Energistics Transfer Protocol (ETP)  
Implementation Guide

|  |  |
| --- | --- |
| ETP Overview | The Energistics Transfer Protocol (ETP) is a new data exchange specification that enables the efficient transfer of real-time data between applications. It will initially be implemented as a companion to the WITSML Standard but is also expected to be used with other Energistics standards. |
| Version | CTP v.05 |
| Abstract | This document explains the migration process from WITSML 1.4.1 to adoption of ETP, initially to replace real-time transfer of log data. |
| Prepared by | Energistics and the ETP Work Group |
| Date published | 08 August 2014 |
|  |  |
| Document type | Implementation Guide |
|  |  |
| Keywords: | standards, energy, data, information, process, reservoir model, shared earth model |

Version 1.0 (DRAFT)



|  |  |
| --- | --- |
| **Document Information** | |
| **DOCUMENT VERSION** | 1.0 |
| **Date** | DD Month YYYY |
| **Language** | U.S. English |

Acknowledgements

Energistics would like to thank members of the WITSML Special Interest Group and the Energistics Integration Council (EIC) Integration Team. Team members represent several companies, including: Baker Hughes, Chevron, Geologix, Halliburton, Petrolink and Schlumberger.

****Usage, Intellectual Property Rights, and Copyright****

This document was developed using the Energistics Standards Procedures. These procedures help implement Energistics’ requirements for consensus building and openness. Questions concerning the meaning of the contents of this document or comments about the standards procedures may be sent to Energistics at [info@energistics.org](mailto:info@energistics.org).

The material described in this document was developed by and is the intellectual property of Energistics. Energistics develops material for open, public use so that the material is accessible and can be of maximum value to everyone.

Use of the material in this document is governed by the Energistics Intellectual Property Policy document and the Product Licensing Agreement, both of which can be found on the Energistics website, <http://www.energistics.org/legal-policies>.

All Energistics published materials are freely available for public comment and use. Anyone may copy and share the materials but must always acknowledge Energistics as the source. No one may restrict use or dissemination of Energistics materials in any way.

Trademars

Energistics®, Epicentre™, WITSML™, PRODML™, RESQML™, Upstream Standards. Bottom Line Results.®, The Energy Standards Resource Centre™ and their logos are trademarks or registered trademarks of Energistics in the United States. Access, receipt, and/or use of these documents and all Energistics materials are generally available to the public and are specifically governed by the Energistics Product Licensing Agreement (<http://www.energistics.org/product-license-agreement>).

Other company, product, or service names may be trademarks or service marks of others.

|  |  |  |  |
| --- | --- | --- | --- |
| **Amendment History** | | | |
| **Version** | **Date** | **Comment** | **By** |
| DRAFT | July | Updated formatting to latest template; edits | Donna Marcotte |
| DRAFT | June 10 | Initial draft | John Shields |
| DRAFT | August 8 | Community Technology Preview (CTP) – protocols 0,1 | Architecture Team |

Table of Contents

[Executive Summary 5](#_Toc392491140)

[1 Introduction 6](#_Toc392491141)

[2 Technology 7](#_Toc392491142)

[2.1 WebSocket 7](#_Toc392491143)

[2.2 Apache Avro 7](#_Toc392491144)

[2.3 ETP Protocols and Messages 7](#_Toc392491145)

[3 Implementation 8](#_Toc392491146)

[3.1 Server 8](#_Toc392491147)

[3.1.1 Java Implementation 9](#_Toc392491148)

[3.1.2 C# .NET Implementation 11](#_Toc392491149)

[3.2 Client 11](#_Toc392491150)

[3.2.1 Java Implementation 11](#_Toc392491151)

[3.2.2 C# .NET Implementation 12](#_Toc392491152)

[3.3 Implementation alongside WITSML 1.4.1 12](#_Toc392491153)

[Appendix A. Glossary 14](#_Toc392491154)

[Appendix B. Style Name = Appd Heading 1 2](#_Toc392491155)

[B.1 Appendix Heading 2 2](#_Toc392491156)

[Appendix C. Standards Used in RESQML 3](#_Toc392491157)

[C.1 Appendix Heading 2 3](#_Toc392491158)

Executive Summary

<OPTIONAL: Not all documents require an executive summary. Most technical documents do not.>

# Introduction

ETP is a new data exchange specification that enables the efficient transfer of data between applications and systems. It defines a data streaming 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. The initial use case is for real-time data; however, it is anticipated that ETP will be expanded to include functionality for data discovery and historical data queries.

