Open Access Web Service API

Developers Guide

This Document is focused for API developers only. Refers to the Configurations and the WSDL Documentation

Sean Glazier, Reinout Heeck, Luz Morales
7/11/2017

OPEN ACCESS API

Contents

1. Design Guide 4
2. Introduction 4
3. Architecture 5
4. Synchronous System Messaging 5
5. Open Access Login 7
6. Sessions 8
7. Security 8
8. Smalltalk Client Example 9
9. Client development guidelines 11

1. Design Guide

This Design Guide is for developers writing client applications to communicate with the Open Access Servers. For installation and configuration refer to documents regarding the configuration and installation of the Open Access Servers. For developers that want a WSDL document there is one available in a separate file. Configuration of certificates that are required for HTTPS connections can be obtained from your Open Access system administrator (the exchange).

2. Introduction

This Document describes high level design of the Open Access API. The API is designed for synchronous communication, the API supports only the synchronous mode of operation as described in the following chapters.

When the Open Access API receives a request the users expects some sort of response. The mode of operation describes how that response gets to the requestor. The response arrives either right away, allowing the caller to proceed with work.

With the synchronous communication the next message cannot be sent until the previous on is finished processing.

3. Architecture

Customer applications access the Open Access API via a web service that runs on the EPEX network. The following figure shows one or more Open Access servers running on the EPEX network and the client applications using HTTPS web transport protocol to connect to the Open Access Server.

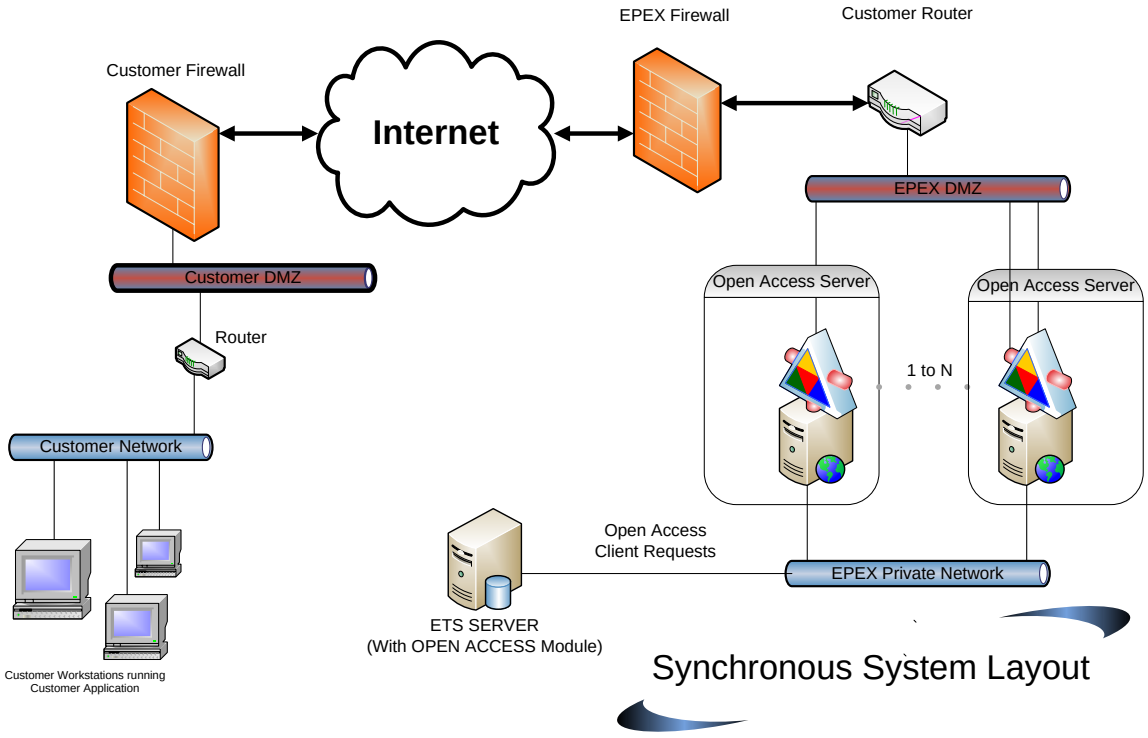

Figure 1: Synchronous System Layout

4. Synchronous System Messaging

Using HTTPS, Client Applications running on customer workstations will send Method: EstablishConnection and provide a user login name and password. Since the HTTPS protocol is used along with client certificates, all login information is secure and encrypted. For setting up server and certificates see *Open Access Servers Configuration Guide* (separate document).

Once a session is established a session token is returned that is used in the header of following API calls. This ensures session security and that orders are applied to the correct user. For most messages, a SOAP header Complex Type: AsynchronousResponseHeader is required. The 'asynchronousResponse' value, when set to 'false', will tell the system to return a reply when the server has completed the API call, this is known as using the system in synchronous mode. In this mode each API call made must be complete (received a response)

before another call can be made. The developers must use the system in this mode, it requires only a trivial processing model on the client.

