in ETP has a field named *storeLastWrite*; for more information about it, see Section **3.12.5.1**. 2. For operations in ChannelDataLoad (Protocol 22) that result in any change to the Channel data object or its channel data, the store MUST update the *storeLastWrite* field with the time of the change. For example, for data changes to the Channel from these messages in this protocol, the store MUST update *storeLastWrite*:

Row#	Requirement	Behavior
		a. ChannelData ***b. TruncateChannels*** ***c. ReplaceRange***
9.	Resetting channel end indexes (***TruncateChannels*** message/operation)	1. To reset the end indexes of channels, deleting any data past the new end indexes (e.g., to correct "index jump" errors), a customer MUST send a ***TruncateChannels*** message, which contains a map of ***TruncateInfo*** records; each record contains the channel ID and its new end index. 2. On receipt of a ***TruncateChannels*** message, the store MUST do all the following: a. Reset its end index to the *newEndIndex* specified in the ***TruncateChannels*** message. b. Delete any data that was previously sent (i.e., after the old erroneous *endIndex* value). c. For the channels it successfully truncates, the store MUST send one or more ***TruncateChannelsResponse*** map response messages. i. For more information on how map response messages work, see Section **3.7.3**. d. For the channels it could NOT successfully truncate, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors, such as EREQUEST_DENIED (6). i. For more information on how ***ProtocolException*** messages work with a plural messages, see Section **3.7.3**. 3. When the customer resumes sending ***ChannelData*** messages (for the channel whose index it corrected), the index for each new *DataItem* MUST be greater than (for increasing data) or less than (for decreasing data) the new end index in the ***TruncateChannels*** message. 4. **NOTIFICATION BEHAVIOR:** When a customer truncates channels, the store MUST send ***ChannelsTruncated*** in ChannelSubscribe (Protocol 21).
10.	Channel data changes: replacing and deleting ranges of historical data (***ReplaceRange*** message/operation)	1. To update or delete an existing range of data in a channel, the customer MUST send ***ReplaceRange*** messages (Section **20.3.7**). 2. When a customer sends a ***ReplaceRange*** message, it MUST honor these protocol capabilities: a. A customer MUST limit the count of channels in a single range request to the store's value for the MaxRangeChannelCount protocol capability. i. If the customer request exceeds this limit, the store MUST deny the request by sending error ELIMIT_EXCEEDED (12). b. A customer MUST limit the total count of ***DataItem*** records in a complete multipart range message to the store's value for MaxRangeDataItemCount protocol capability. i. If the customer request exceeds this limit, the store MUST deny the request by sending error ELIMIT_EXCEEDED (12). ii. If the customer's value for MaxRangeDataItemCount protocol capability is smaller than the store's value, then the customer MAY further limit the total count of ***DataItem*** records to its value. 3. The ***ReplaceRange*** is an ATOMIC operation in that the store is expected to delete the existing data and replace it with the contents of the entire set of multipart messages for the *data* array in a single operation. This is typically implemented as a database transaction. a. Because ***ReplaceRange*** is an atomic operation, the entire operation either succeeds or fails. b. When a store has successfully processed a ***ReplaceRange*** message/operation, it MUST respond with a ***ReplaceRangeResponse*** message. c. If ***ReplaceRange*** fails, the store MUST send a ***ProtocolException*** message with an appropriate error code such as EREQUEST_DENIED (6). 4. ***ReplaceRange*** messages MUST be sent in primary index order. a. For more information, see Row **5 above**. 5. All data items in the *data* array of a ***ReplaceRange*** message MUST conform to these rules: a. The *channelIds* of each ***DataItem*** in the *data* array MUST match one of the channels listed in the *channelIds* array in the first message.

Row#	Requirement	Behavior
		i. If any ***DataItem*** does not match a channel ID, the store MUST reject the request and send error EINVALID_ARGUMENT (5). b. The index of each ***DataItem*** MUST be between (inclusive) of the *startIndex* and *endIndex* defined in the *changedInterval* field. i. If any data falls outside of these bounds, the store MUST reject the request and send error EINVALID_ARGUMENT (5) 6. The behavior of the ***ReplaceRang***e operation is a 'delete and replace'. The store MUST observe the following behavior for ***ReplaceRange*** message: a. Delete the data between (inclusive of) *startIndex* and *endIndex*. b. Insert the ***DataItems*** provided in the *data* array. 7. To simply DELETE a range, the customer MUST send the ***ReplaceRange*** message and leave the *data* array empty (i.e., nothing to insert). a. The store MUST delete the specified range in *changedInterval*, for each of the channel IDs specified in *channelIds*. 8. If a ***ReplaceRange*** operation includes the current end (maximum) index of a channel, the store MUST reset the end index to reflect the new end index after the operation is complete. a. Depending on the specifics of the ***ReplaceRange*** operation, the new end index may be "behind", "ahead of" or the same as the previous end index. 9. **NOTIFICATION BEHAVIOR:** When a customer updates or deletes a range of data, the store MUST send ***RangeReplaced*** in ChannelSubscribe (Protocol 21). If the end index of the channel was also reset, the store MUST also send ***ChannelsTruncated***. For more information, see Section **19.2.2**, Row **9**.
11.	**Store Behavior:** Creating and managing change annotations	1. For a definition of change annotations and related terms, see Section **11.1.4**. 2. See Sections **11.2.2.2**, **11.2.2.4**, and **11.2.2.5** for the requirements on how to create and manage change annotations for channel data objects.
12.	Closing Channels	1. A store MUST close a channel when: a. The customer closes the channel by sending a ***CloseChannels*** message. b. The channel data object is deleted. c. The customer loses access to the channel data object. 2. When closing a channel: a. The store MUST send ***ChannelsClosed*** either as a response to a customer ***CloseChannels*** request or as a notification. b. The store MUST include in the ***ChannelsClosed*** message a human-readable reason why the channels were closed. 3. When a store closes a channel in response to a customer's ***CloseChannels*** request, the store MUST process any ***ChannelData***, ***ReplaceRange***, and ***TruncateChannels*** messages received for the channel before the ***CloseChannels*** message was received. 4. When a store end's a subscription without a customer request, the store MUST deny any pending ***ReplaceRange*** and ***TruncateChannels*** messages with EREQUEST_DENIED before sending ***ChannelsClosed***. 5. After sending the ***ChannelsClosed*** message, the store MUST NOT accept any further data for the closed channels. 6. After a channel has been closed: a. A customer MAY request that the channel be reopened by sending a new ***OpenChannels*** message with the channel's URI. 7. A store MUST NOT reopen the channel without a request from the customer.
13.	Index metadata	1. A channel data object's index metadata MUST be consistent: a. The index units and vertical datums MUST match the channel's index metadata. 2. When sending messages, both the store AND the customer MUST ensure that all index metadata and data derived from index metadata are consistent in all fields in the message, including in XML or JSON object data or part data. a. **EXAMPLE:** The *uom* and *depthDatum* in an ***IndexInterval*** record MUST be consistent with the channel's index metadata. b. A store MUST reject requests with inconsistent index metadata with an appropriate error such as EINVALID_OBJECT (14) or EINVALID_ARGUMENT (5).

20.2.3 ChannelDataLoad: Capabilities

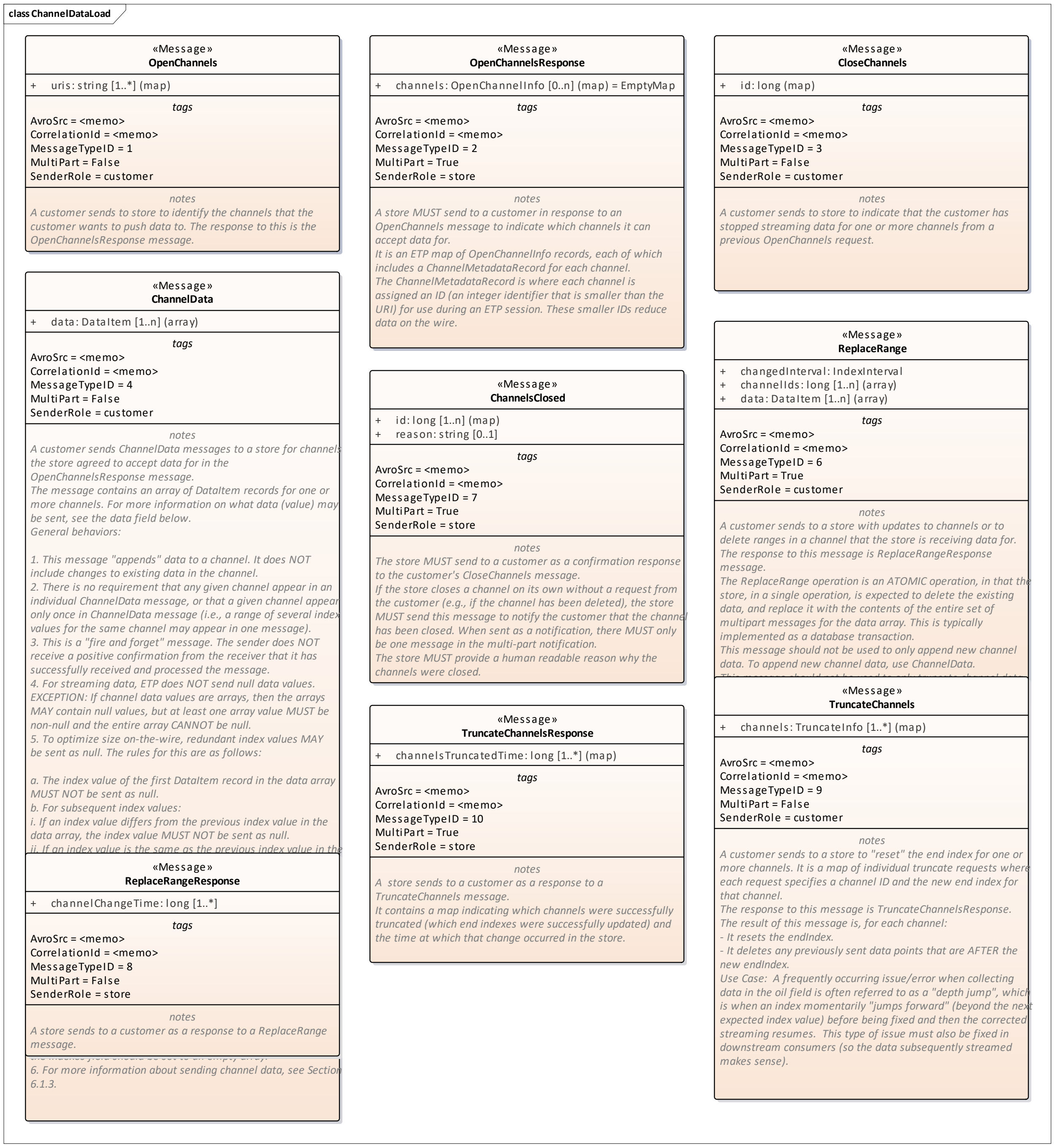
The table below lists key capabilities for this protocol. The protocol capabilities are all defined here.

- For protocol-specific behavior relating to using these capabilities in this protocol, see◦Sections **20.2.1**and◦**20.2.2**.
- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section **3.3**.

ChannelDataLoad (Protocol 22): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of each endpoint capability, see Section **3.3**.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section **3.3.2**. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP** Server.			
Data Object Capabilities (For definitions of each data object capability, see Section **3.3.4**.)			
ActiveTimeoutPeriod: (This is also an endpoint capability.) The minimum time period in seconds that a store keeps the GrowingStatus for a growing data object or channel "active" after the last new part or data point resulting in a change to the data object's end index was added to the data object. This capability can be set for an endpoint and/or for a data object. A data object-specific value overrides an endpoint-specific value.	long	seconds <number of seconds>	**Default:** 3,600 **MIN:** 60 seconds
Protocol Capabilities			
MaxRangeChannelCount: The maximum count of channels allowed in a single range request.	long	count <count of channels>	**MIN:** Should be equivalent to MaxContained DataObjectCount for ChannelSet
MaxRangeDataItemCount: The maximum total count of DataItem records allowed in a complete multipart range message.	long	count <count of records>	**MIN:** 1,000,000
MaxStreamingChannelsSessionCount: The maximum total count of channels allowed to be concurrently open for streaming in a session. The limit applies separately for each protocol with the capability. **EXAMPLE:** Different values can be specified for ChannelSubscribe (Protocol 21) and ChannelDataLoad (Protocol 22).	long	count <count of sessions>	**MIN:** 10,000

20.3 ChannelDataLoad: Message Schemas

This section provides a figure that displays all messages defined in ChannelDataLoad (Protocol 22). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.



«Message»
OpenChannels

+ uris: string [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1
MultiPart = False
SenderRole = customer

notes
A customer sends to store to identify the channels that the customer wants to push data to. The response to this is the OpenChannelsResponse message.

«Message»
ChannelData

+ data: DataItem [1..n] (array)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 4
MultiPart = False
SenderRole = customer

notes
A customer sends ChannelData messages to a store for channels the store agreed to accept data for in the OpenChannelsResponse message.
The message contains an array of DataItem records for one or more channels. For more information on what data (value) may be sent, see the data field below.
General behaviors:

1. This message "appends" data to a channel. It does NOT include changes to existing data in the channel.
2. There is no requirement that any given channel appear in an individual ChannelData message, or that a given channel appear only once in ChannelData message (i.e., a range of several index values for the same channel may appear in one message).
3. This is a "fire and forget" message. The sender does NOT receive a positive confirmation from the receiver that it has successfully received and processed the message.
4. For streaming data, ETP does NOT send null data values. EXCEPTION: If channel data values are arrays, then the arrays MAY contain null values, but at least one array value MUST be non-null and the entire array CANNOT be null.
5. To optimize size on-the-wire, redundant index values MAY be sent as null. The rules for this are as follows:

a. The index value of the first DataItem record in the data array MUST NOT be sent as null.
b. For subsequent index values:
i. If an index value differs from the previous index value in the data array, the index value MUST NOT be sent as null.
ii. If an index value is the same as the previous index value in the
6. For more information about sending channel data, see Section 6.1.3.

«Message»
ReplaceRangeResponse

+ channelChangeTime: long [1..*]

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 8
MultiPart = False
SenderRole = store

notes
A store sends to a customer as a response to a ReplaceRange message.

«Message»
OpenChannelsResponse

+ channels: OpenChannelInfo [0..n] (map) = EmptyMap

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 2
MultiPart = True
SenderRole = store

notes
A store MUST send to a customer in response to an OpenChannels message to indicate which channels it can accept data for.
It is an ETP map of OpenChannelInfo records, each of which includes a ChannelMetadataRecord for each channel.
The ChannelMetadataRecord is where each channel is assigned an ID (an integer identifier that is smaller than the URI) for use during an ETP session. These smaller IDs reduce data on the wire.

«Message»
ChannelsClosed

+ id: long [1..n] (map)
+ reason: string [0..1]

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 7
MultiPart = True
SenderRole = store

notes
The store MUST send to a customer as a confirmation response to the customer's CloseChannels message.
If the store closes a channel on its own without a request from the customer (e.g., if the channel has been deleted), the store MUST send this message to notify the customer that the channel has been closed. When sent as a notification, there MUST only be one message in the multi-part notification.
The store MUST provide a human readable reason why the channels were closed.

«Message»
TruncateChannelsResponse

+ channelsTruncatedTime: long [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 10
MultiPart = True
SenderRole = store

notes
A store sends to a customer as a response to a TruncateChannels message.
It contains a map indicating which channels were successfully truncated (which end indexes were successfully updated) and the time at which that change occurred in the store.

«Message»
CloseChannels

+ id: long (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 3
MultiPart = False
SenderRole = customer

notes
A customer sends to store to indicate that the customer has stopped streaming data for one or more channels from a previous OpenChannels request.

«Message»
ReplaceRange

+ changedInterval: IndexInterval
+ channelIds: long [1..n] (array)
+ data: DataItem [1..n] (array)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 6
MultiPart = True
SenderRole = customer

notes
A customer sends to a store with updates to channels or to delete ranges in a channel that the store is receiving data for.
The response to this message is ReplaceRangeResponse message.
The ReplaceRange operation is an ATOMIC operation, in that the store, in a single operation, is expected to delete the existing data, and replace it with the contents of the entire set of multipart messages for the data array. This is typically implemented as a database transaction.
This message should not be used to only append new channel data. To append new channel data, use ChannelData.

«Message»
TruncateChannels

+ channels: TruncateInfo [1..*] (map)

tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 9
MultiPart = False
SenderRole = customer

notes
A customer sends to a store to "reset" the end index for one or more channels. It is a map of individual truncate requests where each request specifies a channel ID and the new end index for that channel.
The response to this message is TruncateChannelsResponse.
The result of this message is, for each channel:
- It resets the endIndex.
- It deletes any previously sent data points that are AFTER the new endIndex.
Use Case: A frequently occurring issue/error when collecting data in the oil field is often referred to as a "depth jump", which is when an index momentarily "jumps forward" (beyond the next expected index value) before being fixed and then the corrected streaming resumes. This type of issue must also be fixed in downstream consumers (so the data subsequently streamed makes sense).

Figure 30: ChannelDataLoad: message schemas

20.3.1 Message: OpenChannels

A customer sends to store to identify the channels that the customer wants to push data to. The response to this is the OpenChannelsResponse message.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uris	General ETP map where the values MUST be the URI for each channel that the customer wants to push data to. The URIS MUST be URIs for channel data objects. If both endpoints support alternate URIs for the session, these MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	string	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataLoad",
     "name": "OpenChannels",
     "protocol": "22",
     "messageType": "1",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "uris",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

20.3.2 Message: OpenChannelsResponse

A store MUST send to a customer in response to an OpenChannels message to indicate which channels it can accept data for.

It is an ETP map of OpenChannelInfo records, each of which includes a ChannelMetadataRecord for each channel.

The ChannelMetadataRecord is where each channel is assigned an ID (an integer identifier that is smaller than the URI) for use during an ETP session. These smaller IDs reduce data on the wire.

Message Type ID: 2

Correlation Id Usage: MUST be set to the *messageId* of the ***OpenChannels*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
channels	A general ETP map of OpenChannelInfo records, one for each channel the store can accept data for. Each ***OpenChannelInfo*** record references a ChannelMetadataRecord, which contains the URI of the channel and other metadata for interpreting the channel data. The URIs MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**. **NOTE:** For additional information on how to populate these attributes for each of the Energistics data models (WITSML, RESQML or PRODML), see the ML-specific ETP implementation specification.	OpenChannelInfo	0	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataLoad",
     "name": "OpenChannelsResponse",
     "protocol": "22",
     "messageType": "2",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "channels",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.ChannelData.OpenChannelInfo" }, "default": {}
         }
     ]
}
```

20.3.3 Message: CloseChannels

A customer sends to store to indicate that the customer has stopped streaming data for one or more channels from a previous OpenChannels request.

Message Type ID: 3

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
id	A general ETP map whose values MUST be the IDs of the channels that are being closed.	long	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataLoad",
     "name": "CloseChannels",
     "protocol": "22",
     "messageType": "3",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,
     "fields":
     [
         {
             "name": "id",
             "type": { "type": "map", "values": "long" }
         }
     ]
}
```

20.3.4 Message: TruncateChannels

A customer sends to a store to "reset" the end index for one or more channels. It is a map of individual truncate requests where each request specifies a channel ID and the new end index for that channel.

The response to this message is TruncateChannelsResponse.

The result of this message is, for each channel:

- It resets the endIndex.
- It deletes any previously sent data points that are AFTER the new endIndex.

Use Case: A frequently occurring issue/error when collecting data in the oil field is often referred to as a "depth jump", which is when an index momentarily "jumps forward" (beyond the next expected index value) before being fixed and then the corrected streaming resumes. This type of issue must also be fixed in downstream consumers (so the data subsequently streamed makes sense).

Message Type ID: 9

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
channels	A general ETP map of TruncateInfo records, one for each channel to be truncated. Each ***TruncateInfo*** record lists the ID of each channel to be truncated, and the details of each channel and new end index specified.	TruncateInfo	1	*

Avro Source

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Protocol.ChannelDataLoad",
    "name": "TruncateChannels",
    "protocol": "22",
    "messageType": "9",
    "senderRole": "customer",
    "protocolRoles": "store,customer",
    "multipartFlag": false,

    "fields":
    [
        {
            "name": "channels",
            "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.ChannelData.TruncateInfo" }
        }
    ]
}
```

20.3.5 Message: TruncateChannelsResponse

A store sends to a customer as a response to a TruncateChannels message.

It contains a map indicating which channels were successfully truncated (which end indexes were successfully updated) and the time at which that change occurred in the store.

Message Type ID: 10

Correlation Id Usage: MUST be set to the *messageId* of the ***TruncateChannels*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
channelsTruncatedTime	A map whose value is the time each channel in the map was truncated/updated in the store. Must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataLoad",
     "name": "TruncateChannelsResponse",
     "protocol": "22",
     "messageType": "10",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "channelsTruncatedTime",
             "type": { "type": "map", "values": "long" }
         }
     ]
}
```

20.3.6 Message: ChannelData

A customer sends ***ChannelData*** messages to a store for channels the store agreed to accept data for in the OpenChannelsResponse message.

The message contains an array of DataItem records for one or more channels. For more information on what data (value) may be sent, see the *data* field below.

General behaviors and rules:

1. This message "appends" data to a channel. It does NOT include changes to existing data in the channel.
2. There is no requirement that any given channel appear in an individual ***ChannelData*** message, or that a given channel appear only once in ***ChannelData*** message (i.e., a range of several index values for the same channel may appear in one message).
3. This is a "fire and forget" message. The sender does NOT receive a positive confirmation from the receiver that it has successfully received and processed the message.
4. The index values in each ***DataValue*** record are in the same order as their corresponding ***IndexMetadataRecord*** records in the corresponding channel's ***ChannelMetadataRecord*** record, and the primary index is always first.

5. For streaming data, ETP does NOT send null data values. **EXCEPTION:** If channel data values are arrays, then the arrays MAY contain null values, but at least one array value MUST be non-null and the entire array CANNOT be null.
6. To optimize size on-the-wire, redundant index values MAY be sent as null. The rules for this are as follows:

   a. The index value of the first ***DataItem*** record in the *data* array MUST NOT be sent as null.

   b. For subsequent index values:

      i. If an index value differs from the previous index value in the *data* array, the index value MUST NOT be sent as null.

      ii. If an index value is the same as the previous index value in the *data* array, the index value MAY be sent as null.

   c. **EXAMPLE:** These index values from adjacent ***DataItem*** records in the *data* array:

      [1.0, 1.0, 2.0, 3.0, 3.0]

      MAY be sent as:

      [1.0, null, 2.0, 3.0, null].

   d. When the ***DataItem*** records have both primary and secondary index values, these rules apply separately to each index.

   e. **EXAMPLE:** These primary and secondary index values from adjacent ***DataItem*** records in the *data* array:

      [[1.0, 10.0], [1.0, 11.0], [2.0, 11.0], [3.0, 11.0], [3.0, 12.0]]

      MAY be sent as:

      [[1.0, 10.0], [null, 11.0], [2.0, null], [3.0, null], [null, 12.0]].

   f. If ALL index values for a ***DataItem*** record are to be sent as null, the *indexes* field should be set to an empty array.
6. For more information about sending channel data, see Section **6.1.3**.

Message Type ID: 4

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	**Description**	**Data Type**	**Min**	**Max**
data	Contains the data points for channels, which is an array of DataItem records. Note that the value must be one of the types specified in ***DataValue*** (Section **23.30**)—which include options to send a single data value (of various types such as integers, longs, doubles, etc.) OR arrays of values. For more information, see Section **6.1.3**.	DataItem	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataLoad",
     "name": "ChannelData",
     "protocol": "22",
     "messageType": "4",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "data",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.DataItem" }
         }
     ]
}
```

20.3.7 Message: ReplaceRange

A customer sends to a store with updates to channels or to delete ranges in a channel that the store is receiving data for. The response to this message is ReplaceRangeResponse message.

The ***ReplaceRange*** operation is an ATOMIC operation, in that the store, in a single operation, is expected to delete the existing data, and replace it with the contents of the entire set of multipart messages for the data array. This is typically implemented as a database transaction.

This message should not be used to only append new channel data. To append new channel data, use ***ChannelData***.

This message should not be used to only truncate channel data. To truncate channel data, use ***TruncateChannels***.

General behaviors and rules:

1. A complete, multi-part ***ReplaceRange*** message MUST include all replacement data for all channels affected by the message, but there is no requirement that any given channel appear in an individual ***ReplaceRange*** message, or that a given channel appear only once in a ***ReplaceRange*** message (i.e., a range of several index values for the same channel may appear in one message).
2. It is recommended but NOT required to send data in row order rather than column order (i.e., send all data for all channels, one primary index value at a time rather than sending all data for each channel, one channel at a time).
3. For range replacement data, ETP does NOT send null data values. If there is no replacement value for a channel for a particular primary index value, omit that primary index value for that channel from the request. **EXCEPTION:** If channel data values are arrays, then the arrays MAY contain null values, but at least one array value MUST be non-null and the entire array CANNOT be null.
4. The index values in each ***DataValue*** record are in the same order as their corresponding ***IndexMetadataRecord*** records in the corresponding channel's ***ChannelMetadataRecord*** record, and the primary index is always first.
5. To optimize size on-the-wire, redundant index values MAY be sent as null. The rules for this are as follows:

   a. The index value of the first ***DataItem*** record in the *data* array MUST NOT be sent as null.

   b. For subsequent index values:

      i. If an index value differs from the previous index value in the *data* array, the index value MUST NOT be sent as null.

      ii. If an index value is the same as the previous index value in the *data* array, the index value MAY be sent as null.

   c. **EXAMPLE:** These index values from adjacent ***DataItem*** records in the *data* array:

      [1.0, 1.0, 2.0, 3.0, 3.0]

MAY be sent as:

[1.0, null, 2.0, 3.0, null].

d. When the ***DataItem*** records have both primary and secondary index values, these rules apply separately to each index.

e. **EXAMPLE:** These primary and secondary index values from adjacent ***DataItem*** records in the *data* array:

[[1.0, 10.0], [1.0, 11.0], [2.0, 11.0], [3.0, 11.0], [3.0, 12.0]]

MAY be sent as:

[[1.0, 10.0], [null, 11.0], [2.0, null], [3.0, null], [null, 12.0]].

f. If ALL index values for a ***DataItem*** record are to be sent as null, the *indexes* field should be set to an empty array.

For more information about sending channel data, see Section **6.1.3**.

Message Type ID: 6

Correlation Id Usage: For the first message, MUST be set to 0. If this is a multimessage request, the *correlationId* of all successive messages that comprise the request MUST be set to the *messageId* of the first message of the multipart request.

Multi-part: True

Sent by: customer

Field Name	Description	Data Type	Min	Max
changedInterval	The indexes that define the interval that is changing as specified in the IndexInterval record.	IndexInterval	1	1
channelIds	An array of the IDs of the channels that are being updated.	long	1	n
data	An array of channel data as defined in DataItem that will replace the data defined by the *changedInterval* field. To delete an interval in a channel, leave this field blank. The interval identified in *changedInterval* is deleted (essentially replaced with nothing).	DataItem	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataLoad",
     "name": "ReplaceRange",
     "protocol": "22",
     "messageType": "6",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         { "name": "changedInterval", "type":
"Energistics.Etp.v12.Datatypes.Object.IndexInterval" },
         {
             "name": "channelIds",
             "type": { "type": "array", "items": "long" }
         },
         {
             "name": "data",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.DataItem" }
         }
     ]
}
```

20.3.8 Message: ReplaceRangeResponse

A store sends to a customer as a response to a ReplaceRange message.

Message Type ID: 8

Correlation Id Usage: MUST be set to the messageId of the FIRST (or only) ***ReplaceRange*** message in the multipart request that this message is in response to.

Multi-part: False

Sent by: store

Field Name	Description	Data Type	Min	Max
channelChangeTime	The time the ranges were replaced (written in store). Because this is an atomic operation (all fail or all succeed), this message has this single change time (i.e., all channels were replaced at the same time). Must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataLoad",
     "name": "ReplaceRangeResponse",
     "protocol": "22",
     "messageType": "8",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "channelChangeTime", "type": "long" }
     ]
}
```

20.3.9 Message: ChannelsClosed

The store MUST send to a customer as a confirmation response to the customer's ***CloseChannels*** message.

If the store closes a channel on its own without a request from the customer (e.g., if the channel has been deleted), the store MUST send this message to notify the customer that the channel has been closed. When sent as a notification, there MUST only be one message in the multi-part notification.

The store MUST provide a human readable reason why the channels were closed.

Message Type ID: 7

Correlation Id Usage: When sent as a response: MUST be set to the *messageId* of the ***CloseChannels*** message that this message is a response to. When sent as a notification: MUST be ignored and SHOULD be set to 0.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
reason	A reason why the channels have been closed.	string	1	1
id	ETP general map where the values must be the IDs of the channels being closed.	long	1	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.ChannelDataLoad",
     "name": "ChannelsClosed",
     "protocol": "22",
     "messageType": "7",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         { "name": "reason", "type": "string" },
         {
             "name": "id",
             "type": { "type": "map", "values": "long" }
         }
     ]
}
```

21 Dataspace (Protocol 24)

ProtocolID: 24

Defined Roles: store, customer

A customer endpoint uses Dataspace (Protocol 24) to discover dataspaces on a store. After locating the target dataspace, the customer then uses Discovery (Protocol 3) (see Chapter **8**) to discover its content.

IMPORTANT! For information on how to format URIs for ETP, including URIs for dataspaces, see **Appendix: Energistics Identifiers**.

This chapter includes main sections for:

- Key ETP concepts that are important to understanding how this protocol is intended to work (see Section **21.1**.
- Required behavior, which includes:
  - Description of the message sequence for main tasks, along with required behavior, usage of ETP-defined capabilities, and possible errors (see Section **21.2.1**).
  - Other functional requirements (not covered in the message sequence) including use of ETP-defined endpoint and protocol capabilities for preventing and protecting against aberrant behavior (see Section **21.2.2**).
  - Definitions of the ETP-defined endpoint and protocol capabilities used in this protocol (see Section **21.2.3**).
- Sample schemas of the messages defined in this protocol, which are identical to the Avro schemas published with this version of ETP. However, only the schema content in this specification includes documentation for each field (see Section **21.3**).

21.1 Dataspace: Key Concepts

21.1.1 Dataspace: Definition

A dataspace is an abstract concept representing a distinct collection of data objects. Dataspaces have been kept as general as possible to support a variety of use cases. ETP does not assign a specific meaning to dataspaces, but different use cases may use a dataspace to represent a project on disk, a specific database, a specific tenant in a multi-tenant store, a specific back-end data store, etc. When dataspaces are used, dataspaces within a store are identified by unique ETP URIs (see **Appendix: Energistics Identifiers**), and URIs for data objects within a dataspace include the dataspace URI as a prefix.

EXAMPLE: When working on a large oil and gas asset, it is common to organize work into projects. Subsequently, the data may also be organized and stored as projects. In this type of organization, it is possible for the same data object to be worked on by multiple project teams and exist in multiple locations (data stores), and as multiple versions. In these situations, users keep track of which projects they are working on and which data store the project is stored in. ETP dataspaces may be used to represent these different data stores and the projects in them. URIs for data objects within the projects will be prefixed with the project's dataspace URI.

21.2 Dataspace: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.

- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.
- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**

21.2.1 Dataspace: Message Sequence

This section explains the basic message sequence for main tasks to be done using this protocol and includes related key behaviors, capabilities usage, and possible errors. The following General Requirements section provides additional requirements and rules for how this protocol works.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section.

Dataspaces (Protocol 24): Basic Message-Response flow by ETP Role	
Message from customer	**Response Message from store**
GetDataspaces: A customer sends to a store to discover all dataspaces available on the store.	**GetDataspacesResponse** (multipart): Response that lists the dataspaces the store could return.
PutDataspaces: A customer sends to a store to create one or more dataspaces.	**PutDataspacesResponse** (multipart): The "success only" response indicating which dataspaces the store successfully put.
DeleteDataspaces: A customer sends to a store to delete one or more dataspaces.	**DeleteDataspacesResponse** (multipart): The "success only" response indicating which dataspaces the store successfully deleted.

21.2.1.1 To get a list of dataspaces on a store:

1. A customer MUST send a store a ***GetDataspaces*** message (Section **21.3.1**).
    a. An optional *storeLastWriteFilter* allows the customer to filter by the date/time stamp.
2. If the store successfully returns dataspaces that match the criteria specified in the ***GetDataspaces*** message, the store MUST send one or more ***GetDataspacesResponse*** messages (Section **21.3.2**), each of which has an array of dataspaces available in the store.
    a. A store MUST limit the total count of responses to the customer's value for the MaxResponseCount protocol capability.
    b. If the store exceeds the customer's MaxResponseCount value, the customer MAY send error ERESPONSECOUNT_EXCEEDED (30).
    c. If a store's MaxResponseCount value is less than the customer's MaxResponseCount value, the store MAY further limit the total count of responses (to its value).
    d. If a store cannot send all responses to a request because it would exceed the lower of the customer's or the store's MaxResponseCount value, the store:
        i. MUST terminate the multipart response by sending error ERESPONSECOUNT_EXCEEDED (30).

ii. MUST NOT terminate the response until it has sent MaxResponseCount responses.

3. If the store has no dataspaces that meet the criteria specified in the ***GetDataspaces*** message, the store MUST send a ***GetDataspacesResponse*** message with the FIN bit set and the *dataspaces* field set to an empty array.
4. If the store does NOT successfully return dataspaces, it MUST send a non-map ***ProtocolException*** message with an appropriate error, such as EREQUEST_DENIED (06).

21.2.1.2 To create a dataspace on a store:

1. A customer MUST send a store a ***PutDataspaces*** message (Section **21.3.3**), which is a map of the dataspaces it wants to create.
2. For the dataspaces it successfully creates, the store MUST send one or more ***PutDataspacesResponse*** map response messages (Section **21.3.4**) where presence of the map key indicates success.
   a. For more information on how map response messages work, see Section **3.7.3**.
3. For the dataspaces it does NOT create, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors, such as EREQUEST_DENIED (6).
   a. For lack of permissions, send error EREQUEST_DENIED (6).
   b. For the complete list of ETP error codes, see Section **24.3**.
   c. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.

21.2.1.3 To delete a dataspaces from a store:

1. A customer MUST send a store a ***DeleteDataspaces*** message (Section **21.3.5**), which is a map of URIs for the dataspaces it wants to delete.
2. For the dataspaces it successfully deletes, the store MUST send one or more ***DeleteDataspacesResponse*** map response messages (Section **21.3.6**) where presence of the map key indicates success.
   a. For more information on how map response messages work, see Section **3.7.3**.
3. For the dataspaces it does NOT successfully delete, the store MUST send one or more map ***ProtocolException*** messages where values in the *errors* field (a map) are appropriate errors, such as ENOT_FOUND (11).
   a. For more information on how ***ProtocolException*** messages work with plural messages, see Section **3.7.3**.

21.2.2 Dataspace: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) additional rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for

Row#	Requirement	Behavior
		plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter **5**. b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter **5**) and MUST be used/honored as defined in the relevant ETP sub-protocol. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section **3.3**. a. For the list of global capabilities and related behavior, see Section **3.3.2**. 3. Section **21.2.3** identifies the capabilities most relevant to this ETP sub-protocol. Additional details for how to use the protocol capabilities are included below in this table and in Section **21.2.1 Dataspace: Message Sequence**.
3.	Message Sequence See Section **21.2.1**.	1. The Message Sequence section above (Section **21.2.1**) describes requirements for the main tasks listed there and also defines required behavior.
4.	Plural message (which includes maps)	1. This protocol uses plural messages. For detailed rules on handling plural messages (including ***ProtocolException*** handling), see Section **3.7.3**.
5.	Default dataspace	1. A store MUST support the default dataspace. a. The default dataspace MAY be empty (i.e., have no data objects). b. The URI for the default dataspace is **eml:///** c. The path for the empty dataspace is a 0 length string (i.e., the empty string).

21.2.3 Dataspace: Capabilities

The table below lists key capabilities for this protocol. The protocol capabilities are all defined here.

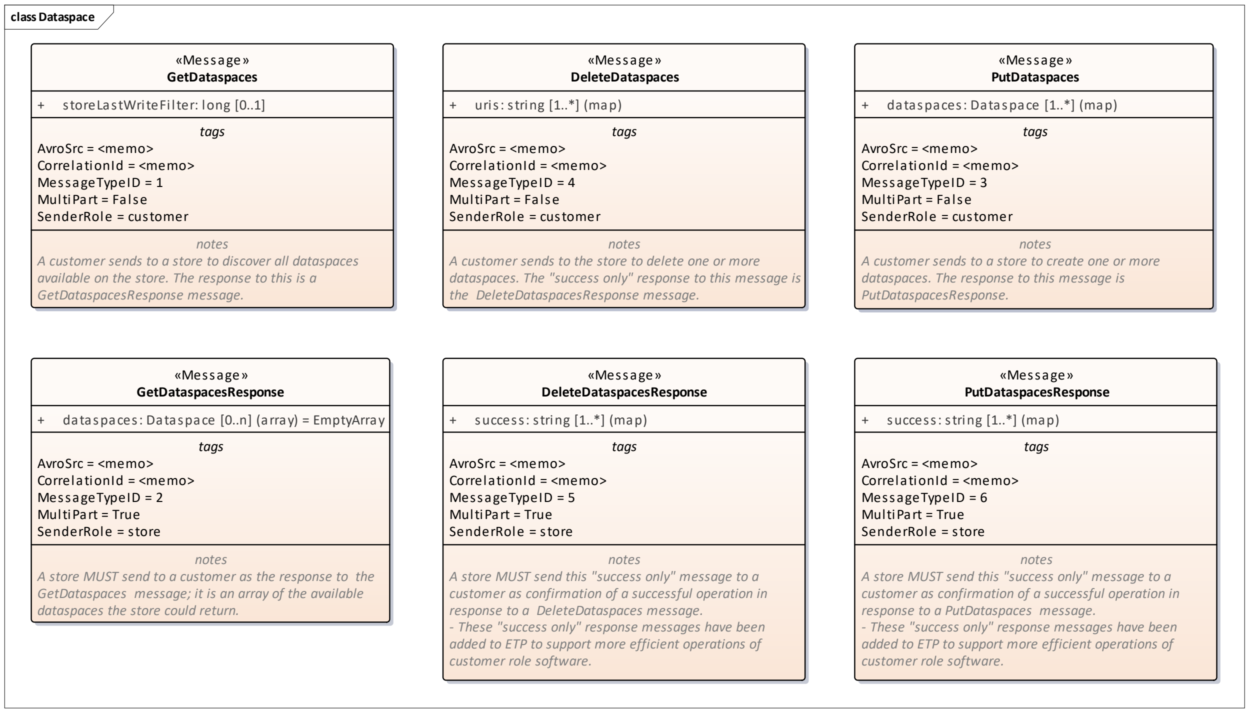
- For protocol-specific behavior relating to using these capabilities in this protocol, see◦Sections **21.2.1** and◦**21.2.2**.

- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section **3.3**.

Dataspace (Protocol 24): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of each endpoint capability, see Section **3.3**.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section **3.3.2**. Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP** Server.			
Protocol Capabilities			
MaxResponseCount: The maximum total count of responses allowed in a complete multipart message response to a single request.	long	count <count of responses>	**MIN:** 10,000

21.3 Dataspace: Message Schemas

This section provides a figure that displays all messages defined in Dataspace (Protocol 24). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.



class Dataspace

«Message»
GetDataspaces
+ storeLastWriteFilter: long [0..1]
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 1
MultiPart = False
SenderRole = customer
notes
A customer sends to a store to discover all dataspaces available on the store. The response to this is a GetDataspacesResponse message.