ETP has initially been implemented as a companion to the WITSML standard and has the potential to be the cornerstone for the next major revision of WITSML. ETP is also expected to be used by other Energistics standards.

This document provides a high-level overview of ETP and its implementation. For more detailed information, see the *ETP Specification*.

NOTE: This document will be updated as needed to reflect additional development of the ETP specification prior to final publication.

## Goals

The main goals of ETP are:

* Provide mechanism for the efficient transfer of high frequency real-time data, with low latency.
* WITSML-compatible alternative to WITS0
* Eliminate polling (push mechanism)
* Implement new transfer mechanism for WITSML 2.0
* Provide a base architecture for data transport for all Energistics standards, with goals that include:
* Easier implementation for simple providers with one connection.
* Flexible enough for enterprise-level software with thousands of clients.
* Internet and firewall friendly. Acceptable to major oil companies.
* Use open standards.

## Use Cases

The initial main use cases for ETP are to move real-time data between applications, including transfer:

* From a wellsite provider to a WITSML store (server).
* From a WITSML store to WITSML store (replication).
* From a WITSML store to client applications.

## Audience Purpose and Scope

This document is intended for information technology professionals (e.g., software developers, programmers, etc.) who want to implement ETP.

Currently, the scope of this document is for implementing ETP to replace current WITSML log functionality.

## Resource Set

The following table lists and describes all resources available to help people understand and implement ETP.

|  | Document/Resource | Description |
| --- | --- | --- |
|  | *Schemas* |  |
|  | *Energistics Transfer Protocol (ETP) Specification* | Document that specifies detailed operation of ETP, including all protocols and data messages. |
|  | *Energistics Transfer Protocol (ETP) Implementation Guide* (this document) | Guidelines for implementing ETP to replace current WITSML log capabilities. |
|  | *Example files* |  |
|  | *WITSML Store Application Program Interface (API) v1.4.1* (or higher) | Specification document for WITSML. |

# Technology

ETP is implemented using these main open source technologies:

* WebSocket for data transport
* Apache Avro for data encoding

On top of these technologies, ETP defines a set of protocols and messages that are used to exchange oil industry data.

This chapter briefly explains each of these technologies.

## WebSocket

The WebSocket protocol is defined in the Internet Engineering Task Force (IETF) RFC 6455 (<http://tools.ietf.org/html/rfc6455>). It was initially delivered with HTML5, but it now has been moved to its own W3C specification (<http://www.w3.org/TR/2009/WD-websockets-20091222/>).

WebSocket enables a bi-directional socket connection between a Web client and Web server. It typically uses the same ports as standard HTTP or HTTPS traffic (ports 80/443) and can traverse firewalls.

Data are exchanged via the WebSocket in the form of “messages,” which are collections of bytes that can represent ASCII or binary data. The communication is similar to using a TCP/IP socket but with enhancements that facilitate data message framing and handshaking. The WebSocket protocol does not make any assumptions about or place any restrictions on the data content of messages.

For a good introduction to WebSocket technology, see the WebSocket website: <http://www.websocket.org/aboutwebsocket.html>

## Apache Avro

Apache Avro is a technique for encoding and serializing data. Avro is used for data serialization in "Big Data" storage systems such as Hadoop. Data structures are defined in Avro schemas, which are represented as JSON documents. These data structures can be instantiated in most programming languages, then populated and serialized into either an efficient binary format or a more bulky JSON ASCII format.

Avro allows data to be transferred between different software environments, e.g., from .NET on Windows to Java on UNIX. Avro also includes specifications for storage of Avro data in a file format that includes the schema and the data, which enables the data file to be decoded by any programming language that understands the Avro specification.

## ETP Protocols and Messages

The table below lists and describes the protocols planned for ETP; this initial release includes protocols 0 and 1 to deliver the core messaging functions and exchange of real-time channel data (log data).