For running Open Access in the synchronous mode, only communications between the Open Access Server and client is required.

Shown below is a diagram of the message flow in the synchronous mode of operation.

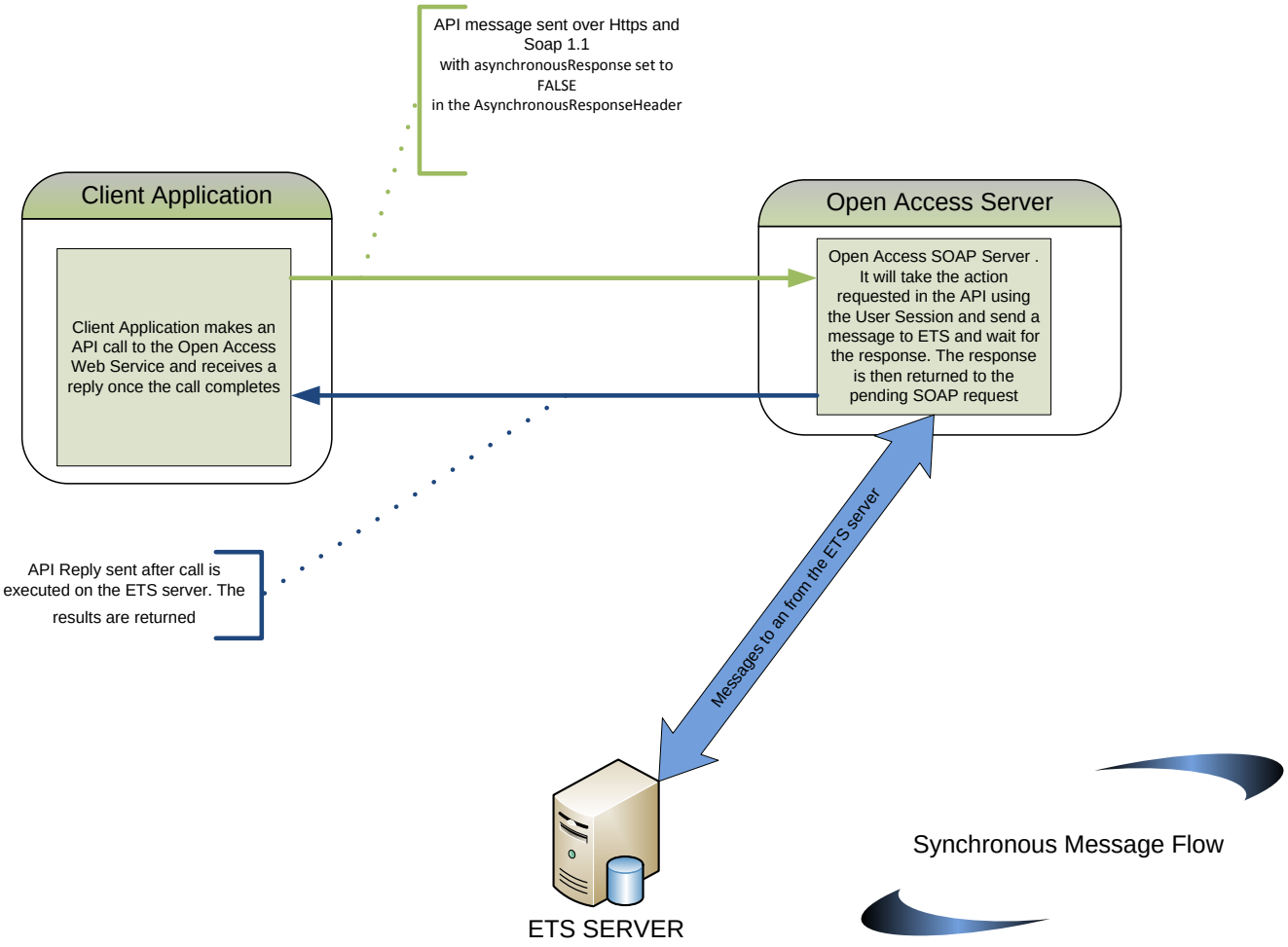

Figure 2: Synchronous Message Flow

5. Open Access Login

The first thing a client application must do to work with the Open Access API is to login and retain the resulting Session Token. The Establish Connection message implements this login functionality. The Client sends over the ETS a userloginname and password as parameters to this call. The Establish Connection Response contains the Session Token to use as a required SOAP header in later API calls.

The userloginname must be associated with the API client type in the ETS server setup. A username associated with the 'fat' ETS clienttype cannot be used to perform API calls and vice verse. This aspect is set up by the ETS server administrator.