«Message»
DeleteDataspaces
+ uris: string [1..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 4
MultiPart = False
SenderRole = customer
notes
A customer sends to the store to delete one or more dataspaces. The "success only" response to this message is the DeleteDataspacesResponse message.

«Message»
PutDataspaces
+ dataspaces: Dataspace [1..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 3
MultiPart = False
SenderRole = customer
notes
A customer sends to a store to create one or more dataspaces. The response to this message is PutDataspacesResponse.

«Message»
GetDataspacesResponse
+ dataspaces: Dataspace [0..n] (array) = EmptyArray
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 2
MultiPart = True
SenderRole = store
notes
A store MUST send to a customer as the response to the GetDataspaces message; it is an array of the available dataspaces the store could return.

«Message»
DeleteDataspacesResponse
+ success: string [1..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 5
MultiPart = True
SenderRole = store
notes
A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a DeleteDataspaces message.
- These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

«Message»
PutDataspacesResponse
+ success: string [1..*] (map)
tags
AvroSrc = <memo>
CorrelationId = <memo>
MessageTypeID = 6
MultiPart = True
SenderRole = store
notes
A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutDataspaces message.
- These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

Figure 31: Dataspace: message schemas

21.3.1 Message: GetDataspaces

A customer sends to a store to discover all dataspaces available on the store. The response to this is a GetDataspacesResponse message.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
storeLastWriteFilter	An optional filter to limit the dataspaces returned by date/time last saved to the store (value in *storeLastWrite* field). The store returns a list of dataspaces whose last changed date/time is greater than the specified date/time. It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	0	1

Avro Source

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Protocol.Dataspace",
    "name": "GetDataspaces",
    "protocol": "24",
    "messageType": "1",
    "senderRole": "customer",
    "protocolRoles": "store,customer",
    "multipartFlag": false,

    "fields":
    [
        { "name": "storeLastWriteFilter", "type": ["null", "long"] }
    ]
}
```

21.3.2 Message: GetDataspacesResponse

A store MUST send to a customer as the response to the GetDataspaces message; it is an array of the available dataspaces the store could return.

Message Type ID: 2

Correlation Id Usage: MUST be set to the *messageId* of the ***GetDataspaces*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
dataspaces	An array of Dataspace records, each of which specifies data for each dataspace being returned.	Dataspace	0	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Dataspace",
     "name": "GetDataspacesResponse",
     "protocol": "24",
     "messageType": "2",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "dataspaces",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.Object.Dataspace" }, "default": []
         }
     ]
}
```

21.3.3 Message: PutDataspaces

A customer sends to a store to create one or more dataspaces. The response to this message is PutDataspacesResponse.

Message Type ID: 3

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
dataspaces	ETP general map where the values MUST be Dataspace records, one each for each dataspace the customer wants to add or update. Each record contains the fields of data for each dataspace. The URIs MUST be canonical Energistics dataspace URIs; for more information, see **Appendix: Energistics Identifiers**.	Dataspace	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Dataspace",
     "name": "PutDataspaces",
     "protocol": "24",
     "messageType": "3",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "dataspaces",
             "type": { "type": "map", "values":
"Energistics.Etp.v12.Datatypes.Object.Dataspace" }
         }
     ]
}
```

21.3.4 Message: PutDataspacesResponse

A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a PutDataspaces message.

These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

Message Type ID: 6

Correlation Id Usage: MUST be set to the *messageId* of the ***PutDataspaces*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
success	• The presence of the map key represents success. • The associated map string value SHOULD be empty because its content is ignored.	string	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Dataspace",
     "name": "PutDataspacesResponse",
     "protocol": "24",
     "messageType": "6",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "success",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

21.3.5 Message: DeleteDataspaces

A customer sends to the store to delete one or more dataspaces. The "success only" response to this message is the DeleteDataspacesResponse message.

Message Type ID: 4

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uris	ETP general map where the values must the URIs for the dataspaces the customer wants to delete. The URIs MUST be canonical Energistics dataspace URIs; for more information, see **Appendix: Energistics Identifiers**.	string	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Dataspace",
     "name": "DeleteDataspaces",
     "protocol": "24",
     "messageType": "4",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         {
             "name": "uris",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

21.3.6 Message: DeleteDataspacesResponse

A store MUST send this "success only" message to a customer as confirmation of a successful operation in response to a DeleteDataspaces message.

These "success only" response messages have been added to ETP to support more efficient operations of customer role software.

Message Type ID: 5

Correlation Id Usage: MUST be set to the *messageId* of the ***DeleteDataspaces*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
success	• The presence of the map key represents success. • The associated map string value SHOULD be empty because its content is ignored.	string	1	*

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.Dataspace",
     "name": "DeleteDataspacesResponse",
     "protocol": "24",
     "messageType": "5",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "success",
             "type": { "type": "map", "values": "string" }
         }
     ]
}
```

22 SupportedTypes (Protocol 25)

ProtocolID: 25

Defined Roles: store, customer

A customer uses SupportedTypes (Protocol 25) to discover a store's data model—that is, to dynamically understand what data object types and relationships are *possible* in the store at a given location (i.e., at a node in the data model), without prior knowledge of the overall data model and graph connectivity.

Other ETP sub-protocols that may be used with SupportedTypes (Protocol 25):

- To discover dataspaces in a store that you might want to get supported types for, use Dataspace (Protocol 24), see Chapter **21**.
- To discover data objects in a store that you might want to get supported types for, use Discovery (Protocol 3), see Chapter **8**.

This chapter includes main sections for:

- Required behavior, which includes:
  - Description of the message sequence for main tasks, along with required behavior and possible errors (Section **22.2.1**).
  - Other functional requirements (not covered in the message sequence) including use of endpoint and protocol capabilities for preventing and protecting against aberrant behavior (Section **22.2.2**).
  - Definitions of the endpoint and protocol capabilities used in this protocol (Section **22.2.3**).
- Sample schemas of the messages defined in this protocol (which are identical to the Avro schemas published with this version of ETP). However, only the schema content in this specification includes documentation for each field (Section **22.3**).

22.1 SupportedTypes: Key Concepts

This section explains concepts that are important to understanding how SupportedTypes (Protocol 25) works.

22.1.1 Data Model as Graph

The messages in SupportedTypes (Protocol 25) have been developed to work with data models as graphs. When understood and used properly, this graph approach allows customers to specify precisely and in a single request the desired set of objects to monitor for notifications, thereby reducing traffic on the wire.

- For general definition of a graph, how it works, and key concepts and how they are used as inputs, see Section **8.1.1**.

22.2 SupportedTypes: Required Behavior

This section contains functional and non-functional requirements for this protocol. It is organized in these sub-sections:

- **Message Sequence.** Summarizes all messages defined by this protocol, identifies main tasks that can be done with this protocol and describes the response/request pattern for the messages needed to perform the tasks, including usage of ETP-defined capabilities, error scenarios, and resulting ETP error codes.
- **General Requirements.** Identifies high-level (across ETP) and protocol-wide general behavior and rules that must be observed (in addition to behavior specified in Message Sequence), including usage of ETP-defined endpoint, data object and protocol capabilities, error scenarios, and resulting error codes.

- **Capabilities.** Lists and defines the ETP-defined parameters most relevant for this sub-protocol. ETP defines these parameters to set necessary limits to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

Prerequisites for using this protocol:

- An ETP session has been established using Core (Protocol 0) as described in Chapter **5.**

22.2.1 SupportedTypes: Message Sequence

This section explains the basic message sequence for main tasks to be done using this protocol and includes related key behaviors and possible errors; it assumes that an ETP session has been established using Core (Protocol 0) as described in Chapter **5**. The following General Requirements section provides additional requirements and rules for how this protocol works.

The following table lists all messages defined in this sub-protocol, the basic request/response usage patterns per ETP role, and whether it may be multipart. The detailed content of each message is explained in the Message Schema section (Section **22.3**).

SupportedTypes (Protocol 25): Basic Message-Response flow by ETP Role	
Message from customer	**Response Message from store**
GetSupportedTypes: Used to discover the types of data objects that a store instantiates or supports	***GetSupportedTypesResponse*** (multipart): An array of supported types that the store could return.

22.2.1.1 To discover data object types that are instantiated or supported in the store:

1. The customer sends the ***GetSupportedTypes*** message (Section **22.3.1**). This request message:
    a. MUST specify a URI from where the data types will be searched.
        i. If the URI is a dataspace URI (for example eml:///), then all datatypes supported in the dataspace are returned.
        ii. If the URI is a data object, then only datatypes that may have a link with the URI data object type are potentially returned.
    b. MUST specify the scope (Section **22.3.1**).
        i. If the URI is a data object, then the scope MUST be either "sources" or "targets". If the scope is NOT "sources" or "targets", the store MUST reject the request and send error EINVALID_OPERATION (32).
        ii. If the URI is not a data object, then this scope is ignored by the store.
    c. Includes an option to count the number of instances of each data object type that matches the request (*countObjects* MUST be set to true).
    d. Includes an option to see the entire list of supported types (include types of which the store does not have any instances). To do this, the *returnEmptyTypes* flag MUST be set. Otherwise only data objects that currently have data are returned.
2. The store MUST respond with a ***GetSupportedTypesResponse*** message (Section **22.3.2**) or a ***ProtocolException*** message.
    a. If the store does return supported types, it MUST send one or more ***GetSupportedTypesResponse*** messages, which are arrays of ***SupportedType*** records that the store supports and an optional count of each type.
        i. If the request URI (in the ***GetSupportedTypes*** request) was a dataspace, then the *relationshipKind* field in the ***SupportedType*** record MUST be set to "Primary" for all datatypes that are returned.

ii. If the request URI (in the ***GetSupportedTypes*** request) was a data object, then the *relationshipKind* field in the ***SupportedType*** record MUST be set to the appropriate value ("Primary" or "Secondary") for all datatypes that are returned.

iii. A store MUST limit the total count of responses to the customer's value for the MaxResponseCount protocol capability.

iv. If the store exceeds the customer's MaxResponseCount value, the customer MAY send error ERESPONSECOUNT_EXCEEDED (30).

v. If a store's MaxResponseCount value is less than the customer's MaxResponseCount value, the store MAY further limit the total count of responses (to its value).

vi. If a store cannot return all responses to a request because it would exceed the lower or the customer's or the store's value for MaxResponseCount, the store MUST terminate the multipart message with error ERESPONSECOUNT_EXCEEDED (30).

vii. A store MUST NOT send ERESPONSECOUNT_EXCEEDED (30) until it has sent MaxResponseCount responses.

b. If no supported types meet the criteria specified in the ***GetSupportedTypes*** message:

i. If the dataspace or data object specified by the URI in the context does not exist, the store MUST send error ENOT_FOUND (11).

ii. If the URI in the context exists, but no supported types could be found matching the request, the store MUST send the ***GetSupportedTypesResponse*** message with the FIN bit set and the *supportedTypes* field set to an empty array.

22.2.2 SupportedTypes: General Requirements

In addition to the basic message sequence described in the previous section, this protocol has the additional requirements listed in the table below. For easy reference, the rows and behaviors in this table are numbered.

NOTE: This table has been organized to reduce redundancy but also to help developers more easily locate specific requirements. **EXAMPLE:** There are rows with general requirements that apply to all protocols (e.g., Rows 1–2), rows for general requirements for this protocol, and (possibly) additional rows with additional requirements for specific types of operations.

Row#	Requirement	Behavior
1.	ETP-wide behavior that MUST be observed in all protocols	1. Requirements for general behavior consistent across all of ETP are defined in Chapter **3**. This behavior includes information such as: all details of message handling (such as message headers, handling compression, use of message IDs and correlation IDs, requirements for plural and multipart message patterns) use of acknowledgements, general rules for sending ***ProtocolException*** messages, URI encoding, serialization and more. **RECOMMENDATION: Read Chapter 3 first.** 2. For information about Energistics identifiers and prescribed ETP URI formats, see **Appendix: Energistics Identifiers**. a. In MOST cases, endpoints performing operations in this protocol MUST use the canonical Energistics URI. For more information, see Section **3.7.4**. 3. For the complete list of error codes defined by ETP, see Chapter **24**. 4. ALL operations in an ETP session are performed **on the set of supported data object types that were negotiated to be used when the session was initiated and established.** a. The client and server exchange the list of data object types in the *supportedDataObjects* field of the ***RequestSession*** and ***OpenSession*** messages in Core (Protocol 0). For more information, see Chapter **5**.

Row#	Requirement	Behavior
		b. In general, the list of supported objects for a session will most likely be the intersection of the data objects that the server supports and the data objects that the client requested for the ETP session. c. A store MUST support all messages (in each supported ETP sub-protocol) for each supported data object, whether the data object is supported explicitly or by wild card. d. An endpoint MUST only send messages with the URI of a data object that is a type supported by the other endpoint for this ETP session. i. If an endpoint sends a URI for an unsupported type of data object, the other endpoint MUST send error EDATAOBJECTTYPE_NOTSUPPORTED (16).
2.	Capabilities-related behavior	1. Relevant endpoint, data object, and/or protocol capabilities MUST be specified when the ETP session is established (see Chapter 5) and MUST be used/honored as defined in the relevant ETP sub-protocol. 2. For an explanation of endpoint, data object, and protocol capabilities, see Section 3.3. a. For the list of global capabilities and related behavior, see Section 3.3.2. 3. Section 22.2.3 identifies the capabilities most relevant to this ETP sub-protocol. Additional details for how to use the protocol capabilities are included below in this table and in Section **22.2.1 SupportedTypes: Message Sequence**.
3.	Message Sequence See Section 22.2.1.	1. The Message Sequence section above (Section 22.2.1) describes requirements for the main tasks listed there and also defines required behavior.
4.	Maps and plural message (which includes maps)	1. This protocol uses plural messages. For detailed rules on handling plural messages (including ***ProtocolException*** handling), see Section 3.7.3.
5.	Session negotiation: specify "all" for *supportedDataObjects*	1. For best results using this protocol, in the ***RequestSession*** message, in the *supportedDataObjects* field, the customer SHOULD specify "all" data objects (**EXAMPLE:** witsml20.*) a. For more information, see Row 1, Para **4 above** or the ***RequestSession*** message Section 5.3.

22.2.3 SupportedTypes: Capabilities

The table below lists key capabilities for this protocol. The protocol capabilities are all defined here.

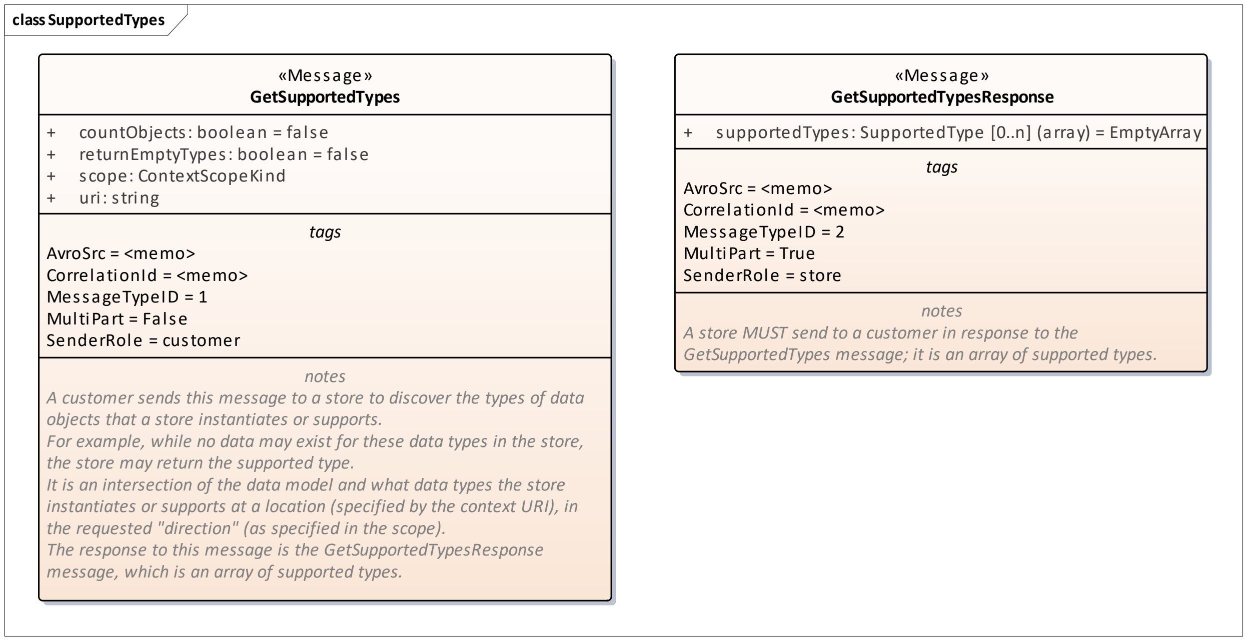
- For protocol-specific behavior relating to using these capabilities in this protocol, see Section **22.2.2 SupportedTypes: General Requirements**.
- For definitions for endpoint and data object capabilities, see the links in the table.
- For general information about the types of capabilities and how they may be used, see Section 3.3.

SupportedTypes (Protocol 25): Capabilities			
Name: Description	**Type**	**Units Value Units**	**Defaults and/or MIN/MAX**
Endpoint Capabilities (For definitions of each endpoint capability, see Section 3.3.)			
NOTE: Many endpoint capabilities are "universal", used in all or most of the ETP protocols. For more information, see Section 3.3.2.			

SupportedTypes (Protocol 25): Capabilities			
Name: Description	**Type**	**Units** **Value Units**	**Defaults and/or MIN/MAX**
Behavior associated with other endpoint capabilities are defined in relevant chapters. **EXAMPLE:** The capabilities defined for limiting ETP sessions between 2 endpoints are discussed in Section **4.3**, **How a Client Establishes a WebSocket Connection to an ETP** Server.			
Protocol Capabilities			
MaxResponseCount: The maximum total count of responses allowed in a complete multipart message response to a single request.	long	count <count of responses>	**MIN:** 10,000

22.3 SupportedTypes: Message Schemas

This section provides a figure that displays all messages defined in SupportedTypes (Protocol 25). Subsequent sub-sections provide an example schema for each message and definitions of the data fields contained in each message.



A customer sends this message to a store to discover the types of data objects that a store instantiates or supports.
For example, while no data may exist for these data types in the store, the store may return the supported type.
It is an intersection of the data model and what data types the store instantiates or supports at a location (specified by the context URI), in the requested "direction" (as specified in the scope).
The response to this message is the GetSupportedTypesResponse message, which is an array of supported types.

A store MUST send to a customer in response to the GetSupportedTypes message; it is an array of supported types.

Figure 32: SupportedTypes: message schemas

22.3.1 Message: GetSupportedTypes

A customer sends this message to a store to discover the types of data objects that a store instantiates or supports.

For example, while no data may exist for these data types in the store, the store may return the supported type.

It is an intersection of the data model and what data types the store instantiates or supports at a location (specified by the context URI), in the requested "direction" (as specified in the scope).

The response to this message is the GetSupportedTypesResponse message, which is an array of supported types.

Message Type ID: 1

Correlation Id Usage: MUST be ignored and SHOULD be set to 0.

Multi-part: False

Sent by: customer

Field Name	Description	Data Type	Min	Max
uri	The URI for the location in the data model where you want to begin discovering the instantiated or supported data types in a store. This MUST be a canonical Energistics URI. The URI MUST be a canonical Energistics data object or dataspace URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
scope	Scope is specified in reference to the URI (which is entered in the *uri* field). It indicates which direction in the graph that the operation should proceed (targets or sources).The enumerated values to choose from are specified in ContextScopeKind. For ***GetSupportedTypes***, the value MUST be either "sources" or "targets". For definitions of targets and sources, see Section **8.1.1**.	ContextScopeKind	1	1
returnEmptyTypes	Flag, if set to true, the store to returns data types that it supports but for which it currently has no data (no instance). Default is false.	boolean	1	1
countObjects	Flag, if set to true, the store provides counts for each supported type of resource identified. Default is false.	boolean	1	1

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.SupportedTypes",
     "name": "GetSupportedTypes",
     "protocol": "25",
     "messageType": "1",
     "senderRole": "customer",
     "protocolRoles": "store,customer",
     "multipartFlag": false,

     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "scope", "type": "Energistics.Etp.v12.Datatypes.Object.ContextScopeKind" },
         { "name": "returnEmptyTypes", "type": "boolean", "default": false },
         { "name": "countObjects", "type": "boolean", "default": false }
     ]
}
```

22.3.2 Message: GetSupportedTypesResponse

A store MUST send to a customer in response to the GetSupportedTypes message; it is an array of supported types.

Message Type ID: 2

Correlation Id Usage: MUST be set to the *messageId* of the ***GetSupportedTypes*** message that this message is a response to.

Multi-part: True

Sent by: store

Field Name	Description	Data Type	Min	Max
supportedTypes	An array of SupportedType records, of the types of data objects that the store instantiates or supports. If the request set the flag to true for *returnedEmptyTypes*, then the array includes types of data objects that the store supports but may currently have no data for.	SupportedType	0	n

Avro Source

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Protocol.SupportedTypes",
     "name": "GetSupportedTypesResponse",
     "protocol": "25",
     "messageType": "2",
     "senderRole": "store",
     "protocolRoles": "store,customer",
     "multipartFlag": true,
     "fields":
     [
         {
             "name": "supportedTypes",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.Object.SupportedType" }, "default": []
         }
     ]
}
```

23 ETP Datatypes

The datatypes package is intended to hold only low-level types that are broadly re-used in various protocols. In general, primitive datatypes follow the rules for Avro itself. These are the lower-level datatypes defined for the protocols. They are only used as fields of messages, not as messages in their own right.

For more information and definitions, see Section **3.4.1.1**.

Figure 33 shows examples of some frequently used datatypes and the messages (and other datatypes) that use those datatypes.

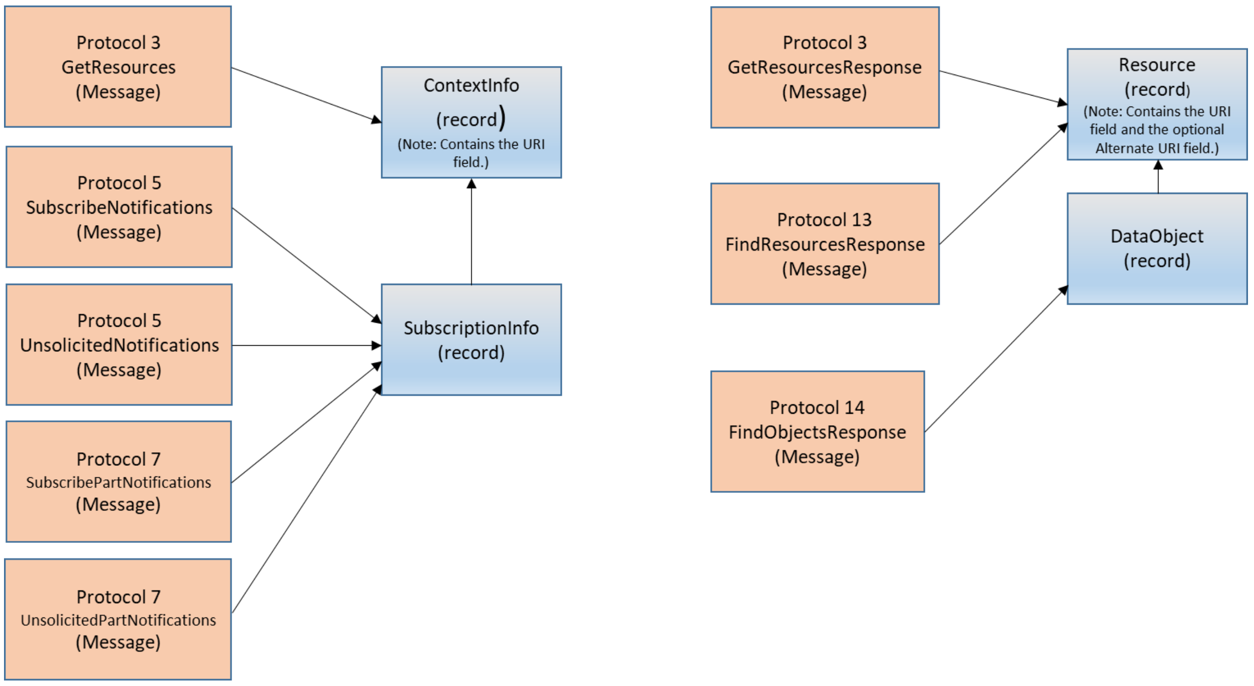
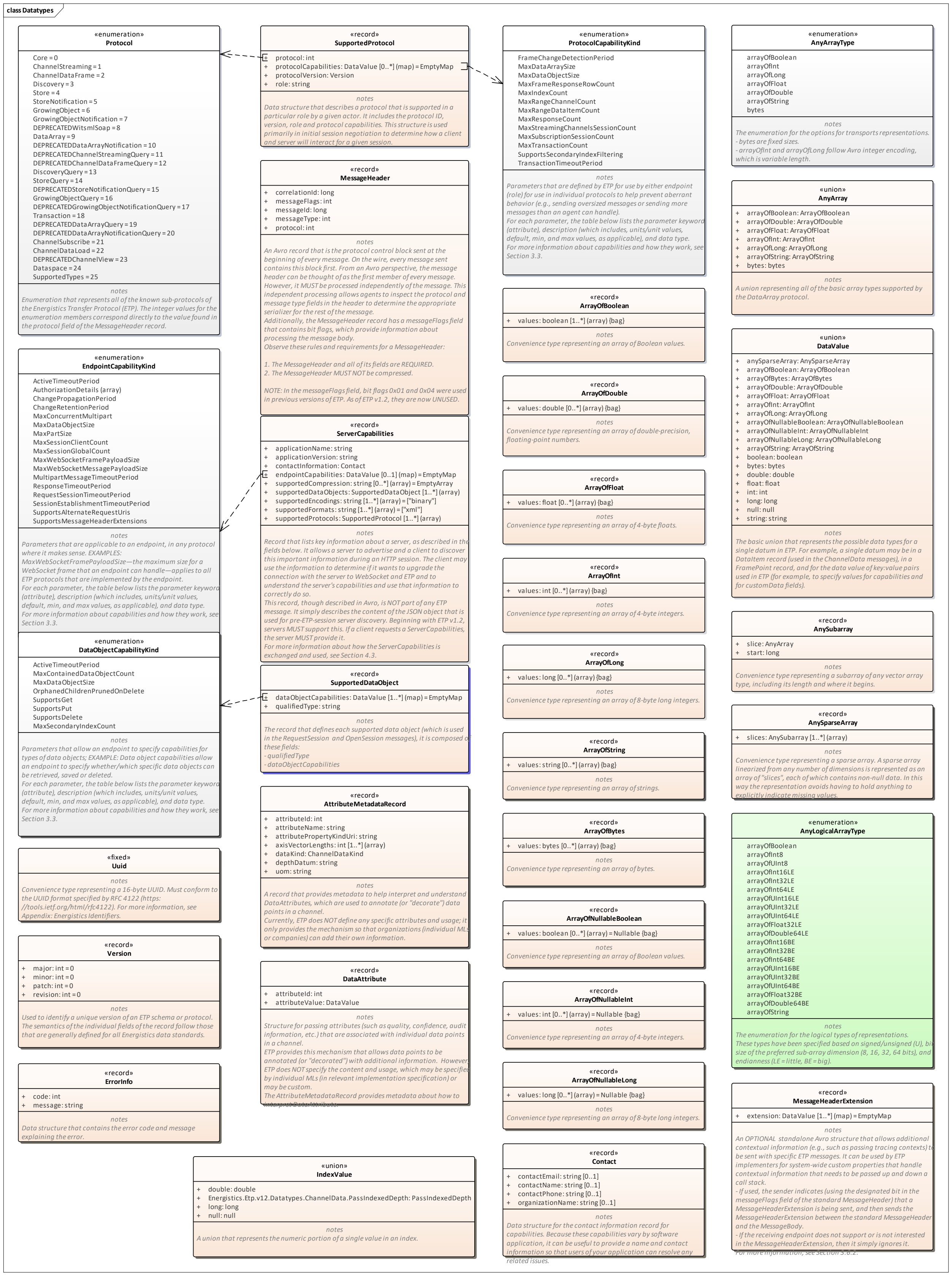

Protocol 3 GetResources (Message)
Protocol 5 SubscribeNotifications (Message)
Protocol 5 UnsolicitedNotifications (Message)
Protocol 7 SubscribePartNotifications (Message)
Protocol 7 UnsolicitedPartNotifications (Message)
ContextInfo (record) (Note: Contains the URI field.)
SubscriptionInfo (record)
Protocol 3 GetResourcesResponse (Message)
Protocol 13 FindResourcesResponse (Message)
Protocol 14 FindObjectsResponse (Message)
Resource (record) (Note: Contains the URI field and the optional Alternate URI field.)
DataObject (record)

Figure 33: Examples of ETP-defined datatypes (Avro records) that are used by multiple messages and other records.


class Datatypes

«enumeration»
Protocol
Core = 0
ChannelStreaming = 1
ChannelDataFrame = 2
Discovery = 3
Store = 4
StoreNotification = 5
GrowingObject = 6
GrowingObjectNotification = 7
DEPRECATEDWitsmlSoap = 8
DataArray = 9
DEPRECATEDDataArrayNotification = 10
DEPRECATEDChannelStreamingQuery = 11
DEPRECATEDChannelDataFrameQuery = 12
DiscoveryQuery = 13
StoreQuery = 14
DEPRECATEDStoreNotificationQuery = 15
GrowingObjectQuery = 16
DEPRECATEDGrowingObjectNotificationQuery = 17
Transaction = 18
DEPRECATEDDataArrayQuery = 19
DEPRECATEDDataArrayNotificationQuery = 20
ChannelSubscribe = 21
ChannelDataLoad = 22
DEPRECATEDChannelView = 23
Dataspace = 24
SupportedTypes = 25
notes
Enumeration that represents all of the known sub-protocols of the Energistics Transfer Protocol (ETP). The integer values for the enumeration members correspond directly to the value found in the protocol field of the MessageHeader record.

«enumeration»
EndpointCapabilityKind
ActiveTimeoutPeriod
AuthorizationDetails (array)
ChangePropagationPeriod
ChangeRetentionPeriod
MaxConcurrentMultipart
MaxDataObjectSize
MaxPartSize
MaxSessionClientCount
MaxSessionGlobalCount
MaxWebSocketFramePayloadSize
MaxWebSocketMessagePayloadSize
MultipartMessageTimeoutPeriod
ResponseTimeoutPeriod
RequestSessionTimeoutPeriod
SessionEstablishmentTimeoutPeriod
SupportsAlternateRequestUris
SupportsMessageHeaderExtensions
notes
Parameters that are applicable to an endpoint, in any protocol where it makes sense. EXAMPLES: MaxWebSocketFramePayloadSize—the maximum size for a WebSocket frame that an endpoint can handle—applies to all ETP protocols that are implemented by the endpoint.
For each parameter, the table below lists the parameter keyword (attribute), description (which includes, units/unit values, default, min, and max values, as applicable), and data type.
For more information about capabilities and how they work, see Section 3.3.

«enumeration»
DataObjectCapabilityKind
ActiveTimeoutPeriod
MaxContainedDataObjectCount
MaxDataObjectSize
OrphanedChildrenPrunedOnDelete
SupportsGet
SupportsPut
SupportsDelete
MaxSecondaryIndexCount
notes
Parameters that allow an endpoint to specify capabilities for types of data objects; EXAMPLE: Data object capabilities allow an endpoint to specify whether/which specific data objects can be retrieved, saved or deleted.
For each parameter, the table below lists the parameter keyword (attribute), description (which includes, units/unit values, default, min, and max values, as applicable), and data type.
For more information about capabilities and how they work, see Section 3.3.

«fixed»
Uuid
notes
Convenience type representing a 16-byte UUID. Must conform to the UUID format specified by RFC 4122 (https://tools.ietf.org/html/rfc4122). For more information, see Appendix: Energistics Identifiers.

«record»
Version
+ major: int = 0
+ minor: int = 0
+ patch: int = 0
+ revision: int = 0
notes
Used to identify a unique version of an ETP schema or protocol. The semantics of the individual fields of the record follow those that are generally defined for all Energistics data standards.

«record»
ErrorInfo
+ code: int
+ message: string
notes
Data structure that contains the error code and message explaining the error.

«record»
SupportedProtocol
+ protocol: int
+ protocolCapabilities: DataValue [0..*] (map) = EmptyMap
+ protocolVersion: Version
+ role: string
notes
Data structure that describes a protocol that is supported in a particular role by a given actor. It includes the protocol ID, version, role and protocol capabilities. This structure is used primarily in initial session negotiation to determine how a client and server will interact for a given session.

«record»
MessageHeader
+ correlationId: long
+ messageFlags: int
+ messageId: long
+ messageType: int
+ protocol: int
notes
An Avro record that is the protocol control block sent at the beginning of every message. On the wire, every message sent contains this block first. From an Avro perspective, the message header can be thought of as the first member of every message. However, it MUST be processed independently of the message. This independent processing allows agents to inspect the protocol and message type fields in the header to determine the appropriate serializer for the rest of the message.
Additionally, the MessageHeader record has a messageFlags field that contains bit flags, which provide information about processing the message body.
Observe these rules and requirements for a MessageHeader:

1. The MessageHeader and all of its fields are REQUIRED.
2. The MessageHeader MUST NOT be compressed.

NOTE: In the messageFlags field, bit flags 0x01 and 0x04 were used in previous versions of ETP. As of ETP v1.2, they are now UNUSED.

«record»
ServerCapabilities
+ applicationName: string
+ applicationVersion: string
+ contactInformation: Contact
+ endpointCapabilities: DataValue [0..1] (map) = EmptyMap
+ supportedCompression: string [0..*] (array) = EmptyArray
+ supportedDataObjects: SupportedDataObject [1..*] (array)
+ supportedEncodings: string [1..*] (array) = ["binary"]
+ supportedFormats: string [1..*] (array) = ["xml"]
+ supportedProtocols: SupportedProtocol [1..*] (array)
notes
Record that lists key information about a server, as described in the fields below. It allows a server to advertise and a client to discover this important information during an HTTP session. The client may use the information to determine if it wants to upgrade the connection with the server to WebSocket and ETP and to understand the server's capabilities and use that information to correctly do so.
This record, though described in Avro, is NOT part of any ETP message. It simply describes the content of the JSON object that is used for pre-ETP-session server discovery. Beginning with ETP v1.2, servers MUST support this. If a client requests a ServerCapabilities, the server MUST provide it.
For more information about how the ServerCapabilities is exchanged and used, see Section 4.3.

«record»
SupportedDataObject
+ dataObjectCapabilities: DataValue [1..*] (map) = EmptyMap
+ qualifiedType: string
notes
The record that defines each supported data object (which is used in the RequestSession and OpenSession messages), it is composed of these fields:
- qualifiedType
- dataObjectCapabilities

«record»
AttributeMetadataRecord
+ attributeId: int
+ attributeName: string
+ attributePropertyKindUri: string
+ axisVectorLengths: int [1..*] (array)
+ dataKind: ChannelDataKind
+ depthDatum: string
+ uom: string
notes
A record that provides metadata to help interpret and understand DataAttributes, which are used to annotate (or "decorate") data points in a channel.
Currently, ETP does NOT define any specific attributes and usage; it only provides the mechanism so that organizations (individual MLs or companies) can add their own information.

«record»
DataAttribute
+ attributeId: int
+ attributeValue: DataValue
notes
Structure for passing attributes (such as quality, confidence, audit information, etc.) that are associated with individual data points in a channel.
ETP provides this mechanism that allows data points to be annotated (or "decorated") with additional information. However, ETP does NOT specify the content and usage, which may be specified by individual MLs (in relevant implementation specification) or may be custom.
The AttributeMetadataRecord provides metadata about how to interpret DataAttribute.

«union»
IndexValue
+ double: double
+ Energistics.Etp.v12.Datatypes.ChannelData.PassIndexedDepth: PassIndexedDepth
+ long: long
+ null: null
notes
A union that represents the numeric portion of a single value in an index.

«enumeration»
ProtocolCapabilityKind
FrameChangeDetectionPeriod
MaxDataArraySize
MaxDataObjectSize
MaxFrameResponseRowCount
MaxIndexCount
MaxRangeChannelCount
MaxRangeDataItemCount
MaxResponseCount
MaxStreamingChannelsSessionCount
MaxSubscriptionSessionCount
MaxTransactionCount
SupportsSecondaryIndexFiltering
TransactionTimeoutPeriod
notes
Parameters that are defined by ETP for use by either endpoint (role) for use in individual protocols to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an agent can handle).
For each parameter, the table below lists the parameter keyword (attribute), description (which includes, units/unit values, default, min, and max values, as applicable), and data type.
For more information about capabilities and how they work, see Section 3.3.

«record»
ArrayOfBoolean
+ values: boolean [1..*] (array) {bag}
notes
Convenience type representing an array of Boolean values.

«record»
ArrayOfDouble
+ values: double [0..*] (array) {bag}
notes
Convenience type representing an array of double-precision, floating-point numbers.

«record»
ArrayOfFloat
+ values: float [0..*] (array) {bag}
notes
Convenience type representing an array of 4-byte floats.

«record»
ArrayOfInt
+ values: int [0..*] (array) {bag}
notes
Convenience type representing an array of 4-byte integers.

«record»
ArrayOfLong
+ values: long [0..*] (array) {bag}
notes
Convenience type representing an array of 8-byte long integers.

«record»
ArrayOfString
+ values: string [0..*] (array) {bag}
notes
Convenience type representing an array of strings.

«record»
ArrayOfBytes
+ values: bytes [0..*] (array) {bag}
notes
Convenience type representing an array of bytes.

«record»
ArrayOfNullableBoolean
+ values: boolean [0..*] (array) = Nullable {bag}
notes
Convenience type representing an array of Boolean values.

«record»
ArrayOfNullableInt
+ values: int [0..*] (array) = Nullable {bag}
notes
Convenience type representing an array of 4-byte integers.

«record»
ArrayOfNullableLong
+ values: long [0..*] (array) = Nullable {bag}
notes
Convenience type representing an array of 8-byte long integers.

«record»
Contact
+ contactEmail: string [0..1]
+ contactName: string [0..1]
+ contactPhone: string [0..1]
+ organizationName: string [0..1]
notes
Data structure for the contact information record for capabilities. Because these capabilities vary by software application, it can be useful to provide a name and contact information so that users of your application can resolve any related issues.

«enumeration»
AnyArrayType
arrayOfBoolean
arrayOfInt
arrayOfLong
arrayOfFloat
arrayOfDouble
arrayOfString
bytes
notes
The enumeration for the options for transports representations.
- bytes are fixed sizes.
- arrayOfInt and arrayOfLong follow Avro integer encoding, which is variable length.

«union»
AnyArray
+ arrayOfBoolean: ArrayOfBoolean
+ arrayOfDouble: ArrayOfDouble
+ arrayOfFloat: ArrayOfFloat
+ arrayOfInt: ArrayOfInt
+ arrayOfLong: ArrayOfLong
+ arrayOfString: ArrayOfString
+ bytes: bytes
notes
A union representing all of the basic array types supported by the DataArray protocol.