Each protocol has a number of defined messages that are Avro data structures used to transfer the data.

| Protocol | Name | Description |
| --- | --- | --- |
| 0 | Core | Basic connection protocol to establish communication session |
| 1 | Channel\_Data | Streaming and historical channel data |
| 2 | Channel\_Tabular | Special requests for gridded data (WITSML log) |
| 3 | Discovery | Navigate the server/producer data hierarchy |
| 4 | Object\_Store | Replacement for the WITSML Store API |
| 5 | Object\_Query | For more complex queries |
| 6-999 | Undefined | Reserved for future use |
| 1000-1999 | Cross-protocol | Messages that are used in multiple protocols, such as Error or Acknowledge (work in progress) |
| 2000+ | Custom | Available for custom use (not Energistics) |

# Implementation

ETP uses different technology from the WITSML Store API, so client and server software need to link to different software development libraries to implement ETP. There are three layers of technology required to implement ETP are described in the table below.

WebSocket connections are made between a Web server and a client application or Web page. After the connection is made, the ETP data transfer may take place in either direction. A wellsite data producer may act as a Web client and send data to the Web server, or an end-user display application may request data to be streamed in real time from a WITSML/ETP server.

| Technology | Description |
| --- | --- |
| ETP Avro Schemas | Definitions of the ETP domain-specific data objects. These are defined within the ETP project and delivered as JSON Avro schema files. The schema files can then be compiled into a language-specific form.   * For the Java implementation, an Apache utility named avro\_tools-1.7.6.jar can be used to generate Java source code from Avro schemas. * To support implementation of other languages, similar utilities are available from Apache and third-party sources.   The main languages supported by the ETP project are C# (.NET) and Java. |
| Apache Avro | Requires libraries from Apache and/or third parties to implement the Avro data serialization and de-serialization of the objects defined for ETP.   * For Java, the org.apache.avro library is used, along with the Jackson libraries for JSON serialization. * For .NET, the Apache.Avro package is used. |
| WebSocket | For Microsoft platforms, the .NET System.Net.WebSockets namespace provides the interfaces for transmitting and receiving data over WebSockets.  For Java, the WebSocket API is defined in the javax.websocket package.  Third parties provide the actual classes for implementation of the interfaces, for example:   * SuperWebSocket provides server side .NET components * WebSocket4Net library provides client side .NET components. * Microsoft provides implementations of client and server WebSocket classes for Windows 8 and Windows Server 2012 * The Jetty project provides a Java implementation of the client and server processing classes. Other Java implementations are also available |

## Server

Not all Web servers provide support for WebSocket. Here is a list of some of the major servers and their support capabilities.

A Web server must first be set up to support WebSocket connections. This means that a server URL must be defined that invokes the creation and return of the WebSocket to the client application. The sections below show how to set up WebSocket handling in different languages.

| Web Server | Comments |
| --- | --- |
| Microsoft IIS | Supports WebSocket from version 8.0. |
| Apache httpd | No direct support for WebSocket. Can use php\_ws for server-side PHP language support or be linked to ApacheTomcat for Java Servlets. |
| Jetty | Open-source server with good support for WebSocket. Provides client and server Java implementations. |
| SuperWebSocket | C#/.NET WebSocket server implementation |
| NGINX | Supports WebSocket. Typically used with languages such as PHP or Python. |
| Node.js | JavaScript-based Web server with good support for WebSocket. |
| Tornado | Python-based web server with WebSocket support |

### Java Implementation

Here’s how to set up WebSockets in Java on a Jetty server, setting up the URL “/etp” for the server to call EtpWebSocketServlet.

Server server = **new** Server(8080);

ServletContextHandler context = **new** ServletContextHandler(ServletContextHandler.*SESSIONS*);

context.addServlet(EtpWebSocketServlet.**class**,"/etp");

The code below is the EtpWebSocketServlet, which sets the EtpWebSocket class to handle WebSocket traffic.

@WebServlet(name = "ETP WebSocket Servlet", urlPatterns = { "/etp" })

**public** **class** EtpWebSocketServlet **extends** WebSocketServlet

{

@Override

**public** **void** configure(WebSocketServletFactory factory)

{

factory.getPolicy().setIdleTimeout(20000);

factory.register(EtpWebSocket.**class**);

System.*out*.println("Servlet Installed");

}

}

#### WebSocketListener Interface

The Jetty WebSocket library defines the WebSocketListener interface and an abstract class called WebSocketAdapter that implements the methods of the interface. The EtpWebSocket class in the above example extends WebSocketAdapter into an actual implementation of WebSocket handling code. The methods in the WebSocketListener interface are described below.

##### public void onWebSocketConnect(Session session)

This method is called when an application connects to the WebSocket. It is passed a Session object that provides information about the HTTP/WebSocket session. This session object can be used to extract information about the calling application.