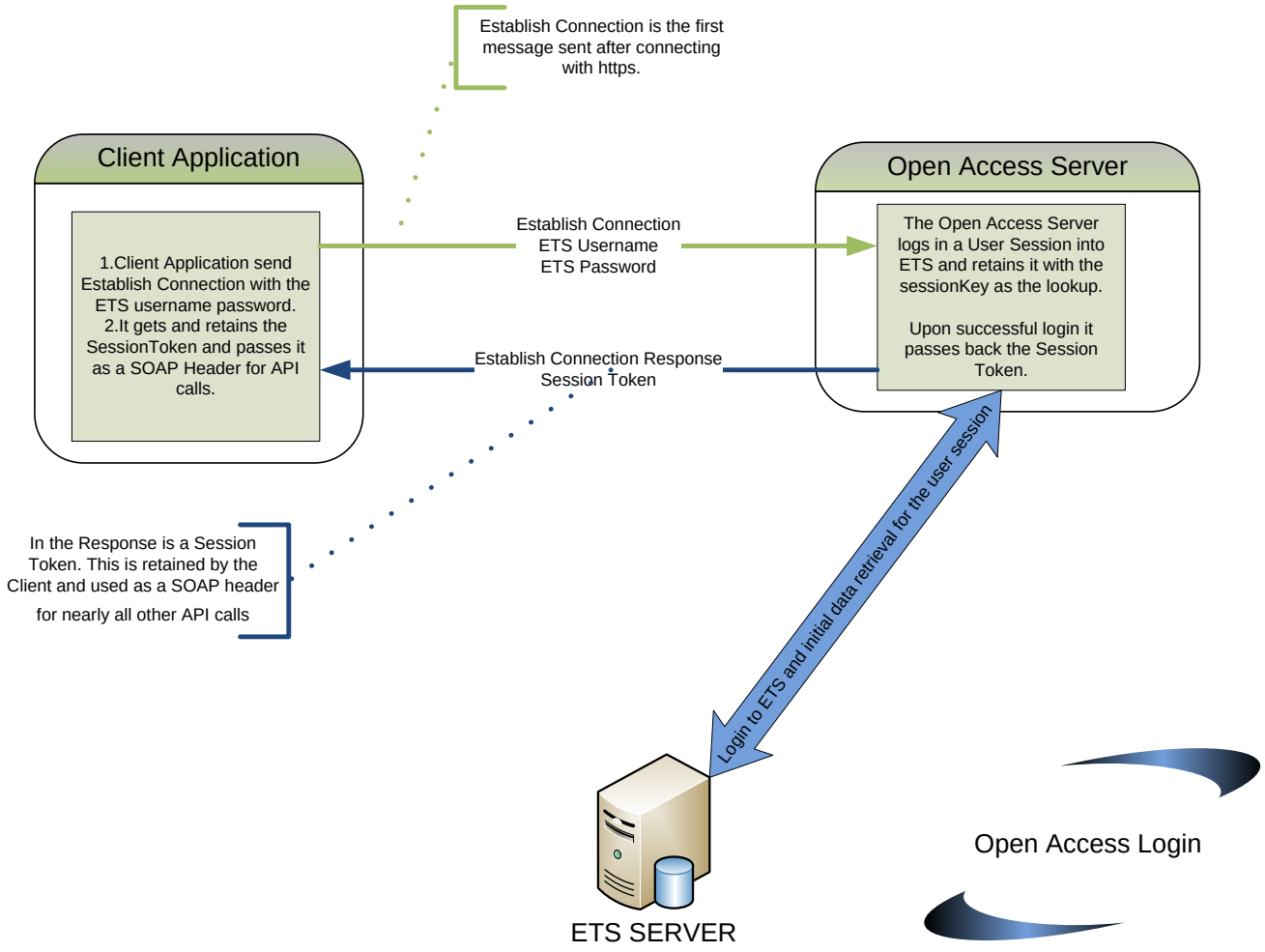

Figure 3: Open Access Login

Once a session is established it remains active on the Open Access Server until a Logout call is sent. The Open Access Server then logs the session out and de-allocates used memory resources.

6. Sessions

Retaining the User Session on the Open Access Server has the benefit of maintaining a session even if the HTTPS connection times out or gets disconnected. The client application can re-connect the HTTPS connection and continue with the same Session Token. This client does not even need to be run on the same machine but that client would need the valid client certificates in order to re-establish the https connection to begin with. This also avoids extra overhead when HTTPS connection time-outs occur. It also means the client should take care to log out his session at the end of the day. Since Https connections are used there is not a security issue with the session being held and running since establishing connection requires certificates and a valid username password.

A session can be active for a maximum of 24 hours, after that time, the HTTPS connection is considered expired and it will be closed automatically.