«union»
DataValue
+ anySparseArray: AnySparseArray
+ arrayOfBoolean: ArrayOfBoolean
+ arrayOfBytes: ArrayOfBytes
+ arrayOfDouble: ArrayOfDouble
+ arrayOfFloat: ArrayOfFloat
+ arrayOfInt: ArrayOfInt
+ arrayOfLong: ArrayOfLong
+ arrayOfNullableBoolean: ArrayOfNullableBoolean
+ arrayOfNullableInt: ArrayOfNullableInt
+ arrayOfNullableLong: ArrayOfNullableLong
+ arrayOfString: ArrayOfString
+ boolean: boolean
+ bytes: bytes
+ double: double
+ float: float
+ int: int
+ long: long
+ null: null
+ string: string
notes
The basic union that represents the possible data types for a single datum in ETP. For example, a single datum may be in a DataItem record (used in the ChannelData messages), in a FramePoint record, and for the data value of key:value pairs used in ETP (for example, to specify values for capabilities and for customData fields).

«record»
AnySubarray
+ slice: AnyArray
+ start: long
notes
Convenience type representing a subarray of any vector array type, including its length and where it begins.

«record»
AnySparseArray
+ slices: AnySubarray [1..*] (array)
notes
Convenience type representing a sparse array. A sparse array linearized from any number of dimensions is represented as an array of "slices", each of which contains non-null data. In this way the representation avoids having to hold anything to explicitly indicate missing values.

«enumeration»
AnyLogicalArrayType
arrayOfBoolean
arrayOfInt8
arrayOfUInt8
arrayOfInt16LE
arrayOfInt32LE
arrayOfInt64LE
arrayOfUInt16LE
arrayOfUInt32LE
arrayOfUInt64LE
arrayOfFloat32LE
arrayOfDouble64LE
arrayOfInt16BE
arrayOfInt32BE
arrayOfInt64BE
arrayOfUInt16BE
arrayOfUInt32BE
arrayOfUInt64BE
arrayOfFloat32BE
arrayOfDouble64BE
arrayOfString
notes
The enumeration for the logical types of representations. These types have been specified based on signed/unsigned (U), bit size of the preferred sub-array dimension (8, 16, 32, 64 bits), and endianness (LE = little, BE = big).

«record»
MessageHeaderExtension
+ extension: DataValue [1..*] (map) = EmptyMap
notes
An OPTIONAL standalone Avro structure that allows additional contextual information (e.g., such as passing tracing contexts) to be sent with specific ETP messages. It can be used by ETP implementers for system-wide custom properties that handle contextual information that needs to be passed up and down a call stack.
- If used, the sender indicates (using the designated bit in the messageFlags field of the standard MessageHeader) that a MessageHeaderExtension is being sent, and then sends the MessageHeaderExtension between the standard MessageHeader and the MessageBody.
- If the receiving endpoint does not support or is not interested in the MessageHeaderExtension, then it simply ignores it.
For more information, see Section 3.6.2.

Figure 34: Datatypes

23.1 AnyLogicalArrayType

The enumeration for the logical types of representations.

These types have been specified based on signed/unsigned (U), bit size of the preferred sub-array dimension (8, 16, 32, 64 bits), and endianness (LE = little, BE = big).

For more information about use of this enumeration, see Section 13.2.2.1.

Enumeration	Description
arrayOfBoolean	
arrayOfInt8	
arrayOfUInt8	
arrayOfInt16LE	
arrayOfInt32LE	
arrayOfInt64LE	
arrayOfUInt16LE	
arrayOfUInt32LE	
arrayOfUInt64LE	
arrayOfFloat32LE	
arrayOfDouble64LE	
arrayOfInt16BE	
arrayOfInt32BE	
arrayOfInt64BE	
arrayOfUInt16BE	
arrayOfUInt32BE	
arrayOfUInt64BE	
arrayOfFloat32BE	
arrayOfDouble64BE	
arrayOfString	
arrayOfCustom	

Avro Source

```
{
     "type": "enum",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "AnyLogicalArrayType",
     "symbols":
     [
         "arrayOfBoolean",
         "arrayOfInt8",
         "arrayOfUInt8",
         "arrayOfInt16LE",
         "arrayOfInt32LE",
         "arrayOfInt64LE",
         "arrayOfUInt16LE",
         "arrayOfUInt32LE",
         "arrayOfUInt64LE",
         "arrayOfFloat32LE",
         "arrayOfDouble64LE",
         "arrayOfInt16BE",
         "arrayOfInt32BE",
         "arrayOfInt64BE",
         "arrayOfUInt16BE",
         "arrayOfUInt32BE",
         "arrayOfUInt64BE",
         "arrayOfFloat32BE",
         "arrayOfDouble64BE",
          "arrayOfString",
          "arrayOfCustom"
     ]
}
```

23.2 AnyArrayType

The enumeration for the options for transports representations.

- bytes are fixed sizes.
- arrayOfInt and arrayOfLong follow Avro integer encoding, which is variable length.

Enumeration	Description
arrayOfBoolean	
arrayOfInt	
arrayOfLong	
arrayOfFloat	
arrayOfDouble	
arrayOfString	
bytes	

Avro Source

```
{
     "type": "enum",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "AnyArrayType",
     "symbols":
     [
          "arrayOfBoolean",
          "arrayOfInt",
          "arrayOfLong",
          "arrayOfFloat",
          "arrayOfDouble",
          "arrayOfString",
          "bytes"
     ]
}
```

23.3 DataObjectCapabilityKind

Parameters that allow an endpoint to specify capabilities for types of data objects; **EXAMPLE:** Data object capabilities allow an endpoint to specify whether/which specific data objects can be retrieved, saved or deleted.

For each parameter, the table below lists the parameter keyword (data object capability), description (which includes, units/unit values, default, as applicable), and data type.

For more information about capabilities and how they work, see Section **3.3**.

Data Object Capability	Description	Data Type
ActiveTimeoutPeriod	The minimum time period in seconds that a store keeps the active status (*activeStatus* field in ETP) for a data object as "active" after the most recent update causing the data object's active status to be set to true. For growing data objects, this is any change to its parts. For channels, this is any change to its data points. This capability can be set for an endpoint and/or for a data object. A data object-specific value overrides an endpoint-specific value. **Units/Value units:** seconds, <number of seconds>	long

Data Object Capability	Description	Data Type
	Min: 60 seconds **Default:** 3,600 seconds	
MaxContainedDataObjectCount	The maximum count of contained data objects allowed in a single instance of the data object type that the capability applies to. **EXAMPLE:** If this capability is set to 2000 for a ChannelSet, then the ChannelSet may contain a maximum of 2000 Channels. **Units/Value units:** count, <count of objects> **Min:** should be specified by the relevant domain	long
MaxDataObjectSize	The maximum size in bytes of a data object allowed in a complete multipart message. Size in bytes is the size in bytes of the uncompressed string representation of the data object in the format in which it is sent or received. This capability can be set for an endpoint, a protocol, and/or a data object. If set for all three, here is how they generally work: • An object-specific value overrides an endpoint-specific value. • A protocol-specific value can further lower (but NOT raise) the limit for the protocol. **EXAMPLE:** A store may wish to generally support sending and receiving any data object that is one megabyte or less with the exceptions of Wells that are 100 kilobytes or less and Attachments that are 5 megabytes or less. A store may further wish to limit the size of any data object sent as part of a notification in StoreNotification (Protocol 5) to 256 kilobytes. **Units/Value units:** bytes, <number of bytes> **Min:** 100,000 bytes	long
OrphanedChildrenPrunedOnDelete	For a container data object type (i.e., a data object type that may contain other data objects by value), this capability indicates whether contained data objects that are orphaned as a result of an operation on its container data object may be deleted (pruned). **NOTES:** 1. Both delete or put operations of a container data object may result in contained data objects being orphaned. 2. For successful pruning operations to occur on a specific data object type, both of these conditions MUST be true: * This capability MUST be set to true. * The *pruneContainedObjects* Boolean flag on the request message MUST be set to true. **EXAMPLE:** If this capability is set to true for ChannelSet, and on a ***DeleteDataObjects*** message (Store (Protocol 4) for a ChannelSet the *pruneContainedObjects* Boolean flag is set to true, and (after the container is deleted) a Channel in that ChannelSet belongs to no other ChannelSets, then that "orphaned" Channel is also deleted. **Default:** false	boolean
SupportsGet	Indicates whether get operations are supported for the data object type. **Default:** true	boolean
SupportsPut	Indicates whether put operations are supported for the data object type. If the operation can be technically supported by an endpoint, this capability should be true. **Default:** true	boolean
SupportsDelete	Indicates whether delete operations are supported for the data object type. If the operation can be technically supported by an endpoint, this capability should be true. **Default:** true	boolean
MaxSecondaryIndexCount	The maximum count of secondary indexes allowed in a single instance of the data object type that the capability applies to, which may be Channel or ChannelSet.	long

Data Object Capability	Description	Data Type
	Units/Value Units: count, <count of secondary indexes> **Default:** 1 **Min:** 1	

Avro Source

```
{
     "type": "enum",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "DataObjectCapabilityKind",
     "symbols":
     [
         "ActiveTimeoutPeriod",
         "MaxContainedDataObjectCount",
         "MaxDataObjectSize",
         "OrphanedChildrenPrunedOnDelete",
         "SupportsGet",
         "SupportsPut",
         "SupportsDelete",
         "MaxSecondaryIndexCount"
     ]
}
```

23.4 EndpointCapabilityKind

Parameters that are applicable to an endpoint, in any protocol where it makes sense. **EXAMPLES:** MaxWebSocketFramePayloadSize—the maximum size for a WebSocket frame that an endpoint can handle—applies to all ETP protocols that are implemented by the endpoint.

For each parameter, the table below lists the parameter keyword (endpoint capability), description (which includes, units/unit values, default, as applicable), and data type.

For more information about capabilities and how they work, see Section **3.3**.

Endpoint Capability	Description	Data Type
ActiveTimeoutPeriod	The minimum time period in seconds that a store keeps the active status (*activeStatus* field in ETP) for a data object as “active”, after the most recent update causing the data object’s active status to be set to true. For growing data objects, this is any change to its parts. For channels, this is any change to its data points. This capability can be set for an endpoint and/or for a data object. A data object-specific value overrides an endpoint-specific value. **Units/Value units:** seconds, <number of seconds> **Min:** 60 seconds **Default:** 3,600 seconds	long
AuthorizationDetails	1. Contains an ArrayOfString with WWW-Authenticate style challenges. 2. To support the required authorization workflow (to enable an endpoint to acquire an access token with the necessary scope from the designated authorization server), the AuthorizationDetails endpoint capability MUST include at least one challenge with the Bearer scheme which must include the ‘authz_server' and ‘scope’ parameters. a. The 'authz_server' parameter MUST be a URI for an authorization server to enable the endpoint to acquire any other needed metadata about the authorization server using OpenID Connect Discovery.	ArrayOfString

Endpoint Capability	Description	Data Type
	3. An ETP server MUST have the AuthorizationDetails endpoint capability, which must meet the requirements of Point 2 above. 4. If an ETP client does NOT need to authorize ETP servers, it MAY omit the AuthorizationDetails.	
ChangePropagationPeriod	The maximum time period in seconds--under normal operation on an uncongested session--for these conditions: • after a change in an endpoint before that endpoint sends a change notification covering the change to any subscribed endpoint in any session. • **if the change was the result of a message WITHOUT a positive response**, it is the maximum time until the change is reflected in read operations in any session. • **If the change was the result of a message WITH a positive response**, it is the maximum time until the change is reflected in sessions other than the session where the change was made. **RECOMMENDATION:** Set as short as possible (i.e. a few seconds). **Units/Value units:** seconds, <number of seconds> **Min:** 1 second **Max:** 600 seconds **Default:** 5 seconds	long
ChangeRetentionPeriod	The minimum time period in seconds that a store retains the Canonical URI of a deleted data object and any change annotations for channels and growing objects. **RECOMMENDATION:** This period should be as long as is feasible in an implementation. When the period is shorter, the risk is that additional data will need to be transmitted to recover from outages, leading to higher initial load on sessions. **Units/Value units:** seconds, <number of seconds> **Min:** 86,400 seconds **Default:** 86,400 seconds	long
MaxConcurrentMultipart	The maximum count of multipart messages allowed in parallel, in a single protocol, from one endpoint to another. The limit applies separately to each protocol, and separately from client to server and from server to client. The limit for an endpoint applies to the multipart messages that the endpoint can *receive*. **EXAMPLE:** If an endpoint's MaxConcurrentMultipart is 5, then it can receive 5 messages--each with any number of parts--at one time, in Store (Protocol 4) and another 5 messages in process in Discovery (Protocol 3). In Discovery (Protocol 3), this could be the multipart responses to 5 distinct ***GetResources*** request messages. **Units, Value Units:** count, <count of messages> **Min:** 1	long
MaxDataObjectSize	The maximum size in bytes of a data object allowed in a complete multipart message. Size in bytes is the size in bytes of the uncompressed string representation of the data object in the format in which it is sent or received. This capability can be set for an endpoint, a protocol, and/or a data object. If set for all three, here is how they generally work: • An object-specific value overrides an endpoint-specific value. • A protocol-specific value can further lower (but NOT raise) the limit for the protocol. **EXAMPLE:** A store may wish to generally support sending and receiving any data object that is one megabyte or less with the exceptions of Wells that are 100 kilobytes or less and	long

Endpoint Capability	Description	Data Type
	Attachments that are 5 megabytes or less. A store may further wish to limit the size of any data object sent as part of a notification in StoreNotification (Protocol 5) to 256 kilobytes. **Units/Value units:** bytes, <number of bytes> **Min:** 100,000 bytes	
MaxSessionClientCount	The maximum count of concurrent ETP sessions that may be established for a given endpoint, by a specific client. If possible, the determination of whether this limit is exceeded should be made at the time of receiving the HTTP WebSocket upgrade or connect request based on the authorization details provided with the request. At the latest, it should be based on an authorized ***RequestSession*** message. **Units/Value Units:** count, <count of sessions> **Min:** 2 sessions	long
MaxPartSize	The maximum size in bytes of each data object part allowed in a standalone message or a complete multipart message. Size in bytes is the total size in bytes of the uncompressed string representation of the data object part in the format in which it is sent or received. **Units/Value Units:** bytes, <number of bytes> **Min:** 10,000 bytes	long
MaxSessionGlobalCount	The maximum count of concurrent ETP sessions that may be established for a given endpoint across all clients. The determination of whether this limit is exceeded should be made at the time of receiving the HTTP WebSocket upgrade or connect request. **NOTE:** Exposing this information may have security implications, so it should only be exposed if an implementation is comfortable with any potential associated risks. **Units/Value Units:** count, <count of sessions> **Min:** 2 sessions	long
MaxWebSocketFramePayloadSize	The maximum size in bytes allowed for a single WebSocket frame payload. The limit to use during a session is the smaller of the client's and the server's value for MaxWebSocketFramePayloadSize, which should be determined by the limits imposed by the WebSocket library used by each endpoint. **Units/Value Units:** bytes, <number of bytes>	long
MaxWebSocketMessagePayloadSize	The maximum size in bytes allowed for a complete WebSocket message payload, which is composed of one or more WebSocket frames. The limit to use during a session is the smaller of the client's and the server's value for MaxWebSocketMessagePayloadSize, which should be determined by the limits imposed by the WebSocket library used by each endpoint. **Units/Value Units:** bytes, <number of bytes>	long
MultipartMessageTimeoutPeriod	The maximum time period in seconds--under normal operation on an uncongested session--allowed between subsequent messages in the SAME multipart request or response. The period is measured as the time between when each message has been fully sent or received via the WebSocket. **Units/Value Units:** seconds, <count of seconds> **Max:** 60 seconds	long
ResponseTimeoutPeriod	The maximum time period in seconds allowed between a request and the standalone response message or the first message in the multipart response message. The period is measured as the time between when the request message has been successfully sent via the WebSocket and when the first or only response message has been fully received via the WebSocket. When calculating this period, any Acknowledge messages or empty placeholder responses are ignored	long

Endpoint Capability	Description	Data Type
	EXCEPT where these are the only and final response(s) to the request. **Units/Value Units:** seconds, <number of seconds> **Min:** 60 seconds **Default:** 300 seconds	
RequestSessionTimeoutPeriod	The maximum time period in seconds a server will wait to receive a ***RequestSession*** message from a client after the WebSocket connection has been established. **Units/Value Units:** seconds, <number of seconds> **Min:** 5 seconds **Default:** 45 seconds	long
SessionEstablishmentTimeoutPeriod	The maximum time period in seconds a client or server will wait for a valid ETP session to be established. **For a server:** • A valid session is established when it sends an ***OpenSession*** message to the client, which indicates a session has been successfully established. • The time period starts when it receives the initial ***RequestSession*** message from the client. **For a client:** • A valid session is established when it receives an ***OpenSession*** message from the server. • The time period starts when it sends the initial ***RequestSession*** message to the server. **Units/Value Units:** seconds, <number of seconds> **Min:** 5 seconds **Default:** 60 seconds	long
SupportsAlternateRequestUris	Indicates whether an endpoint supports alternate URI formats--beyond the canonical Energistics URIs, which MUST be supported for requests. **Default:** false	boolean
SupportsMessageHeaderExtensions	Indicates whether an endpoint supports message header extensions. For more information about message header extensions and their use, see Section 3.6.2. **Default:** false	boolean

Avro Source

```
{
     "type": "enum",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "EndpointCapabilityKind",
     "symbols":
     [
         "ActiveTimeoutPeriod",
         "AuthorizationDetails",
         "ChangePropagationPeriod",
         "ChangeRetentionPeriod",
         "MaxConcurrentMultipart",
         "MaxDataObjectSize",
         "MaxPartSize",
         "MaxSessionClientCount",
         "MaxSessionGlobalCount",
         "MaxWebSocketFramePayloadSize",
         "MaxWebSocketMessagePayloadSize",
         "MultipartMessageTimeoutPeriod",
         "ResponseTimeoutPeriod",
         "SupportsAlternateRequestUris",
         "SupportsMessageHeaderExtensions",
         "RequestSessionTimeoutPeriod",
         "SessionEstablishmentTimeoutPeriod"
     ]
}
```

23.5 ProtocolCapabilityKind

Parameters that are defined by ETP for use by either endpoint (role) for use in individual protocols to help prevent aberrant behavior (e.g., sending oversized messages or sending more messages than an endpoint can handle).

For each parameter, the table below lists the parameter keyword (protocol capability), description (which includes, units/unit values, default, as applicable), and data type.

For more information about capabilities and how they work, see Section **3.3**.

Protocol Capability	Description	Data Type
FrameChangeDetectionPeriod	The maximum time period in seconds for updates to a channel to be visible in ChannelDataFrame (Protocol 2). Updates to channels are not guaranteed to be visible in responses in less than this period. (**EXAMPLE:** If your requested range includes rows that just received new data, the store may not return those rows. The store may be allowing time to potentially receive additional values for the rows before including them in responses.) The intent for this capability is that rows in ***ChannelDataframe*** messages are complete, and not 'partially updated'. ChannelDataFrame (Protocol 2) should not be used to poll for realtime data. **Units/Value Units:** seconds, <number of seconds> **Min:** 1 second **Max:** 600 seconds **Default:** 60 seconds	long
MaxDataArraySize	The maximum size in bytes of a data array allowed in a store. Size in bytes is the product of all array dimensions multiplied by the size in bytes of a single array element. **Units/Value Units:** bytes, <number of bytes> **Min:** 100,000 bytes	long
MaxDataObjectSize	The maximum size in bytes of a data object allowed in a complete multipart message. Size in bytes is the size in bytes of the uncompressed string representation of the data object in the format in which it is sent or received. This capability can be set for an endpoint, a protocol, and/or a data object. If set for all three, here is how they generally work: • An object-specific value overrides an endpoint-specific value. • A protocol-specific value can further lower (but NOT raise) the limit for the protocol. **EXAMPLE:** A store may wish to generally support sending and receiving any data object that is one megabyte or less with the exceptions of Wells that are 100 kilobytes or less and Attachments that are 5 megabytes or less. A store may further wish to limit the size of any data object sent as part of a notification in StoreNotification (Protocol 5) to 256 kilobytes. **Units/Value Units:** bytes, <number of bytes> **Min:** 100,000 bytes	long
MaxFrameResponseRowCount	The maximum total count of rows allowed in a complete multipart message response to a single request. **Units/Value Units:** count, <count of rows> **Min:** 100,000 rows	long

Protocol Capability	Description	Data Type
MaxIndexCount	The maximum index count value allowed for a channel streaming request. **Units/Value Units:** count, <count of indexes> **Min:** 1 index **Default:** 100 indexes	long
MaxRangeChannelCount	The maximum count of channels allowed in a single range request. **Units/Value Units:** count, <count of channels> **Min:** Should be equivalent of MaxContainedDataObjectCount for a ChannelSet	long
MaxRangeDataItemCount	The maximum total count of DataItem records allowed in a complete multipart range message. **Units/Value Units:** count, <count of records> **Min:** 1,000,000 records	long
MaxResponseCount	The maximum total count of responses allowed in a complete multipart message response to a single request. **Units/Value Units:** count, <count of responses> **Min:** 10,000 responses	long
MaxStreamingChannelsSessionCount	The maximum total count of channels allowed to be concurrently open for streaming in a session. The limit applies separately for each protocol with the capability. **EXAMPLE:** Different values can be specified for ChannelSubscribe (Protocol 21) and ChannelDataLoad (Protocol 22). **Units/Value Units:** count, <count of channels> **Min:** 10,000 channels	long
MaxSubscriptionSessionCount	The maximum total count of concurrent subscriptions allowed in a session. The limit applies separately for each protocol with the capability. **EXAMPLE:** Different values can be specified for StoreNotification (Protocol 5) and GrowingObjectNotification (Protocol 7). **Units/Value Units:** count, <count of subscriptions> **Min:** 100 subscriptions	long
MaxTransactionCount	The maximum count of transactions allowed in parallel in a session. **Units/Value Units:** count, <count of transactions> **Min:** 1 transaction **Max:** 1 transaction **Default:** 1 transaction	long
SupportsSecondaryIndexFiltering	Indicates whether an endpoint supports filtering requested data by secondary index values. If the filtering can be technically supported by an endpoint, this capability should be true. **Default:** false	boolean
TransactionTimeoutPeriod	The maximum time period in seconds allowed between receiving a ***StartTransactionResponse*** message and sending the corresponding ***CommitTransaction*** or ***RollbackTransaction*** request. **Units/Value Units:** seconds, <number of seconds> **Min:** 5 seconds	long

Avro Source

```
{
     "type": "enum",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "ProtocolCapabilityKind",
     "symbols":
     [
         "FrameChangeDetectionPeriod",
         "MaxDataArraySize",
         "MaxDataObjectSize",
         "MaxFrameResponseRowCount",
         "MaxIndexCount",
         "MaxRangeChannelCount",
         "MaxRangeDataItemCount",
         "MaxResponseCount",
         "MaxStreamingChannelsSessionCount",
         "MaxSubscriptionSessionCount",
         "MaxTransactionCount",
         "TransactionTimeoutPeriod",
         "SupportsSecondaryIndexFiltering"
     ]
}
```

23.6 fixed: Uuid

Convenience type representing a UUID as an array of 16 bytes. The format and byte order of the UUID MUST conform to RFC 4122 (https://tools.ietf.org/html/rfc4122). For more information, see **Appendix: Energistics Identifiers**.

Avro Schema

```
{
     "type": "fixed",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "Uuid",
     "size": 16
}
```

23.7 record: ArrayOfBoolean

Convenience type representing an array of Boolean values.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "ArrayOfBoolean",
     "fields":
     [
         {
             "name": "values",
             "type": { "type": "array", "items": "boolean" }
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
values	An array of Boolean values.	boolean	0	*

23.8 record: ArrayOfNullableBoolean

Convenience type representing an array of nullable Boolean values.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "ArrayOfNullableBoolean",
     "fields":
     [
         {
             "name": "values",
             "type": { "type": "array", "items": ["null", "boolean"] }
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
values	An array of nullable Boolean values.	boolean	0	*

23.9 record: ArrayOfInt

Convenience type representing an array of 4-byte integers.

Avro Schema

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Datatypes",
    "name": "ArrayOfInt",
    "fields":
    [
        {
            "name": "values",
            "type": { "type": "array", "items": "int" }
        }
    ]
}
```

Field Name	Description	Data Type	Min	Max
values	An array of integers.	int	0	*

23.10 record: ArrayOfNullableInt

Convenience type representing an array of nullable 4-byte integers.

Avro Schema

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Datatypes",
    "name": "ArrayOfNullableInt",
    "fields":
    [
        {
            "name": "values",
            "type": { "type": "array", "items": ["null", "int"] }
        }
    ]
}
```

Field Name	Description	Data Type	Min	Max
values	An array of nullable integers.	int	0	*

23.11 record: ArrayOfLong

Convenience type representing an array of 8-byte long integers.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "ArrayOfLong",
     "fields":
     [
         {
             "name": "values",
             "type": { "type": "array", "items": "long" }
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
values	An array of long integers.	long	0	*

23.12 record: ArrayOfNullableLong

Convenience type representing an array of nullable8-byte long integers.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "ArrayOfNullableLong",
     "fields":
     [
         {
             "name": "values",
             "type": { "type": "array", "items": ["null", "long"] }
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
values	An array of nullable long integers.	long	0	*

23.13 record: ArrayOfFloat

Convenience type representing an array of 4-byte floats. NaN should be used to represent null values.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "ArrayOfFloat",
     "fields":
     [
         {
             "name": "values",
             "type": { "type": "array", "items": "float" }
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
values	An array of single-precision numbers.	float	0	*

23.14 record: ArrayOfDouble

Convenience type representing an array of double-precision, floating-point numbers. NaN should be used to represent null values.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "ArrayOfDouble",
     "fields":
     [
         {
             "name": "values",
             "type": { "type": "array", "items": "double" }
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
values	An array of double precision numbers.	double	0	*

23.15 record: ArrayOfString

Convenience type representing an array of strings. Empty strings should be used to represent null values.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "ArrayOfString",
     "fields":
     [
         {
             "name": "values",
             "type": { "type": "array", "items": "string" }
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
values	An array of strings.	string	0	*

23.16 record: ArrayOfBytes

Convenience type representing an array of byte arrays (i.e., blobs).

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "ArrayOfBytes",
     "fields":
     [
         {
             "name": "values",
             "type": { "type": "array", "items": "bytes" }
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
values	An array of byte arrays (i.e., blobs).	bytes	0	*

23.17 record: AnySparseArray

Convenience type representing a sparse array. A sparse array linearized from any number of dimensions is represented as an array of "slices", each of which contains non-null data. In this way the representation avoids having to hold anything to explicitly indicate missing values.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "AnySparseArray",
     "fields":
     [
         {
             "name": "slices",
             "type": { "type": "array", "items": "Energistics.Etp.v12.Datatypes.AnySubarray" }
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
slices	A slice is a sub-part pf a sparse array which contains non-null data.	AnySubarray	1	*

23.18 record: AnySubarray

Convenience type representing a subarray of any vector array type linearized from any number of dimensions, including its length and where it begins.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "AnySubarray",
     "fields":
     [
         { "name": "start", "type": "long" },
         { "name": "slice", "type": "Energistics.Etp.v12.Datatypes.AnyArray" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
start	The index of the start of a subarray within its parent array's linearized indexing scheme.	long	1	1
slice	A linearized slice of data within the parent array.	AnyArray	1	1

23.19 record: ServerCapabilities

Record that lists key information about a server, as described in the fields below. It allows a server to advertise and a client to discover this important information during an HTTP session. The client may use the information to determine if it wants to upgrade the connection with the server to WebSocket and ETP and to understand the server's capabilities and use that information to correctly do so.

This record is NOT part of any ETP message. It is used for pre-ETP-session server discovery. Beginning with ETP v1.2, servers MUST support this. If a client requests a ServerCapabilities, the server MUST provide it.

For more information about how the ServerCapabilities is exchanged and used, see Section **4.3**.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "ServerCapabilities",
     "fields":
     [
         { "name": "applicationName", "type": "string" },
         { "name": "applicationVersion", "type": "string" },
         { "name": "contactInformation", "type": "Energistics.Etp.v12.Datatypes.Contact" },
         {
             "name": "supportedCompression",
             "type": { "type": "array", "items": "string" }, "default": []
         },
         {
             "name": "supportedEncodings",
             "type": { "type": "array", "items": "string" }, "default": ["binary"]
         },
         {
             "name": "supportedFormats",
             "type": { "type": "array", "items": "string" }, "default": ["xml"]
         },
         {
             "name": "supportedDataObjects",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.SupportedDataObject" }
         },
         {
             "name": "supportedProtocols",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.SupportedProtocol" }
         },
         {
             "name": "endpointCapabilities",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
applicationName	The string by which the server identifies itself, normally a software product or system name. The name may or may not include a version. The format is entirely application dependent. Vendors are encouraged to identify their company name as part of this string.	string	1	1
applicationVersion	The version of the application identified in *applicationName.*	string	1	1
contactInformation	The email and phone number of the organization/person to contact for questions or issues with this application.	Contact	1	1
supportedCompression	An array of compression algorithms supported by the server. An empty array indicates compression is not supported. **EXAMPLES:** "gzip" or "gzip, deflate"	string	0	*
supportedEncodings	The encodings that a server supports for transport of ETP messages on the wire. The allowed encodings: • binary	string	1	*

Field Name	Description	Data Type	Min	Max
	• json • binary;json			
supportedFormats	An array of data formats supported by the server, in order of preference. The format(s) are used when sending data objects or growing data object parts in ETP messages. They the receiver of messages know how to deserialize the array of bytes representing the data object or growing data object part in the message. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats. **Default:** xml **NOTE:** An endpoint indicates in the message, per request and response, which format it wants to use or is being used.	string	1	*
supportedDataObjects	A list of data objects that the server supports for any of the ETP sub-protocols it supports and the data object capabilities for each with the endpoint's values for each, which is defined in the SupportedDataObject record. For the list of data object capabilities defined by ETP, see DataObjectCapabilityKind.	SupportedDataObject	1	*
supportedProtocols	The list of identifiers of the ETP sub-protocols supported by the server and the protocol capabilities for each with the endpoint's value for each capability, as defined on the SupportedProtocol record. If the server is able to support both roles in a protocol, the protocol will appear twice in the list, once with each supported role. For the list of protocol capabilities defined by ETP, see ProtocolCapabilityKind.	SupportedProtocol	1	*
endpointCapabilities	A map of key-value pairs of endpoint-specific capability data (i.e., constraints, limitations). The names, defaults, optionality, and expected data types are defined by this specification. These endpoint capabilities are the ones that will normally be provided by the server in ***OpenSession*** messages. • Map keys are capability names, which are case-sensitive strings. For ETP-defined capabilities, the name must spelled exactly as listed in EndpointCapabilityKind. • Map values are of type DataValue. • For more information about capabilities and rules for using them, see Section **3.3**.	DataValue	0	1

23.20 record: SupportedDataObject

The record that defines each supported data object (which is used in the RequestSession and OpenSession messages), it is composed of these fields:

- qualifiedType
- dataObjectCapabilities

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "SupportedDataObject",
     "fields":
     [
         { "name": "qualifiedType", "type": "string" },
         {
             "name": "dataObjectCapabilities",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
qualifiedType	This must be a value of dataObjectType as described in Appendix: Energistics Identifiers and serialized as JSON. It is the semantic equivalent of a qualifiedEntityType in OData. They ARE case sensitive. EXAMPLES: "witsml20.Well", "witsml20.Wellbore", "prodml21.WellTest", "resqml20.obj_TectonicBoundaryFeature" "eml21.DataAssuranceRecord" To indicate that all data objects within a data schema version are supported, you can use a star (*) as a wildcard, EXAMPLE: "witsml20.*", "prodml21.*", "resqml20.*", So "witsml20.*" means all data objects defined by WITSML v2.0 data schemas.	string	1	1
dataObjectCapabilities	A map of key-value pairs that allow an endpoint to specify capabilities, parameters, and limits for types of data objects. • Map keys are capability names, which are case-sensitive strings. For ETP-defined capabilities, the name must be spelled exactly as listed in DataObjectCapabilityKind. • Map values are of type DataValue. For more information about capabilities and rules for using them, see Section **3.3**.	DataValue	1	*

23.21 record: SupportedProtocol

Record that describes a protocol that is supported in a particular role by a given actor. It includes the protocol ID, version, role and protocol capabilities. This structure is used primarily in initial session negotiation to determine how a client and server will interact for a given session.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "SupportedProtocol",
     "fields":
```

```
     [
         { "name": "protocol", "type": "int" },
         { "name": "protocolVersion", "type": "Energistics.Etp.v12.Datatypes.Version" },
         { "name": "role", "type": "string" },
         {
             "name": "protocolCapabilities",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
protocol	The ID of the protocol that this ***SupportedProtocol*** record represents, as defined by this specification or the relevant custom protocol. The value is an integer value rather than an enumerated value to allow custom protocols to be supported.	int	1	1
protocolVersion	The specific version of the protocol to be used.	Version	1	1
role	Most of the supported protocols involve two mutually exclusive roles: store and customer. (ChannelStreaming (Protocol 1) uses producer and consumer.) The values expected for this string are defined by each sub-protocol and included as decorations in the Avro schemas. Values are case-sensitive and, by modeling convention, should be all lower case in the specification and Avro schema files.	string	1	1
protocolCapabilities	A map of key-value pairs of protocol-specific configuration or capability data used to prevent aberrant behavior. The names, defaults, optionality, and expected data types may be defined by each protocol. • Map keys are capability names, which are case-sensitive strings. For ETP-defined capabilities, the name must spelled exactly as listed in ProtocolCapabilityKind. • Map values are of type DataValue. • For more information about capabilities and rules for using them, see Section **3.3**.	DataValue	0	*

