WSO2 ESB - A Troubleshoot Guide

version 1.0

Date: 19th May 2014

Document Revisions

Date	Version	Description
19/05/2014	1.0	First version of troubleshooting guide

Author

Isuru Udana
Senior Software Engineer, WSO2

Table of Content

Summary

The objective of this article is to provide you guidance on troubleshooting issues which may occur when developing integration scenarios with WSO2 ESB.

The first part of this article describes the major troubleshooting techniques and the next part covers the common issues which you may come across during development due to misconfigurations.

Troubleshooting Techniques

In this section we will discuss the steps that can be taken to troubleshoot issues in developing integration scenarios using WSO2 ESB.

Debug logs

When there is an error in the configuration or if something goes wrong while executing the mediation flow, we usually get an error printed in the ESB console. At the same time these errors get written in to the log file (wso2carbon.log). The wso2carbon.log log file is located at the <ESB_HOME>/repository/logs directory unless we change the default location. These log statements are very much self explanatory. However, if we want to get more information to debug an issue, we can use the debug logs. By looking at the debug logs we can extract useful information of the message flow.

Debug logs can be enabled by following the below steps;

- Shutdown the ESB server
- open log4j.properties file from a text editor. The log4j.properties file is located at <ESB_HOME>/repository/conf directory.
- Set log level to debug for log4j.category.org.apache.synapse as follows
    log4j.category.org.apache.synapse=DEBUG
- Start ESB server

org.apache.synapse is the package of synapse core which does most of the processing in WSO2 ESB. However you may have to enable debug logs for other packages depending on the requirement.
For example, synapse transport level debugging can be enabled by setting the following entry;
log4j.category.org.apache.synapse.transport=DEBUG

The following is an example of DEBUG level logs getting printed when iterate mediator is executed.

```
[2014-05-04 19:58:38,783] DEBUG - IterateMediator Start : Iterate mediator
[2014-05-04 19:58:38,787] DEBUG - IterateMediator Splitting with XPath : //m0:getQuote/m0:request resulted in 4 elements
[2014-05-04 19:58:38,788] DEBUG - IterateMediator Submitting 1 of 0 messages for processing in parallel
[2014-05-04 19:58:38,788] DEBUG - Axis2SynapseEnvironment Creating Message Context
[2014-05-04 19:58:38,798]  WARN - MessageHelper Deep clone not happened for property : tenant.info.id. Class type : java.lang.Integer
[2014-05-04 19:58:38,798]  INFO - MessageHelper Parent's Fault Stack : [] : Child's Fault Stack :[]
[2014-05-04 19:58:38,799] DEBUG - Target Target mediation : START
[2014-05-04 19:58:38,799] DEBUG - Target Asynchronously mediating using the in-lined anonymous sequence
[2014-05-04 19:58:38,799] DEBUG - Axis2SynapseEnvironment Injecting MessageContext for asynchronous mediation using the : Anonymous Sequence
[2014-05-04 19:58:38,800] DEBUG - Target Target mediation : END
```

In the above logs we can extract useful information like the number of elements that a message is split into and whether the message processing is happening in parallel or sequential among others.

Trace Logs

Trace logs trace the entire path when a message travels along a mediation sequence. Tracing can be enabled for a proxy or for a sequence. This can be done by adding the following attribute for the proxy/sequence configuration;

trace="enable"

When tracing is enabled, trace logs can be seen from the mediation tracer UI or from the wso2-esb-trace.log file. This file is located at the <ESB_HOME>/repository/logs directory.

More fine grained trace logs can be enabled by setting the log level to TRACE in trace logger. The following is the log4j.properties entry to make that change;