##### public void onWebSocketBinary(byte[] payload, int offset, int len)

This method is called when a binary data message is received on the WebSocket. For ETP, this binary data is made up of ETP Avro data objects. The first object in a message is always a MessageHeader. The Message header describes the protocol and message type that can be used to decode the rest of the message. This is the JSON schema that describes the MessageHeader object:

{

"type": "record",

"namespace": "Energistics.Datatypes",

"name": "MessageHeader",

"fields":

[

{ "name": "protocol", "type": "int" },

{ "name": "messageType", "type": "int" },

{ "name": "correlationId", "type": "long" },

{ "name": "messageId", "type": "long" },

{ "name": "messageFlags", "type": "int" }

]

}

The first two fields define the protocol number and message type. These are used by the server to determine how to read the rest of the message from the WebSocket. The code to read the MessageHeader from the binary WebSocket data stream looks like this:

ByteBuffer bb = ByteBuffer.*wrap*(payload); // Create ByteBuffer from received byte array

Parser parser = **new** org.apache.avro.Schema.Parser(); // Create an Avro parser

ArrayList<ByteBuffer> l = **new** ArrayList<>(); // Put data into a List object

l.add(bb);

ByteBufferInputStream in = **new** ByteBufferInputStream(l); Set up an input stream

BinaryDecoder bd = DecoderFactory.*get*().binaryDecoder(in,**null**); // Avro decoder

// Create a MessageHeader reader

SpecificDatumReader<MessageHeader> reader = **new**  SpecificDatumReader<MessageHeader>(MessageHeader.**class**);

MessageHeader head = reader.read(**null**, bd); // Read in the Message header

System.*out*.print("Protocol: "+head.getProtocol());

System.*out*.print(" Message Type: "+head.getMessageType());

System.*out*.print(" Correlation Id: "+head.getCorrelationId());

System.*out*.print(" Message Id: "+head.getMessageId());

System.*out*.println(" Message Flags: "+head.getMessageFlags());

After the binary decoder has been set up, it can be used for subsequent calls to read data objects from the WebSocket using SpecificDatumReaders for the required data objects.

After the MessageHeader has been obtained, the server code can set up switch statements to process the protocols and their messages as follows:

**switch**(protocol)

{

**case** 0:

**switch** (msgType)

{

**case** 1: // Request a session – read the RequestSession message

SpecificDatumReader<RequestSession> rrs = **new**

SpecificDatumReader<RequestSession>(RequestSession.**class**);

RequestSession request = rrs.read(**null**, bd);

// Insert request session handling code here

**break**;

**case** 5: // Close the session

SpecificDatumReader<CloseSession> rcs = **new**

SpecificDatumReader<CloseSession>(CloseSession.**class**);

CloseSession cs = rcs.read(null, bd);

Etc….

Details of the ETP protocols and messages can be found in the *ETP Specification Document*.

##### public void onWebSocketText(String message)

This method is called when a text data message is received on the WebSocket. For ETP, the text consists of ETP data objects encoded in JSON format.

##### public void onWebSocketError(Throwable cause)

Called when an error occurs on the WebSocket.

##### public void onWebSocketClose(int statusCode, String reason)

Called when the WebSocket is closed. The statusCode can be used to identify the reason for the WebSocket being closed. This method can be used to implement any clean-up of local data objects when the socket connection is closed.

### C# .NET Implementation

Download SuperWebSocket from source <https://superwebsocket.codeplex.com/>. Open the Superwebsocket solution and modify the BasicSubProtocol.cs to replace Basic subprotocol to energistics-tp

Change this line of code :

/// <summary>

/// Default basic sub protocol name

/// </summary>

public const string DefaultName = "Basic";

With this

/// <summary>

/// Default basic sub protocol name

/// </summary>

public const string DefaultName = "energistics-tp";

Below is the code to start the server

var appServer = new WebSocketServer();

appServer.NewSessionConnected += appServer\_NewSessionConnected;

appServer.NewMessageReceived += appServer\_NewMessageReceived;

appServer.NewDataReceived += appServer\_NewDataReceived;

appServer.SessionClosed += appServer\_SessionClosed;

//Setup the appServer

if (!appServer.Setup(2012)) //Setup with listening port

{

Console.WriteLine("Failed to setup!");

Console.ReadKey();

return;

}

if (!appServer.Start())

{

Console.WriteLine("Failed to start!");

Console.ReadKey();

return;

}

Console.WriteLine("The server started successfully");

#### Superwebsocket Event Handler

The Superwebsocket library defines the callback as regular .NET event handler. The code above attaches the event handler before starting up the server. The event handler in WebSocket server is described below.