23.22 record: Version

Used to identify a unique version of an ETP schema or protocol. The semantics of the individual fields of the record follow those that are generally defined for all Energistics data standards.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "Version",
     "fields":
     [
         { "name": "major", "type": "int", "default": 0 },
         { "name": "minor", "type": "int", "default": 0 },
         { "name": "revision", "type": "int", "default": 0 },
         { "name": "patch", "type": "int", "default": 0 }
     ]
}
```

Field Name	Description	Data Type	Min	Max
major	Involves significant change to all schemas, protocols, and business rules of a specification.	int	1	1
minor	Includes significant changes to schemas, most probably with breaking changes. The overall protocols and approach should not change significantly.	int	1	1
revision	May contain additions to existing schemas but does not remove any schema elements. Enumerated types may also change.	int	1	1
patch	Involves minor changes only, usually bug fixes, and should not create breaking changes for other clients and servers on the same version.	int	1	1

23.23 record: DataAttribute

Record for passing attributes (such as quality, confidence, audit information, etc.) that are associated with individual data points in a channel.

ETP provides this mechanism that allows data points to be annotated (or "decorated") with additional information. However, ETP does NOT specify the content and usage, which may be specified by individual MLs (in relevant implementation specification) or may be custom.

The AttributeMetadataRecord provides metadata about how to interpret DataAttribute.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "DataAttribute",
     "fields":
     [
         { "name": "attributeId", "type": "int" },
         { "name": "attributeValue", "type": "Energistics.Etp.v12.Datatypes.DataValue" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
attributeId	The identifier for the attribute, as received in the AttributeMetadataRecord..	int	1	1
attributeValue	The value of an attribute for the given ID, which must be of type DataValue. The AttributeMetadataRecord can be used to specify relevant information about this field.	DataValue	1	1

23.24 record: AttributeMetadataRecord

A record that provides metadata to help interpret and understand DataAttributes, which are used to annotate (or "decorate") data points in a channel.

Currently, ETP does NOT define any specific attributes and usage; it only provides the mechanism so that organizations (individual MLs or companies) can add their own information.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "AttributeMetadataRecord",
     "fields":
     [
         { "name": "attributeId", "type": "int" },
         { "name": "attributeName", "type": "string" },
         { "name": "dataKind", "type":
"Energistics.Etp.v12.Datatypes.ChannelData.ChannelDataKind" },
         { "name": "uom", "type": "string" },
         { "name": "depthDatum", "type": "string" },
         { "name": "attributePropertyKindUri", "type": "string" },
         {
             "name": "axisVectorLengths",
             "type": { "type": "array", "items": "int" }
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
attributeName	The name of the attribute.	string	1	1
attributeId	The identifier assigned to this attribute for this ETP session. Each attribute associated with a channel must have a unique identifier within a session, but attribute identifiers may be reused across channels. For example, no channel may have 2 attributes with attributeId = 3, but 2 different channels may each have 1 attribute with attributeId = 3.	int	1	1
dataKind	The kind of data contained in the channel, which must be one of the values in ChannelDataKind.	ChannelDataKind	1	1
uom	The unit of measure for the attribute, if applicable.	string	1	1
depthDatum	If the attribute data is a depth value, this is the datum it references.	string	1	1
attributePropertyKindUri	An optional field that allows an endpoint to specify the URI of a property kind data object, which MUST be available from the endpoint and MAY be from the Energistics PropertyKindDictionary. Use of this field lets an endpoint indicate more specifically what the attribute is representing. The PropertyKindDictionary is an implementation of the Practical Well Log Standard (PWLS). For more information on how PWLS is used in ETP, see Section **3.12.7**.	string	1	1
axisVectorLengths	If the metadata values are arrays, then this field MUST be populated. It provides the context for how to decode a flattened 1D array back into the higher dimension array. **Rules for populating this field:** • You MUST encode the positional information as an absolute 'start' offset between it and the length of each subarray that Avro will encode onto the wire. • The number of elements in the array indicates the number of dimensions in the data. • The elements in the array indicate the length of each dimension. **Rules for ordering:** • Slowest to fastest. • Index 0 is the slowest moving dimension. • Last index is the fastest moving dimension. **EXAMPLE:** If you a have a 2D array of 3 rows and 20 columns, then this field would contain: 3,20	int	1	*

23.25 record: MessageHeader

A record that is the protocol control block sent at the beginning of every message. On the wire, every message sent contains this block first. From an Avro perspective, the message header can be thought of as the first member of every message. However, it MUST be processed independently of the message. This independent processing allows agents to inspect the protocol and message type fields in the header to determine the appropriate serializer for the rest of the message.

Additionally, the ***MessageHeader*** record has a *messageFlags* field that contains bit flags, which provide information about processing the message body.

Observe these rules and requirements for a ***MessageHeader***:

1. The ***MessageHeader*** and all of its fields are REQUIRED.
2. The ***MessageHeader*** MUST NOT be compressed.

NOTE: In the *messageFlags* field, bit flags 0x01 and 0x04 were used in previous versions of ETP. As of ETP v1.2, they are now UNUSED.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "MessageHeader",
     "fields":
     [
         { "name": "protocol", "type": "int" },
         { "name": "messageType", "type": "int" },
         { "name": "correlationId", "type": "long" },
         { "name": "messageId", "type": "long" },
         { "name": "messageFlags", "type": "int" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
protocol	The ETP sub-protocol number in which this message is defined or used. **EXAMPLES:** Core (Protocol 0), ChannelStreaming (Protocol 1), ChannelDataFrame (Protocol 2), Discovery (Protocol 3), etc.). In MOST cases, the protocol number where a message is defined is also the only place that message may be used. **EXCEPTIONS:** So-called ETP "universal" messages--***ProtocolException*** and ***Acknowledge***--are defined in Core (Protocol 0) but MAY be used in ANY ETP sub-protocol. When these universal messages are used, this *protocol* field is populated with the ETP sub-protocol number that was responsible for sending the message. **EXAMPLE:** If an error resulting in an endpoint sending a ***ProtocolException*** occurs in Discovery (Protocol 3), then its *protocol* field = 3.	int	1	1
messageType	Contains the enumerated, protocol-specific value for the accompanying message, which implicitly defines the schema for the message body. ETP identifies message types by assigning an integer to each unique message in a sub-protocol (**EXAMPLE:** In Protocol 0, ***RequestSession*** is message type 1 and ***OpenSession*** is message type 2. Thus a client or server can read the message type from the message header and know which schema proxy to use to decode the rest of the message.	int	1	1
correlationId	Used to allow servers and clients to match related messages. **EXAMPLES:**	long	1	1

Field Name	Description	Data Type	Min	Max
	• A response message MUST have as its *correlationId* the *messageId* of the request message it is responding to. • A ProtocolException message MUST have as its *correlationId* the number of the message that caused the exception to be raised. Each individual message defined in ETP provides guidance for setting its *correlationId*.			
messageId	The unique identifier for a message within an ETP session and endpoint. ETP requires that messages within an ETP session and endpoint each be uniquely numbered. • A *messageId* of "0" is invalid. **Message IDs MUST be:** • Unique within a session, and for a given endpoint (i.e., client/server). The IDs used by clients and servers are completely independent of one another. Put another way, the 'primary key' of any given message could be thought of as endpointType + messageId. • To help with de-bugging and problem solving, ETP has adopted this *messageId* numbering convention, which endpoints MUST observe: * the client side of the connection MUST use ONLY non-zero **even-numbered** message IDs. * the server side of the connection MUST use ONLY non-zero **odd-numbered** message IDs. **NOTE:** Message IDs ARE NOT required to be sequential or for any correlation between message IDs and any particular sub-protocol.	long	1	1
messageFlags	A bit field of flags that apply to a message. When an endpoint receives a ***MessageHeader***, it MUST inspect these flags and it MUST use the provided information for processing and/or perform the requested action (see Section 3.5.4). The bit field of flags has the following bits defined (as hexadecimal values): • 0x01: UNUSED. • 0x02: Serves as a "finish bit" (FIN bit) to indicate the "end" of a request, response, notification or data message. Observe these rules for setting the FIN bit: 1) It MUST be set by the sender role only. 2) For multipart messages (requests, responses and notifications), it MUST be set on the last message in the multipart message. For messages that are not multipart (i.e., they only have a single ETP message), it MUST be set on the single message. **EXAMPLES:** a) If an ETP server is streaming ***ChannelData*** messages; set this flag on each message. b) If a request or response is composed of only 1 ETP message, set this flag. c) If a request or response is composed of a set of related messages, set this flag on the last message of the set only (e.g., for a request composed of 5 messages, set this flag on the 5th message only.) For more information on how to set this flag for multipart requests and responses, see Section **3.7.3**. • 0x04: UNUSED. • 0x08: Indicates that the body of this message is compressed. It may be set by either role. The body of any message (except those explicitly excluded in Protocol 0), can be compressed, regardless of role, based on the compression encoding negotiated during initialization of the ETP session. • 0x10: Indicates that the sender is requesting an ***Acknowledge*** message receipt of the message. An	int	1	1

Field Name	Description	Data Type	Min	Max
	Acknowledge message MUST NOT have this bit flag set. • 0x20: Indicates that the sender is sending an optional MessageHeaderExtension (after the ***MessageHeader*** and before sending the message body). For more information, see Section **3.6.2**.			

23.26 record: MessageHeaderExtension

An OPTIONAL standalone record that allows additional contextual information (e.g., such as passing tracing contexts) to be sent with specific ETP messages. It can be used by ETP implementers for system-wide custom properties that handle contextual information that needs to be passed up and down a call stack.

- If used, the sender indicates (using the designated bit in the messageFlags field of the standard MessageHeader) that a MessageHeaderExtension is being sent, and then sends the MessageHeaderExtension between the standard MessageHeader and the MessageBody.
- If the receiving endpoint does not support or is not interested in the MessageHeaderExtension, then it simply ignores it.

For more information, see Section **3.6.2**.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "MessageHeaderExtension",
     "fields":
     [
         {
             "name": "extension",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
extension	A map of key/value pairs where the value must be a type in DataValue, indicating the extension and its value. • Map keys are case sensitive. • Any key that a receiver does not understand MUST be ignored (i.e.., no error). • Use of properties are either protocol-specific or refer to things OUTSIDE of the ETP Specification (such as tracing).	DataValue	1	*

23.27 record: Contact

A record for the contact information record for capabilities. Because these capabilities vary by software application, it can be useful to provide a name and contact information so that users of your application can resolve any related issues.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "Contact",
     "fields":
     [
         { "name": "organizationName", "type": "string", "default": "" },
         { "name": "contactName", "type": "string", "default": "" },
         { "name": "contactPhone", "type": "string", "default": "" },
         { "name": "contactEmail", "type": "string", "default": "" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
organizationName	The name of your company or organization within a company.	string	0	1
contactName	Name of the person to contact.	string	0	1
contactPhone	Phone number of the person to contact.	string	0	1
contactEmail	Email address of the person to contact.	string	0	1

23.28 record: ErrorInfo

A record that contains the error code and message explaining the error.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes",
     "name": "ErrorInfo",
     "fields":
     [
         { "name": "message", "type": "string" },
         { "name": "code", "type": "int" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
message	Text explaining the nature of the error and any additional information that an application chooses to provide. Chapter 24 identifies human-readable error names, numbers, and usage notes. These names and notes are informational only; they are used in the documentation but have no meaning on the wire. Implementers may want to use these names as a #define or constant name in their code, but this is not part of the specification. Additionally, use the name and/or notes as part of the text description in this field.	string	1	1
code	The error code. Positive error codes MUST be one of the error codes defined by Energistics. The error codes currently defined by Energistics are in Chapter **24**. New error codes may be defined in future versions of this specification or in ML implementation guides. Custom error codes MUST be negative.	int	1	1

23.29 union: AnyArray

A union representing all of the basic array types supported by DataArray (Protocol 9).

Field Name	Description	Data Type	Min	Max
arrayOfBoolean	Array of Boolean values.	ArrayOfBoolean	1	1
arrayOfInt	Array of signed 32-bit integers	ArrayOfInt	1	1
arrayOfLong	Array of signed 64-bit integers.	ArrayOfLong	1	1
arrayOfFloat	Array of 4-byte floats.	ArrayOfFloat	1	1
arrayOfDouble	Array of 8-byte floats.	ArrayOfDouble	1	1
arrayOfString		ArrayOfString	1	1
bytes	Array of bytes.	bytes	1	1

23.30 union: DataValue

The basic union that represents the possible data types for a single data point in ETP. For example, a single data point may be in a DataItem record (used in the ChannelData messages), in a FramePoint record, and for the data value of key:value pairs used in ETP (for example, to specify values for capabilities and for *customData* fields).

Field Name	Description	Data Type	Min	Max
null	Avro null	null	1	1
boolean	Avro Boolean	boolean	1	1
int	Avro int	int	1	1
long	Avro long	long	1	1
float	Avro float	float	1	1
double	Avro double	double	1	1
string	Avro string. NOTE: In ETP, all strings MUST use UTF-8 encoding.	string	1	1
arrayOfBoolean		ArrayOfBoolean	1	1
arrayOfNullableBoolean		ArrayOfNullableBoolean	1	1
arrayOfInt		ArrayOfInt	1	1
arrayOfNullableInt		ArrayOfNullableInt	1	1
arrayOfLong		ArrayOfLong	1	1
arrayOfNullableLong		ArrayOfNullableLong	1	1
arrayOfFloat		ArrayOfFloat	1	1
arrayOfDouble		ArrayOfDouble	1	1
arrayOfString		ArrayOfString	1	1
arrayOfBytes		ArrayOfBytes	1	1
bytes		bytes	1	1
anySparseArray		AnySparseArray	1	1

23.31 union: IndexValue

A union that represents the numeric portion of a single value in an index.

Field Name	Description	Data Type	Min	Max
null	Used to indicate infinity, for example, if you are specifying an interval (see IndexInterval).	null	1	1

Field Name	Description	Data Type	Min	Max
long	Avro long	long	1	1
PassIndexedDepth	Index type for a wireline operation, as defined in PassIndexedDepth.	PassIndexedDepth	1	1
double	Avro double	double	1	1

23.32 DataArrayTypes

This section contains low-level types used for DataArray (Protocol 9).

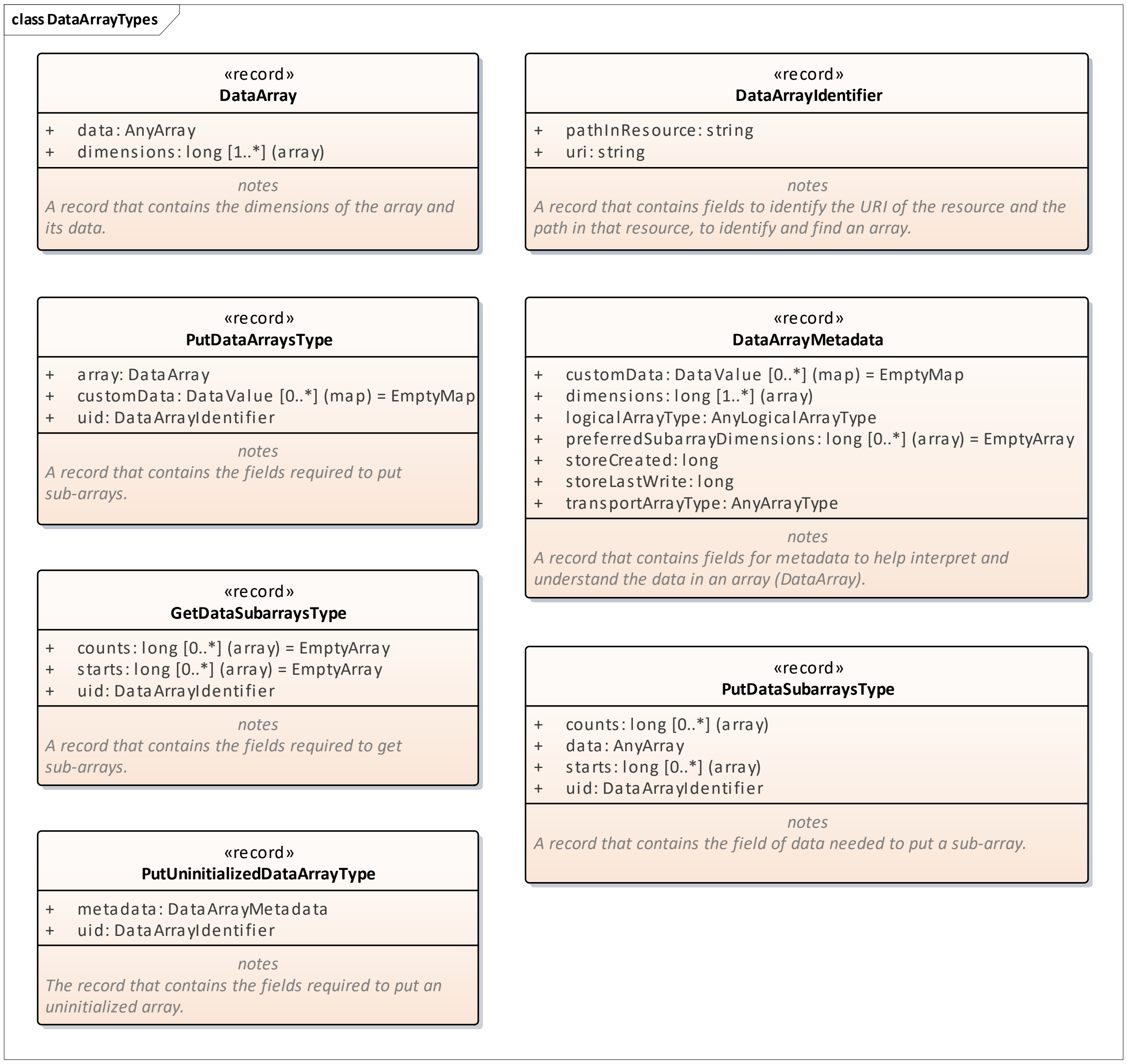

class DataArrayTypes
«record»
DataArray
+ data: AnyArray
+ dimensions: long [1..*] (array)
notes
A record that contains the dimensions of the array and its data.
«record»
DataArrayIdentifier
+ pathInResource: string
+ uri: string
notes
A record that contains fields to identify the URI of the resource and the path in that resource, to identify and find an array.
«record»
PutDataArraysType
+ array: DataArray
+ customData: DataValue [0..*] (map) = EmptyMap
+ uid: DataArrayIdentifier
notes
A record that contains the fields required to put sub-arrays.
«record»
DataArrayMetadata
+ customData: DataValue [0..*] (map) = EmptyMap
+ dimensions: long [1..*] (array)
+ logicalArrayType: AnyLogicalArrayType
+ preferredSubarrayDimensions: long [0..*] (array) = EmptyArray
+ storeCreated: long
+ storeLastWrite: long
+ transportArrayType: AnyArrayType
notes
A record that contains fields for metadata to help interpret and understand the data in an array (DataArray).
«record»
GetDataSubarraysType
+ counts: long [0..*] (array) = EmptyArray
+ starts: long [0..*] (array) = EmptyArray
+ uid: DataArrayIdentifier
notes
A record that contains the fields required to get sub-arrays.
«record»
PutDataSubarraysType
+ counts: long [0..*] (array)
+ data: AnyArray
+ starts: long [0..*] (array)
+ uid: DataArrayIdentifier
notes
A record that contains the field of data needed to put a sub-array.
«record»
PutUninitializedDataArrayType
+ metadata: DataArrayMetadata
+ uid: DataArrayIdentifier
notes
The record that contains the fields required to put an uninitialized array.

Figure 35: DataArrayType: schemas

23.32.1 record: DataArray

A record that contains the dimensions of the array and its data.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.DataArrayTypes",
     "name": "DataArray",
     "fields":
     [
         {
             "name": "dimensions",
             "type": { "type": "array", "items": "long" }
         },
         { "name": "data", "type": "Energistics.Etp.v12.Datatypes.AnyArray" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
dimensions	An array of dimensions for the data array. This MUST be the actual size of the included data, whether or not it is a sub-array of another array.	long	1	*
data	The data in the array, which must be a type of AnyArray.	AnyArray	1	1

23.32.2 record: DataArrayMetadata

A record that contains fields for metadata to help interpret and understand the data in an array (DataArray).

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.DataArrayTypes",
     "name": "DataArrayMetadata",
     "fields":
     [
         {
             "name": "dimensions",
             "type": { "type": "array", "items": "long" }
         },
         {
             "name": "preferredSubarrayDimensions",
             "type": { "type": "array", "items": "long" }, "default": []
         },
         { "name": "transportArrayType", "type": "Energistics.Etp.v12.Datatypes.AnyArrayType"
},
         { "name": "logicalArrayType", "type":
"Energistics.Etp.v12.Datatypes.AnyLogicalArrayType" },
         { "name": "storeLastWrite", "type": "long" },
         { "name": "storeCreated", "type": "long" },
         {
             "name": "customData",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
dimensions	An array of dimension sizes for the data array. This MUST be the actual size of the included data, whether or not it is a sub-array of another array.	long	1	*
preferredSubarrayDimensions	(Optional) Allows a store to advertise its native chunking of array data. A customer is free to read/access arrays in whatever order it wants. However, doing so in ways that go against the	long	0	*

Field Name	Description	Data Type	Min	Max
	store's (backend) native/actual chunking order may result in significant performance issues. This field is an array of the preferred dimensions.			
transportArrayType	The Avro representation of the logical array type; this field must be one of the types specified in AnyArrayType. **NOTE:** Only certain transport types can be used with specific logical array types (*logicalArrayType* field). For rules and the mapping of allowable types, see Section **13.2.2.1**.	AnyArrayType	1	1
logicalArrayType	The type of array data that is being transferred; this field must be one of the types in AnyLogicalArrayType. **NOTE:** Only certain transport types (*transportArrayType* field) can be used with specific logical array types. For rules and the mapping of allowable types, see Section **13.2.2.1**.	AnyLogicalArrayType	1	1
storeLastWrite	The last time the data array was *written in a particular store.* (See also *storeCreated* in this ***DataArrayMetadata*** record.) • Its main purpose is for use in workflows for eventual consistency between 2 stores. • It is maintained by the ETP store. • For ANY CHANGES to the array data (E.g., values in the array are updated) a store MUST update *storeLastWrite* in this metadata. • When array data is first created in a store, the store MUST set *storeLastWrite* to the same value as *storeCreated*. The value must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
storeCreated	The time that the data array was created in the store. (See also *storeLastWrite* in this ***DataArrayMetadata*** record.) • Its main purpose is for use in workflows for eventual consistency between 2 stores. Specifically, this field helps with an important edge case: on reconnect, an endpoint can more easily determine if while disconnected a data array was modified OR deleted and recreated. • Like *storeLastWrite*, this *storeCreated* field is maintained by the ETP store. The value must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
customData	Allows an endpoint to send custom data, which is a data type defined by an organization other than Energistics (i.e., it's not defined by ETP or any of the Energistics domain data models). This custom data is also informally referred to as "proprietary data or content". It contains a key-value pair of custom key names and associated values. Observe these rules for specifying custom data:	DataValue	0	*

Field Name	Description	Data Type	Min	Max
	1. The keys MAY BE both well-known (and thus, reserved) names as well as application- and vendor-specific names. **RECOMMENDATION:** To specify the authority for a key use this convention "authority:key". 2. Keys are case sensitive. 3. The value MUST be one of the types specified in DataValue.			

23.32.3 record: DataArrayIdentifier

A record that contains fields to identify the URI of the resource and the path in that resource, to identify and find an array.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.DataArrayTypes",
     "name": "DataArrayIdentifier",
     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "pathInResource", "type": "string" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
uri	The URI of the resource (NOT a ***Resource*** record) containing the array data. For more information, see the relevant ML ETP implementation specification. The resource may be an Energistics data object (that references the array), a file, or content in a store. For some Energistics domain standards (e.g., RESQML or PRODML), the URI may identify the EpcExternalPartReference, which acts as a proxy for the repository containing the DataArray. (For more information, see Section **13.1.1**.) **Some example URIs (from RESQML v2.0.1):** • eml:///dataspace/resqml20.obj_IjkGridRepresentation(uuid) • eml:///dataspace/eml20.obj_EpcExternalPartReference(uuid) If both endpoints support alternate URIs for the session, the URIs MAY be alternate data object URIs. Otherwise, they MUST be canonical Energistics data object URIs. For more information, see **Appendix: Energistics Identifiers**.	string	1	1
pathInResource	The path within the resource for the array data. If the resource is an HDF file, this may be a path within the HDF file. For more information, see the relevant ML ETP implementation specification.	string	1	1

23.32.4 record: GetDataSubarraysType

A record that contains the fields required to get sub-arrays.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.DataArrayTypes",
     "name": "GetDataSubarraysType",
     "fields":
     [
         { "name": "uid", "type":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.DataArrayIdentifier" },
         {
             "name": "starts",
             "type": { "type": "array", "items": "long" }, "default": []
         },
         {
             "name": "counts",
             "type": { "type": "array", "items": "long" }, "default": []
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
uid	The required identifying information needed as defined in DataArrayIdentifier.	DataArrayIdentifier	1	1
starts	The starting indexes of the sub-array, per dimension.	long	0	*
counts	The count of values along each dimension.	long	0	*

23.32.5 record: PutDataArraysType

A record that contains the fields required to put sub-arrays.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.DataArrayTypes",
     "name": "PutDataArraysType",
     "fields":
     [
         { "name": "uid", "type":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.DataArrayIdentifier" },
         { "name": "array", "type": "Energistics.Etp.v12.Datatypes.DataArrayTypes.DataArray" },
         {
             "name": "customData",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
uid	The required identifying information needed as defined in DataArrayIdentifier.	DataArrayIdentifier	1	1
array	The data being put for array as defined in DataArray.	DataArray	1	1
customData	Allows an endpoint to send custom data, which is a data type defined by an organization other than Energistics (i.e., it's not defined by ETP or any of the Energistics domain data models). This custom data is also informally referred to as "proprietary data or content". It contains a key-value pair of custom key names and associated values. Observe these rules for specifying custom data: 1. The keys MAY BE both well-known (and thus, reserved) names as well as application- and vendor-specific names. **RECOMMENDATION:** To specify the	DataValue	0	*

Field Name	Description	Data Type	Min	Max
	authority for a key use this convention "authority:key". 2. Keys are case sensitive. 3. The value MUST be one of the types specified in DataValue.			

23.32.6 record: PutUninitializedDataArrayType

The record that contains the fields required to put an uninitialized array.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.DataArrayTypes",
     "name": "PutUninitializedDataArrayType",
     "fields":
     [
         { "name": "uid", "type":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.DataArrayIdentifier" },
         { "name": "metadata", "type":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.DataArrayMetadata" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
uid	The required identifying information needed as defined in DataArrayIdentifier.	DataArrayIdentifier	1	1
metadata	The metadata for each uninitialized array being put as defined in DataArrayMetadata.	DataArrayMetadata	1	1

23.32.7 record: PutDataSubarraysType

A record that contains the field of data needed to put a sub-array.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.DataArrayTypes",
     "name": "PutDataSubarraysType",
     "fields":
     [
         { "name": "uid", "type":
"Energistics.Etp.v12.Datatypes.DataArrayTypes.DataArrayIdentifier" },
         { "name": "data", "type": "Energistics.Etp.v12.Datatypes.AnyArray" },
         {
             "name": "starts",
             "type": { "type": "array", "items": "long" }
         },
         {
             "name": "counts",
             "type": { "type": "array", "items": "long" }
         }
     ]
}
```

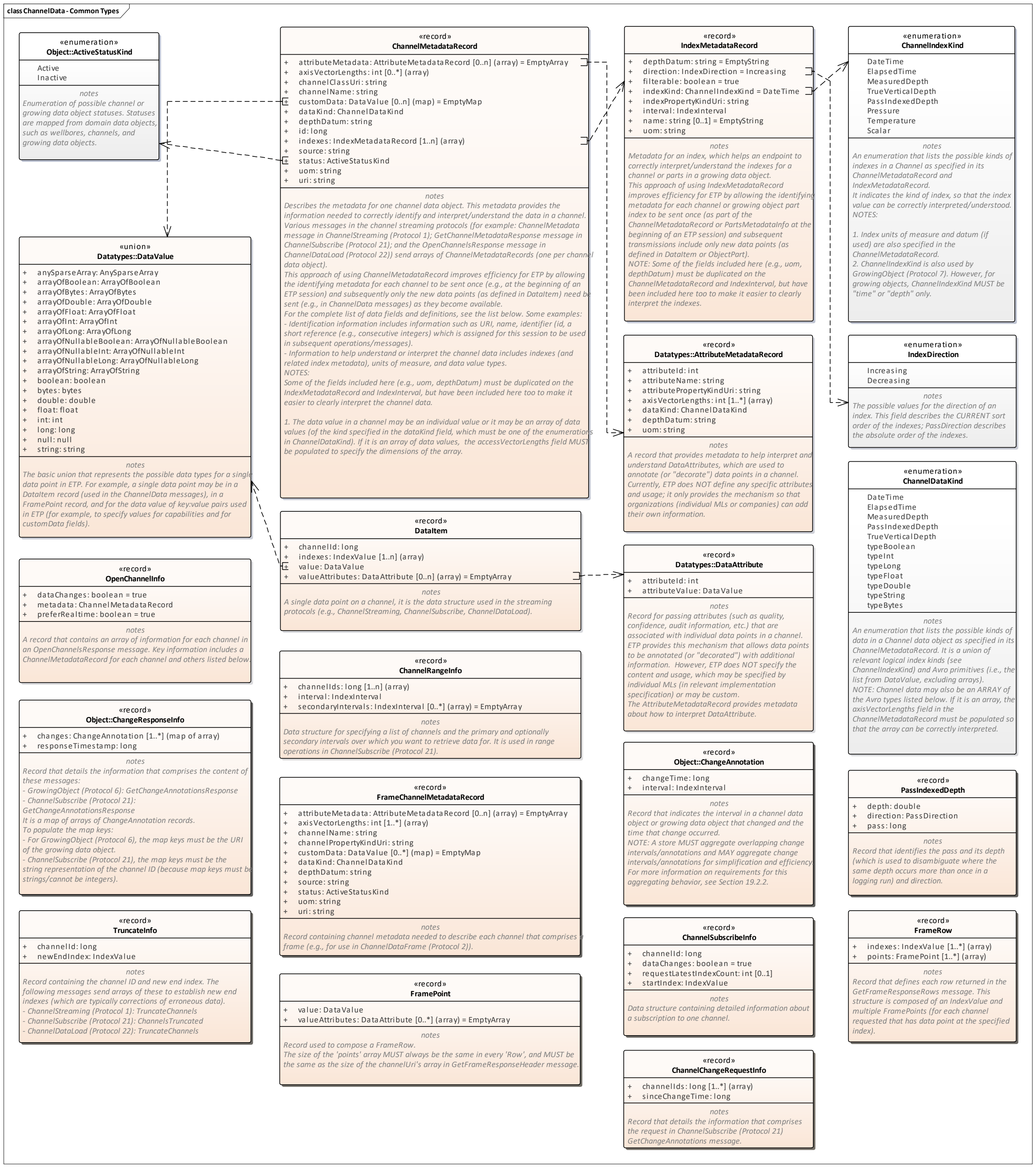
Field Name	Description	Data Type	Min	Max
uid	The required identifying information needed as defined in DataArrayIdentifier.	DataArrayIdentifier	1	1
data	The data in the array, which must of type AnyArray.	AnyArray	1	1

Field Name	Description	Data Type	Min	Max
starts	The starting indexes of the sub-array, per dimension.	long	0	*
counts	The count of values along each dimension.	long	0	*

23.33 ChannelData

This section contains low-level types used for protocols that stream and handle historical channel data, which include:

- ChannelStreaming (Protocol 1)
- ChannelDataFrame (Protocol 2)
- ChannelSubscribe (Protocol 21)
- ChannelDataLoad (Protocol 22)



«enumeration»
Object::ActiveStatusKind

```
Active
Inactive
```

notes
Enumeration of possible channel or growing data object statuses. Statuses are mapped from domain data objects, such as wellbores, channels, and growing data objects.

«record»
ChannelMetadataRecord

```
+   attributeMetadata: AttributeMetadataRecord [0..n] (array) = EmptyArray
+   axisVectorLengths: int [0..*] (array)
+   channelClassUri: string
+   channelName: string
+   customData: DataValue [0..n] (map) = EmptyMap
+   dataKind: ChannelDataKind
+   depthDatum: string
+   id: long
+   indexes: IndexMetadataRecord [1..n] (array)
+   source: string
+   status: ActiveStatusKind
+   uom: string
+   uri: string
```

notes
Describes the metadata for one channel data object. This metadata provides the information needed to correctly identify and interpret/understand the data in a channel. Various messages in the channel streaming protocols (for example: ChannelMetadata message in ChannelStreaming (Protocol 1); GetChannelMetadataResponse message in ChannelSubscribe (Protocol 21); and the OpenChannelsResponse message in ChannelDataLoad (Protocol 22)) send arrays of ChannelMetadataRecords (one per channel data object).
This approach of using ChannelMetadataRecord improves efficiency for ETP by allowing the identifying metadata for each channel to be sent once (e.g., at the beginning of an ETP session) and subsequently only the new data points (as defined in DataItem) need be sent (e.g., in ChannelData messages) as they become available.
For the complete list of data fields and definitions, see the list below. Some examples:
- Identification information includes information such as URI, name, identifier (id, a short reference (e.g., consecutive integers) which is assigned for this session to be used in subsequent operations/messages).
- Information to help understand or interpret the channel data includes indexes (and related index metadata), units of measure, and data value types.
NOTES:
Some of the fields included here (e.g., uom, depthDatum) must be duplicated on the IndexMetadataRecord and IndexInterval, but have been included here too to make it easier to clearly interpret the channel data.

1. The data value in a channel may be an individual value or it may be an array of data values (of the kind specified in the dataKind field, which must be one of the enumerations in ChannelDataKind). If it is an array of data values, the accessVectorLengths field MUST be populated to specify the dimensions of the array.

«record»
IndexMetadataRecord

```
+   depthDatum: string = EmptyString
+   direction: IndexDirection = Increasing
+   filterable: boolean = true
+   indexKind: ChannelIndexKind = DateTime
+   indexPropertyKindUri: string
+   interval: IndexInterval
+   name: string [0..1] = EmptyString
+   uom: string
```

notes
Metadata for an index, which helps an endpoint to correctly interpret/understand the indexes for a channel or parts in a growing data object.
This approach of using IndexMetadataRecord improves efficiency for ETP by allowing the identifying metadata for each channel or growing object part index to be sent once (as part of the ChannelMetadataRecord or PartsMetadataInfo at the beginning of an ETP session) and subsequent transmissions include only new data points (as defined in DataItem or ObjectPart).
NOTE: Some of the fields included here (e.g., uom, depthDatum) must be duplicated on the ChannelMetadataRecord and IndexInterval, but have been included here too to make it easier to clearly interpret the indexes.

«enumeration»
ChannelIndexKind

```
DateTime
ElapsedTime
MeasuredDepth
TrueVerticalDepth
PassIndexedDepth
Pressure
Temperature
Scalar
```

notes
An enumeration that lists the possible kinds of indexes in a Channel as specified in its ChannelMetadataRecord and IndexMetadataRecord.
It indicates the kind of index, so that the index value can be correctly interpreted/understood.
NOTES:

1. Index units of measure and datum (if used) are also specified in the ChannelMetadataRecord.
2. ChannelIndexKind is also used by GrowingObject (Protocol 7). However, for growing objects, ChannelIndexKind MUST be "time" or "depth" only.

«union»
Datatypes::DataValue

```
+   anySparseArray: AnySparseArray
+   arrayOfBoolean: ArrayOfBoolean
+   arrayOfBytes: ArrayOfBytes
+   arrayOfDouble: ArrayOfDouble
+   arrayOfFloat: ArrayOfFloat
+   arrayOfInt: ArrayOfInt
+   arrayOfLong: ArrayOfLong
+   arrayOfNullableBoolean: ArrayOfNullableBoolean
+   arrayOfNullableInt: ArrayOfNullableInt
+   arrayOfNullableLong: ArrayOfNullableLong
+   arrayOfString: ArrayOfString
+   boolean: boolean
+   bytes: bytes
+   double: double
+   float: float
+   int: int
+   long: long
+   null: null
+   string: string
```

notes
The basic union that represents the possible data types for a single data point in ETP. For example, a single data point may be in a DataItem record (used in the ChannelData messages), in a FramePoint record, and for the data value of key:value pairs used in ETP (for example, to specify values for capabilities and for customData fields).

«record»
Datatypes::AttributeMetadataRecord

```
+   attributeId: int
+   attributeName: string
+   attributePropertyKindUri: string
+   axisVectorLengths: int [1..*] (array)
+   dataKind: ChannelDataKind
+   depthDatum: string
+   uom: string
```

notes
A record that provides metadata to help interpret and understand DataAttributes, which are used to annotate (or "decorate") data points in a channel. Currently, ETP does NOT define any specific attributes and usage; it only provides the mechanism so that organizations (individual MLs or companies) can add their own information.

«enumeration»
IndexDirection

```
Increasing
Decreasing
```

notes
The possible values for the direction of an index. This field describes the CURRENT sort order of the indexes; PassDirection describes the absolute order of the indexes.

«enumeration»
ChannelDataKind

```
DateTime
ElapsedTime
MeasuredDepth
PassIndexedDepth
TrueVerticalDepth
typeBoolean
typeInt
typeLong
typeFloat
typeDouble
typeString
typeBytes
```

notes
An enumeration that lists the possible kinds of data in a Channel data object as specified in its ChannelMetadataRecord. It is a union of relevant logical index kinds (see ChannelIndexKind) and Avro primitives (i.e., the list from DataValue, excluding arrays).
NOTE: Channel data may also be an ARRAY of the Avro types listed below. If it is an array, the axisVectorLengths field in the ChannelMetadataRecord must be populated so that the array can be correctly interpreted.

«record»
DataItem

```
+   channelId: long
+   indexes: IndexValue [1..n] (array)
+   value: DataValue
+   valueAttributes: DataAttribute [0..n] (array) = EmptyArray
```

notes
A single data point on a channel, it is the data structure used in the streaming protocols (e.g., ChannelStreaming, ChannelSubscribe, ChannelDataLoad).

«record»
Datatypes::DataAttribute

```
+   attributeId: int
+   attributeValue: DataValue
```

notes
Record for passing attributes (such as quality, confidence, audit information, etc.) that are associated with individual data points in a channel. ETP provides this mechanism that allows data points to be annotated (or "decorated") with additional information. However, ETP does NOT specify the content and usage, which may be specified by individual MLs (in relevant implementation specification) or may be custom.
The AttributeMetadataRecord provides metadata about how to interpret DataAttribute.

«record»
OpenChannelInfo

```
+   dataChanges: boolean = true
+   metadata: ChannelMetadataRecord
+   preferRealtime: boolean = true
```

notes
A record that contains an array of information for each channel in an OpenChannelsResponse message. Key information includes a ChannelMetadataRecord for each channel and others listed below.

«record»
ChannelRangeInfo

```
+   channelIds: long [1..n] (array)
+   interval: IndexInterval
+   secondaryIntervals: IndexInterval [0..*] (array) = EmptyArray
```

notes
Data structure for specifying a list of channels and the primary and optionally secondary intervals over which you want to retrieve data for. It is used in range operations in ChannelSubscribe (Protocol 21).

«record»
Object::ChangeResponseInfo

```
+   changes: ChangeAnnotation [1..*] (map of array)
+   responseTimestamp: long
```

notes
Record that details the information that comprises the content of these messages:
- GrowingObject (Protocol 6): GetChangeAnnotationsResponse
- ChannelSubscribe (Protocol 21): GetChangeAnnotationsResponse
It is a map of arrays of ChangeAnnotation records.
To populate the map keys:
- For GrowingObject (Protocol 6), the map keys must be the URI of the growing data object.
- ChannelSubscribe (Protocol 21), the map keys must be the string representation of the channel ID (because map keys must be strings/cannot be integers).

«record»
Object::ChangeAnnotation

```
+   changeTime: long
+   interval: IndexInterval
```

notes
Record that indicates the interval in a channel data object or growing data object that changed and the time that change occurred.
NOTE: A store MUST aggregate overlapping change intervals/annotations and MAY aggregate change intervals/annotations for simplification and efficiency. For more information on requirements for this aggregating behavior, see Section 19.2.2.

«record»
PassIndexedDepth

```
+   depth: double
+   direction: PassDirection
+   pass: long
```

notes
Record that identifies the pass and its depth (which is used to disambiguate where the same depth occurs more than once in a logging run) and direction.

«record»
FrameChannelMetadataRecord

```
+   attributeMetadata: AttributeMetadataRecord [0..n] (array) = EmptyArray
+   axisVectorLengths: int [1..*] (array)
+   channelName: string
+   channelPropertyKindUri: string
+   customData: DataValue [0..*] (map) = EmptyMap
+   dataKind: ChannelDataKind
+   depthDatum: string
+   source: string
+   status: ActiveStatusKind
+   uom: string
+   uri: string
```

notes
Record containing channel metadata needed to describe each channel that comprises a frame (e.g., for use in ChannelDataFrame (Protocol 2)).

«record»
TruncateInfo

```
+   channelId: long
+   newEndIndex: IndexValue
```

notes
Record containing the channel ID and new end index. The following messages send arrays of these to establish new end indexes (which are typically corrections of erroneous data).
- ChannelStreaming (Protocol 1): TruncateChannels
- ChannelSubscribe (Protocol 21): ChannelsTruncated
- ChannelDataLoad (Protocol 22): TruncateChannels

«record»
ChannelSubscribeInfo

```
+   channelId: long
+   dataChanges: boolean = true
+   requestLatestIndexCount: int [0..1]
+   startIndex: IndexValue
```

notes
Data structure containing detailed information about a subscription to one channel.

«record»
FrameRow

```
+   indexes: IndexValue [1..*] (array)
+   points: FramePoint [1..*] (array)
```

notes
Record that defines each row returned in the GetFrameResponseRows message. This structure is composed of an IndexValue and multiple FramePoints (for each channel requested that has data point at the specified index).

«record»
FramePoint

```
+   value: DataValue
+   valueAttributes: DataAttribute [0..*] (array) = EmptyArray
```

notes
Record used to compose a FrameRow.
The size of the 'points' array MUST always be the same in every 'Row', and MUST be the same as the size of the channelUri's array in GetFrameResponseHeader message.

«record»
ChannelChangeRequestInfo

```
+   channelIds: long [1..*] (array)
+   sinceChangeTime: long
```

notes
Record that details the information that comprises the request in ChannelSubscribe (Protocol 21) GetChangeAnnotations message.

Figure 36: ChannelData: data type schemas

23.33.1 ChannelDataKind

An enumeration that lists the possible kinds of data in a Channel data object as specified in its ChannelMetadataRecord. It is a union of relevant logical index kinds (see ChannelIndexKind) and Avro primitives (i.e., the list from DataValue, excluding arrays).

NOTE: Channel data may also be an ARRAY of the Avro types listed below. If it is an array, the *axisVectorLengths* field in the ChannelMetadataRecord must be populated so that the array can be correctly interpreted.

Channel Data Kind	Description	Data Type
DateTime	Each value for channel data is a timestamp. The actual channel data is a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	string
ElapsedTime	Each value for channel data is an elapsed time. The actual channel data is the number of microseconds from zero and is an Avro long. **NOTE:** 1. This value is NOT related to any time datum. 2. The index UOM MUST be set to "us". **EXAMPLE elapsed time use case:** Engine hours for equipment, which is how long the equipment has been running.	string
MeasuredDepth	Each value for channel data represents a measured depth (MD).	string
PassIndexedDepth	Each value for channel data represents a pass indexed depth.	string
TrueVerticalDepth	Each value for channel data represents a true vertical depth (TVD).	string
typeBoolean		string
typeInt		string
typeLong		string
typeFloat		string
typeDouble		string
typeString		string
typeBytes		string

Avro Source

```
{
     "type": "enum",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "ChannelDataKind",
     "symbols":
     [
         "DateTime",
         "ElapsedTime",
         "MeasuredDepth",
         "PassIndexedDepth",
         "TrueVerticalDepth",
         "typeBoolean",
         "typeInt",
         "typeLong",
         "typeFloat",
         "typeDouble",
         "typeString",
         "typeBytes"
     ]
}
```

23.33.2 ChannelIndexKind

An enumeration that lists the possible kinds of indexes in a Channel as specified in its ChannelMetadataRecord and IndexMetadataRecord.

It indicates the kind of index, so that the index value can be correctly interpreted/understood.

NOTES:

1. Index units of measure and datum (if used) are also specified in the ChannelMetadataRecord.
2. ChannelIndexKind is also used by GrowingObject (Protocol 7). However, for growing objects, ChannelIndexKind MUST be "time" or "depth" only.

Channel Index Kind	Description	Data Type
DateTime	The index for the channel is a timestamp. Each actual index value is a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	
ElapsedTime	The index for the channel is an elapsed time. Each actual index value is the number of microseconds from zero and is an Avro long. **NOTES:** 1. This value is NOT related to any time datum. 2. The index UOM MUST be set to "us". **EXAMPLE** elapsed time use case: Engine hours for equipment, which is how long the equipment has been running.	
MeasuredDepth	The index of the channel is measure depth (MD).	
TrueVerticalDepth	The index of the channel is a true vertical depth (TVD).	
PassIndexedDepth	The index of the channel is a pass indexed depth.	
Pressure	The index of the channel is a pressure.	
Temperature	The index of a channel is a temperature.	
Scalar	The index of the channel represents values that are temperature or pressure. It indicates that the index is of type Avro double. **NOTES:** 1. Even if the index values are integer numbers, the index values MUST be sent as Avro doubles. 2. Optionally, you may specify a datum (in the ChannelMetadataRecord).	

Avro Source

```
{
     "type": "enum",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "ChannelIndexKind",
     "symbols":
     [
         "DateTime",
         "ElapsedTime",
         "MeasuredDepth",
         "TrueVerticalDepth",
         "PassIndexedDepth",
         "Pressure",
         "Temperature",
         "Scalar"
     ]
}
```

23.33.3 IndexDirection

The possible values for the direction of an index. This field describes the CURRENT sort order of the indexes; PassDirection describes the absolute order of the indexes.