log4j.category.TRACE_LOGGER=TRACE, TRACE_APPENDER, TRACE_MEMORYAPPENDER

The following is a sample trace log for a proxy service

```
22:15:43,020 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER Proxy Service SplitAggregateProxy received a new message from :
127.0.0.1
22:15:43,020 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER Message To: /services/SplitAggregateProxy
22:15:43,020 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER SOAPAction: urn:getQuote
22:15:43,020 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER WSA-Action: urn:getQuote
22:15:43,021 [-] [PassThroughMessageProcessor-1] TRACE TRACE_LOGGER Envelope : <?xml version="1.0"
encoding="utf-8"?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Body></soapenv:Body></soapenv:Envelope>
22:15:43,023 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER Using the anonymous in-sequence of the proxy service for
mediation
22:15:43,023 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER Start : Sequence <anonymous>
22:15:43,023 [-] [PassThroughMessageProcessor-1] TRACE TRACE_LOGGER Message : <?xml version="1.0"
encoding="utf-8"?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Body></soapenv:Body></soapenv:Envelope>
22:15:43,024 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER Sequence <SequenceMediator> :: mediate()
22:15:43,024 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER Mediation started from mediator position : 0
22:15:43,032 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER Start : Iterate mediator
22:15:47,728 [-] [PassThroughMessageProcessor-1] TRACE TRACE_LOGGER Message : <?xml version="1.0"
encoding="utf-8"?><soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Header
xmlns:wsa="http://www.w3.org/2005/08/addressing"><wsa:To>http://localhost:8280/services/SplitAggregateProxy</wsa:To><wsa:Messa
geID>urn:uuid:be8621bc-3018-4251-adf7-81c5229388be</wsa:MessageID><wsa:Action>urn:getQuote</wsa:Action></soapenv:Header><so
apenv:Body><m0:getQuote
xmlns:m0="http://services.samples"><m0:request><m0:symbol>IBM</m0:symbol></m0:request><m0:request><m0:symbol>IBM</m0:symbol>
</m0:request><m0:request><m0:symbol>IBM</m0:symbol></m0:request><m0:request><m0:symbol>IBM</m0:symbol></m0:request></m0:ge
tQuote></soapenv:Body></soapenv:Envelope>
22:15:49,396 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER Splitting with XPath : //m0:getQuote/m0:request resulted in 4
elements
22:15:49,396 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER Submitting 1 of 0 messages for processing in parallel
22:15:49,412 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER Submitting 2 of 1 messages for processing in parallel
22:15:49,413 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER Submitting 3 of 2 messages for processing in parallel
22:15:49,414 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER Submitting 4 of 3 messages for processing in parallel
22:15:49,415 [-] [PassThroughMessageProcessor-1]  INFO TRACE_LOGGER End : Iterate mediator
22:15:49,416 [-] [PassThroughMessageProcessor-1] TRACE TRACE_LOGGER Message : <?xml version="1.0"
```

encoding="utf-8"?><soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing"><wsa:To>http://localhost:8280/services/SplitAggregateProxy</wsa:To><wsa:MessageID>urn:uuid:be8621bc-3018-4251-adf7-81c5229388be</wsa:MessageID><wsa:Action>urn:getQuote</wsa:Action></soapenv:Header><soapenv:Body><m0:getQuote xmlns:m0="http://services.samples"><m0:request><m0:symbol>IBM</m0:symbol></m0:request><m0:request><m0:symbol>IBM</m0:symbol></m0:request><m0:request><m0:symbol>IBM</m0:symbol></m0:request><m0:request><m0:symbol>IBM</m0:symbol></m0:request></m0:getQuote></soapenv:Body></soapenv:Envelope>
22:15:49,416 [-] [PassThroughMessageProcessor-1] INFO TRACE_LOGGER End : Sequence <anonymous>

Monitoring Messages

While developing integration scenarios with the ESB, sometimes we might get into situations where the expected results are not received for the service invocations. There can be various reasons for this. The following are some of the reasons;

- The message payload sent by the ESB is not in the format that is expected by the backend service
- The content type of the sent message is not supported by the backend service
- Some of the required headers like 'Authorization Header' is missing in the request sent from the ESB.

To debug these type of issues, we could inspect the messages that are transferred over the connection and verify whether they are in the format that is expected by the backend service. After inspecting the transferred messages, we can amend the messages to make it compatible with the backend service.

To inspect messages passed in the connections, we can use two mechanisms.