In case of a client crash for instance the open access session is still there and another client can log in with the aforementioned certificate and continue on from where it left off.

The server will not keep track of message states and no ID numbers will be used etc. It will be up to the client to know what API messages it has sent and what responses have been received etc.

Order states can be easily obtained by searching for either block orders with: Method: RetrieveBlockOrders or hourly orders with: Method: RetrieveOrder API calls.

Session Tokens, once received during the log in process, must be passed in the SOAP header of every API call except for Keep Alive calls. Keep Alive messages are used to keep socket sessions alive and to verify connectivity and do not need a session logged in to send one.

Note: The servers will be restarted at 3:00 AM and all the connections and sessions will be closed.

7. Security

Because communication between the Customer Application and the Open Access Portal is over the internet, it must be secure. Both authentication and encryption are required for all communications between the Customer Application and Open Access Server. HTTPS connections are used for all the SOAP requests. This requires both client and server side certificates. During the development phase, the developer will use its own generated client and server certificates. For deployment, the EPEX will need to supply server and client certificates for use by server and clients for the SSL layer communications. For development and testing an Open Access Server can be configured to run with HTTP rather than HTTPS, ask your exchange for details.

Note: The customer should take care of the expiration date of their certificates. It is recommended to implement a warning system to inform the customer when a certificate is about to expire.

8. Smalltalk Client Example

The following is an example in Smalltalk. Note the WSDL wizard has been run to build the interface classes. This example is shown to illustrate using the API. Other client languages also build service interfaces using generators that parse the WSDL. Smalltalk is used for its English-like readability.

```
"create an instance of the client"
        client := OpenAccessV3Client new.

"Services invocation"
"log on and header set for calls"
  "First Login and get the session token needed for later calls"
        args := Array with: ('John Doe') with: ('my.secret.password').
        establishConnection := client executeSelector: #'EstablishConnection' args: args.

"the establishSession response instance contains the session token we need so we ask for it and
remember it in a variable"
        sessionToken := establishConnection sessionToken.


"further requests will need to set up the required Soap headers. We set it up for synchronous
communication"
        asyncHeader :=      AsynchronousResponseHeader new
                                asynchronousResponse: (false);
                                responseToken: ('');
                                yourself.

"set up the SessionToken and the AsynchronousResponseHeader both headers are required "
        ( client headerFor: #'AsynchronousResponseHeader')
                value: (asyncHeader).
                client requestHeaderAt:    #'AsynchronousResponseHeader' put: ( client headerAt:
        #'AsynchronousResponseHeader' ifAbsent:[nil]).

        ( client headerFor: #'SessionToken') value: (sessionToken).
        client requestHeaderAt: #'SessionToken' put: (    client headerAt: #'SessionToken' ifAbsent:[nil] ).
```

```
"Services invocation"
" Now the client is set up we can invoke service methods"
        args := Array new.
        result := client executeSelector: #'RetrieveTradableAreaSets' args: args.

  "the result is an instance of the complex type RetrieveTradeableAreaSetsAcknowledgement

It will contain the areaSetNames instance variable which is an array of strings"
```

This example is a small illustration of the using the API in synchronous mode (indicated in the required SOAP header). First we log in, then we ask the result for the sessionToken needed for future client invocations. The SOAP headers are set up: the AsynchronousResponseHeader has 'asynchronousResponse' variable set to 'false' to request synchronous responses and the 'SessionToken' header will carry the session token obtained from the login sequence. Now that the SOAP Headers are set up we can make an API call. As an example we execute 'RetrieveTradableAreaSets' and we get back an instance of RetrieveTradeableAreaSetsAcknowledgement which contains the 'areaSetNames' we wanted.

The API is fairly straight forward to use in any language especially if there is a class builder that reads the WSDL and generates the interfaces and classes for you. The steps outlined above will be the same but in the language of your choice.

9. Client development guidelines

The development of a client should follow the following rules:

The client cannot do polls that keep the server busy continuously. If the customer is implementing retries, this should be done with an exponential back off algorithm.
E.g.: Ask retrieveMarketResults every 3 seconds will produce a collapse in the server, a possible poll example can be this:

Time (seconds)	Tries	Status
0	1	Fail
5	2	Fail
15	3	Fail
30	4	Success

The client should not take up unnecessary resources. E.g.: a client should be logged out when it has finished its actions.

No undocumented methods should be used.

The client must not try asynchronous mode in the client, as this is not supported yet.

The client must use its own credentials (every customer should have its own certificates issued by or through EPEX).

The developer should know that the API is protected at the moment by one measure. There are internal limits for the information to be returned and these limits depend of the kind of collection.

The developer should know the http connection will be closed if the initial handshake is not done in less than six seconds, therefore the code should not introduce delays during the handshake.

These guidelines and this measure should be enough to guard the well functioning of the API.