Field Name	Description	Data Type	Min	Max
Increasing	The index values increase.		1	1

Field Name	Description	Data Type	Min	Max
Decreasing	The index values decrease.		1	1
Unordered	The index values are unordered. This MUST NOT be used for primary indexes. This value ONLY applies to secondary indexes.		1	1

Avro Source

```
{
     "type": "enum",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "IndexDirection",
     "symbols":
     [
         "Increasing",
         "Decreasing",
         "Unordered"
     ]
}
```

23.33.4 PassDirection

The possible values for the direction of a pass in a wireline operation. It defines the absolute ordering for PassIndexedDepth data (compared to IndexDirection which is the current sort order).

Field Name	Description	Data Type	Min	Max
Up	The wireline tool is moving up in the hole/wellbore.		1	1
HoldingSteady	The wireline tool is not moving in the hole/wellbore. NOTE: This MUST NOT be used for primary indexes. This value ONLY applies to secondary indexes.		1	1
Down	The wireline tool is moving down in the hole/wellbore.		1	1

Avro Source

```
{
     "type": "enum",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "PassDirection",
     "symbols":
     [
         "Up",
         "HoldingSteady",
         "Down"
     ]
}
```

23.33.5 record: DataItem

A single data point on a channel, it is the data structure used in the streaming protocols (e.g., ChannelStreaming, ChannelSubscribe, ChannelDataLoad).

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "DataItem",
     "fields":
     [
         { "name": "channelId", "type": "long" },
         {
             "name": "indexes",
             "type": { "type": "array", "items": "Energistics.Etp.v12.Datatypes.IndexValue" }
         },
         { "name": "value", "type": "Energistics.Etp.v12.Datatypes.DataValue" },
         {
             "name": "valueAttributes",
             "type": { "type": "array", "items": "Energistics.Etp.v12.Datatypes.DataAttribute"
}, "default": []
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
channelId	The identifier of the channel for this point, as received in a ChannelMetadata record.	long	1	1
indexes	The value of the indexes for this data point, which MUST be of type IndexValue. • The array MUST be of length 0, or the length of the corresponding index metadata array for the channelId. • If the length is 0, then the index values are the same as the indexes in the previous item in the array of DataItem, which MUST have identical index metadata as the channelId of this record. • If the length is not 0, individual elements in the *indexes* may be null to indicate that specific index value is the same as the previous item in the array of DataItem.	IndexValue	1	n
value	The value of this data point, which must be of a type specified in DataValue.	DataValue	1	1
valueAttributes	(Optional) Any qualifiers, such as quality, accuracy, etc., attached to this data point. It is an array of ID-value pairs, where the IDs and the values are NOT described as part of this specification. Use of this field is defined by the relevant implementation specification or can be for custom use. MUST be of type DataAttribute. The AttributeMetadataRecord contains the metadata for interpreting/understanding the data attributes.	DataAttribute	0	n

23.33.6 record: IndexMetadataRecord

Metadata for an index, which helps an endpoint to correctly interpret/understand the indexes for a channel or parts in a growing data object.

This approach of using IndexMetadataRecord improves efficiency for ETP by allowing the identifying metadata for each channel or growing object part index to be sent once (as part of the ChannelMetadataRecord or PartsMetadataInfo at the beginning of an ETP session) and subsequent transmissions include only new data points (as defined in DataItem or ObjectPart).

NOTE: Some of the fields included here (e.g., uom, depthDatum) must be duplicated on the ChannelMetadataRecord and IndexInterval, but have been included here too to make it easier to clearly interpret the indexes.

Avro Schema

```
{
      "type": "record",
      "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
      "name": "IndexMetadataRecord",
      "fields":
      [
          { "name": "indexKind", "type":
"Energistics.Etp.v12.Datatypes.ChannelData.ChannelIndexKind", "default": "DateTime" },
          { "name": "interval", "type": "Energistics.Etp.v12.Datatypes.Object.IndexInterval" },
          { "name": "direction", "type":
"Energistics.Etp.v12.Datatypes.ChannelData.IndexDirection", "default": "Increasing" },
          { "name": "name", "type": "string", "default": "" },
          { "name": "uom", "type": "string" },
          { "name": "depthDatum", "type": "string", "default": "" },
          { "name": "indexPropertyKindUri", "type": "string" },
          { "name": "filterable", "type": "boolean", "default": true }
      ]
}
```

Field Name	Description	Data Type	Min	Max
indexKind	Main type of the index (time, depth, etc.) as defined in ChannelIndexKind.	ChannelIndexKind	1	1
interval	The information that defines the interval as specified in the IndexInterval record, which includes the pair of indexes that define the interval and other data common to the interval including unit and depth datum. The *uom* and *depthDatum* fields on the *interval* record MUST match the *uom* and *depthDatum* fields on this record.	IndexInterval	1	1
direction	The direction of the index values, increasing or decreasing. Must remain constant for the life of a channel. If the IndexMetadataRecord is describing a part in a growing data object (ObjectPart), then direction MUST BE increasing. Primary indexes MUST be either Increasing or Decreasing. For primary indexes, Increasing means strictly increasing, and Decreasing means strictly decreasing. The direction for secondary indexes MUST reflect the order that the secondary index values will appear when data is ordered by the primary index in its specified direction. When ordered by the primary index, secondary indexes may be monotonically increasing, monotonically decreasing or unordered. For secondary indexes, Increasing means monotonically increasing, Decreasing means monotonically decreasing, and Unordered means neither monotonically increasing nor monotonically decreasing.	IndexDirection	1	1
name	A mnemonic description of the index. This is an optional field; in the absence of a value, the string representation of the indexType enumeration SHOULD be considered the mnemonic.	string	0	1
uom	The unit of measure of the index.	string	1	1
depthDatum	If the index is depth, this is the depth datum it references.	string	1	1
indexPropertyKindUri	An optional field that allows an endpoint to specify the URI of a property kind data object, which MUST be available from the endpoint and MAY be from the Energistics PropertyKindDictionary. Use of this field lets an endpoint indicate more specifically what the index described by the IndexMetatdataRecord is representing. The PropertyKindDictionary is an implementation of the Practical Well Log Standard (PWLS). For	string	1	1

Field Name	Description	Data Type	Min	Max
	more information on how PWLS is used in ETP, see Section **3.12.7**.			
filterable	Flag that indicates that the index described by this record can be used as a filter. Use of this field is optional; if not used it must be false.	boolean	1	1

23.33.7 record: ChannelMetadataRecord

Describes the metadata for one channel data object. This metadata provides the information needed to correctly identify and interpret/understand the data in a channel.

Various messages in the channel streaming protocols (for example: ***ChannelMetadata*** message in ChannelStreaming (Protocol 1); ***GetChannelMetadataResponse*** message in ChannelSubscribe (Protocol 21); and the ***OpenChannelsResponse*** message in ChannelDataLoad (Protocol 22)) send arrays of ChannelMetadataRecords (one per channel data object).

This approach of using ChannelMetadataRecord improves efficiency for ETP by allowing the identifying metadata for each channel to be sent once (e.g., at the beginning of an ETP session) and subsequently only the new data points (as defined in DataItem) need be sent (e.g., in ***ChannelData*** messages) as they become available.

For the complete list of data fields and definitions, see the list below. Some examples:

- Identification information includes information such as URI, name, identifier (**id**, a short reference (e.g., consecutive integers) which is assigned for this session to be used in subsequent operations/messages).
- Information to help understand or interpret the channel data includes indexes (and related index metadata), units of measure, and data value types.

NOTES:

1. Some of the fields included here (e.g., uom, depthDatum) must be duplicated on the IndexMetadataRecord and IndexInterval, but have been included here too to make it easier to clearly interpret the channel data.
2. The data value in a channel may be an individual value or it may be an array of data values (of the kind specified in the *dataKind* field, which must be one of the enumerations in ChannelDataKind). If it is an array of data values, the *accessVectorLengths* field MUST be populated to specify the dimensions of the array.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "ChannelMetadataRecord",
     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "id", "type": "long" },
         {
             "name": "indexes",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.IndexMetadataRecord" }
         },
         { "name": "channelName", "type": "string" },
         { "name": "dataKind", "type":
"Energistics.Etp.v12.Datatypes.ChannelData.ChannelDataKind" },
         { "name": "uom", "type": "string" },
         { "name": "depthDatum", "type": "string" },
         { "name": "channelClassUri", "type": "string" },
         { "name": "status", "type": "Energistics.Etp.v12.Datatypes.Object.ActiveStatusKind" },
         { "name": "source", "type": "string" },
         {
             "name": "axisVectorLengths",
             "type": { "type": "array", "items": "int" }
         },
         {
             "name": "attributeMetadata",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.AttributeMetadataRecord" }, "default": []
         },
         {
             "name": "customData",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
uri	MUST be the URI to a domain-specific data object that identifies and describes the channel. The URI MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
id	An integer identifier assigned to the channel in a specific protocol for the ETP session, which must be provided by the producer or store role. These integer IDs are used to reduce data on the wire and are used (instead of URIs) for subsequent operations/messages. Channel IDs are only unique or meaningful within a specific protocol in a given ETP session. If you start a new session or use the channel in a different protocol within the same session, the same channel URI may result in different channel IDs. If the channel has been deleted and recreated during a session, it MUST be assigned a new ID in that session.	long	1	1
indexes	The metadata for the indexes associated with this channel as specified in the IndexMetadataRecord. • The array MUST have a length of at least 1. • The record for the primary index MUST always be the first record in the array. • The values of the primary index MUST be unique within the channel.	IndexMetadataRecord	1	n
channelName	The name for the channel.	string	1	1
uom	The unit of measure for the channel. All DataItem records send data using this UOM. For Energistics domain standards (i.e., WITSML, PRODML, RESQML) version 2.0 or above, the UOM MUST be a valid value from QuantityClass in Energistics *common* (which is an implementation of the Energistics UOM Standard). ETP does not support conversion to a customer-requested system of measurement.	string	1	1
dataKind	The kind of data contained in the channel, which must be one of the values in ChannelDataKind.	ChannelDataKind	1	1
depthDatum	If the channel data is a depth value, this is the datum it references.	string	1	1
channelClassUri	MUST populate this field with the URI of a property kind data object, which MUST be available from the endpoint and MAY be from the Energistics PropertyKindDictionary. Use of this field means an endpoint can specifically	string	1	1

Field Name	Description	Data Type	Min	Max
	describe what property the channel data represents. The PropertyKindDictionary is an implementation of the Practical Well Log Standard (PWLS). For more information on how PWLS is used in ETP, see Section **3.12.7**.			
status	Current status of this channel as defined in ActiveStatusKind.	ActiveStatusKind	1	1
source	It is the provider, typically a company, that is the source of the data in this channel. This field maps to the *source* field on a Channel data object in WITSML.	string	1	1
axisVectorLengths	If the channel data is an array, then this field MUST be populated. It provides the context for how to decode a flattened 1D array back into the higher dimension array. Rules for populating this field: ● You MUST encode the positional information as an absolute 'start' offset between it and the length of each subarray that Avro will encode onto the wire. ● The number of elements in the array indicates the number of dimensions in the data. ● The elements in the array indicate the length of each dimension. Rules for ordering: ● Slowest to fastest. ● Index 0 is the slowest moving dimension. ● Last index is the fastest moving dimension. **EXAMPLE:** If you a have a 2D array of 3 rows and 20 columns, then this field would contain: 3,20	int	0	*
attributeMetadata	An array of metadata (as specified in AttributeMetadataRecord) that describes the DataAttributes that may be provided for individual DataItems in a channel.	AttributeMetadataRecord	0	n
customData	Allows an endpoint to send custom data, which is a data type defined by an organization other than Energistics (i.e., it's not defined by ETP or any of the Energistics domain data models). This custom data is also informally referred to as "proprietary data or content". It contains a key-value pair of custom key names and associated values. Observe these rules for specifying custom data: 1. The keys MAY BE both well-known (and thus, reserved) names as well as application- and vendor-specific names. **RECOMMENDATION:** To specify the authority for a key use this convention "authority:key". 2. Keys are case sensitive. 3. The value MUST be one of the types specified in DataValue.	DataValue	0	n

23.33.8 record: ChannelRangeInfo

Data structure for specifying a list of channels and the primary and optionally secondary intervals over which you want to retrieve data for. It is used in range operations in ChannelSubscribe (Protocol 21).

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "ChannelRangeInfo",
     "fields":
     [
         {
             "name": "channelIds",
             "type": { "type": "array", "items": "long" }
         },
         { "name": "interval", "type": "Energistics.Etp.v12.Datatypes.Object.IndexInterval" },
         {
             "name": "secondaryIntervals",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.Object.IndexInterval" }, "default": []
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
channelIds	One or more channel IDs for which this range is requested. All channels MUST have a common index type, UOM, and direction.	long	1	n
interval	Specifies the primary interval, which defines the range of interest, as defined in IndexInterval.	IndexInterval	1	1
secondaryIntervals	(Optional) Specifies one or more secondary intervals, as defined in IndexInterval. This secondary interval is additional filtering of the data returned from the primary interval (specified in the *interval* field above). **NOTE**: If a store's SupportsSecondaryIndexFiltering protocol capability is false and a customer populates this field, then the Store MUST deny the request and send error ENOTSUPPORTED (7).	IndexInterval	0	*

23.33.9 record: ChannelSubscribeInfo

Data structure containing detailed information about a subscription to one channel.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "ChannelSubscribeInfo",
     "fields":
     [
         { "name": "channelId", "type": "long" },
         { "name": "startIndex", "type": "Energistics.Etp.v12.Datatypes.IndexValue" },
         { "name": "dataChanges", "type": "boolean", "default": true },
         { "name": "requestLatestIndexCount", "type": ["null", "int"] }
     ]
}
```

Field Name	Description	Data Type	Min	Max
channelId	The ID of the channel to be started or stopped. This is the UID that was assigned to the channel (in place of the longer channel URI) with the	long	1	1

Field Name	Description	Data Type	Min	Max
	ChannelMetadataRecord returned in the GetChannelMetadataResponse.			
startIndex	The starting index that the customer is requesting the store start streaming from. It must be of a type specified in IndexValue . **NOTE:** Optionally, an endpoint can specify a *requestLatestIndexCount*; if it is populated this start index is ignored.	IndexValue	1	1
requestLatestIndexCount	If specified, the store MUST return the latest *n* values from the specified channel and continue streaming per the subscription. If this property is provided, i.e., not null, the store MUST ignore the start index (*startIndex* field).	int	0	1
dataChanges	Boolean. If true, it indicates that data changes are being requested (in addition to real-time streaming data).	boolean	1	1

23.33.10 record: OpenChannelInfo

A record that contains an array of information for each channel in an OpenChannelsResponse message. Key information includes a ChannelMetadataRecord for each channel and others listed below.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "OpenChannelInfo",
     "fields":
     [
         { "name": "metadata", "type":
"Energistics.Etp.v12.Datatypes.ChannelData.ChannelMetadataRecord" },
         { "name": "preferRealtime", "type": "boolean", "default": true },
         { "name": "dataChanges", "type": "boolean", "default": true }
     ]
}
```

Field Name	Description	Data Type	Min	Max
metadata	The metadata for each channel as specified in the ChannelMetadataRecord.	ChannelMetadataRecord	1	1
preferRealtime	Boolean. If true, it indicates the receiver has a preference to receive realtime data first (before historical data). Default = true	boolean	1	1
dataChanges	Boolean. If true, indicates that data changes (which are sent with ReplaceRange and TruncateChannels messages) are also being requested (in addition to realtime streaming data). Default = true	boolean	1	1

23.33.11 record: FrameChannelMetadataRecord

Record containing channel metadata needed to describe each channel that comprises a frame (e.g., for use in ChannelDataFrame (Protocol 2)).

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "FrameChannelMetadataRecord",
     "fields":
```

```
     [
         { "name": "uri", "type": "string" },
         { "name": "channelName", "type": "string" },
         { "name": "dataKind", "type":
"Energistics.Etp.v12.Datatypes.ChannelData.ChannelDataKind" },
         { "name": "uom", "type": "string" },
         { "name": "depthDatum", "type": "string" },
         { "name": "channelPropertyKindUri", "type": "string" },
         { "name": "status", "type": "Energistics.Etp.v12.Datatypes.Object.ActiveStatusKind" },
         { "name": "source", "type": "string" },
         {
             "name": "axisVectorLengths",
             "type": { "type": "array", "items": "int" }
         },
         {
             "name": "attributeMetadata",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.AttributeMetadataRecord" }, "default": []
         },
         {
             "name": "customData",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
uri	The Energistics URI for an Energistics channel set data object. The URI MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
channelName	The name for the channel.	string	1	1
dataKind	The kind of data contained in the channel, which must be one of the values in ChannelDataKind.	ChannelDataKind	1	1
uom	The unit of measure for the channel. All FramePoint data use this UOM. For Energistics domain standards (i.e., WITSML, PRODML, RESQML) version 2.0 or above, the UOM MUST be a valid value from QuantityClass in Energistics *common*. ETP does not support conversion to a consumer-requested system of measurement.	string	1	1
depthDatum	If the channel data is a depth value, this is the datum it references.	string	1	1
channelPropertyKindUri	MUST populate this field with the URI of a property kind (PropertyKind) from the Energistics PropertyKindDictionary. Use of this field means an endpoint can specifically describe what property the channel data represents. The PropertyKindDictionary is an implementation of the Practical Well Log Standard (PWLS). For more information on how PWLS is used in ETP, see Section **3.12.7**.	string	1	1
status	Current status of this channel as defined in ActiveStatusKind.	ActiveStatusKind	1	1
source	It is the provider, typically a company, that is the source of the data in this channel. This field maps to the *source* field on a Channel data object in WITSML.	string	1	1
axisVectorLengths	If the channel data is an array, then this field MUST be populated. It provides the context for how to decode a flattened 1D array back into the higher dimension array. Rules for populating this field:	int	1	*

Field Name	Description	Data Type	Min	Max
	• You MUST encode the positional information as an absolute 'start' offset between it and the length of each subarray that Avro will encode onto the wire. • The number of elements in the array indicates the number of dimensions in the data. • The elements in the array indicate the length of each dimension. Rules for ordering: • Slowest to fastest. • Index 0 is the slowest moving dimension. • Last index is the fastest moving dimension. **EXAMPLE:** If you a have a 2D array of 3 rows and 20 columns, then this field would contain: 3,20			
attributeMetadata	An array of metadata (as specified in AttributeMetadataRecord) that describes the DataAttributes that may be provided for individual FramePoints in a channel frame.	AttributeMetadataRecord	0	n
customData	Allows an endpoint to send custom data, which is a data type defined by an organization other than Energistics (i.e., it's not defined by ETP or any of the Energistics domain data models). This custom data is also informally referred to as "proprietary data or content". It contains a key-value pair of custom key names and associated values. Observe these rules for specifying custom data: 1. The keys MAY BE both well-known (and thus, reserved) names as well as application- and vendor-specific names. **RECOMMENDATION:** To specify the authority for a key use this convention "authority:key". 2. Keys are case sensitive. 3. The value MUST be one of the types specified in DataValue.	DataValue	0	*

23.33.12 record: FramePoint

Record used to compose a FrameRow.

The size of the 'points' array MUST always be the same in every 'Row', and MUST be the same as the size of the *channelUri*'s array in GetFrameResponseHeader message.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "FramePoint",
     "fields":
     [
         { "name": "value", "type": "Energistics.Etp.v12.Datatypes.DataValue" },
         {
             "name": "valueAttributes",
             "type": { "type": "array", "items": "Energistics.Etp.v12.Datatypes.DataAttribute"
}, "default": []
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
value	The value of a given point, which must be of a type in DataValue.	DataValue	1	1
valueAttributes	(Optional) Any qualifiers, such as quality, accuracy, etc., attached to this data point. It is an array of ID-value pairs, where the IDs and the values are NOT described as part of this specification. Use of this field is defined by the relevant implementation specification or can be for custom use. MUST be of type DataAttribute. The AttributeMetadataRecord contains the metadata for interpreting/understanding the data attributes.	DataAttribute	0	*

23.33.13 record: FrameRow

Record that defines each row returned in the GetFrameResponseRows message. This structure is composed of an IndexValue and multiple FramePoints (for each channel requested that has data point at the specified index).

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "FrameRow",
     "fields":
     [
         {
             "name": "indexes",
             "type": { "type": "array", "items": "Energistics.Etp.v12.Datatypes.IndexValue" }
         },
         {
             "name": "points",
             "type": { "type": "array", "items":
"Energistics.Etp.v12.Datatypes.ChannelData.FramePoint" }
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
indexes	The index for each row as defined in IndexValue.	IndexValue	1	*
points	The value of the points as defined in FramePoint.	FramePoint	1	*

23.33.14 record: TruncateInfo

Record containing the channel ID and new end index. The following messages send arrays of these to establish new end indexes (which are typically corrections of erroneous data).

- ChannelStreaming (Protocol 1): TruncateChannels
- ChannelSubscribe (Protocol 21): ChannelsTruncated
- ChannelDataLoad (Protocol 22): TruncateChannels

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "TruncateInfo",
     "fields":
     [
         { "name": "channelId", "type": "long" },
         { "name": "newEndIndex", "type": "Energistics.Etp.v12.Datatypes.IndexValue" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
channelId	The channel ID of the channel whose index you want to change.	long	1	1
newEndIndex	The new end index for the specified channel, which must be of type IndexValue.	IndexValue	1	1

23.33.15 record: ChannelChangeRequestInfo

Record that details the information that comprises the request in ChannelSubscribe (Protocol 21) GetChangeAnnotations message.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "ChannelChangeRequestInfo",
     "fields":
     [
         { "name": "sinceChangeTime", "type": "long" },
         {
             "name": "channelIds",
             "type": { "type": "array", "items": "long" }
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
sinceChangeTime	The start time for changes. That is, the customer is requesting changes that happened since this time. This time MUST BE less than or equal to the store's ChangeRetentionPeriod endpoint capability. MUST be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
channelIds	The UIDs of the channels for which change annotations are being requested.	long	1	*

23.33.16 record: PassIndexedDepth

Record that identifies the pass and its depth (which is used to disambiguate where the same depth occurs more than once in a logging run) and direction.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.ChannelData",
     "name": "PassIndexedDepth",
     "fields":
```

```
     [
         { "name": "pass", "type": "long" },
         { "name": "direction", "type":
"Energistics.Etp.v12.Datatypes.ChannelData.PassDirection" },
         { "name": "depth", "type": "double" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
pass	A unique identifier for a logging pass.	long	1	1
direction	The direction that the tool is moving in a wireline operation as defined in PassDirection.	PassDirection	1	1
depth	The depth for this pass.	double	1	1

23.34 Object

This section contains datatypes for working with data objects. These datatypes are used by Discovery (Protocol 3), Store (Protocol 4), Store Notification (Protocol 5), GrowingObject (Protocol 6), GrowingObjectNotification (Protocol 7), StoreQuery (Protocol 14), and GrowingObjectQuery (Protocol 16).

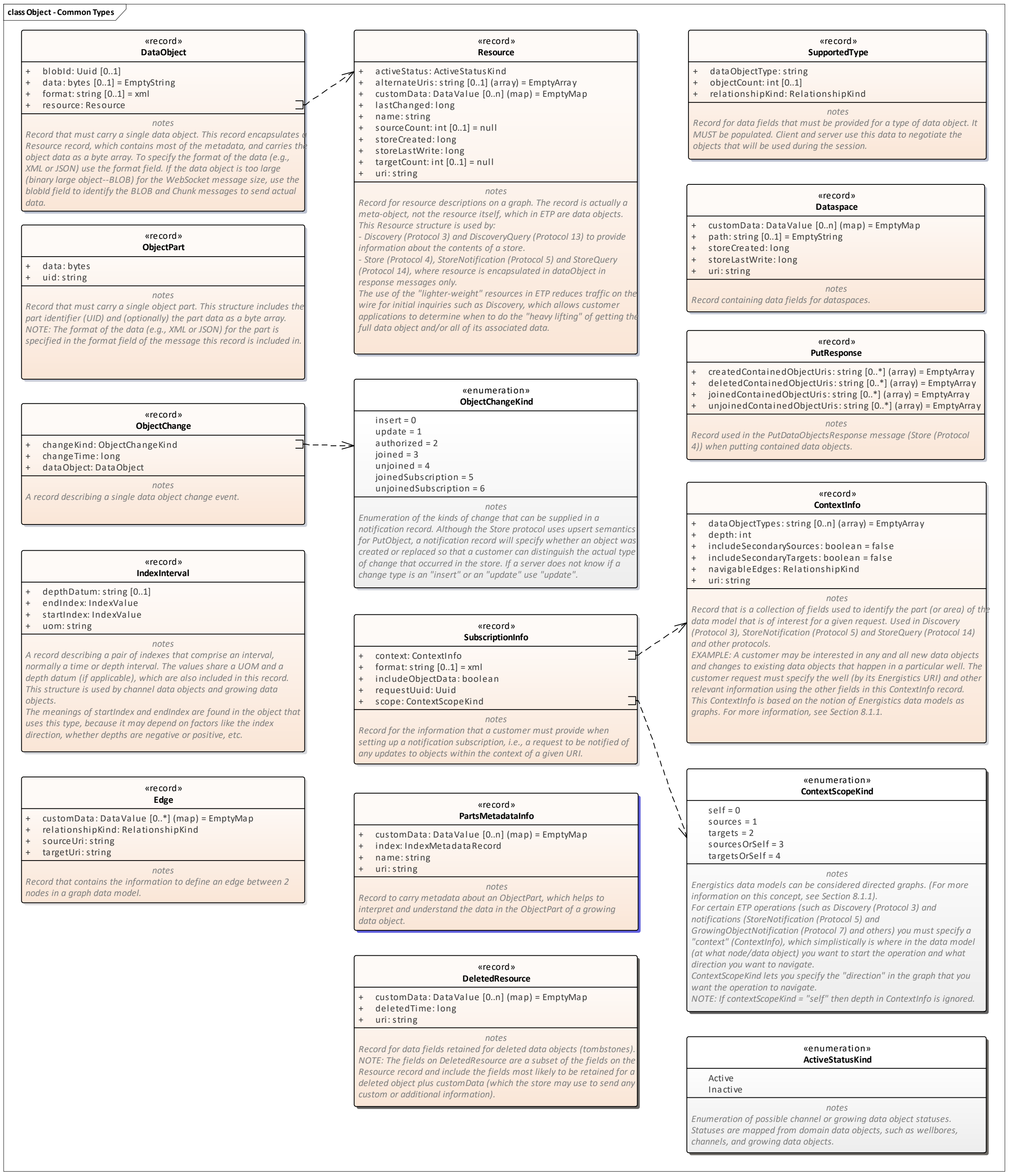

class Object - Common Types
«record» DataObject
+ blobId: Uuid [0..1]
+ data: bytes [0..1] = EmptyString
+ format: string [0..1] = xml
+ resource: Resource
notes
Record that must carry a single data object. This record encapsulates a Resource record, which contains most of the metadata, and carries the object data as a byte array. To specify the format of the data (e.g., XML or JSON) use the format field. If the data object is too large (binary large object--BLOB) for the WebSocket message size, use the blobId field to identify the BLOB and Chunk messages to send actual data.
«record» ObjectPart
+ data: bytes
+ uid: string
notes
Record that must carry a single object part. This structure includes the part identifier (UID) and (optionally) the part data as a byte array. NOTE: The format of the data (e.g., XML or JSON) for the part is specified in the format field of the message this record is included in.
«record» ObjectChange
+ changeKind: ObjectChangeKind
+ changeTime: long
+ dataObject: DataObject
notes
A record describing a single data object change event.
«record» IndexInterval
+ depthDatum: string [0..1]
+ endIndex: IndexValue
+ startIndex: IndexValue
+ uom: string
notes
A record describing a pair of indexes that comprise an interval, normally a time or depth interval. The values share a UOM and a depth datum (if applicable), which are also included in this record. This structure is used by channel data objects and growing data objects.
The meanings of startIndex and endIndex are found in the object that uses this type, because it may depend on factors like the index direction, whether depths are negative or positive, etc.
«record» Edge
+ customData: DataValue [0..*] (map) = EmptyMap
+ relationshipKind: RelationshipKind
+ sourceUri: string
+ targetUri: string
notes
Record that contains the information to define an edge between 2 nodes in a graph data model.
«record» Resource
+ activeStatus: ActiveStatusKind
+ alternateUris: string [0..1] (array) = EmptyArray
+ customData: DataValue [0..n] (map) = EmptyMap
+ lastChanged: long
+ name: string
+ sourceCount: int [0..1] = null
+ storeCreated: long
+ storeLastWrite: long
+ targetCount: int [0..1] = null
+ uri: string
notes
Record for resource descriptions on a graph. The record is actually a meta-object, not the resource itself, which in ETP are data objects. This Resource structure is used by:
- Discovery (Protocol 3) and DiscoveryQuery (Protocol 13) to provide information about the contents of a store.
- Store (Protocol 4), StoreNotification (Protocol 5) and StoreQuery (Protocol 14), where resource is encapsulated in dataObject in response messages only.
The use of the "lighter-weight" resources in ETP reduces traffic on the wire for initial inquiries such as Discovery, which allows customer applications to determine when to do the "heavy lifting" of getting the full data object and/or all of its associated data.
«enumeration» ObjectChangeKind
insert = 0
update = 1
authorized = 2
joined = 3
unjoined = 4
joinedSubscription = 5
unjoinedSubscription = 6
notes
Enumeration of the kinds of change that can be supplied in a notification record. Although the Store protocol uses upsert semantics for PutObject, a notification record will specify whether an object was created or replaced so that a customer can distinguish the actual type of change that occurred in the store. If a server does not know if a change type is an "insert" or an "update" use "update".
«record» SubscriptionInfo
+ context: ContextInfo
+ format: string [0..1] = xml
+ includeObjectData: boolean
+ requestUuid: Uuid
+ scope: ContextScopeKind
notes
Record for the information that a customer must provide when setting up a notification subscription, i.e., a request to be notified of any updates to objects within the context of a given URI.
«record» PartsMetadataInfo
+ customData: DataValue [0..n] (map) = EmptyMap
+ index: IndexMetadataRecord
+ name: string
+ uri: string
notes
Record to carry metadata about an ObjectPart, which helps to interpret and understand the data in the ObjectPart of a growing data object.
«record» DeletedResource
+ customData: DataValue [0..n] (map) = EmptyMap
+ deletedTime: long
+ uri: string
notes
Record for data fields retained for deleted data objects (tombstones). NOTE: The fields on DeletedResource are a subset of the fields on the Resource record and include the fields most likely to be retained for a deleted object plus customData (which the store may use to send any custom or additional information).
«record» SupportedType
+ dataObjectType: string
+ objectCount: int [0..1]
+ relationshipKind: RelationshipKind
notes
Record for data fields that must be provided for a type of data object. It MUST be populated. Client and server use this data to negotiate the objects that will be used during the session.
«record» Dataspace
+ customData: DataValue [0..n] (map) = EmptyMap
+ path: string [0..1] = EmptyString
+ storeCreated: long
+ storeLastWrite: long
+ uri: string
notes
Record containing data fields for dataspaces.
«record» PutResponse
+ createdContainedObjectUris: string [0..*] (array) = EmptyArray
+ deletedContainedObjectUris: string [0..*] (array) = EmptyArray
+ joinedContainedObjectUris: string [0..*] (array) = EmptyArray
+ unjoinedContainedObjectUris: string [0..*] (array) = EmptyArray
notes
Record used in the PutDataObjectsResponse message (Store (Protocol 4)) when putting contained data objects.
«record» ContextInfo
+ dataObjectTypes: string [0..n] (array) = EmptyArray
+ depth: int
+ includeSecondarySources: boolean = false
+ includeSecondaryTargets: boolean = false
+ navigableEdges: RelationshipKind
+ uri: string
notes
Record that is a collection of fields used to identify the part (or area) of the data model that is of interest for a given request. Used in Discovery (Protocol 3), StoreNotification (Protocol 5) and StoreQuery (Protocol 14) and other protocols.
EXAMPLE: A customer may be interested in any and all new data objects and changes to existing data objects that happen in a particular well. The customer request must specify the well (by its Energistics URI) and other relevant information using the other fields in this ContextInfo record.
This ContextInfo is based on the notion of Energistics data models as graphs. For more information, see Section 8.1.1.
«enumeration» ContextScopeKind
self = 0
sources = 1
targets = 2
sourcesOrSelf = 3
targetsOrSelf = 4
notes
Energistics data models can be considered directed graphs. (For more information on this concept, see Section 8.1.1).
For certain ETP operations (such as Discovery (Protocol 3) and notifications (StoreNotification (Protocol 5) and GrowingObjectNotification (Protocol 7) and others) you must specify a "context" (ContextInfo), which simplistically is where in the data model (at what node/data object) you want to start the operation and what direction you want to navigate.
ContextScopeKind lets you specify the "direction" in the graph that you want the operation to navigate.
NOTE: If contextScopeKind = "self" then depth in ContextInfo is ignored.
«enumeration» ActiveStatusKind
Active
Inactive
notes
Enumeration of possible channel or growing data object statuses. Statuses are mapped from domain data objects, such as wellbores, channels, and growing data objects.

Figure 37: Object: datatype schemas

23.34.1 ActiveStatusKind

Enumeration of possible channel or growing data object statuses. Statuses are mapped from domain data objects, such as wellbores, channels, and growing data objects.

Active Status	Description	Data Type
Active	The data object is currently producing data points. Same as ObjectGrowing = true in WITSML 1.x	
Inactive	The data object is not currently producing data points. Same as ObjectGrowing = False in WITSML 1.x	

Avro Source

```
{
     "type": "enum",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "ActiveStatusKind",
     "symbols":
     [
         "Active",
         "Inactive"
     ]
}
```

23.34.2 RelationshipKind

Energistics data models can be considered directed graphs. (For more information on this concept, see Section **8.1.1**).

For discovery and notification operations, a customer can specify the kinds of relationship it wants to be included.

Relationship	Description	Data Type
Primary	The nature of a Primary relationship has to do with organizing or grouping data objects, for example organizing Channels into ChannelSets or organizing ChannelSets into Logs. Characteristics of a Primary relationship: • One end of the relationship is almost always mandatory; that is, one object cannot exist (as a data object in the system) without the other. In the above example: A ChannelSet cannot exist without at least 1 Channel. • In Energistics data models, a ByValue relationship is ALWAYS organizational. **NOTE:** A ByValue relationship is one where one data object "contains" one or more other data objects, indicated with the ByValue construct in XML, such as ChannelSets containing Channels.	
Secondary	Secondary relationships provide additional contextual information about a data object, to improve understanding. For example, the reference from a Channel to a Wellbore. Characteristics of a Secondary relationship: • Both ends of the relationship are usually optional. • It is always specified using the Energistics Data Object Reference (DOR) construct (never the ByValue construct). For more information about DORs, see Energistics Online.	
Both	Refers to both Primary and Secondary relationships.	

Avro Source

```
{
     "type": "enum",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "RelationshipKind",
     "symbols":
     [
         "Primary",
         "Secondary",
         "Both"
     ]
}
```

23.34.3 ContextScopeKind

Energistics data models can be considered directed graphs. (For more information on this concept, see Section **8.1.1**).

For certain ETP operations (such as Discovery (Protocol 3) and notifications (StoreNotification (Protocol 5) and GrowingObjectNotification (Protocol 7) and others) you must specify a "context" (ContextInfo), which simplistically is where in the data model (at what node/data object) you want to start the operation and what direction you want to navigate.

ContextScopeKind lets you specify the "direction" in the graph that you want the operation to navigate.

NOTE: If contextScopeKind = "self" then **depth** in ContextInfo is ignored.

Context Scope	Description	Data Type
self	The data object as specified in the context URI. If contextScopeKind = "self", then *depth* in ContextInfo is ignored.	int
sources	For a complete definition of *sources*, see Section **8.1.1**.	int
targets	For a complete definition of *targets*, see Section **8.1.1**.	int
sourcesOrSelf	Those objects in the data model that are sources of self or self (the data object referred to by the URI in ContextInfo). For a complete definition of *sources*, see Section **8.1.1**.	int
targetsOrSelf	Those objects in the data model that are targets of self or self (the data object referred to by the URI in ContextInfo). For a complete definition of *targets*, see Section **8.1.1**.	int

Avro Source

```
{
    "type": "enum",
    "namespace": "Energistics.Etp.v12.Datatypes.Object",
    "name": "ContextScopeKind",
    "symbols":
    [
        "self",
        "sources",
        "targets",
        "sourcesOrSelf",
        "targetsOrSelf"
    ]
}
```

23.34.4 ObjectChangeKind

Enumeration of the kinds of change that can be supplied in a notification record. Although the Store protocol uses upsert semantics for PutObject, a notification record will specify whether an object was created or replaced so that a customer can distinguish the actual type of change that occurred in the store. If a server does not know if a change type is an "insert" or an "update" use "update".

Object Change	Description	Data Type
insert	Object has been inserted (or added) to a store.	int
update	The object has been updated in the store or the store cannot determine if the object has been inserted or updated.	int
authorized	A user has been authorized (given permissions) to a data object.	int
joined	A data object now references another data object with a ByValue reference. The contained object is said to be "joined" to the container object. For more information, about containers and contained objects, see Section **9.1.3**.	int
unjoined	A contained data object has been "removed" from its container data object. The contained object is said to be "unjoined" from the container object. For more information, about containers and contained objects, see Section **9.1.3**.	int
joinedSubscription	A data object has been added to the scope and context of a StoreNotification (Protocol 5) subscription.	int
unjoinedSubscription	A data object has been removed from the scope and context of a StoreNotification (Protocol 5) subscription.	int

Avro Source

```
{
     "type": "enum",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "ObjectChangeKind",
     "symbols":
     [
         "insert",
         "update",
         "authorized",
         "joined",
         "unjoined",
         "joinedSubscription",
         "unjoinedSubscription"
     ]
}
```

23.34.5 record: DataObject

Record that must carry a single data object. This record encapsulates a Resource record, which contains most of the metadata, and carries the object data as a byte array. To specify the format of the data (e.g., XML or JSON) use the *format* field. If the data object is too large (binary large object--BLOB) for the WebSocket message size, use the *blobId* field to identify the BLOB and ***Chunk*** messages to send actual data.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "DataObject",
     "fields":
     [
         { "name": "resource", "type": "Energistics.Etp.v12.Datatypes.Object.Resource" },
         { "name": "format", "type": "string", "default": "xml" },
         { "name": "blobId", "type": ["null", "Energistics.Etp.v12.Datatypes.Uuid"] },
         { "name": "data", "type": "bytes", "default": "" }
```

```
    ]
}
```

Field Name	Description	Data Type	Min	Max
resource	Contains high-level metadata about the data object being transferred, in the form of a Resource record.	Resource	1	1
format	Specifies the format (e.g., XML or JSON) of the data for the data object in this record or in correlated ***Chunk*** messages. When included in a request, this MUST be a format that was negotiated when establishing the session. When included in a response, this MUST match the format in the request. When included in a notification, this MUST match the format in the SubscriptionInfo record for the subscription. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.	string	0	1
blobId	Used when a binary large object (BLOB) will be sent in several ***Chunk*** messages (NOT in the *data* field of this ***DataObject*** record). Must be of type ***Uuid*** (Section **23.6**). The *blobId* MUST be a UUID, and it MUST be unique within an ETP session. When you populate the *blobId* field, the *data* field (on the DataObject record) MUST be empty. Populating the *blobId* field means that the actual data will be sent in the ***Chunk*** message (not in the ***DataObject*** record). For more information about how blob IDs are assigned and used, see Section **3.7.3.2**.	Uuid	0	1
data	A byte array containing the encoded object, as per the *format* field above. Note, for StoreNotification (Protocol 5) messages, if *includeObjectData* is false in the ***NotificationRequest*** record, this field has zero bytes. If the *blobId* field is populated, this field MUST be empty.	bytes	0	1

23.34.6 record: ObjectPart

Record that must carry a single object part. This structure includes the part identifier (UID) and (optionally) the part data as a byte array.

NOTE: The format of the data (e.g., XML or JSON) for the part is specified in the *format* field of the message this record is included in.

Avro Schema

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Datatypes.Object",
    "name": "ObjectPart",
    "fields":
    [
        { "name": "uid", "type": "string" },
        { "name": "data", "type": "bytes" }
    ]
}
```

Field Name	Description	Data Type	Min	Max
uid	The ID of the part contained in this record. The UID MUST be unique within the parent growing data object.	string	1	1
data	The data being sent for one part.	bytes	1	1

23.34.7 record: ObjectChange

A record describing a single data object change event.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "ObjectChange",
     "fields":
     [
         { "name": "changeKind", "type":
"Energistics.Etp.v12.Datatypes.Object.ObjectChangeKind" },
         { "name": "changeTime", "type": "long" },
         { "name": "dataObject", "type": "Energistics.Etp.v12.Datatypes.Object.DataObject" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
changeKind	The kind of change that occurred, which must be one of the enumerations listed in ObjectChangeKind.	ObjectChangeKind	1	1
changeTime	The time the data-change event occurred. This is not the time the event happened in the "real world"; it is the time that the change occurred in the store database (as indicated by the *storeLastWrite* field; for more information, see Resource). It must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
dataObject	If the customer requested that object data be included in the subscription (the *includeObjectData* field was true on the SubscriptionInfo record of the SubscribeNotifications message that created the subscription) then this field contains the full object data, as specified in DataObject. Otherwise it only contains the resource, which is also specified in DataObject.	DataObject	1	1

23.34.8 record: IndexInterval

A record describing a pair of indexes that comprise an interval, normally a time or depth interval. The values share a UOM and a depth datum (if applicable), which are also included in this record.