TCPMon

TCPMon is a very handy tool that can be used to monitor both messages coming into and going out from the ESB. The following diagram shows the typical message flow when an ESB proxy service is involved in a client server communication;

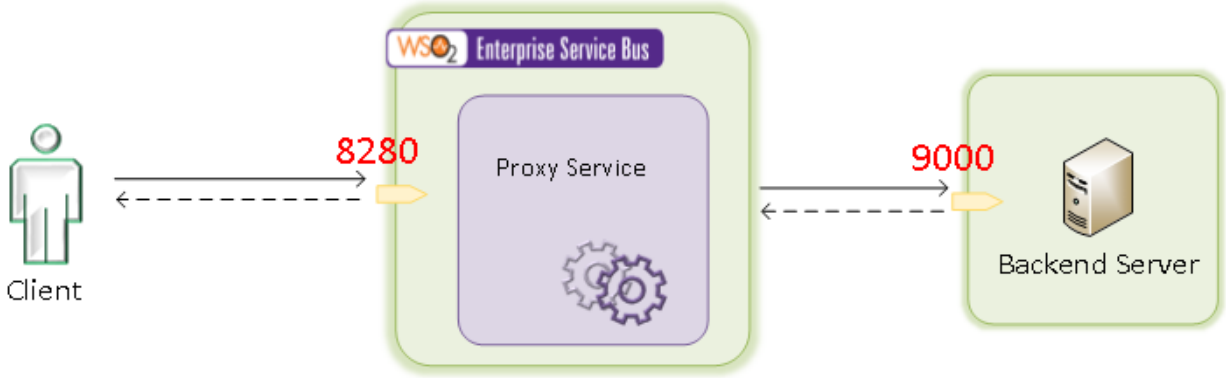

Fig. 1

There are two connections in this scenario.
1. Client-to-Proxy Service connection
2. Proxy-to-Backend Server connection

We can monitor messages flowing between the Proxy service and the Backend by placing TCPMon as follows. In this occasion we need to temporarily change the port of the endpoint to point to the TCPMon which can be done without restarting the ESB.

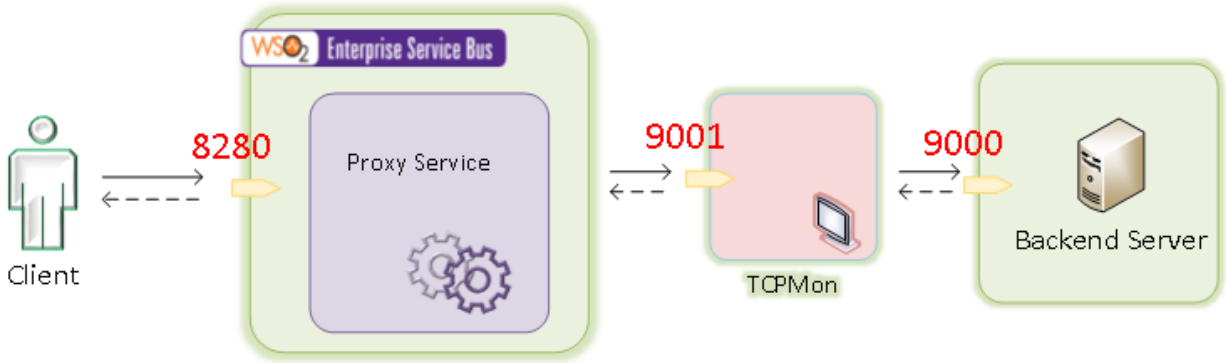

Fig. 2

We can monitor messages flowing between client and the Proxy service by placing TCPMon as follows;

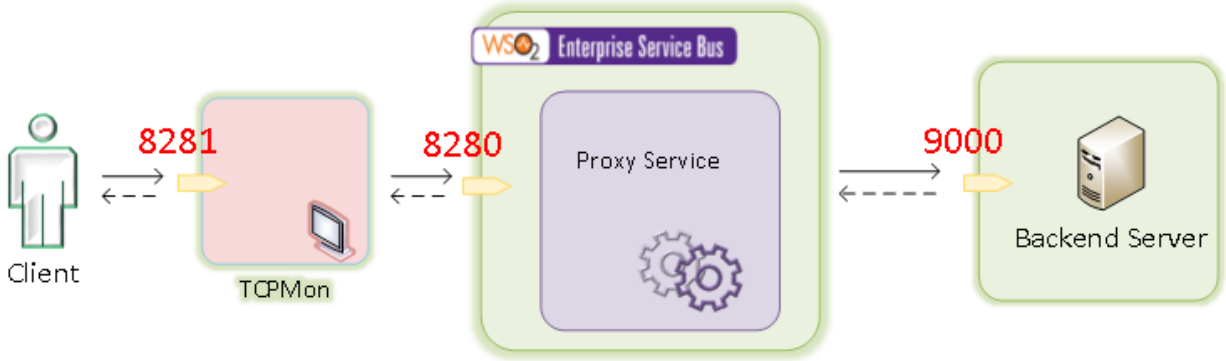

Fig. 3

Please refer to this resource for more information on using TCPMon for monitoring. Please note that we can use TCPMon for monitoring messages transferred only over a http transport.