##### public event SessionHandler<TAppSession> NewSessionConnected

This event handler will be fired when a client connects to the server. It passed a Session object that provides information about the HTTP/WebSocket session. This session object can be used to extract information about the calling application and also sending the message back to the client.

##### public event SessionHandler<TWebSocketSession, byte[]> NewDataReceived

This event handler will be fired when a binary message is received on server. For ETP, this binary data is made up of ETP Avro data objects. The first part in a message is always a MessageHeader. The Message header describes the protocol and message type that can be used to decode the rest of the message.

This is the JSON schema that describes the MessageHeader object:

{

"type": "record",

"namespace": "Energistics.Datatypes",

"name": "MessageHeader",

"fields":

[

{ "name": "protocol", "type": "int" },

{ "name": "messageType", "type": "int" },

{ "name": "correlationId", "type": "long" },

{ "name": "messageId", "type": "long" },

{ "name": "messageFlags", "type": "int" }

]

}

The first two fields define the protocol number and message type. These are used by the server to determine how to read the rest of the message from the WebSocket. The code to read the MessageHeader from binary WebSocket data stream looks like this :

// create memory stream to wrap byte[] payload

var inputStream = new MemoryStream(payload);

// create avro binary decoder to read from memory stream

var decoder = new BinaryDecoder(inputStream);

// get message header schema, need to cache this one.

var headerSchema = new MessageHeader().Schema;

// create specific reader for message header

var headerReader = new SpecificReader<MessageHeader>(headerSchema, headerSchema);

// deserialize the header

var messageHeader = headerReader.Read(null, decoder);

Console.WriteLine("Protocol : {0}", messageHeader.Protocol);

Console.WriteLine("Message Type : {0}", messageHeader.MessageType);

Console.WriteLine("Correlation Id : {0}", messageHeader.CorrelationId);

Console.WriteLine("Message Id : {0}", messageHeader.MessageId);

Console.WriteLine("Message Flags : {0}", messageHeader.MessageFlags);

After the binary decoder has been setup, it can be used for subsequent calls to read the ETP message body from the WebSocket using SpecificDatumReaders for the required data objects.

The server code can set up switch statements to process the protocols and their messages as follows:

switch (messageHeader.Protocol)

{

case 0:

switch (messageHeader.MessageType)

{

// type of RequestSession

case 1:

var requestSession = new RequestSession();

// **TODO need to cache the schema**

var requestSchema = requestSession.Schema;

var rrs = new SpecificReader<RequestSession>(requestSchema, requestSchema);

var request = rrs.Read(requestSession, decoder);

// insert request session handling code here

break;

case 5:

var closeSession = new CloseSession();

// **TODO need to cache the schema**

var closeSchema = closeSession.Schema;

var rcs = new SpecificReader<CloseSession>(closeSchema, closeSchema);

var close = rcs.Read(closeSession, decoder);

// insert close session handling code here

break;

}

break;

// another protocol handler

}

##### public event SessionHandler<TWebSocketSession, string> NewMessageReceived

This event handler will be fired when a text data message is received on the server. For ETP, the text consists of ETP data objects encoded in JSON format.

##### public event SessionHandler<TAppSession, CloseReason> SessionClosed

This event handler will be fire when WebSocket connection is closed. The close reason can be used to identify the reason for the WebSocket being closed. This method can be used to implement any clean-up of local data objects when the socket connection is closed.

## Client

### Java Implementation

Create a WebSocket client using the Jetty Java libraries using one of the following methods:

* Use an annotated plain old java objects (POJO) implementation
* Extend the Jetty implementation classes

The examples below use the annotation POJO method. The Jetty libraries provide annotations that can be used to build a WebSocket client application. The @WebSocket annotation is used to convert a plain Java class into an implementation of a WebSocket endpoint.

The annotation can also take a number of arguments that control the behavior of the WebSocket connection, which include:

* inputBufferSize
* maxBinaryMessageSize
* maxIdleTime
* maxTextMessageSize

#### Opening the WebSocket

This example code shows how to open and connect to the WebSocket.