This structure is used by channel data objects and growing data objects.

The meanings of startIndex and endIndex are found in the object that uses this type, because it may depend on factors like the index direction, whether depths are negative or positive, etc.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "IndexInterval",
     "fields":
     [
         { "name": "startIndex", "type": "Energistics.Etp.v12.Datatypes.IndexValue" },
         { "name": "endIndex", "type": "Energistics.Etp.v12.Datatypes.IndexValue" },
         { "name": "uom", "type": "string" },
         { "name": "depthDatum", "type": "string", "default": "" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
startIndex	The index that defines the beginning of the interval. Must be of data type IndexValue. Use of a null implies infinity.	IndexValue	1	1
endIndex	The index that defines the end of the interval. Must be of data type IndexValue. Use of a null implies infinity.	IndexValue	1	1
uom	The unit of measure for the indexes in this interval.	string	1	1
depthDatum	If the indexes are depths, a value must be provided. For a time or other non-depth index, the datum is implied and is described elsewhere in the documentation.	string	0	1

23.34.9 record: PutResponse

Record used in the PutDataObjectsResponse message (Store (Protocol 4)) when putting contained data objects.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "PutResponse",
     "fields":
     [
         {
             "name": "createdContainedObjectUris",
             "type": { "type": "array", "items": "string" }, "default": []
         },
         {
             "name": "deletedContainedObjectUris",
             "type": { "type": "array", "items": "string" }, "default": []
         },
         {
             "name": "joinedContainedObjectUris",
             "type": { "type": "array", "items": "string" }, "default": []
         },
         {
             "name": "unjoinedContainedObjectUris",
             "type": { "type": "array", "items": "string" }, "default": []
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
createdContainedObjectUris	An array of the URIs of the contained data objects that were created as a result of the put operation. The URIs MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.	string	0	*
deletedContainedObjectUris	An array of the URIs of the contained data objects that were deleted (pruned) as a result of the put operation. The URIs MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.	string	0	*
joinedContainedObjectUris	An array of the URIs of the existing contained data objects that were joined ("linked") to a	string	0	*

Field Name	Description	Data Type	Min	Max
	container data object as a result of the put operation. The URIs MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.			
unjoinedContainedObjectUris	An array the URIs of the contained data objects that were unjoined ("unlinked") from a container data object as a result of the put operation. The URIs MUST be canonical Energistics data object URIs; for more information, see **Appendix: Energistics Identifiers**.	string	0	*

23.34.10 record: Dataspace

Record containing data fields for dataspaces.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "Dataspace",
     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "path", "type": "string", "default": "" },
         { "name": "storeLastWrite", "type": "long" },
         { "name": "storeCreated", "type": "long" },
         {
             "name": "customData",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
uri	The URI for a dataspace. It MUST be a canonical Energistics URI. For more information, see Section **25.3.6**. The URI MUST be a canonical Energistics dataspace URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
path	The unique location associated with the dataspace, which is used to construct the dataspace's URI. **EXAMPLE:** /folder-name/project-name	string	0	1
storeLastWrite	The last time the dataspace was *written in a particular store.* This is an ETP-only field. (See also *storeCreated*). ● Its main purpose is for use in workflows for eventual consistency between 2 stores. ● It is carried in ETP only, thereby separating transport properties from data object properties. ● When a dataspace is first created in a store, the store MUST set *storeLastWrite* to the same value as *storeCreated*. ● **RECOMMENDATION:** The *storeLastWrite* value be maintained indefinitely or as long as possible.	long	1	1

Field Name	Description	Data Type	Min	Max
	The value must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).			
storeCreated	The time that the dataspace was *first created in a particular store.* This is an ETP-only field. (See also *storeLastWrite*). • Its main purpose is for use in workflows for eventual consistency between 2 stores. • It is carried in ETP only, thereby separating transport properties from data object properties. • When a dataspace is first created in a store, the store MUST set *storeLastWrite* to the same value as *storeCreated*. • **RECOMMENDATION:** The *storeLastWrite* value be maintained indefinitely or as long as possible. The value must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
customData	Allows an endpoint to send custom data, which is a data type defined by an organization other than Energistics (i.e., it's not defined by ETP or any of the Energistics domain data models). This custom data is also informally referred to as "proprietary data or content". It contains a key-value pair of custom key names and associated values. Observe these rules for specifying custom data: 1. The keys MAY BE both well-known (and thus, reserved) names as well as application- and vendor-specific names **RECOMMENDATION:** To specify the authority for a key use this convention "authority:key". 2. Keys are case sensitive. 3. The value MUST be one of the types specified in DataValue.	DataValue	0	n

23.34.11 record: Resource

Record for resource descriptions on a graph. The structure is actually a meta-object, not the resource itself, which in ETP are data objects. This Resource record is used by:

- Discovery (Protocol 3) and DiscoveryQuery (Protocol 13) to provide information about the contents of a store.
- Store (Protocol 4), StoreNotification (Protocol 5) and StoreQuery (Protocol 14), where resource is encapsulated in dataObject in response messages only.

The use of the "lighter-weight" resources in ETP reduces traffic on the wire for initial inquiries such as Discovery, which allows customer applications to determine when to do the "heavy lifting" of getting the full data object and/or all of its associated data.

Avro Schema

```
{
    "type": "record",
      "namespace": "Energistics.Etp.v12.Datatypes.Object",
      "name": "Resource",
      "fields":
      [
          { "name": "uri", "type": "string" },
          {
              "name": "alternateUris",
              "type": { "type": "array", "items": "string" }, "default": []
          },
          { "name": "name", "type": "string" },
          { "name": "sourceCount", "type": ["null", "int"], "default": null },
          { "name": "targetCount", "type": ["null", "int"], "default": null },
          { "name": "lastChanged", "type": "long" },
          { "name": "storeLastWrite", "type": "long" },
          { "name": "storeCreated", "type": "long" },
          { "name": "activeStatus", "type":
 "Energistics.Etp.v12.Datatypes.Object.ActiveStatusKind" },
          {
              "name": "customData",
              "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
 "default": {}
          }
      ]
 }
```

Field Name	Description	Data Type	Min	Max
uri	The Energistics URI for an Energistics data object. The URI MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
alternateUris	In addition to the canonical URI, a store MAY support alternate URI formats. Use this field to send one or more alternate URI format(s). **Usage Rules for STORES:** 1. To use this field, the store MUST set the endpoint capability (see EndpointCapabilityKind) *SupportsAlternateRequestUris* to true. 2. If *SupportsAlternateRequestUris* is set to false, and a store receives an alternate URI format, it MUST send error EINVALID_URI (9). 3. Alternate URIs MUST be valid Energistics URIs, but they need not be canonical URIs. (For more information, see **Appendix: Energistics Identifiers**. 4. If a store supports alternate URIs, it MUST return its allowed alternate URIs in Discovery (Protocol 3) in the GetResourcesResponse message (which uses this Resource data structure). 5. If a store supports alternate URIs, it is expected to support them in ALL protocols that it supports (see exceptions below). 6. There is no expectation that alternate URIs can be used in a different store. **Usage Rules for CUSTOMERS** 1. A customer should not populate this *alternateUris* field in the ***PutDataObjects*** message in Store (Protocol 4) or the ***PutGrowingDataObjectsHeader*** message in GrowingObject (Protocol 6) (both of those messages use Resource); it should be set to an empty array. If present, the store MUST ignore it. 2. A customer SHOULD only send/use alternate URIs (e.g., in other protocols/messages) that it received from the store (i.e., an alternate URI that the store returned in Discovery (Protocol 3) in the ***GetResourcesResponse*** message).	string	0	1

Field Name	Description	Data Type	Min	Max
name	A human-readable name for the data object. There is no expectation of uniqueness or any particular semantic for this value. The name comes from the Title field in the Citation of the data object; for more information on Citation, see http://docs.energistics.org/#COM/COM_TOPICS/COM-000-005-0-R-sv2100.html	string	1	1
sourceCount	Indicates that the DataObject resource node has links whose source is the node. This value must be one of the following: • null: there is no count or count is not relevant in the given context. • 0 (no source links exist). • a positive integer (the count of source links). **NOTE:** This field is NOT used in the following protocols and must be set to null in all operations: • Store (Protocol 4) • StoreNotification (Protocol 5) • StoreQuery (Protocol 14)	int	0	1
targetCount	Indicates that the DataObject resource node has links whose target is the node. This value must be one of the following: • null: there is no count or count is not relevant in the given context. • 0 (no target links exist). • a positive integer (the count of target links). **NOTE:** This field is NOT used in the following protocols and must be set to null in all operations: • Store (Protocol 4) • StoreNotification (Protocol 5) • StoreQuery (Protocol 14)	int	0	1
storeLastWrite	The last time the data object was *written in a particular store,* which IS NOT the same as the *lastChanged field* on a data object's ***Citation*** element. (See also *storeCreated,* which is also only on the ***Resource***). This *storeLastWrite* field may be the last time the data object was saved to a database or the last time a file was written (depending on the store). • Its main purpose is for use in workflows for eventual consistency between 2 stores. • It is carried on the ***Resource*** only (not the data object), thereby separating transport properties from data object properties. • For ANY CHANGES to a data object or its data (E.g., parts of a growing data object or channel data in a channel data object) a store MUST update *storeLastWrite*. • When a data object is first created in a store, the store MUST set *storeLastWrite* to the same value as *storeCreated*. • **RECOMMENDATION:** The *storeLastWrite* value be maintained indefinitely or as long as possible. The value must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1

Field Name	Description	Data Type	Min	Max
lastChanged	The date and time (time stamp) of the last change to the data object that this resource represents. This field must be populated from data in the Citation of the relevant data object as follows: ● If *lastUpdate* field is populated, use that value. ● If lastUpdate is NOT populated, use the value in the Creation field. For more information about Citation, see http://docs.energistics.org/#COM/COM_TOPICS/COM-000-005-0-R-sv2100.html The value must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
storeCreated	The time that the data object (that the ***Resource*** represents) was created in the store, which IS NOT the same as the *creation* field in the ***Citation*** in Energistics *common*. (See also the *storeLastWrite* field, also on the ***Resource***.) ● Its main purpose is for use in workflows for eventual consistency between 2 stores. Specifically, this field helps with an important edge case: on reconnect, an endpoint can more easily determine if while disconnected a data object was modified OR deleted and recreated. (Each of these scenarios would require different actions.) ● Like *storeLastWrite*, this *storeCreated* field is also only stored on the ***Resource***. The value must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
activeStatus	The active status for channel data object or growing data object, which must be a value in ActiveStatusKind enumeration. This field is for WITSML channel data objects and growing data objects based on the value in the data object's GrowingStatus field, which may be: ● active = A channel or growing data object is actively producing data points. ● inactive = A channel or growing object is offline or not currently producing data points.	ActiveStatusKind	1	1
customData	Allows an endpoint to send custom data, which is a data type defined by an organization other than Energistics (i.e., it's not defined by ETP or any of the Energistics domain data models). This custom data is also informally referred to as "proprietary data or content". It contains a key-value pair of custom key names and associated values. Observe these rules for specifying custom data: 1. The keys MAY BE both well-known (and thus, reserved) names as well as application- and vendor-specific names. **RECOMMENDATION:** To specify the authority for a key use this convention "authority:key". 2. Keys are case sensitive. 3. The value MUST be one of the types specified in DataValue.	DataValue	0	n

23.34.12 record: DeletedResource

Record for data fields retained for deleted data objects (tombstones). **NOTE:** The fields on ***DeletedResource*** are a subset of the fields on the Resource record and include the fields most likely to be retained for a deleted object plus *customData* (which the store may use to send any custom or additional information).

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "DeletedResource",
     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "deletedTime", "type": "long" },
         {
             "name": "customData",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
uri	The URI of a deleted Energistics data object. The URI MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
deletedTime	The time the object was deleted from the store. Must be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).	long	1	1
customData	Allows an endpoint to send custom data, which is a data type defined by an organization other than Energistics (i.e., it's not defined by ETP or any of the Energistics domain data models). This custom data is also informally referred to as "proprietary data or content". It contains a key-value pair of custom key names and associated values. Observe these rules for specifying custom data: 1. The keys MAY BE both well-known (and thus, reserved) names as well as application- and vendor-specific names. **RECOMMENDATION:** To specify the authority for a key use this convention "authority:key". 2. Keys are case sensitive. 3. The value MUST be one of the types specified in DataValue.	DataValue	0	n

23.34.13 record: Edge

Record that contains the information to define an edge between 2 nodes in a graph data model.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "Edge",
     "fields":
     [
         { "name": "sourceUri", "type": "string" },
         { "name": "targetUri", "type": "string" },
         { "name": "relationshipKind", "type":
"Energistics.Etp.v12.Datatypes.Object.RelationshipKind" },
         {
             "name": "customData",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
sourceUri	The URI for the source Energistics data object. The URI MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**. For definitions of sources and targets, see Section **8.1.1.1.1**.	string	1	1
targetUri	The URI for the target Energistics data object. The URI MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**. For definitions of sources and targets, see Section **8.1.1.1.1**.	string	1	1
relationshipKind	The kind of relationship that this edge represents as specified in RelationshipKind, which may be Contextual or Organizational. **NOTE**: For edges, RelationshipKind CANNOT be Both.	RelationshipKind	1	1
customData	Allows an endpoint to send custom data, which is a data type defined by an organization other than Energistics (i.e., it's not defined by ETP or any of the Energistics domain data models). This custom data is also informally referred to as "proprietary data or content". It contains a key-value pair of custom key names and associated values. Observe these rules for specifying custom data: 1. The keys MAY BE both well-known (and thus, reserved) names as well as application- and vendor-specific names. **RECOMMENDATION:** To specify the authority for a key use this convention "authority:key". 2. Keys are case sensitive. 3. The value MUST be one of the types specified in DataValue.	DataValue	0	*

23.34.14 record: SupportedType

Record for data fields that must be provided for a type of data object. It MUST be populated. Client and server use this data to negotiate the objects that will be used during the session.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "SupportedType",
     "fields":
```

```
     [
         { "name": "dataObjectType", "type": "string" },
         { "name": "objectCount", "type": ["null", "int"] },
         { "name": "relationshipKind", "type":
"Energistics.Etp.v12.Datatypes.Object.RelationshipKind" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
objectCount	The value is the number of instances of the type: ● -1 (instance count is unknown) ● 0 (no instances) ● a positive integer (the count of instances)	int	0	1
dataObjectType	This must be a value of dataObjectType as described in **Appendix: Energistics Identifiers**. It is the semantic equivalent of a qualifiedEntityType in OData. They ARE case sensitive. **EXAMPLES:** "witsml20.Well", "witsml20.Wellbore", "prodml21.WellTest", "resqml20.obj_TectonicBoundaryFeature" "eml21.DataAssuranceRecord" To indicate that all data objects within a data schema version are supported, you can use a star (*) as a wildcard, EXAMPLE: "witsml20.*", "prodml21.*", "resqml20.*", So "witsml20.*" means all data objects defined by WITSML v2.0 data schemas.	string	1	1
relationshipKind	The kind of relationship which must be a value from the RelationshipKind enumeration. Relationship kinds can be used in Discovery (Protocol 3) when discovering data objects (resources) on a graph. For more information, see the descriptions in the enumeration and Section **8.1.1.1.2**.	RelationshipKind	1	1

23.34.15 record: ContextInfo

Record that is a collection of fields used to identify the part (or area) of the data model that is of interest for a given request. Used in Discovery (Protocol 3), StoreNotification (Protocol 5) and StoreQuery (Protocol 14) and other protocols.

EXAMPLE: A customer may be interested in any and all new data objects and changes to existing data objects that happen in a particular well. The customer request must specify the well (by its Energistics URI) and other relevant information using the other fields in this ***ContextInfo*** record.

This ***ContextInfo*** record is based on the notion of Energistics data models as graphs. For more information, see Section **8.1.1**.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "ContextInfo",
     "fields":
```

```
     [
         { "name": "uri", "type": "string" },
         { "name": "depth", "type": "int" },
         {
             "name": "dataObjectTypes",
             "type": { "type": "array", "items": "string" }, "default": []
         },
         { "name": "navigableEdges", "type": 
"Energistics.Etp.v12.Datatypes.Object.RelationshipKind" },
         { "name": "includeSecondaryTargets", "type": "boolean", "default": false },
         { "name": "includeSecondarySources", "type": "boolean", "default": false }
     ]
}
```

Field Name	Description	Data Type	Min	Max
uri	The URI where you want to begin discovering, subscribing to notifications, or querying a store (the operation depends on which ETP sub-protocol message is using this ContextInfo record). The URI MAY be either: • A root URI, such as eml:/// or another dataspace URI (NOTE: When discovering a dataspace URI, the *scope* and *depth* fields MUST be ignored. • An Energistics URI for a data object • For DiscoveryQuery (Protocol 13), it must be a data object query URI (which includes the OData-style string used in ETP). For more information, see Chapter **14**. Depending on the message that this record is used in, it may either be required to be a canonical Energistics URI or allowed to be an alternate URI. For the rules that apply to that message, see the documentation for the message where this record is used. For more information on Energistics URIs, see **Appendix: Energistics Identifiers**.	string	1	1
depth	The "depth" or how many "levels" (or "jumps") in the data model (graph) from the starting point (specified by the URI) that you want to discover, search, or receive notifications for. **NOTES**: 1. *Depth* MUST always be greater than zero. 2. Individual domain data models specify appropriate values for *depth*. For details, see the relevant ML for ETP implementation specification (which is a companion to this main ETP Specification). **RECOMMENDATION:** For maximum efficiency in discovery and notification operations, understand how the graph is intended to work and specify an appropriate value here (i.e., for Discovery (Protocol 3) DO NOT simply set depth =1 and iterate). For more information, see Section **8.1.1**. • If *scope* = "self", then *depth* is ignored.	int	1	1
dataObjectTypes	Optionally, specify the types of data objects that you want. The default is an empty array, which means ALL data types negotiated for the current ETP session. If specific values are specified each must be a value of dataObjectType as described in Appendix: Energistics Identifiers and serialized as	string	0	n

Field Name	Description	Data Type	Min	Max
	JSON. It is the semantic equivalent of a qualifiedEntityType in OData. They ARE case sensitive. EXAMPLES: "witsml20.Well", "witsml20.Wellbore", "prodml21.WellTest", "resqml20.obj_TectonicBoundaryFeature" "eml21.DataAssuranceRecord" To indicate that all data objects within a data schema version are supported, you can use a star (*) as a wildcard, EXAMPLE: "witsml20.*", "prodml21.*", "resqml20.*", So "witsml20.*" means all data objects defined by WITSML v2.0 data schemas.			
navigableEdges	Edges in a graph indicate relationships between objects. This field indicates the type of edge (relationship) to be navigated during the discovery operation, as specified in RelationshipKind. Choices are Primary or Secondary (NOTE: This field SHOULD NOT be set to Both.) Only edges of the specified type are navigated during discovery. Use of this field helps to exclude unwanted objects being returned in Discovery.	RelationshipKind	1	1
includeSecondaryTargets	Boolean. If true, the initial candidate set of nodes is expanded with, targets (depth=1) of secondary relationships of nodes in the initial candidate set of nodes. The edges for these secondary relationships are also included. NOTE: This flag and *includeSecondarySources* MUST be applied "simultaneously" (not in sequence) so the candidate set is expanded once, not twice. Default=false	boolean	1	1
includeSecondarySources	Boolean. If true, the initial candidate set of nodes is expanded with sources (depth=1) of secondary relationships of nodes in the initial candidate set of nodes. The edges for these secondary relationships are also included. **NOTE**: This flag and *includeSecondaryTargets* MUST be applied "simultaneously" (not in sequence) so the candidate set is expanded once, not twice. Default=false	boolean	1	1

23.34.16 record: SubscriptionInfo

Record for the information that a customer must provide when setting up a notification subscription, i.e., a request to be notified of any updates to objects within the context of a given URI.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "SubscriptionInfo",
     "fields":
     [
         { "name": "context", "type": "Energistics.Etp.v12.Datatypes.Object.ContextInfo" },
         { "name": "scope", "type": "Energistics.Etp.v12.Datatypes.Object.ContextScopeKind" },
         { "name": "requestUuid", "type": "Energistics.Etp.v12.Datatypes.Uuid" },
         { "name": "includeObjectData", "type": "boolean" },
         { "name": "format", "type": "string", "default": "xml" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
context	Specifies the extent in the data model of a subscription to notifications of change, as defined in the ContextInfo record, which includes the URI of the node/data object to begin an operation, how many "levels" of relationships in the model should be navigated or included, and what specific types of data objects are of interest.	ContextInfo	1	1
scope	The scope of the subscription as enumerated in ContextScopeKind. That is, are you interested in only the data object as specified in the context URI (self)? Those objects that point to "self" (which are called "sources")? Or objects that "self" points to (which are called "targets")? Or a specified combination of these (i.e., "sourcesOrSelf" or "targetOrSelf"? For more information including definitions of targets and sources, see Section **8.1.1**.	ContextScopeKind	1	1
requestUuid	A UUID for this request (e.g., in StoreNotification (Protocol 5), SubscribeNotifications message). This MUST be a newly-generated UUID from the customer sending the request message. This ID can be used to cancel the notification later. This MUST be of datatype ***Uuid*** (Section **23.6**).	Uuid	1	1
includeObjectData	• If true, then notification MUST contain the complete data object, corresponding to the put operation in the ***ObjectChange*** message. **NOTE:** Does not apply to the ***DeleteNotification*** message. • If false, only a ***Resource*** record for the data object is sent. If the customer requires the data object, it MUST use Store (Protocol 4) to get the data object. Growing parts of data objects, such as trajectory stations or data arrays, MUST NOT be sent as a result of this parameter being set to true. To work with parts of growing data objects, you MUST use GrowingObject (Protocol 6). To work with data arrays, you must use DataArray (Protocol 9).	boolean	1	1
format	Specifies the format (e.g., XML or JSON) for data for the data objects or parts that may be sent as part of a notification (i.e., if the *includeObjectData* field is set to true). This MUST be a format that was negotiated when establishing the session. Currently, ETP MAY support "xml" and "json". Other formats may be supported in the future, and endpoints may agree to use custom formats.	string	0	1

23.34.17 record: PartsMetadataInfo

Record to carry metadata about an ObjectPart, which helps to interpret and understand the data in the ObjectPart of a growing data object.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "PartsMetadataInfo",
     "fields":
     [
         { "name": "uri", "type": "string" },
         { "name": "name", "type": "string" },
         { "name": "index", "type":
"Energistics.Etp.v12.Datatypes.ChannelData.IndexMetadataRecord" },
         {
             "name": "customData",
             "type": { "type": "map", "values": "Energistics.Etp.v12.Datatypes.DataValue" },
"default": {}
         }
     ]
}
```

Field Name	Description	Data Type	Min	Max
uri	The Energistics URI for an Energistics growing data object. The URI MUST be a canonical Energistics data object URI; for more information, see **Appendix: Energistics Identifiers**.	string	1	1
name	A one-line description or name of that growing data object to which a part belongs, which is the Title field in the data object's Citation element. NOTE: For more information on the Citation and its elements, see Energistics Online.	string	1	1
index	The metadata about the index for the parts as specified in the IndexMetadataRecord.	IndexMetadataRecord	1	1
customData	Allows an endpoint to send custom data, which is a data type defined by an organization other than Energistics (i.e., it's not defined by ETP or any of the Energistics domain data models). This custom data is also informally referred to as "proprietary data or content". It contains a key-value pair of custom key names and associated values. Observe these rules for specifying custom data: 1. The keys MAY BE both well-known (and thus, reserved) names as well as application- and vendor-specific names. **RECOMMENDATION:** To specify the authority for a key use this convention "authority:key". 2. Keys are case sensitive. 3. The value MUST be one of the types specified in DataValue.	DataValue	0	n

23.34.18 record: ChangeAnnotation

Record that indicates the interval in a channel data object or growing data object that changed and the time that change occurred.

NOTE: A store MUST aggregate overlapping change intervals/annotations and MAY aggregate change intervals/annotations for simplification and efficiency. For more information on requirements for this aggregating behavior, see Section **19.2.2**.

Avro Schema

```
{
     "type": "record",
     "namespace": "Energistics.Etp.v12.Datatypes.Object",
     "name": "ChangeAnnotation",
     "fields":
     [
         { "name": "changeTime", "type": "long" },
         { "name": "interval", "type": "Energistics.Etp.v12.Datatypes.Object.IndexInterval" }
     ]
}
```

Field Name	Description	Data Type	Min	Max
changeTime	The time of the change in the store, which MUST be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC). **NOTE:** Stores can aggregate change intervals and annotations, but must reflect the most recent change for an interval. For more information about this aggregation behavior, see Section **19.2.2**.	long	1	1
interval	The interval where the change occurred as specified in IndexInterval.	IndexInterval	1	1

23.34.19 record: ChangeResponseInfo

Record that details the information that comprises the content of these messages:

- GrowingObject (Protocol 6): GetChangeAnnotationsResponse
- ChannelSubscribe (Protocol 21): GetChangeAnnotationsResponse

It is a map of arrays of ChangeAnnotation records.

To populate the map keys:

- For GrowingObject (Protocol 6), the map keys must be the URI of the growing data object.
- ChannelSubscribe (Protocol 21), the map keys must be the string representation of the channel ID (because map keys must be strings/cannot be integers).

Avro Schema

```
{
    "type": "record",
    "namespace": "Energistics.Etp.v12.Datatypes.Object",
    "name": "ChangeResponseInfo",
    "fields":
    [
        { "name": "responseTimestamp", "type": "long" },
        {
            "name": "changes",
            "type": { "type": "map", "values": { "type": "array", "items": 
"Energistics.Etp.v12.Datatypes.Object.ChangeAnnotation" } }
        }
    ]
}
```

Field Name	Description	Data Type	Min	Max
responseTimestamp	The time that the ***GetChangeAnnotationsResponse*** message was sent. When there are no ***ChangeAnnotation*** *changes* available for a particular channel or growing data object, this timestamp serves as a "high-water mark" for that channel or growing data object. That is, it is the timestamp at which it is known that the channel or growing data object has no known historical changes that were made within the ChangeRetentionPeriod.	long	1	1

Field Name	Description	Data Type	Min	Max
	MUST be a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC).			
changes	The interval(s) that were changed for each channel data object or growing data object, as specified in ChangeAnnotation.	ChangeAnnotation	1	*

24 ETP Error Codes

When an error occurs, an endpoint sends a ***ProtocolException*** message, with an appropriate error code, as defined throughout this specification. The table in Section **24.3** lists the ETP-defined error codes.

Each error code has a Code/number (column 1), Name (column 2) and Notes (column 3), which provide a description and possible usage comments. Note that the Name and Notes columns are informational only; they are used in the documentation to make references to error codes more human readable but have no meaning on the wire. Implementers may wish to use these Names as a #define or constant name in their code, but this is not part of the ETP Specification. Additionally, implementers may want to use the Name as part of the text description in the ***ProtocolException*** message.

24.1 Error Code Numbering Scheme

The following are the general rules for assigning error codes:

1. Positive error codes are reserved for error codes defined by Energistics. The error codes currently defined by Energistics are in this chapter. New error codes may be defined in future versions of this specification or in ML implementation guides.
2. Custom error codes MAY be defined but MUST be negative. Implementers should make consistent use of any custom error codes that they define, but nothing prevents multiple implementers from each assigning their own meaning to the same error code. Implementers should use care when interpreting the meaning of custom error codes that they receive from other implementations, especially when the custom error code number matches one that they have defined. Because of this, it is recommended to use the error codes defined by Energistics whenever possible.
3. Beginning in ETP v1.2, error codes may be used in ANY ETP protocol as appropriate. Previous versions of ETP specified error codes for each protocol; only a subset of error codes were specified as "global" so those codes were defined in Core (Protocol 0). The error code numbering scheme reflects this previous assignment to specific protocols (e.g., error codes for Store (Protocol 4) are numbered 4003, 4004, etc.).

   For changes to the error codes since the previous ETP version, see Section **2.1.5**.

24.2 Domain Model-Defined Error Codes

Each Energistics domain model (WITSML, RESQML and PRODML) may define additional error codes, required for that domain. Those error codes are published in the respective ML's ETP implementation specification (a companion document to this specification). However, periodically, those ML-assigned codes will be added to the ETP error code table below. Please refer to ML implementation specifications for any domain-specific error codes that may have been issued since this specification was published.

24.3 Current ETP Error Codes

Code	Name	Notes
1	ENOROLE	The server does not support the requested role.
2	ENOSUPPORTEDPROTOCOLS	The server does not support any of the requested protocols.
3	EINVALID_MESSAGETYPE	The message type ID is either: 1) not defined at all in the ETP Specification (e.g., no schema for it); or 2) not a correct message type ID for the receiving role (**EXAMPLE:** Per this specification, only the store role may SEND a ***GetDataObjectsResponse*** message; if the store RECEIVES a ***GetDataObjectsResponse*** message, it MUST send this error code.)
4	EUNSUPPORTED_PROTOCOL	The endpoint does not support the protocol identified in a message header.
5	EINVALID_ARGUMENT	Logically invalid argument. Use this error code in any situation where a logically invalid argument is encountered.

Code	Name	Notes
6	EREQUEST_DENIED	The receiving endpoint has denied the request. **RECOMMENDATION:** Endpoints should supply an error message explaining why the request was denied. (**EXAMPLE**: If a customer attempts to change immutable fields in a data object, the store should send EREQUEST_DENIED, and the message could be “Cannot change the unit of measure for a channel".)
7	ENOTSUPPORTED	The endpoint does not support the operation.
8	EINVALID_STATE	Indicates that the message is not allowed in the current state of the protocol. **EXAMPLE:** In Protocol 21, a customer sending a ***SubscribeChannels*** message for a channel that the customer is already subscribed to, or receiving a message that is not applicable for the current operation (as defined in this specification).
9	EINVALID_URI	The URI sent is either a malformed URI, is not a valid URI format for ETP, or is not appropriate for specific requirements of a field in a message. **EXAMPLE:** If a customer sends an alternate URI format to a store that does not accept/support alternate URIs, the store MUST send this error code.
10	EAUTHORIZATION_EXPIRED	Sent from server to client when the server is about to terminate the session because of an expired authorization.
11	ENOT_FOUND	Used when a resource, a data object, part or range is not found. May be used in any situation, as appropriate.
12	ELIMIT_EXCEEDED	Sent by either endpoint if a request, response, or notification exceeds allowed or stated limits specified by the other endpoint when a more specific error code has not been specified. **EXAMPLES:** • In Protocol 13 (DataArray) if the customer attempts to put an array into the store that exceeds the store’s MaxDataArraySize capability. • In Protocol 21 (ChannelSubscribe) if a customer exceeds a store’s value for MaxStreamingChannelsSessionCount capability.
13	ECOMPRESSION_NOTSUPPORTED	Sent by either endpoint when it receives a message whose ***MessageHeader*** has a *protocolId* field = 0 AND whose message body is compressed. (That is, messages defined and used in Core (Protocol 0) MUST NEVER be compressed.)
14	EINVALID_OBJECT	Sent in any protocol when either role sends an invalid XML document. **NOTE:** ETP does not distinguish between malformed and well-formed but invalid for this purpose. The same error message is used in both cases.
15	EMAX_TRANSACTIONS_EXCEEDED.	The maximum number of transactions per ETP session has been exceeded. Currently, Transaction (Protocol 18) is the only ETP protocol that has the notion of a "transaction" and allows only 1 transaction per session.
16	EDATAOBJECTTYPE_NOTSUPPORTED	The data object type is not supported by the endpoint or was not negotiated for use during the current ETP session.
17	EMAXSIZE_EXCEEDED	Sent from a store to a customer when the customer attempts a get or put operation that exceeds the store's maximum advertised values for MaxDataObjectSize, MaxPartSize, or MaxDataArraySize capabilities.
18	EMULTIPART_CANCELLED	Sent by either role to notify of canceled transmission of multipart response or request. **EXAMPLE:** When an endpoint's advertised value for the MaxConcurrentMultipart endpoint capability has been exceeded.
19	EINVALID_MESSAGE	Sent by either role when it is unable to de-serialize the header or body of a message.
20	EINVALID_INDEXKIND	Sent by either role when an index kind used in a message is invalid for the data.
21	ENOSUPPORTEDFORMATS	The server does not support any of the client's supported formats.
22	EREQUESTUUID_REJECTED	Sent by the store when it rejects a customer-assigned request UUID (*requestUuid*), most likely because the request UUID is not unique.

Code	Name	Notes
23	EUPDATEGROWINGOBJECT_DENIED	Sent by a store when a customer tries to update an existing growing data object (i.e., do a put operation) using Store (Protocol 4) or includes parts when updating a growing data object header using GrowingObject (Protocol 6).
24	EBACKPRESSURE_LIMIT_EXCEEDED	If the sender's queuing capacity is exhausted and it is imminently unable to send a message to the receiver, the sender MUST attempt to send this error and then attempt to send the ***CloseSession*** message. Sender MUST then close the connection, regardless of whether or not the ***ProtocolException*** and ***CloseSession*** messages were sent.
25	EBACKPRESSURE_WARNING	If sender starts to detect sending backpressure (e.g., queues of outgoing messages are starting to fill up), sender MAY send this warning.
26	ETIMED_OUT	May be sent by either role to cancel an operation when the response time for a relevant operation is exceeded, such as ResponseTimeoutPeriod or MultipartMessageTimeoutPeriod capabilities.
27	EAUTHORIZATION_REQUIRED	Sent from an endpoint during session negotiation (and ONLY during session negotiation) to indicate that the other endpoint requires authorization.
28	EAUTHORIZATION_EXPIRING	Optionally sent from an endpoint when the other endpoint's authorization will expire soon. The receiving endpoint should follow the necessary authorization workflow to renew its authorization. If it does not, the sending endpoint will eventually terminate the connection. The precise definition of "soon" and the required re-authorization workflow are intentionally out of the scope of the ETP Specification.
29	ENOSUPPORTEDDATAOBJECTTYPES	The server does not support any of the client's supported data object types.
30	ERESPONSECOUNT_EXCEEDED	Sent by a store endpoint to terminate a non-map response once the number of responses sent has reached the allowed or stated limits specified by the relevant capabilities. This lets customers know that the store has more data than it could return to the customer. **EXAMPLES:** • In Protocol 3 (Discovery) and all query protocols, sent by the store if it must stop sending responses to the customer because it has already sent MaxResponseCount responses to a customer request. • In Protocol 21 (ChannelSubscribe), sent by the store if it must stop sending data points to a customer in response to a ***GetRanges*** request because the store has already sent MaxRangeDataItemCount data points in response to the request.
31	EINVALID_APPEND	Sent in response to a ***ChannelData*** message that is not appending data to a channel.
32	EINVALID_OPERATION	Sent in response to a request when the requested operation would be invalid. **EXAMPLE:** In Protocol 6 (GrowingObject), a ***ReplacePartsByRange*** message where some replacement parts are not covered by the delete range is an invalid operation.
1002	EINVALID_CHANNELID	Sent by either role when operations are requested on a channel ID that is not valid for the session.
4003	ENOCASCADE_DELETE	Sent when an attempt is made to delete an object that has children and the store does not support cascading deletes (prune operations).
4004	EPLURAL_OBJECT	Sent when an endpoint attempts put operations for more than one data object under the plural root of a 1.x Energistics data object. ETP only supports a single data object, one XML document. ETP 1.2 is not designed to work with 1.x Energistics data objects, but this error code is left for use with custom protocols.
5001	ERETENTION_PERIOD_EXCEEDED	Sent from a store to a customer when the client asks for changes beyond the stated change period of a server.

Code	Name	Notes
6001	ENOTGROWINGOBJECT	Sent from a store to a customer when the customer attempts to perform a growing object operation on an object that is not defined as a growing data object type. **EXAMPLE:** A store would send this if the customer attempted to add parts to a WITSML well object, which is not a growing data object.

25 Appendix: Energistics Identifiers

This appendix serves as an interim *Energistics Identifier Specification* until that document can be updated, reviewed, and published. These definitions and rules MUST be observed with the Energistics Transfer Protocol (ETP) v1.2.

This appendix describes the syntax and semantics of data object and dataspace identifiers as used within the Energistics family of data exchange standards and ETP.

25.1 Definitions: Data Objects, Resources, and Dataspaces

Messages in ETP are the mechanism to communicate about and perform actions on data objects and resources. Some definitions for context:

- Energistics domain specifications—WITSML, RESQML, PRODML and EML (i.e., Energistics *common*, which is shared by the other 3 MLs)—define **data objects,** which represent real-world business objects such as wellbores, logs, channels, earth models, faults, production reports, and PVT data, to name a few.

  A **data object** is a valid document of the specified format (XML, JSON, other), which conforms to one of the schemas specified in the Energistics namespace and inherits from AbstractObject, which is defined in Energistics *common*.

  Energistics has these broad categories of data objects, each of which has some specific considerations when being operated upon by the various sub-protocols that comprise ETP:

  – **"static" data objects**. These are informally referred to as "static" (compared to "growing"; see below) because they change only when people, process, and/or software change them. Additionally, they may have a "main" object (sometimes called a "header" object) and associated arrays of numeric data.

  – **"growing" data objects**. Objects that change inherently over time by adding to them. These objects typically exist in the drilling domain and are defined in WITSML, such as trajectories (grows as new trajectory stations are added), and "mud logs" (now called wellbore geology) which grow in several ways with the evaluation and recordings at different intervals for geological cuttings samples, lithology sequences along the length of the wellbore, and interpretations of the quality of hydrocarbon shows along the wellbore).