Wire Logs

If we want to inspect the messages sent over http/https transports without using a tool like TCPMon, we can use the inbuilt functionality of the ESB - 'wire logs'.

Passthrough http transport is the main transport which handles the http/https messages in WSO2 ESB. Messages coming in to and going out from the ESB through passthrough http transport can be seen by using the wire logs functionality. One of the advantages of using wire logs over TCPMon is that you can see the messages transferred over https protocol. This is something which is not supported by TCPMon. Also, we don't have to do any modifications to the synapse configuration.

The following steps shows how we can enable wire logs for passthrough http transport;

- Shutdown the ESB server
- Open log4j.properties file from a text editor. This file is located at the <ESB_HOME>/repository/conf directory.
- Uncomment the following entry;

    log4j.logger.org.apache.synapse.transport.http.wire=DEBUG

- Start the ESB server

The above described procedure applies to WSO2 ESB 4.7.0 or higher versions. The entry to uncomment in log4j.properties is slightly different in previous versions.

For ESB 4.6.0 :

log4j.logger.org.apache.synapse.transport.passthru.wire=DEBUG (add this line to log4j.properties)

For ESB 4.5.1 or previous versions (PTT is not available in these versions):

log4j.logger.org.apache.synapse.transport.nhttp.wire=DEBUG

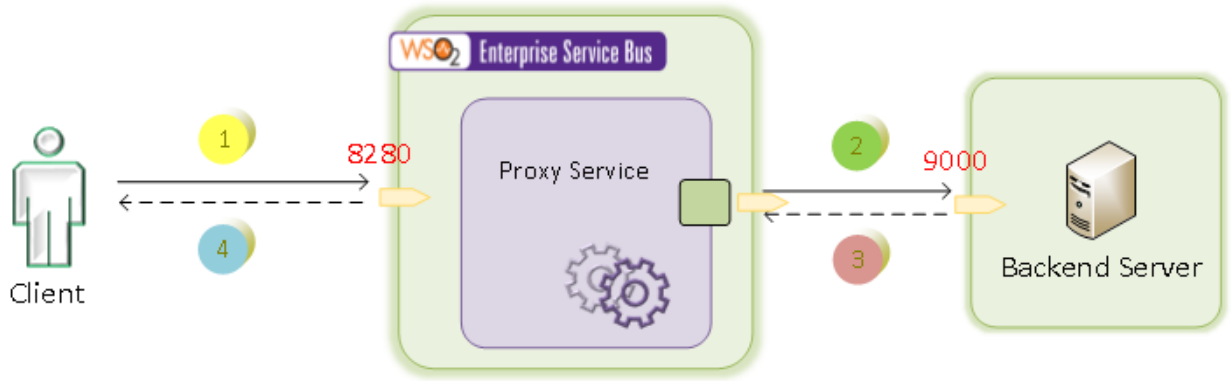

Fig. 4

```
[2013-09-22 19:47:57,797] DEBUG - wire >> "POST /services/StockQuoteProxy HTTP/1.1[\r][\n]"
[2013-09-22 19:47:57,798] DEBUG - wire >> "Content-Type: text/xml; charset=UTF-8[\r][\n]"
[2013-09-22 19:47:57,798] DEBUG - wire >> "SOAPAction: "urn:getQuote"[\r][\n]"
[2013-09-22 19:47:57,799] DEBUG - wire >> "User-Agent: Axis2[\r][\n]"
[2013-09-22 19:47:57,799] DEBUG - wire >> "Host: localhost:8280[\r][\n]"
[2013-09-22 19:47:57,799] DEBUG - wire >> "Transfer-Encoding: chunked[\r][\n]"
[2013-09-22 19:47:57,800] DEBUG - wire >> "[\r][\n]"
[2013-09-22 19:47:57,800] DEBUG - wire >> "215[\r][\n]"
[2013-09-22 19:47:57,800] DEBUG - wire >> "http://localhost:8280/services/StockQuoteProxyurn:uuid:9e1b0def-a24b-4fa2-8016-86cf3b458f67urn:getQuoteIBM[\r][\n]"
[2013-09-22 19:47:57,801] DEBUG - wire >> "0[\r][\n]"
[2013-09-22 19:47:57,801] DEBUG - wire >> "[\r][\n]"