@WebSocket

**public** **class** MyEtpClient

{

**public** MyEtpClient(URI uri)

{

WebSocketClient client = **new** WebSocketClient();

SendBinaryData socket = **this**; // This class is annotated as a WebSocket

client.start();

ClientUpgradeRequest request = **new** ClientUpgradeRequest();

// Set HTTP header to indicate ETP binary transfer

request.setHeader("etp-encoding", "binary");

// Set up the ETP protocol HTTP header parameter

request.setHeader("Sec-WebSocket-Protocol", "energistics-tp");

client.connect(socket, uri, request);

Within the class, the following annotations (explained in Sections 3.2.1.2 through 3.2.1.4) are used to define the methods that handle WebSocket events.

#### @OnWebSocketConnect

Used to annotate the method used to handle new connections.

@OnWebSocketConnect

**public** **void** onConnect(Session session)

{

System.*out*.printf("Got connect: %s%n", session);

// Add any required connection code here

}

#### @OnWebSocketMessage

Used to handle data received on the WebSocket. There are two versions of this method: one for handling text messages and one for handling binary messages.

@OnWebSocketMessage

**public** **void** onMessage(Session session, String message )

{

System.*out*.println("Received: " + message);

// Add message handling code here

}

@OnWebSocketMessage

**public** **void** onMessage**(Session session, byte[] bytes, int offset, int length )**

{

System.*out*.println("Received binary data”);

// Add message handling code for binary data here

}

#### @OnWebSocketClose

Used to handle the closing of the WebSocket.

@OnWebSocketClose

**public** **void** onClose(**int** statusCode, String reason)

{

System.*out*.printf("Connection closed: status: " + statusCode + " reason: " + reason);

// Add any required cleanup code here

}

### C# .NET Implementation

Create a WebSocket client using Websocket4Net. This can be downloaded with NuGet or from Codeplex <https://websocket4net.codeplex.com/>

This library also uses event handler approach for maintaining connection with server.

#### Opening the WebSocket

This example code shows how to open and connect to the WebSocket.

var websocket = new WebSocket(

"ws://localhost:2012/",

subProtocol: "energistics-tp",

customHeaderItems: new List<KeyValuePair<string, string>>

{

new KeyValuePair<string, string>("etp-encoding", "binary")

});

websocket.Opened += websocket\_Opened;

websocket.Error += websocket\_Error;

websocket.Closed += websocket\_Closed;

websocket.MessageReceived += websocket\_MessageReceived;

websocket.DataReceived += websocket\_DataReceived;

websocket.Open();

#### public event EventHandler Opened

Used to notify that the server accepted the connection request

private void websocket\_Opened(object sender, EventArgs e)

{

Console.WriteLine("Opened");

}

#### public event EventHandler<MessageReceivedEventArgs> MessageReceived

Used to handle text data received on the WebSocket.

private void websocket\_MessageReceived(object sender, MessageReceivedEventArgs e)

{

Console.WriteLine("Received text message: " + e.Message);

// add message handling code for text mesage here

}

#### public event EventHandler<DataReceivedEventArgs> DataReceived

Used to handle binary data received on the WebSocket.

private void websocket\_DataReceived(object sender, DataReceivedEventArgs e)

{

Console.WriteLine("Received binary data : " + e.Data);

// add message handling code for binary data here

}

#### public event EventHandler Closed

Used to handle connection closed.

private void websocket\_Closed(object sender, EventArgs e)

{

Console.WriteLine("Connection closed");

// Add any required cleanup code here

}

#### public event EventHandler<ErrorEventArgs> Error

Used to handle error in connection.

private void websocket\_Error(object sender, ErrorEventArgs e)

{

Console.WriteLine("Exception : {0}", e.Exception.Message);

}

## Implementation with WITSML 1.4.1

ETP can be used alongside an existing WITSML v1.4.1 system to enable more efficient transfer of real-time log data. In ETP, WITSML log curves are represented by ETP channels. The channels are identified by a URI, which will in future be represented by a UUID value. Log curves in WITSML v1.4.1 do not have universally unique identifiers in their logCurveInfo elements, so it is proposed to use a PRODML style of identifier as the ETP channel identifier. This identifier takes the form of a well/wellbore/log/curve hierarchy such as:

well(uidWell)/wellbore(uidWellbore)/log(uidLog)/curve(mnemonic)

This is used in the channelUri field of the ChannelMetadataRecord message of ETP. The structure of ChannelMetadataRecord is:

{

"type": "record",

"namespace": "Energistics.Datatypes.ChannelData",

"name": "ChannelMetadataRecord",

"fields":

[

{ "name": "indexId", "type": "int" },

{ "name": "channelUri", "type": "string" },

{ "name": "channelID", "type": "int" },

{ "name": "mnemonic", "type": "string" },

{ "name": "dataType", "type": "string" },

{ "name": "uom", "type": "string" },

{ "name": "startIndex", "type": ["Energistics.Datatypes.ChannelData.RowIndex", "null"] },

{ "name": "endIndex", "type": ["Energistics.Datatypes.ChannelData.RowIndex", "null"] },

{ "name": "description", "type": "string" },

{ "name": "additionalIndexes", "type": { "type": "array", "items": "int" } },

{ "name": "status", "type": "Energistics.Datatypes.ChannelData.ChannelStatuses" }

]

}

A raw data producer populates the URI into a ChannelMetadata message that it sends before the channel streaming data starts. An ETP consumer populates the required channelUris into a ChannelDescribe message that it sends to an ETP producer to request a transfer of real-time data.

### Implementation Scenarios

There are a number of different use cases for ETP data transfer to and from a WITSML v1.4.1 server. ETP protocol 1 recognizes two types of real-time data producer, a Simple Streaming producer and a Normal Streaming producer. The differences between these are:

|  |  |
| --- | --- |
| **Simple Streaming Producer** | **Normal Streaming Producer** |
| Does not retain any history of channel data | Retains channel data history |
| Ignores **ChannelDescribe** messages, streams all channels as they are available | Uses **ChanelDescribe** messages from consumer to control which data items are streamed |
| After receiving the Start message from the consumer, sends **ChannelMetadata** for all channels. Does not send **ChannelData** messages until **ChannelStreamingStart** is received. | After receiving the Start message from the consumer, waits for **ChannelDescribe**, then only sends **ChannelMetadata** for the channels requested in the **ChannelDescribe** message. Does not send **ChannelData** messages until **ChannelStreamingStart** is received. |
| Ignores the content of **ChannelStreamingStart**/**Stop** but uses the **ChannelStreamingStart**/**Stop** to control the flow of streaming data. All channels are started and stopped. | Reads the content of **ChannelStreamingStart**/**Stop** and uses the data to start and stop the streaming of individual channels |
| Does not send **ChannelDataChange** messages | Sends **ChannelDataChange** messages to inform of any changes to the metadata or the historical data for a channel |
| Does not send **ChannelDelete** messages | Sends **ChannelDelete** messages to indicate that a channel is no longer available in the data stream. |

#### Simple Streaming Producer to Aggregating Server

In this scenario, a simple data streaming producer, e.g. a ‘smart sensor’ pushes data to an ETP aggregating server that also supports WITSML v1.4.1. The data producer does not have the capability to query the server to determine the current well and wellbore and does not store any measurement history. It is the responsibility of the aggregating server to map received channel data into its WITSML hierarchy of well/wellbore/log/curve. This will require some prior communication between the producer and consumer companies to define the meaning and mapping of the data channels. Here is a suggested sequence of activities to perform the ETP data transfer:

|  |  |
| --- | --- |
| **Producer (Simple Streaming)** | **Consumer (ETP/WITSML Aggregator)** |
| Provides an ETP WebSocket server at a known URL |  |
|  | Open WebSocket via HTTP call to WebSocket URL |
|  | Sends **RequestSession** Message, requesting protocol 1 and defining role as a consumer |
| Sends **OpenSession** |  |
|  | Sends protocol 1 **Start** message which may contain throttling parameters |
| Sends **ChannelMetadata** message which contains local identifiers for the channel(s) |  |
|  | Decodes **ChannelMetadata**, assigning channels to a well/wellbore/log/curve hierarchy. This must be completed before any channel data can be received. |
|  | Sends protocol 1 **ChannelStreamingStart** message to specify an array of the required channels |
| Goes into sending loop, sending **ChannelData** messages for all channel data as they are available. |  |
|  | Decodes **ChannelData** and appends it to appropriate WITSML log objects and/or makes it available as ETP channel data to other consumers. |
|  | While receiving data, sends data flow control messages as appropriate. **ChannelStreamingStop,ChannelStreamingStart** |
|  | Sends **CloseSession** to terminate data transfer |

At the end of a session, the consumer would normally send the CloseSession message to terminate the session but the producer could also send a CloseSession message if, for some reason, it had to stop sending data.