  – **channel data objects**. A channel is a series of values, usually measured or calculated, that are referenced to one or more indexes, usually time or depth. Groups of channels are informally called “logs” and individual channels are sometimes referred to as “curves”. Channels are similar to growing data objects, but they are important enough and different enough to be treated as a distinct type of object. In ETP, the channel protocols are dedicated to handling channel data.

  – **"contained" and "container" data object.** A contained data object refers to a data object that is contained by another data object (the container) with a ByValue reference (and ONLY a ByValue reference; i.e., relationships specified by an Energistics Data Object Reference (DOR) do not result in container/contained objects). An Energistics data object MAY be included in one or more container data objects.

    One of the best-known examples come from WITSML where:

    ▪ One or more Channel data objects can be contained in one or more ChannelSet data objects. In this example, the Channels are the "contained" data objects and the Channel Set is the "container".

    ▪ One or more ChannelSet data objects can be contained in one or more Log data objects. In this example, the ChannelSets are the "contained" data objects and the Log is the "container".

    NOTE: Individual data objects that may be containers or contained data objects are listed in the relevant ML's ETP implementation specification (which is a companion document to this *ETP*

Specification). For example, Channel, Channel Set and other contained data objects defined in WITSML are listed in the *ETP v1.2 for WITSML v2.0 Implementation Specification*.

- A **resource** is a meta-object that contains information that identifies an actual data object. A resource contains a mix of fields: some fields are from the data object and some fields are from the data object as instantiated on a particular store, for example, the *storeLastWrite* field. The use of the "lighter-weight" resources for some use cases in ETP reduces traffic on the wire for initial inquiries (such as discovery operations), which allows customer applications to determine when to do the "heavy lifting" of getting the full data object and/or all of its associated data.
- A **dataspace** is an abstraction representing a distinct collection of data objects, such as a project or a specific database. (For more information, see Section **21.1.1**.)

25.2 Mechanisms for Identification: UUIDs and URIs

Energistics has 2 main mechanisms for identification: UUIDs and URIs. This section explains UUIDs; for information about URIs, see Section **25.3**.

- Unique instances of **data objects** defined by Energistics data models must be identified with a **UUID** as defined by RFC 4122 (https://tools.ietf.org/html/rfc4122). A UUID is an array of 16 unsigned bytes (or a single 128-bit unsigned integer), and can be printed and serialized in various ways.
  - For use in Energistics domain standards, for string representation of a data object, a UUID MUST be serialized using Microsoft Registry Format; that is, with dashes inside the UUID and without curly braces.
  - For use in ETP messages–with the exception of string representation of data objects that may be conveyed with message (see next bullet) –ETP uses the ***Uuid*** datatype (Section **23.6**) to send UUIDs. The ***Uuid*** data type is encoded as an array of 16 bytes in big-endian format. EXAMPLE: The UUID "00112233-4455-6677-8899-aabbccddeeff" is encoded as the byte array [0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff] in the ***Uuid*** data type.
- The application that first creates a data object assigns its UUID. If during data transfer an application changes the UUID of an object, that application MUST preserve the original UUID as an alias and it is up to the application to change the authority.

25.3 Energistics URIs

An Energistics uniform resource identifier (URI) provides a mechanism to identify dataspaces and data objects. Energistics URIs are formatted according to RFC 3986 (Uniform Resource Identifier (URI): Generic Syntax (https://tools.ietf.org/html/rfc3986)), and, beginning with ETP v1.2, Energistics URIs are based on OData URI syntax (http://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part2-url-conventions.html) with some tailoring as described here.

IMPORTANT: The URI formats specified here are for ETP v1.2 and higher (ETP v1.2+). Previous versions of ETP used a different URI format.

This section:

- Defines the term canonical URI.
- Specifies the form of canonical Energistics URIs for dataspaces, data objects, and queries that will match a collection of data objects within a dataspace and provides examples.
- Defines so-called "alternate" URIs and their usage.
- Provides regular expressions (REGEXes) and examples of each.

25.3.1 Requirements for Supporting URIs

An endpoint in an ETP session:

- MUST support the canonical Energistics URIs.

- MUST support **eml:///**, which is the URI for the default dataspace, which may or may not be empty.
- MAY support alternate URI formats, which are explained in Section **25.3.9**.

25.3.2 Overview

Energistics URIs provide a flexible way to identify dataspaces and data objects within dataspaces. By building on OData URI syntax, Energistics URIs can represent:

- individual dataspaces and data objects
- hierarchical relationships between objects
- sub-elements within data objects
- queries for collections of objects

ETP v1.2+ uses a subset of Energistics URIs to identify:

- Dataspaces
- Individual data objects within a dataspace
- A query that will match a collection of data objects within a dataspace

When both ETP endpoints in a session can support them, ETP v1.2+ allows optional use of other forms of Energistics URIs in some protocol messages, which may have application-specific meaning. For more information on optional use of other URI forms, see Sections **25.3.4** and Section **25.3.9**.

25.3.3 URI Notation

When describing the form of URIs in this document:

- { } indicate a parameter that is substituted with an actual value
- [ ] indicate an optional portion of the URI, which may be omitted

25.3.4 Canonical URIs

Because of the flexibility of Energistics URIs, many URIs can be semantically equivalent—that is, they identify the same unique dataspace or data object or they represent the same query.

A **canonical URI** is the preferred, and often shortest, URI out of a set of semantically equivalent URIs.

In ETP, observe these rules about use of canonical URIs:

- ETP endpoints MUST support canonical Energistics URIs. In some ETP messages, their use is always required. In other messages, their use is required unless both ETP endpoints in the session support alternate forms of Energistics URIs.
- Even when the use of canonical URIs is optional, use of canonical URIs MUST always be supported.

25.3.5 Canonical Energistics URIs

This section defines the form of canonical Energistics URIs for ETP v1.2+.

25.3.6 Dataspace URIs

Dataspace URIs identify individual dataspaces, which contain zero or more data objects. ETP supports named dataspaces, which use a path as a name, and the default dataspace, which has no name.

- The canonical form of the default dataspace URI is:
  eml:///
- The canonical form for named dataspace URIs is:
  eml:///dataspace('{path}')
- An example dataspace URI is:
  eml:///dataspace('/folder-name/project-name')

- For named dataspaces, the path may be a relative path. For example:
  eml:///dataspace('rdms-db')

Observe these rules for dataspace URIs:

- In addition to named dataspaces, all ETP stores and producers MUST support the default, nameless dataspace, which is identified by the empty string.
  - While the default dataspace MUST be supported, it MAY be empty; that is, it may not have any data objects in it.

IMPORTANT: The default dataspace is NOT an alias for a named dataspace. It is a simplification for ETP stores and producers that do not need to support named dataspaces.

25.3.7 Data Object URIs

A data object URI is one that provides direct reference to a single data object in a dataspace contained behind an ETP endpoint. A data object URI may optionally refer to a specific version of a data object.

- The canonical form of a data object URI without a version is:
  eml:///[dataspace('{path}')/]{DataObjectType}({uuid})
- The canonical form of a data object URI with a version is:
  eml:///[dataspace('{path}')/]{DataObjectType}(uuid={uuid},version='{version}')

Example data object URIs are:

- eml:///witsml20.ChannelSet(2c0f6ef2-cc54-4104-8523-0f0fbaba3661)
- eml:///dataspace('rdms-db')/resqml20.obj_HorizonInterpretation(uuid=421a7a05-033a-450d-bcef-051352023578,version='2.0')

25.3.7.1 Data Object Types

In ETP, a data object type (dataObjectType) is the semantic equivalent of a qualifiedEntityType in OData. It is composed of:

- The Energistics domain standard or Energistics *common* (designated by *eml*) and version where the data object type is defined.
- The data object type name as defined by its schema.

Examples:

- witsml20.Well
- resqml20.UnstructuredGridRepresentation
- prodml20.ProductVolume
- eml21.DataAssuranceRecord

NOTES:

1. The *qualifiedType* field on the ***SupportedDataObjec***t record (used on the ***RequestSession*** and ***OpenSession*** messages) is a *dataObjectType* and uses the rules below for deriving a data object type.
2. DataObjectType replaces the ContentType, which was specified in the last-published version of the *Energistics Identifier Specification* (see the topic ContentType in Energistics Online). ContentType is an important component of the DataObjectReference (DOR) (see ObjectReference in Energistics Online), which is an important mechanism for defining the graph—the relationships between objects in Energistics data models. For more information on changes to DORs and how this works, see Section **8.2.2**.

You MUST follow these rules to create a valid dataObjectType (or qualifiedType):

1. The Energistics domain standard MUST be one of the three Energistics domain standards or *eml* (for

data object types that are defined in Energistics *common* for use by the domain standards), all in lower case concatenated with the first 2 digits of its version. For example, RESQML v2.0.1 would use: *resqml20.* Supported versions of the other Energistics standards include: *witsml20, prodml20, prodml21, eml20, eml21,* and *eml22.*

2. The data object type name MUST be the schema name of the data object as defined in the Energistics standard. It IS case sensitive.

25.3.7.2 Rules for Using Dataspaces and Version in Data Object URIs

Observe these rules for using dataspaces and version in data object URIs:

- If the dataspace is the default dataspace, then the dataspace segment MUST be omitted from the canonical URI.
- Data objects identified by Energistics URIs are always in a dataspace, so the data object URI is prefixed with the relevant dataspace.
  - If the dataspace is omitted from the URI, then the URI implicitly refers to the default dataspace.
- If version is omitted and there are multiple versions of a data object behind an ETP endpoint, then the URI implicitly refers to the most recent version.
  - ETP 1.2 does not provide rules that define which of two versions of a data object is the most recent version. The data object version that is most recent is ETP-endpoint-dependent.
  - If the intent is to refer to the most recent version of the data object, then the version segment SHOULD be omitted from the canonical URI.

The data object URI uses these conventions from OData:

- The data object type in a data object URI is semantically equivalent to an OData qualifiedEntityType; for example: witsml20.Well (as described above).
- The specification of the uuid and the optional version are semantically equivalent to keys in OData collections.

25.3.8 Data Object Query URIs

A data object query URI is one that refers to a collection of data objects in a dataspace contained behind an ETP endpoint.

The canonical form of a data object query URI MUST be one of the following:

- **{DataObjectUri}/{DataObjectType}[?{query}]**
- **eml:///[dataspace('{path}')]/{DataObjectType}[?{query}]**
- **{DataObjectUri}?{query}**

Example data object query URIs are:

- **eml:///dataspace('rdms-db')/resqml20.obj_HorizonInterpretation**
- **eml:///witsml20.Well(uuid=ec8c3f16-1454-4f36-ae10-27d2a2680cf2,version='1.0')/witsml20.Wellbore**
- e**ml:///witsml20.Channel?$filter=ChannelClass/Title eq 'Gamma'&$top=300**

25.3.8.1 Rules for Using Dataspaces and Version in Data Object Query URIs

Observe these rules for using dataspaces and version in data object query URIs:

- If the dataspace is the default dataspace, then the dataspace segment MUST be omitted from the canonical URI.
- If the intent is to refer to the most recent version of the data object, then the version segment SHOULD be omitted from the canonical URI.
- A data object query URI MAY specify an OData Entity Collection. That is, a data object type without associated uuid or version. This represents a query for objects of the specified type.

- When a data object query URI includes a specific data object uuid, the query operates on data objects that have some relationship to the data object specified by the uuid.
- Whether the relationship is primary or secondary or goes from sources to targets or targets to sources depends on other contextual information where the URI is used. **EXAMPLE:** In DiscoveryQuery, the *context* and *scope* fields on the ***FindResources*** message will provide this information.
- A data object query URI MAY also include a URI query string (for details, see Chapter 14).
  - When the URI path ends with an OData Entity Collection, the query string is optional (because the OData Entity Collection represents an implicit query).
  - When the URI path ends with a specific data object, the query string is required.

25.3.9 Alternate URIs

In some situations, ETP v1.2+ also allows applications to use so-called **alternate URIs**. These URIs MUST be valid Energistics URIs, but they need not be canonical URIs. Alternate URIs may have application-specific meaning.

NOTE: For alternate URIs to be used in an ETP session, the store MUST return the allowed alternate formats in Discovery (Protocol 3). For more information, see Chapter **8**.

Here is a non-exhaustive list of alternate URI forms and examples of them:

1. URIs prefixed with eml:/ instead of eml:///
2. Hierarchical data object URIs where more than one path segment uniquely identifies a data object and the path indicates a navigable relationship between the objects.
   eml:///witsml20.Well(ec8c3f16-1454-4f36-ae10-27d2a2680cf2)/witsml20.Wellbore(81bb7920-fa42-48cb-b9ac-38031e2703a8)
3. Template URIs where multiple path segments specify a data object type.
   eml:///witsml20.Well/witsml20.Wellbore
4. URIs with hash segments.
   eml:///resqml20.obj_HorizonInterpretation(421a7a05-033a-450d-bcef-051352023578)#hash
5. Dataspace URIs with query segments.
   eml:///dataspace('rdms-db')?$filter=Name eq 'mydb'
6. URIs with path segments that address elements within data objects.
   eml:///witsml20.Channel(53b3bf2b-3aa3-458d-b40c-9a4cb754210e)/ChannelClass/Title
7. URIs that include the primary key name:
   eml:///witsml20.Well(uuid=ec8c3f16-1454-4f36-ae10-27d2a2680cf2)
8. URIs that use alternate OData keys:
   eml:///witsml20.ChannelSet(2c0f6ef2-cc54-4104-8523-0f0fbaba3661)/witsml20.Channel(Mnemonic='HKLD')
9. URIs that support earlier ML versions:
   eml:///witsml14.well(uid='abc')/witsml14.wellbore(uid='def')

Observe these rules for using alternate URIs:

- To use alternate URIs in an ETP session, BOTH ETP endpoints MUST have set the SupportsAlternateRequestUris endpoint capability to "true" in the ***RequestSession*** and ***OpenSession*** messages (in their respective *endpointCapabilities* fields) that were exchanged to establish the ETP session.
- ETP does not provide functionality for an endpoint to advertise all possible alternate URIs it supports.

- An ETP endpoint that wants to use alternate URIs in requests SHOULD assume the other endpoint in the session supports only alternate URIs it has explicitly received in response to previous requests.
- Even if an endpoint indicates it supports alternate URIs, it is NOT required or guaranteed that all possible forms of alternate URIs are supported.

25.3.10 Regular Expressions for Validating Canonical Energistics URIs

The following regular expressions can be used to validate canonical Energistics URIs. These regular expressions use ECMAScript regular expression syntax.

- **Canonical Dataspace URIs:**

  ```
  ^eml:\/\/\/(?:dataspace\('(?<dataspace>[^']*?(?:''[^']*?)*)'\))?$
  ```

  EXAMPLES:

  - eml:///
  - eml:///dataspace('/folder-name/project-name')
  - eml:///dataspace('rdms-db')

- **Canonical Data Object URIs:**

  ```
  ^eml:\/\/\/(?:dataspace\('(?<dataspace>[^']*?(?:''[^']*?)*)'\)\/)?(?<domain>witsml|resqml|prodml|eml)(?<domainVersion>[1-9]\d)\.(?<objectType>\w+)\((?:(?<uuid>[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12})|uuid=(?<uuid2>[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}),version='(?<version>[^']*?(?:''[^']*?)*)')\)$
  ```

  EXAMPLES:

  - eml:///witsml20.Well(ec8c3f16-1454-4f36-ae10-27d2a2680cf2)
  - eml:///witsml20.Well(uuid=ec8c3f16-1454-4f36-ae10-27d2a2680cf2,version='1.0')
  - eml:///dataspace('/folder-name/project-name')/resqml20.obj_HorizonInterpretation(uuid=421a7a05-033a-450d-bcef-051352023578,version='2.0')

- **Canonical Data Object Query URIs with both Data Object and OData Entity Collection:**

  ```
  ^eml:\/\/\/(?:dataspace\('(?<dataspace>[^']*?(?:''[^']*?)*)'\)\/)?(?<domain>witsml|resqml|prodml|eml)(?<domainVersion>[1-9]\d)\.(?<objectType>\w+)\((?:(?<uuid>[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12})|uuid=(?<uuid2>[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}),version='(?<version>[^']*?(?:''[^']*?)*)')\)\/(?<collectionDomain>witsml|resqml|prodml|eml)(?<collectionDomainVersion>[1-9]\d)\.(?<collectionType>\w+)(?:\?(?<query>[^#]+))?$
  ```

  EXAMPLES:

  - eml:///witsml20.Well(ec8c3f16-1454-4f36-ae10-27d2a2680cf2)/witsml20.Wellbore?query
  - eml:///witsml20.Well(uuid=ec8c3f16-1454-4f36-ae10-27d2a2680cf2,version='1.0')/witsml20.Wellbore?query
  - eml:///dataspace('/folder-name/project-name')/witsml20.Well(uuid=ec8c3f16-1454-4f36-ae10-27d2a2680cf2,version='1.0')/witsml20.Wellbore?query

- **Canonical Data Object Query URIs with OData Entity Collection but no Data Object:**

  ```
  ^eml:\/\/\/(?:dataspace\('(?<dataspace>[^']*?(?:''[^']*?)*)'\)\/)?(?<collectionDomain>witsml|resqml|prodml|eml)(?<collectionDomainVersion>[1-9]\d)\.(?<collectionType>\w+)(?:\?(?<query>[^#]+))?$
  ```

  EXAMPLES:

  - eml:///witsml20.Well?query
  - eml:///dataspace('/folder-name/project-name')/resqml20.obj_HorizonInterpretation?query

- **Canonical Data Object Query URIs with Data Object but no OData Entity Collection:**

  ^eml:\/\/\/(?:dataspace\('(?<dataspace>[^']*?(?:''[^']*?)*)'\)\/)?(?<domain>witsml|resqml|prodml|eml)(?<domainVersion>[1-9]\d)\.(?<objectType>\w+)\((?:(?<uuid>[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12})|uuid=(?<uuid2>[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}),version='(?<version>[^']*?(?:''[^']*?)*)')\)\?(?<query>[^#]+)$

  EXAMPLES:

  – eml:///witsml20.Well(ec8c3f16-1454-4f36-ae10-27d2a2680cf2)?query
  – eml:///witsml20.Well(uuid=ec8c3f16-1454-4f36-ae10-27d2a2680cf2,version='1.0')?query
  – eml:///dataspace('/folder-name/project-name')/resqml20.obj_HorizonInterpretation(uuid=421a7a05-033a-450d-bcef-051352023578,version='2.0')?query

26 Appendix: Data Replication and Outage Recovery Workflows

While the definition, support and execution of specific business use cases and related workflows are governed by the Energistics domain standards (WITSML, RESQML and PRODML), data replication and outage recovery can be seen as commonly used, high-level workflows across the domains. This appendix provides an "ML-neutral" overview of these workflows. It also provides a good example of how the ETP sub-protocols are intended to work together.

While the section on outage recovery focuses on recovery during the replication process, the principles apply to and can be used for outage recovery for most domain workflows.

For more detailed ML-specific information on these workflows, see the relevant ETP implementation guide for a particular domain standard.

26.1 Goal and Scope of Replication

The goal of data replication is for a destination data store to be eventually consistent with a source data store. A definition from Wikipedia (https://en.wikipedia.org/wiki/Eventual_consistency):

> *Eventual consistency is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.*

Leveraging new features in ETP v1.2, these replication workflows have been defined so that endpoints can reliably replicate data and recover from unintended disconnects/outages with a significantly reduced likelihood of having to "resend all data again"—which of course is costly and time consuming. The reliability features also support better decision making around when it is necessary to resend everything.

The following scenarios were explicitly NOT considered when designing the ETP features for reliable replication:

- **Multi-master replication** and **Bi-directional replication**. In multi-master replication, more than one entity may be changing data in a store at the same time. In bi-directional replication (which is a special case of multi-master replication) changes to data in either endpoint must be replicated. ETP provides features to detect when another ETP session has changed data in a destination data store, but ETP does not provide features to prevent other ETP sessions from changing data. This means that there can be race conditions when two or more sessions try to update the same data, especially considering that ETP does not support partial edits to data objects. These scenarios were not considered because they are not common, and the features to support them would be complex.
- **Handling changes to clocks in the source store**. ETP is not immune to nor can it automatically detect clock changes. If it's really important to detect clock changes, individual implementations may be able to use out-of-band communications or features of ETP (**EXAMPLE:** ***Ping*** and ***Pong*** messages). This version of ETP does NOT explicitly address this issue because correctly recovering from clock changes usually requires external intervention.

26.2 Key Concepts and Definitions for Replication

This section explains key concepts and definitions that are important to understanding how replication is intended work. RECOMMENDATION: Familiarize yourself with these concepts before reading the workflow sections.

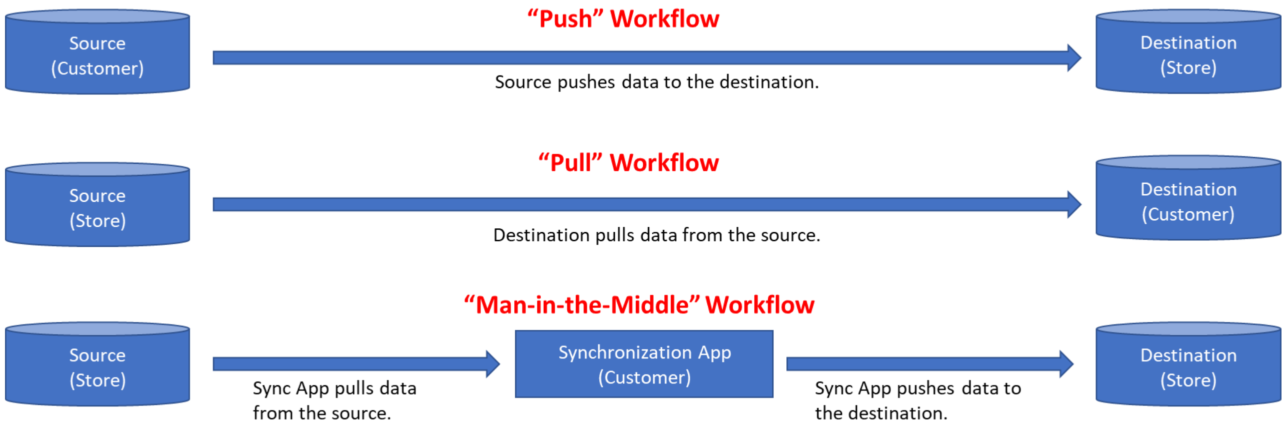
26.2.1 Change Annotations

ETP v1.2 introduces the use of change annotations. For more information, see Section **11.1.4**.

26.2.2 Replication Approaches and Related Definitions

ETP supports these data replication approaches, which are shown in **Figure 38** and further explained below the figure:

- "Push" workflow, where the source data store actively pushes data to the destination data store. For more information, see Section **26.4 Push Workflow**.
- "Pull" workflow, where the destination data store actively pulls data from the source data store. For more information, see Section **26.5** Pull Workflow.
- “Man-in-the-middle” workflow includes a specially designed synchronization application, which replicates data by executing both the pull and the push workflow using only ETP functionality, as shown in **Figure 38.**



“Push” Workflow
Source (Customer)
Source pushes data to the destination.
Destination (Store)
“Pull” Workflow
Source (Store)
Destination pulls data from the source.
Destination (Customer)
“Man-in-the-Middle” Workflow
Source (Store)
Sync App pulls data from the source.
Synchronization App (Customer)
Sync App pushes data to the destination.
Destination (Store)

Figure 38: ETP now supports push and pull data replication workflows, as well as "man-in-the-middle" synchronization applications. The words in parenthesis (either customer or store) refer to the ETP role assigned to that endpoint.

The *source* is the data store from which the data is being replicated; the *destination* is the data store to which the data is be replicated. The goal of replication is for the destination to be eventually consistent with the source.

NOTE: The source and destination DO NOT necessarily coincide with the client and server endpoints in an ETP session. (For more information about clients, servers, and ETP-assigned endpoint roles, see Section **3.1.2**.)

Source and destination are determined by whether the workflow is "push" or "pull". A key factor to how the workflows function is the ETP-assigned role of each endpoint, (in all but one ETP sub-protocol the two defined endpoint roles are "customer" or "store").

- In the push workflow, the endpoint with the customer role is the source. It controls the workflow operations by using "put" messages from the various ETP sub-protocols to push data to the store.
- In the pull workflow, the endpoint with the customer role is the destination. It controls the operations by using "subscribe" and "get" messages to pull data from the store (source).
- In the man-in-the middle workflow, the synchronization application (sync app) is the customer in both the push and pull workflows. (**NOTE:** Though the sync app has the customer role in both the pull and push workflows, two separate ETP sessions must be created: one for the pull workflow and one for the push workflow.)

In both push and pull workflows, the ETP customer role is the *active participant*, which is an informal general term used in this appendix for the ETP endpoint that is in control of the replication operation.

26.2.3 Graphs, Scope and Replication Scope

ETP is designed to interpret the data models that it operates on (WITSML, RESQML and PRODML) as graphs. Scope refers to a specified area of the graph, based on a starting node of interest specified by a URI, and further defined by a direction and depth in the data model. (For more information on graphs and related concepts including scope, see Section **8.1.1**, **Data Model as a Graph**.)

An ETP endpoint can specify a scope in the context of discovery operations (using Discovery (Protocol 3)) and in setting up subscriptions to receive change notifications and new data as they become available, which may occur using StoreNotification (Protocol 5), GrowingObjectNotification (Protocol 7) or ChannelSubscribe (Protocol 21).

The term *replication scope* refers informally to the data that an endpoint is required to replicate. Replication scope may be all of the data in a store, one or more specific data objects, or (a most likely scenario) all of the data objects associated with a particular data object (e.g., all the channels in a particular well).

EXAMPLE (from WITSML)**:** The replication scope is Well XYZ and all data objects associated with it. So this is not an explicit list of data objects (i.e., the actual set of data objects in the replication scope is Well XYZ, Wellbore ABC, ChannelSet DEF that is added later, etc.)

ETP concepts of graphs and scope along with ETP functionality (in discovery and notification protocols) help an endpoint to determine the list of data objects in the replication scope, monitor the scope for changes, and receive the necessary change data for replicating it.

26.2.4 Understanding the Workflows in this Appendix

The following sections in this appendix explain how to do these replication tasks for the "push" and "pull" approaches. (The man-in-the-middle apps use both approaches; pull on one end and push on the other.)

For brevity, the notation used in the workflows is: ***ProtocolName.MessageName***, for example, ***Store.PutDataObjects*** is the ***PutDataObjects*** message from Store (Protocol 4).

NOTE: For the details about the requirements, rules, and error handling for the message flow and protocol-level operations (e.g., how the ***PutDataObjects*** message works, fields, options, etc.) see the specific protocol chapter in this guide (e.g., Chapter **9**, **Store (Protocol 4)**).

26.3 Main Replication Tasks

Currently ETP supports the main data-replication tasks listed below. The scope of all of these tasks includes replicating subsequent changes once a replication operation has begun. (**EXAMPLE:** Data Object A is replicated from source to destination, then Data Object A is updated in the source. These workflows address how this subsequent update to Data Object A in the source is replicated in the destination, as part of the ongoing replication operation.)

All of these replication tasks can be done using either the push or pull workflow, which is explained in subsequent sections:

- **Identify Replication Scope**: Given a replication scope, identify the actual set of data objects in it, including any changes to the set over time (for more information about replication scope, see Section **26.2.3 above**).
- **Replicate** this data:
  - **Data Objects:** Replicate identified data object (for growing data objects, the "header"), including any subsequent changes.
  - **Growing data object Parts:** Replicate parts of identified growing data objects, including any subsequent changes (adds, edits, and deletes).
  - **Channel Data:** Replicate channel data from identified channels, including any subsequent changes (adds/appends, edits, and deletes).

NOTE: ETP has similar features that can be used to support replication for other objects (such as dataspaces and data arrays) however, those workflows have not yet been documented.

26.4 Push Workflow

In the push workflow, the source (with the ETP-assigned role of customer) actively pushes data to the destination (with the ETP-assigned role of store). (For more information about source and destination, see Section **26.2.1**.)

The main replication tasks in the push workflow are accomplished as follows:

- **Replication Scope Identification:**
  - The replication scope comes from an external source, for example, a contract that says what data your endpoint is expected to deliver (**EXAMPLE:** Your rigsite store as a logging/data acquisition company must replicate data for Well XYZ to the destination endpoint (e.g., a client oil company's data store)). That is, you cannot discover the replication scope with ETP functionality.
  - When a data store itself is the source, the scope details are provided in an externally supplied configuration, which the data store must use to identify the specific data objects in itself that fall within the replication scope. The data store must use internal knowledge of itself to track changes to the set of objects that fall within the scope.
  - When a synchronization application (sync app) is the source, the sync app receives the scope details from an externally supplied configuration and uses the pull workflow from the data store it is replicating to identify the specific data objects within the replication scope and any changes to the set of objects that fall within the scope.
- **Data Object Replication:** Once you have identified the data objects within the replication scope, you must replicate these data objects and any changes for each with these operations:

  For creates and updates:

  - For growing data objects (which does not include channels, which are different a type of data object), push the header using ***GrowingObject.PutGrowingDataObjectsHeader***. (For details of the message flows for this protocol, see Chapter **11 GrowingObject (Protocol 6)**.)
  - For all other data objects (including channel), push the data object using ***Store.PutDataObjects***. (For details of the message flows for this protocol, see Chapter **9 Store (Protocol 4)**.)

  For deletes:

  - For all data objects, delete the data objects using ***Store.DeleteDataObjects***. (For details of the message flows for this protocol, see Chapter **9 Store (Protocol 4)**.)
- **Growing data object Part Replication:**
  - Push part creates, changes and deletes using ***GrowingObject.PutParts***, ***GrowingObject.DeleteParts*** and/or ***GrowingObject.ReplacePartsByRange***. The recommendation is always to work as efficiently as possible; **EXAMPLE:** If you are deleting 1000 contiguous parts, you can do this using a single ***ReplacePartsByRange*** message. But the combination of messages used to replicate those changes is up to the implementer. (For details of the message flows for this protocol, see Chapter **11 GrowingObject (Protocol 6)**.)
- **Channel Data Replication:**
  - Push existing channel data using either ***ChannelDataLoad.ChannelData*** or ***ChannelDataLoad.ReplaceRange***.
  - Push new channel data (i.e., appended index/data points) using ***ChannelDataLoad.ChannelData***. (For details of the message flows for this protocol, see Chapter **20 ChannelDataLoad (Protocol 22)**.)

  - Push data edits and deletes using ***ChannelDataLoad.TruncateChannels*** and/or ***ChannelDataLoad.ReplaceRange***. (For details of the message flows for this protocol, see Chapter **20 ChannelDataLoad (Protocol 22)**.)

NOTE: To push data in this data replication workflow, it is **STRONGLY** recommended to use the ETP customer role and send the request messages listed above (instead of using the ETP store role and sending notification messages, which technically can "work") for these reasons:

- By design, request messages must give positive confirmation that the destination data store has successfully processed the requested changes, thereby improving reliability (i.e., the destination confirms it has successfully processed the message pushed by the source).
- Notification messages do NOT give this confirmation, which increases the chances for data loss under failure scenarios.
  - Notification messages are more suitable for applications that consume but do not persist the data (e.g., a visualization or dashboard application) or do not need full eventual consistency.

26.5 Pull Workflow

In the pull workflow, the destination actively pulls data from the source and subscribes to data changes in the source (so it can receive subsequent updates to objects in its replication scope). The pull workflow is more complex than the push workflow, and it relies on the destination endpoint using more information that is available through ETP.

The destination has the ETP role of customer and it pulls data from the source, which has the ETP role of store. In general, replication occurs by getting data (using Store (Protocol 4), GrowingObject (Protocol 6) and ChannelSubscribe (Protocol 21)) and subscribing to notifications of change (StoreNotification (Protocol 5), GrowingObjectNotification (Protocol 7) and ChannelSubscribe (Protocol 21)).

The main replication tasks in the pull workflow are accomplished as follows:

- **Replication Scope Identification**
  - The replication scope comes from an external source, as described for the push workflow (see Section **26.4**).
  - The set of objects in the scope MAY come from an external source when it is a static, explicit list (e.g. replicate Channel 123 and Channel 456 and nothing else).
  - When the set of objects in the scope is dynamic (which is the most likely scenario):
    - Discover the initial set of objects in the scope using ***Discovery.GetResources***. (For details of the message flows for this protocol, see Chapter **8 Discovery (Protocol 3)**.)
    - Subscribe to changes to the set of objects in the scope with ***StoreNotification.SubscribeNotifications***.
    - Receive changes to the set of objects in the scope with ***StoreNotification.ObjectChanged***, ***StoreNotification.ObjectAccessRevoked*** and ***StoreNotification.ObjectDeleted***.
      EXAMPLE: If data objects are added, removed or deleted from a scope, the destination endpoint is notified of these changes through these notification messages. (For details of the message flows for this protocol, see Chapter **10 StoreNotification (Protocol 5)**.)
- **Data Object Replication**
  - Discover the initial set of data objects, including growing data objects, using Discovery (Protocol 3) (which tells you their current state and when they last changed) and, based on the results of discovery, pull the desired objects using ***Store.GetDataObjects***. (For details of the message flows for this protocol, see Chapter **9 Store (Protocol 4)**.)

  For creates, joins, and updates:

  - Receive created, updated, and joined (i.e. existing data objects added to the replication scope) data objects with ***StoreNotification.ObjectChanged***. In some cases, the notification includes the actual data object (which reduces traffic on the wire because you don't have to issue a request for it). However, in some scenarios, you will not get the data object with the notification, so must get it using ***Store.GetDataObjects*** (Chapter **9 Store (Protocol 4)**) or ***GrowingObject.GetDataObjectsHeader*** (Chapter **11 GrowingObject (Protocol 6)**).

For deletes, unjoins and access revocations:

- Receive unjoins (i.e. data objects removed from the replication scope without being deleted) with ***StoreNotification.ObjectChanged*** (see Chapter **10 StoreNotification (Protocol 5)**).
- Receive access revocations with ***StoreNotification.ObjectAccessRevoked*** (see Chapter **10 StoreNotification (Protocol 5)**).
- Receive deletes with ***StoreNotification.ObjectDeleted*** (see Chapter **10 StoreNotification (Protocol 5)**).
- Growing data object Part Replication
- Pull the initial set of parts together with the data object header using ***Store.GetDataObjects*** (Chapter **9 Store (Protocol 4)**).
- Subscribe to growing data objects in the replication scope using ***GrowingObjectNotification.SubscribePartNotificaitons*** (see Chapter **12 GrowingObjectNotification (Protocol 7)**) to receive part creates, changes and deletes.
- Receive part creates, changes and deletes with ***GrowingObjectNoficiation.PartsChanged***, ***GrowingObjectNoficiation.PartsDeleted*** and/or ***GrowingObjectNoficiation.PartsReplacedByRange*** (see Chapter **12 GrowingObjectNotification (Protocol 7)**).

- **Channel Data Replication**
  - Get metadata for channels in the replication scope using ***ChannelSubscribe.GetChannelMetadata***.
  - Pull existing channel data in the data ranges provided in the returned channel metadata using ***ChannelSubscribe.GetRanges***.
  - Subscribe to the channels starting from the end of the existing data range provided in the channel metadata using ***ChannelSubscribe.SubscribeChannels*** (see Chapter **19 ChannelSubscribe (Protocol 21)**) to receive new, streaming channel data.
  - Receive new (i.e. appended) data as it becomes available with ***ChannelSubscribe.ChannelData***.
  - Receive data edits and deletes with ***ChannelSubscribe.ChannelsTruncated*** and/or ***ChannelSubscribe.RangeReplaced***.

26.6 Outage Recovery: Resuming Operations After a Disconnect

The most commonly occurring (at least in the drilling/WITSML world) and problematic failure is the sudden loss of connectivity in the middle of data replication/transmission. The information in this section deals with recovery around that scenario.

This Appendix is focused specifically on data replication workflows (potentially one of the most complex and comprehensive workflows). However, the general principles and tasks in this outage recovery workflow are applicable for recovery from the sudden interruption of nearly any domain workflow.

NOTES:

1. The active participant (an informal term for the endpoint that is controlling the replication operation; for more information see Section **26.2.1**) in the replication workflow must initiate the outage recovery workflow.
2. For the details about the requirements, rules, and error handling for the message flow and protocol-level operations (e.g., how the ***PutDataObjects*** message works, fields, options, etc.) see the specific protocol chapter in this guide (e.g., Chapter **9**, **Store (Protocol 4)**).

26.6.1 Goal of Outage Recovery

The main goal of outage recovery is to resume the data transmission connection after a session was interrupted and resume operations that were in process before the outage without missing or losing any data. Ideally, this operation can resume "where" it was interrupted; however, that resumption isn't always easy because the endpoint "producing" or sending data may continue to do so even after the connection was dropped.

So when the session resumes, in addition to resuming the current operation, the active participant must determine: if any data was "missed" during the outage, what that missed data was, and get the "missed" data—ideally, without having to start the operation all over again (i.e., resend all data from the beginning of the disconnected session).

Leveraging new features in ETP v1.2, these replication and outage-recovery workflows have been defined so that endpoints can reliably replicate data and recover from unintended disconnects/outages with a significantly reduced likelihood of having to "resend everything again"—which of course is costly and time consuming. The reliability features support better decision making around when it is necessary to resend everything.

26.6.2 Key Concepts for Outage Recovery

This section explains key concepts and definitions that are important to understanding how outage recovery is intended work. (For definitions not provided here, see Section **26.2 above**.)
RECOMMENDATION: Familiarize yourself with these concepts before reading the workflow sections.

26.6.2.1 Timestamps from the ETP Store

This section explains what a timestamp is and where (in what ETP messages) they occur. The next section (Section **26.6.2.2 Change Retention Period**) and the workflow sections below explain how timestamps are used to recover from an outage.

<u>Timestamps from the clock of the endpoint that is acting as the ETP store (the destination in the push workflow and the source in the pull workflow)</u> play an important role in outage recovery.

ETP defines timestamps in several messages, for example:

- In Core (Protocol 0), the ***RequestSession*** and ***OpenSession*** messages have the field *currentDateTime*.
- Change messages defined in notification protocols include a *changeTime* indicating when the change happened.

All such timestamps are defined in ETP as a UTC dateTime value, serialized as a long, using the Avro logical type timestamp-micros (microseconds from the Unix Epoch, 1 January 1970 00:00:00.000000 UTC). (For more information about timestamps, see Section **3.12.5**.)

In general, the timestamps are used as follows:

- When an ETP session is established, the store's currentDateTime is exchanged and established.
- When changes (additions, deletions, or updates to data objects) occur in the endpoint acting as the ETP store, the ETP customer receives timestamps of when these changes happened in the store via notifications in the pull workflow and via positive response messages in the push workflow. **NOTE:** Whether the change is pushed or pulled, depends on the replication workflow in use.

26.6.2.2 Change Retention Period

The change retention period (ChangeRetentionPeriod (CRP)) is an endpoint capability defined by ETP. (For more information on capabilities and their purposes, see Section **3.3**.)

Specifically, the CRP is the minimum time period in seconds that a store retains the canonical URI of a deleted data object and any change annotations for data objects, including channels and growing data objects. It is recommended that the CRP be as long as is feasible in an implementation, but per this specification it must be at least 24 hours (84,600 seconds). When the period is shorter, the risk is that additional data will need to be transmitted to recover from outages, leading to higher initial load on sessions.

In some stores, the retention history may be lost from time to time (e.g., if the store application restarts). These stores MUST retain change data for at least the CRP as long as at least one session is connected to the store's endpoint. If the store DOES lose the retention history, the store MUST send the earliest timestamp for which it DOES have retained change data in the *earliestRetainedChangeTime* field in either the ***RequestSession*** or ***OpenSession*** messages. From the customer's perspective, this essentially serves as a potentially shorter CRP than usual when initially connecting. For the remainder of this appendix, when ChangeRetentionPeriod or CRP are used, they mean either the store's advertised ChangeRetentionPeriod OR the shorter period based on the store's *earliestRetainedChangeTime*.