#### Wellsite Service Company to Aggregating Server

In this example we have a service company at the wellsite, e.g. a mud-logging or LWD data provider that is the data producer sending data to an ETP/WITSML aggregating server. In this case, the data consumer acts as the web server and is called by the producer to initiate the connection. After the initial opening of the session, the dialog is identical to that described above except for the producer being able to send ChannelDataChange and ChannelDelete messages.

|  |  |
| --- | --- |
| **Producer (Service Company)** | **Consumer (ETP/WITSML Aggregator)** |
|  | Provides a WITSML v1.4.1 server and a WebSocket server |
| Queries server via WITSML Store API to determine well/wellbore/log hierarchy.  Creates new log objects if necessary to be able to receive the streaming data. |  |
| Open WebSocket via HTTP call to WebSocket URL |  |
| Sends **RequestSession** Message, requesting protocol 1 and defining role as a producer |  |
|  | Sends **OpenSession** message |
|  | Sends protocol 1 **Start** message which may contain throttling parameters |
| Sends **ChannelMetadata** message which contains PRODML style identifiers for the channel(s) well(idwell)/wellbore(idwellbore)/log(idlog)/curve(idcurve) |  |
|  | Internally associates the channel indexes to the appropriate log/curve locations. |
|  | Sends protocol 1 **ChannelStreamingStart** message to specify an array of the required channels |
| Goes into sending loop, sending **ChannelData** messages as requested. |  |
|  | Decodes **ChannelData** messages and appends the data to appropriate WITSML log objects and/or makes it available as ETP channel data to other consumers. |
| While sending data, sends channel change messages as appropriate. **ChannelDataChange, ChannelDelete** | While receiving data, sends data flow control messages as appropriate. **ChannelStreamingStop,ChannelStreamingStart** |
| Sends **CloseSession** to terminate data transfer |  |

#### ETP Server to real-time WITSML v1.4.1 Display Client

Here we have a real-time display application that understands WITSML v1.4.1 and wants to get real-time ETP data from a server to drive displays and data plots. The client is capable of navigating the WITSML data hierarchy using the Store API.

|  |  |
| --- | --- |
| **Producer (ETP/WITSML Server)** | **Consumer (Real-time display)** |
|  | Queries server via WITSML to determine well/wellbore/log/curve hierarchy |
|  | Open WebSocket via HTTP call to WebSocket URL |
|  | Sends **RequestSession** Message, requesting protocol 1 and defining role as a consumer |
| Sends **OpenSession** message defining supported protocols. |  |
|  | Sends protocol 1 **Start** message which may contain throttling parameters |
|  | Sends **ChannelDescribe** message, listing the channels of interest. The channel identifiers are in the PRODML style: well(idwell)/wellbore(idwellbore)/log(idlog)/curve(idcurve)  NOTE: If the producer is a “Simple Streamer” the ChannelDescribe message will not be required. The producer will start sending the metadata for all channels. |
| If the producer is not capable of providing the channel data in the **ChannelDescribe** message, it sends a **ProtocolException** message with an appropriate error code and a description of the error.  If the requested channels are recognized, sends **ChannelMetadata** message which contains PRODML style identifiers for the channel URIs: well(idwell)/wellbore(idwellbore)/log(idlog)/curve(idcurve) |  |
|  | Sends **ChannelStreamingStart** to specify the required channels and their starting points |
| Goes into sending loop, sending the requested **ChannelData** messages |  |
|  | Updates the displays with data from the received **ChannelData** messages. |
| While sending data, sends channel change messages as appropriate. **ChannelDataChange,ChannelDelete** | While receiving data, sends data flow control messages as appropriate. **ChannelStreamingStop,ChannelStreamingStart** |
|  | Sends **CloseSession** to terminate data transfer |

1. Glossary

|  |  |
| --- | --- |
| Term | Description |
| ETP | Energistics Transfer Protocol |
| JSON | JavaScript Object Notation. A text-based, lightweight data interchange format. http://www.json.org/ |
| Simple Streamer | A data provider with limited capabilities e.g. does not store historical data. |
| URI | Uniform Resource Identifier. A string of characters used to identify a resource, typically at a location on the Internet. |
| UUID | Universally Unique Identifier. Internally represented by a 128-bit integer and usually represented as a 36 character sequence such as: 550e8400-e29b-41d4-a716-446655440000. |