The CRP can be exchanged in both of these ways:

- In the server capabilities endpoint, before establishing the WebSocket connection (for more information, see Section **4.3**).
- In the ***RequestSession*** and ***OpenSession*** messages when establishing the ETP session (for more information, see Chapter **5**).

So when you connect to an ETP endpoint, you can discover its CRP and you will have the CRP when you begin exchanging timestamps (which are explained in Section **26.6.2.1 above**) during the session.

NOTE: Information on required behaviors related to use of the ChangeRetentionPeriod are defined in the Required Behavior section of each relevant protocol chapter.

This section provides a brief general explanation of how CRP and timestamps work, based on the simple example in **Figure 39**.

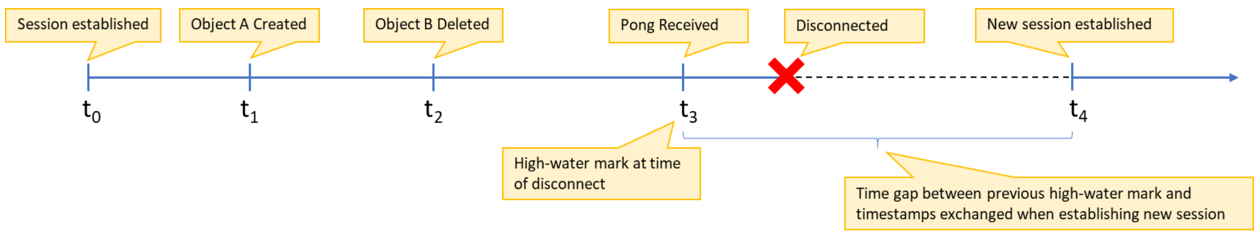

Session established
Object A Created
Object B Deleted
Pong Received
Disconnected
New session established
t0
t1
t2
t3
t4
High-water mark at time of disconnect
Time gap between previous high-water mark and timestamps exchanged when establishing new session

Figure 39: Example showing how timestamps in various messages and change retention period work to retain a "high-water mark" timestamp, which is crucial to determine what content changed during an outage.

In general, the basic idea of how timestamps and the CRP works is as follows:

1. Before connecting and/or as part of the establishing the ETP session, an endpoint's CRP is discovered.
2. When a session is established, initial (t_0) timestamps (currentDateTime) are exchanged in Core (Protocol 0) ***RequestSession*** and ***OpenSession*** messages.
3. While connected, the latest timestamp of changes that have been pushed or pulled can be tracked (t_1, t_2); during periods of inactivity, ***Ping*** and ***Pong*** messages can be used to track updated timestamps (t_3); this latest timestamp of known change(s) is informally referred to as the *high-water mark*. The active participant MUST track this high-water mark during the ETP session.

   DETAILS from Figure 39: At t_1, an object was created, so an ***ObjectChanged*** message is sent saying an object was created at timestamp t_1. Timestamp t_1 is now the high-water mark, so we know about any and all changes that may have happened in the window between t_0 and t_1.

   At t_2, an object is deleted, and an ***ObjectDeleted*** message is sent with the timestamp of t_2, so t_2 becomes the new high-water mark.

Then the data transmission goes idle for a while. So to establish a new (more recent) high-water mark within the source store's CRP, the customer endpoint sends the ***Core.Ping*** message and receives the response ***Core.Pong*** message with a timestamp of t_3. Timestamp t_3 is now the high-water mark.

After t_3, the connection is inadvertently dropped and the session disconnected. So t_3 remains the last known high-water mark.

4. When reconnecting after the disconnect (t_4), the *currentDateTime* timestamps are exchanged when establishing the new session (just like when the initial session was established).
5. After reconnecting, the active participant must compare the gap between the timestamp of the new session start (t_4) and the high-water mark from the previous session (t_3) to the CRP. Necessary actions depend on whether the gap is less than/equal to the CRP (i.e., you have reconnected within the CRP) or greater than the CRP (i.e., you have connected later than the CRP).

   For next steps, see Section **26.6.3 Main Resumption Workflow**.

NOTES:

1. Real-world operations—which may mean hundreds or thousands of messages flowing back and forth between endpoints simultaneously, some operations resulting in multiple notifications with the same timestamp, and receipt of multiple messages at a particular timestamp—will make it more challenging to determine exactly what happened at a particular timestamp. However, if an endpoint receives a message with a particular timestamp, it can be confident it received all changes before that message/timestamp. This is an important semantic that stores must understand and support for this change detection process to work.
2. The process for determining missed data during an outage is more relevant to pull workflows than push workflows (because an active participant that is pushing data "knows/tracks" what data it was pushing). However, the process is important to man-in-the-middle applications, which uses both push and pull workflows.

26.6.3 Main Resumption Workflow

To resume data replication after a previous session was disconnected, the active participant must follow the general workflow below. This main resumption workflow "branches" for difference tasks/steps for push and pull, and it contains links to the push- and pull-specific details.

REMINDER: In both the "push" and "pull" workflows, it is the ETP-assigned role of "customer" that is the active participant (in the disconnected replication session). However, in the push workflow, the customer/active participant is the source (i.e., the content being replicated), and in the pull workflow, the customer/active participant is the destination. (For more information about definitions for these terms, see Section **26.2.1**.)

The active participant must follow these steps:

1. Establish an ETP session. (For detailed instructions for establishing the WebSocket connection, see Chapter **4**. For information about establishing the ETP session, see Section **5.2.1.1**.) If the active participant is also the ETP client, it should establish a new connection to start the ETP session. If the active participant is acting as an ETP server, it must wait for the other endpoint to reconnect before proceeding with session establishment.

   REMINDER: Each ETP session is independent of any other ETP session (i.e., there is no session survivability, no automatic resumption of past activities). So "everything" that was happening in the interrupted session must be recreated in the new session.

2. Determine whether the disconnected period was within the endpoint's change retention period. (For information on how the change retention period (ChangeRetentionPeriod) works, see Section **26.6.2.1**.)

   Whether or not the disconnected period was within or beyond the endpoint's change retention period determines if and how to identify what changed during the disconnected period (next step).

3. Identify what changed during the disconnected period.
    a. **RECOMMENDATION:** When querying for changes since the high-water mark or determining if the high-water mark is older than the ChangeRetentionPeriod or not, it is recommended to subtract a small time delta, such as the store's ChangePropagationPeriod, from the high-water mark to maximize the likelihood of not missing any changes.
    b. **If the disconnect period is within the change retention period**, the active endpoint can understand all changes that may need to be replicated by using an appropriate combination of ***Discovery.GetResources***, ***Discovery.GetDeletedResources***, ***ChannelSubscribe.GetChannelMetadata***, ***ChannelSubscribe.GetChangeAnnotations***, ***GrowingObject.GetPartsMetadata***, and ***GrowingObject.GetChangeAnnotations***.

       NOTE: If a customer makes a request that is greater than the store's CRP, the store MUST send error ERETENTION_PERIOD_EXCEEDED (5001).
    c. **If the disconnected period is longer than the change retention period**, the results and possible required actions depend on the nature of the change.

       The table below provides highlights for key changes that may have occurred and the actions required to determine changes.

       - In some cases (e.g., data objects), you can determine exactly what changed.
       - In other cases, you cannot determine exactly what changed; in these cases, you'll need to decide if you can accept the results the store can provide or if you must re-send or request "all data" (i.e., from the start of the disconnected session).

Nature of Change	**Result/Required Action**
Data object updated	Changes can determined through the *storeLastWrite* element, which can be used in discovery and query protocols to filter data objects. When you get a resource or data object, its *storeLastWrite* element shows when it last changed. If a data object's *storeLastWrite* is greater than your high-water mark from the previous session, then you must get the data object.
Data object deleted or access revoked	This cannot be reliably detected. Here is the process: 1. Use Discovery (Protocol 3) to discover the set of data objects that you were previously replicating when the outage occurred (the previously known set). 2. Do a "diff" between the previously known set of data objects and the set of data objects returned in Step 1. The diff reveals if you have fewer (or more) data objects and which ones are no longer in the replication scope (and which ones have been added). You can assume that the data objects no longer in the replication scope were either deleted, unjoined or your access to them was revoked. You can differentiate between unjoined and deleted or access revoked, but you cannot differentiate between deleted and access revoked. In some cases, these issues are not important. You can simply resume operations for the remaining set of data objects.

Nature of Change	Result/Required Action
Channel data and growing data object parts	Cannot be reliably detected and may need to be replicated "from scratch" (i.e., re-send all data).

3. After the changes have been identified, the active participant must:
   a. Start pushing or pulling any changes detected in the new session.
      - For the push workflow, see Section **26.6.4**.
      - For the pull workflow, see Section **26.6.5**.
   b. Replicate the changes that happened while disconnected and, when needed, replicate any objects from scratch.

26.6.4 Resumption Workflow: Details for Push

In the data push workflow, the burden is on the source to track what data it has successfully pushed to the destination, so that, on reconnect, the source knows what data it must resume sending. This "knowledge" appears to be "session survivability" (which was removed from ETP in v1.2), but it is not. The source can persist this information however it deems appropriate; the method is not specified by ETP.

The data the source must track includes:

- The objects it is pushing to the destination.
- For each object, the most recent change to the object that was successfully pushed.
  NOTE: Put response messages notify an endpoint of what data was successfully put. So in the original replication workflow, the destination will send positive response messages for all previous successful put operations.
- For each growing data object, the most recent change (create, update, delete) to parts in the object that were successfully pushed.
- For each channel, both:
  - The most recent appended data index sent to the destination.
    NOTE: ETP does not specify a success message for this operation. However, as part of the process of the source resuming pushing data, the destination responds to the source with its current end index for each channel. The source must determine if that end index is the same or different than the last one it sent, and then take necessary action.
  - The most recent change to existing data that was successfully pushed.
    NOTE: Like store put operations, a successful replace range operation has a successful confirmation message.

On reconnect, the source must follow these steps:

1. To understand the changes that it must push, the source must compare the tracked information (i.e., the list above) against its current state of data.
   - When the source is a data store, the source must use its **internal knowledge** about the current state of data in itself. For example, a source must determine things such as: Are there new or different data objects in replication scope that weren't there before that it now needs to push? Are there changes to objects that it sent previously that are more recent than what the destination has?
   - If the source is a sync app, the sync app must use the pull workflow to get the current state of data from the data store it is replicating. It will need to issue queries to understand the current state.

- – The source may additionally use aspects of the pull workflow on resumption by issuing queries to the destination to verify that the destination's content matches the source's expectations based on the information it tracked. This option is not described in detail here, but possible actions when the source's expectations are not met are to stop the transfer with an error or triggering a full resend of data for affected data objects

2. On reconnect, after the changes that need to be pushed have been identified (step 1), the changes are pushed using the normal push workflow (described in Section **26.4**).

   In the push workflow, there is no difference between pushing changes that happened while disconnected and pushing changes that happen while connected.

26.6.5 Resumption Workflow: Details for Pull

Resumption of the data pull workflow is inherently more complex than in the data push workflow. In the pull workflow, the burden is on the destination to track what it has successfully pulled or received from the source, which includes:

- The list of objects it is pulling (the replication scope).
- For each object, the most recent change to the object that was received.
- For each growing data object, the most recent change (create, update, delete) to parts in the object that was received.
- For each channel, the most recent appended data index received and the most recent change (i.e., range replaced or channels truncated) to existing data that was pulled or received.

This section (including subsequent sub-sections) gives an overview of what must be tracked, how to determine changes, and how to resume operations for the pull workflow.

On reconnect, the destination must follow these steps:

1. To understand the changes it must pull, the destination compares this tracked information against the current state of data in the source, which is typically done with queries.

   NOTE: In general, the current state of data in the destination is NOT used. If more than one entity may be changing data in the destination, extra care is needed; but multi-master replication is not currently addressed in ETP.

   a. To understand the state of the data in the source, the destination must gather information about the items listed in the first column of this table. For more information on the specifics of what the destination must track for each item in column 1, see Section **26.6.5.1** and the detailed section referenced in the table below.

To understand the types of changes to this and how to detect them…	See this section:
Replication scope	**26.6.5.2.1**
Objects in the replication scope	**26.6.5.2.2**
Growing data object parts in the replication scope	**0**
Channel data in the replication scope	**0**

2. After the changes have been identified, the destination must pull the changes.

26.6.5.1 Information That Must Be Tracked by the Destination and How to Initialize and Track it

During the replication process, the destination in the pull workflow must track what is being replicated (the list in Step 1a **above** and repeated in the table below), and it must tracks changes to those items throughout the replication process, so it can keep the required tracked information current.

NOTE: This information is NOT explicitly listed for the push workflow, because in the push workflow, the source is in control of sending the messages to push the data (from itself!) and must simply track confirmation that the action specified in the messages was successfully completed (with the positive responses from the destination). In the pull workflow, the destination must also pull this "tracking data" from the source.

The following tables summarizes, what must be tracked (column 1), and for each of those items, specifically what is tracked (column 2), how the tracked information is initialized (column 3), and updated (column 4) during the replication process. The text below the table describes the behavior for each "row" in the table.

Information Tracked About	What Is Tracked	How Tracked Information Is Initialized	How Tracked Information Is Updated
Replication Scope	URIs of all data objects in scope	***Discovery.GetResources***	***StoreNotification.ObjectChanged*** ***StoreNotification.ObjectAccessRevoked*** ***StoreNotification.ObjectDeleted***
Data Objects	*storeLastWrite*	***Discovery.GetResources***	*storeLastWrite* on ***DataObject*** in ***StoreNotification.ObjectChanged***
Growing data object Parts	Last timestamp of part change	***GrowingObject.GetChangeAnnotations*** with latestOnly=true Or, if none, timestamp from ***RequestSession*** or ***OpenSession***	***GrowingObjectNotification.PartsChanged*** ***GrowingObjectNotification.PartsDeleted*** ***GrowingObjectNotification.PartsReplacedByRange***
Channel Data	Current end index	***ChannelSubscribe.GetChannelMetadata***	***ChannelSubscribe.ChannelData***
	Last timestamp of edit or delete	***ChannelSubscribe.GetChangeAnnotations*** with latestOnly=true Or, if none, timestamp from ***RequestSession*** or ***OpenSession***	***ChannelSubscribe.ChannelsTruncated*** ***ChannelSubscribe.RangeReplaced***

(Row 1) On initial connection for replication, the destination initializes the replication scope using the ***Discovery.GetResources*** message; the source replies with the list of resources (one for each data object in the scope) which contains the URI for the data object (among other data). The destination must also subscribe to notifications for changes to objects in the replication scope; when changes occur, the source sends the destination ***StoreNotification.ObjectChanged, StoreNotification.ObjectAccessRevoked,*** and ***StoreNotification.ObjectDeleted***. The destination uses these notices to update the data object's *storeLastWrite* time.

(Row 2) For each data object in the replication scope, the destination must track its URI and its *storeLastWrite* time.

(Row 3) The high-water mark for edits or deletes to parts in a growing data object is the timestamp on the most recent ***ChangeAnnotation*** for that growing data object; it conveys that no parts were changed in that growing data object after that timestamp. (For information about how change annotations (CA) work in this workflow, see Section **0**.) The destination initializes this information by sending message ***GrowingObject.GetChangeAnnotations*** (with *latestOnly*=true). If no CAs are returned, the high-water mark is the *currentDateTime* stamp exchanged in Core (Protocol 0) when establishing the session in the ***RequestSession*** and ***OpenSession*** messages. Tracked information is updated in the destination during replication with ***GrowingObjectNotification.PartsChanged***,***GrowingObjectNotification.PartsDeleted,*** and ***GrowingObjectNotification.PartsReplacedByrange*** messages, which will result in changes to the *storeLastWrite* of a growing data object.

(Row 4) For channel data, appended data (Row 4a) must be tracked independently of edits and deletes to existing data (Row 4b).

(Row 4a) For appended channel data, the destination must track the end index (*endIndex*) of each channel in its replication scope. On initialization the destination gets that information by sending the ***GetChannelMetadata*** message; the source responds with the ***GetChannelMetadataResponse*** message, which is part of the process of subscribing to receive data from a channel using ChannelSubscribe (Protocol 21) (NOTE: The *endIndex* is on the ***IndexInterval*** record). As part of the subscription, as new data are available in the source, the destination receives ***ChannelSubscribe.ChannelData*** messages, which is used to update the *endIndex* of the channel.

(Row 4b) The high-water mark for edits or deletes to existing channel data is the timestamp on the most recent ***ChangeAnnotation*** for that channel; it conveys that no data was changed in that channel after that timestamp. (For information about how change annotations (CA) work in this workflow, see Section **0**.) The destination initializes this information by sending message ***ChannelSubscribe.GetChangeAnnotations*** (with *latestOnly*=true). If no CAs are returned, the high-water mark is the *currentDateTime* stamp exchanged in Core (Protocol 0) when upgrading the WebSocket connection to ETP, in the ***RequestSession*** and ***OpenSession*** messages. Tracked information is updated in the destination during replication with ***ChannelSubscribe.ChannelsTruncated*** and ***ChannelSubscribe.RangeReplaced*** messages, which will result in changes to the *storeLastWrite* of a channel and may result in changes to the endIndex of a channel.

26.6.5.2 How to Detect Specific Types of Changes

26.6.5.2.1 Replication Scope

While disconnected, changes can happen to the set of objects in the replication scope (for the definition of replication scope, see Section **26.2.3**.). The changes in the scope occur due to addition and deletion of data objects, and addition or removal of relationships between objects.

Figure 40 shows an example of how a replication scope my change as the result of various operations; these changes are explained below.

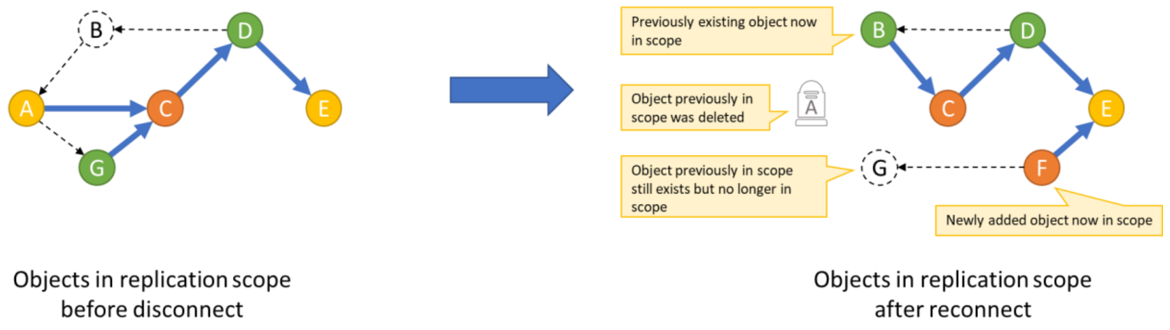

B
D
A
C
E
G
Objects in replication scope before disconnect
Previously existing object now in scope
Object previously in scope was deleted
Object previously in scope still exists but no longer in scope
Newly added object now in scope
B
D
A
C
E
G
F
Objects in replication scope after reconnect

Figure 40: Example replication scope changes while disconnected. Solid colorled circles represent data objects of interest (in the replication scope); dashed-line circles are objects outside the replication scope. After reconnect (righ) in this example, B is now in scope and G is out of scope.

Change Type	How Store Retains It	How Customer Discovers Change on Reconnect	How Customer Requests Change
New data object created	Updates *storeLastWrite* and *storeCreated*	Object URI not in previously tracked replication scope	***Store.GetDataObjects***

Change Type	**How Store Retains It**	**How Customer Discovers Change on Reconnect**	**How Customer Requests Change**
Existing data object added to scope	Updates *storeLastWrite* on one end of the relationship	Object URI not in previously tracked replication scope and *storeCreated* is older than the high-water mark.	***Store.GetDataObjects***
Existing object deleted	DeletedResource	No Resource returned matching previously tracked URI. Discovery.GetDeletedResources has a matching DeletedResource.	
Access to existing object revoked	Not retained in information available through ETP	No ***Resource*** returned matching previously tracked URI. ***Discovery.GetDeletedResources*** does NOT have a matching ***DeletedResource***.	
Existing object removed from scope	Updates *storeLastWrite*	No ***Resource*** returned matching previously tracked URI.	

(Row 1) New objects may be created in the source that fall within the replication scope. Any time a new object is created, the source store initializes both *storeLastWrite* and *storeCreated* to the object's creation time. On reconnect, the destination sends ***Discovery.GetResources*** to get the updated replication scope. Any URIs that were not previously known to the destination are newly created data objects if their *storeCreated* time is newer than the destination's high-water mark. The destination requests the new data object with ***Store.GetDataObjects***.

(Row 2) New relationships may be created in the source between existing data objects that cause the existing data objects to be included in the replication scope. When this happens, the source store initializes the *storeLastWrite* of the container data object or the source of the data object reference in the relationship. On reconnect, the destination sends ***Discovery.GetResources*** to get the updated replication scope. Any URIs that were not previously known to the destination are existing data objects that have been added to the scope if their *storeCreated* time is newer than the destination's high-water mark. The destination requests the new data object with ***Store.GetDataObjects***.

(Row 3) Existing data objects within the replication scope may be deleted. When this happens, the source creates a ***DeletedResource*** for the data object. On reconnect, the destination sends ***Discovery.GetResources*** to get the updated replication scope. If any URIs previously known to the destination are missing from the response, the destination sends ***Discovery.GetDeletedResources*** to get the list of deleted ***DeletedResource*** records. A deleted object will have a corresponding ***DeletedResource*** in the response.

(Row 4) The source may revoke the destination's access to an object that is within the replication scope. When this happens, the store does not track this information in a field on an ETP record. On reconnect, the destination sends ***Discovery.GetResources*** to get the updated replication scope. If any URIs previously known to the destination are missing from the response, the destination sends ***Discovery.GetDeletedResources*** to get the list of deleted ***DeletedResource*** records. If any URIs previously known to the destination do NOT have a corresponding ***DeletedResource*** in the response, the destination sends a ***Discovery.GetDeletedResources*** for each such URI and scoped only to that URI. If no ***Resource*** is returned, the destination has lost access to the data object.

(Row 5) Relationships between objects within the replication scope in the source may be modified causing some of the objects to no longer be in the scope. When this happens, the source store initializes the *storeLastWrite* of the container data object or the source of the data object reference in the relationship. On reconnect, the destination sends ***Discovery.GetResources*** to get the updated replication scope. If any URIs previously known to the destination are missing from the response, the destination sends ***Discovery.GetDeletedResources*** to get the list of deleted ***DeletedResource*** records. If any URIs previously known to the destination do NOT have a corresponding ***DeletedResource*** in the response, the destination sends a ***Discovery.GetDeletedResources*** for each such URI and scoped only to that URI. If a ***Resource*** IS returned, the data object has been removed from the replication scope.

26.6.5.2.2 Objects

While disconnected, data objects in the replication scope may be updated; that is, elements on the data object have changed.

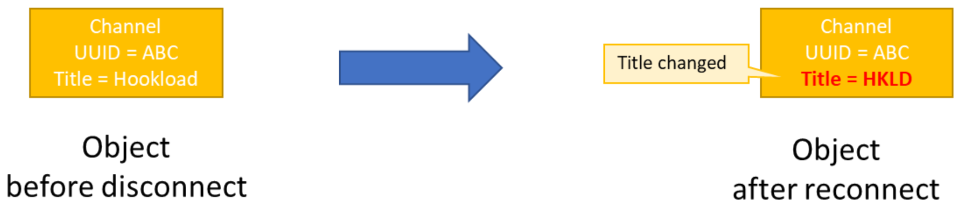


Figure 41: Example of update to data object. While disconnected, the channel's title has been changed (from "Hookload" to "HKLD".

Change Type	How Store Retains It	How Customer Discovers Change on Reconnect	How Customer Requests Change
Object Changed	Updates storeLastWrite	Resource.storeLastWrite newer than previously tracked storeLastWrite and storeCreated is older	Store.GetDataObjects or GrowingObject.GetDataObjectsHeader
Object Deleted and Recreated	Updates *storeLastWrite* and *storeCreated*	***Resource***.*storeCreated* newer than previously tracked *storeLastWrite*	***Store.GetDataObjects*** or ***GrowingObject.GetDataObjectsHeader*** followed by ***ChannelSubscribe.GetRanges*** or ***GrowingObject.GetPartsByRange*** if appropriate

(Row 1) Data objects within the replication scope may be changed in the source. Any time an object is changed, the source updates *storeLastWrite*. On reconnect, the destination sends ***Discovery.GetResources*** to get the updated replication scope. If the ***Resource*** for a data object has a newer *storeLastWrite* than the data object's last known *storeLastWrite* AND the ***Resource*** has a *storeCreated* that is equal to or older than the data object's last known *storeLastWrite*, then the data object was changed while the session was disconnected. To get the last data for the data object, the destination sends ***Store.GetDataObjects*** or ***GrowingObject.GetDataObjectsHeader***.

(Row 2) Data objects within the replication scope may be deleted and recreated in the source. Any time this happens, the source updates both *storeLastWrite* and *storeCreated*. If the ***Resource*** for a data object has a newer *storeCreated* than the data object's last known *storeLastWrite*, then the data object was deleted and recreated while the session was disconnected. To get the last data for the data object, the destination sends ***Store.GetDataObjects*** or ***GrowingObject.GetDataObjectsHeader***. If the data object is a growing data object or a channel, the destination requests the new data with ***GrowingObject.GetPartsByRange*** or ***ChannelSubscribe.GetRanges***.

26.6.5.2.3 Growing data object Parts

While disconnected, parts in growing data objects may be added, updated or deleted.

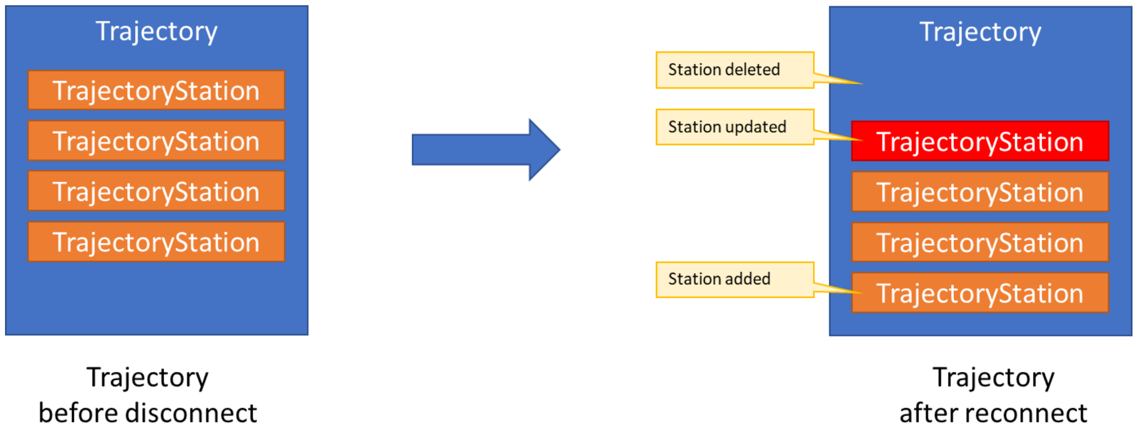


Figure 42: Example: trajectory had 4 trajectory stations; on reconnect there are still 4, but one has been deleted, a new one added, and one has been updated.

The table below lists the four possible actions, which are explained below the table.

Change Type	How Store Retains It	How Customer Discovers Change on Reconnect	How Customer Requests Change
Part Added	Creates ***ChangeAnnotation***	New ***ChangeAnnotation*** covering the changed interval	***GrowingObject.GetChangeAnnotations***
Part Changed	Creates ***ChangeAnnotation***	New ***ChangeAnnotation*** covering the changed interval	***GrowingObject.GetChangeAnnotations***
Part Deleted	Creates ***ChangeAnnotation***	New ***ChangeAnnotation*** covering the changed interval	***GrowingObject.GetChangeAnnotations***
Range Updated	Creates ***ChangeAnnotation***	New ***ChangeAnnotation*** covering the changed interval	***GrowingObject.GetChangeAnnotations***

(Row 1) New parts may be added to growing data objects within the replication scope. When this happens, the source updates the index ranges as necessary and creates a ***ChangeAnnotation*** for the affected data objects. On reconnect, the destination sends ***GrowingObject.GetChangeAnnotations*** with the high-water mark to get any new ***ChangeAnnotations*** that may have been created while disconnected. Steps to take in response to new ***ChangeAnnotations*** are described below.

(Row 2) Existing parts may be modified in growing data objects within the replication scope. When this happens, the source creates a ***ChangeAnnotation*** for the affected data objects. On reconnect, the destination sends ***GrowingObject.GetChangeAnnotations*** with the high-water mark to get any new ***ChangeAnnotations*** that may have been created while disconnected. Steps to take in response to new ***ChangeAnnotations*** are described below.

(Row 3) Parts may be deleted from growing data objects within the replication scope. When this happens, the source updates the index ranges as necessary and creates a ***ChangeAnnotation*** for the affected data objects. On reconnect, the destination sends ***GrowingObject.GetChangeAnnotations*** with the high-water mark to get any new ***ChangeAnnotations*** that may have been created while disconnected. Steps to take in response to new ***ChangeAnnotations*** are described below.

(Row 4) Ranges of parts may be deleted from growing data objects and replaced with new parts within the replication scope. When this happens, the source updates the index ranges as necessary and creates a ***ChangeAnnotation*** for the affected data objects. On reconnect, the destination sends ***GrowingObject.GetChangeAnnotations*** with the high-water mark to get any new ***ChangeAnnotations*** that may have been created while disconnected. Steps to take in response to new ***ChangeAnnotations*** are described below.

Handling New *ChangeAnnotations* on Reconnect

There are three cases that must be handled with ***ChangeAnnotations***:

1. The ***ChangeAnnotation*** is entirely within the growing data object's previously known data range. In this scenario, the destination discards any previously known parts entirely covered by the ***ChangeAnnotation*** interval and requests new parts for the interval with ***GrowingObject.GetPartsByRange***.
2. The growing data object's previously known start or end index falls within the ***ChangeAnnotation*** interval. In this scenario, the destination must discard all previously known parts entirely covered by the ***ChangeAnnotation*** interval and requests the new data for the interval with ***GrowingObject.GetPartsByRange***.
3. The ***ChangeAnnotation*** is entirely beyond the previously known index range for the growing data object. In this scenario, the destination retrieves all new data beyond the previously known index range with one or two ***GrowingObject.GetPartsByRange*** messages.

26.6.5.2.4 Channel Data

While disconnected, new channel data may be appended and existing channel data may be edited or deleted.

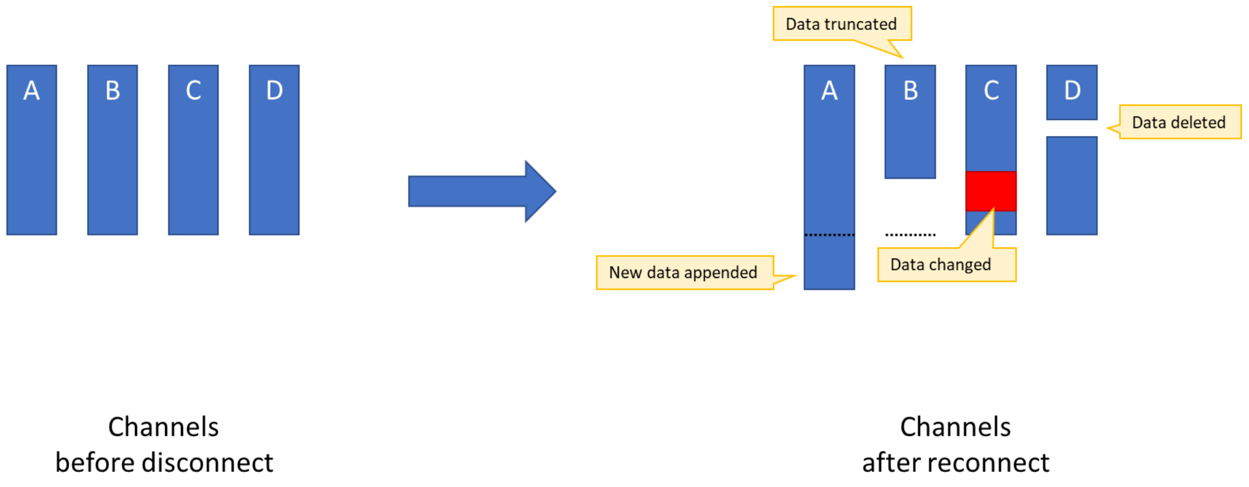


Figure 43: Example of change annotations and how they work (see details below).

The example in **Figure 43** is channel data with indexes increasing downward. The table below lists the changes to channel data that can occur (or some combination of these). A customer must be able to detect all these changes on reconnect.

Change Type	How Store Retains It	How Customer Discovers Change on Reconnect	How Customer Requests Change
New Data Appended	Updates end Index	New end index values in the *interval* field on ***IndexMetadataRecord*** in ***ChannelMetadataRecord***	***ChannelSubscribe.SubscribeChannels***

Change Type	How Store Retains It	How Customer Discovers Change on Reconnect	How Customer Requests Change
Channel Truncated	Updates end Index and creates ***ChangeAnnotation***	New ***ChangeAnnotation*** that may or may not extend beyond current end index.	***ChannelSubscribe.GetRanges***
Data Changed	Creates ***ChangeAnnotation***	New ***ChangeAnnotation*** covering the changed interval.	***ChannelSubscribe.GetRanges***
Data Deleted	Creates ***ChangeAnnotation***	New ***ChangeAnnotation*** covering the deleted interval.	***ChannelSubscribe.GetRanges***

(Row 1) New data may be appended to channels within the replication scope. When this happens, the source updates the end indexes for the affected channels. On reconnect, the destination sends ***ChannelSubscribe.GetChannelMetadata*** and compares the previously known end indexes for each channel against the new ones returned in ***ChannelMetadataRecord***. If the new end indexes are beyond (where beyond may be greater than or less than depending on *direction* in ***IndexMetadataRecord***) the previously known end indexes, new data was appended. The destination sends ***ChannelSubscribe.GetRanges*** to request the new data.

(Row 2) A channel within the replication scope may be truncated, which is when the end index is reset to an earlier value and any data beyond the new end index is deleted. When this happens, the source resets the channel's end index, and it creates a ***ChangeAnnotation*** covering the truncated interval, merging this as needed with existing ***ChangeAnnotation*** records. On reconnect, the destination sends ***ChannelSubscribe.GetChangeAnnotations*** with the high-water mark to get any new ***ChangeAnnotations*** that may have been created while disconnected. Steps to take in response to new ***ChangeAnnotations*** are described below.

(Row 3) Data within a channel within the replication scope may be changed. When this happens, the source creates a ***ChangeAnnotation*** covering the changed interval, merging this as needed with existing ***ChangeAnnotation*** records. On reconnect, the destination sends ***ChannelSubscribe.GetChangeAnnotations*** with the high-water mark to get any new ***ChangeAnnotations*** that may have been created while disconnected. Steps to take in response to new ***ChangeAnnotations*** are described below.

(Row 4) Data within a channel within the replication scope may be deleted. When this happens, the source creates a ***ChangeAnnotation*** covering the deleted interval, merging this as needed with existing ***ChangeAnnotation*** records. On reconnect, the destination sends ***ChannelSubscribe.GetChangeAnnotations*** with the high-water mark to get any new ***ChangeAnnotations*** that may have been created while disconnected. Steps to take in response to new ***ChangeAnnotations*** are described below.

Handling New *ChangeAnnotations* on Reconnect

There are three cases that must be handled with ***ChangeAnnotations***:

1. The ***ChangeAnnotation*** is entirely within the channel's previously known data range. In this scenario, the destination discards any previously known data for the ***ChangeAnnotation*** interval and requests new data for the interval with ***ChannelSubscribe.GetRanges***.
2. The channel's previously known end index falls within the ***ChangeAnnotation*** interval. In this scenario, the destination must discard all data from the start of the ***ChangeAnnotation*** interval to the previously known end index. It this requests the new data for the interval with ***ChannelSubscribe.GetRanges***.
3. The ***ChangeAnnotation*** is entirely beyond the previously known end index for the channel. In this

scenario, the destination retrieves all new data beyond the previously known end index with a single ***ChannelSubscribe.GetRanges***.

27 Appendix: Security Requirements and Rationale for the Current Approach

This appendix lists the requirements for the new security approach added in ETP v1.2. The requirements were driven in large part by a specific request from the WITSML Executive Team, which includes representatives from several large operators, that the Basic Authentication in previous versions of ETP was no longer sufficient.

The Energistics Architecture Team (responsible for the design of ETP v1.2) worked with the Executive Team to clarify the requirements, explored possible solutions, and met with security experts from several operator companies both for input/guidance and to vet the solution.

This appendix also lists other options that were considered and explains why the current approach was selected.

For information about the security in ETP v1.2 and how it works, see Chapter **4**.

27.1 ETP Security Considerations and Requirements

This is the summary of considerations, operating conditions, and the resulting ETP security solution requirements:

- Due to long-lived connections:
  - Sessions must be able to terminate if access has been revoked.
  - ETP clients must be able to retrieve a refreshed bearer token that will be accepted by an ongoing ETP session.
  - Additionally, care must be taken when linking a bearer token to session entitlements (This impacts scenarios where ETP servers live behind intermediate layers like API gateways or load balancers that may handle access authorization).
- Due to low-connectivity scenarios:
  - High-connectivity 2FA/MFA approaches (e.g., phone calls, SMSs, e-mails, physical key generation devices) cannot be made a hard requirement for ETP.
- Due to cross-organization scenarios:
  - Federated authentication must be possible.
  - Reverse data flows, where data flows the opposite direction of HTTP Authorization, must be possible.
- Due to machine-to-machine scenarios, authorization cannot always be strongly tied to an end-user.

Additional considerations:

- ETP is a standard that will be implemented by many server applications and many client applications.
  - It is impractical, for a given server, to know in advance the full list of client applications that may need to connect to it.
  - It is necessary to allow arbitrary clients to connect to arbitrary servers without dedicated, server-specific, connection logic.
- Cloud and rig have different security requirements:
  - Rig security must be as simple as possible.
    - Current workflow is field hand at rig gets e-mailed a URL, username, and password.
    - New workflow must not be much harder than this.
    - New workflow must avoid pitfalls that would potentially disrupt data flowing merely because the sending application could not be authorized.
  - Rig workflow must be “compatible” with the cloud workflow.

  - Same applications/services must be usable with both.
  - Data typically flows between “services” and not end users.

27.2 Approaches Considered and Why the Current One Was Selected

The current approach was determined and designed for these reasons:

- Extensive investigation showed there is no one standard or simple approach that could simply be "picked off the shelf".
- All options that were investigated had limitations; these included: HTTPS, Cookies, Basic, Mutual TLS, URL Query, Sec-WebSocket-Protocol, other Headers.
- Many of the current security systems are for user-driven, interactive workflows, which are not appropriate for most of our device-to-device connectivity scenarios.

Based on the research and the feedback collected from the community, including security experts, the Architecture Team believes this is the best approach because:

- It best supports our use cases.
- It's a minimal but extensive method that appears mainstream enough and is implemented in sufficient packages and languages (i.e., existing tools are available to support it).
- It does not prevent organizations from supporting more advanced, interactive workflows.
- It's believed to be extensible in the future without further schema changes (but, of course, there are no guarantees—Internet security changes fast).
- It's simple to implement Auth Server on an ETP server for small, self-contained installs, while allowing external Auth Servers for larger/corporate configurations.