[2013-09-22 19:47:57,846]  INFO - TimeoutHandler This engine will expire all callbacks after : 120 seconds, irrespective of the timeout action, after the specified or optional
[2013-09-22 19:47:57,867] DEBUG - wire << "POST /services/SimpleStockQuoteService HTTP/1.1[\r][\n]"
[2013-09-22 19:47:57,867] DEBUG - wire << "Content-Type: text/xml; charset=UTF-8[\r][\n]"
[2013-09-22 19:47:57,867] DEBUG - wire << "SOAPAction: "urn:getQuote"[\r][\n]"
[2013-09-22 19:47:57,867] DEBUG - wire << "Transfer-Encoding: chunked[\r][\n]"
[2013-09-22 19:47:57,868] DEBUG - wire << "Host: localhost:9000[\r][\n]"
[2013-09-22 19:47:57,868] DEBUG - wire << "Connection: Keep-Alive[\r][\n]"
[2013-09-22 19:47:57,868] DEBUG - wire << "User-Agent: Synapse-PT-HttpComponents-NIO[\r][\n]"
[2013-09-22 19:47:57,868] DEBUG - wire << "[\r][\n]"
[2013-09-22 19:47:57,868] DEBUG - wire << "215[\r][\n]"
[2013-09-22 19:47:57,868] DEBUG - wire << "http://localhost:8280/services/StockQuoteProxyurn:uuid:9e1b0def-a24b-4fa2-8016-86cf3b458f67urn:getQuoteIBM[\r][\n]"
[2013-09-22 19:47:57,868] DEBUG - wire << "0[\r][\n]"
[2013-09-22 19:47:57,869] DEBUG - wire << "[\r][\n]"
[2013-09-22 19:47:58,002] DEBUG - wire >> "HTTP/1.1 200 OK[\r][\n]"
[2013-09-22 19:47:58,002] DEBUG - wire >> "Content-Type: text/xml; charset=UTF-8[\r][\n]"
[2013-09-22 19:47:58,002] DEBUG - wire >> "Date: Sun, 22 Sep 2013 14:17:57 GMT[\r][\n]"
[2013-09-22 19:47:58,002] DEBUG - wire >> "Transfer-Encoding: chunked[\r][\n]"
[2013-09-22 19:47:58,002] DEBUG - wire >> "Connection: Keep-Alive[\r][\n]"
[2013-09-22 19:47:58,002] DEBUG - wire >> "[\r][\n]"
[2013-09-22 19:47:58,014] DEBUG - wire << "HTTP/1.1 200 OK[\r][\n]"
[2013-09-22 19:47:58,015] DEBUG - wire << "Content-Type: text/xml; charset=UTF-8[\r][\n]"
[2013-09-22 19:47:58,015] DEBUG - wire << "Date: Sun, 22 Sep 2013 14:17:58 GMT[\r][\n]"
[2013-09-22 19:47:58,015] DEBUG - wire << "Server: WSO2-PassThrough-HTTP[\r][\n]"
[2013-09-22 19:47:58,016] DEBUG - wire << "Transfer-Encoding: chunked[\r][\n]"
[2013-09-22 19:47:58,016] DEBUG - wire << "[\r][\n]"
[2013-09-22 19:47:58,016] DEBUG - wire >> "4d8[\r][\n]"
[2013-09-22 19:47:58,017] DEBUG - wire >> "urn:getQuoteResponseurn:uuid:9e1b0def-a24b-4fa2-8016-86cf3b458f673.827143922330303-8.819296796724336-
170.50810412063595170.732189445609445un Sep 22 19:47:57 IST 2013-170.472077024782785.562077973231586E7IBM
Company178.0616712932281324.9438904049222641.9564266653777567195.619084019760041BM6216[\r][\n]"
[2013-09-22 19:47:58,017] DEBUG - wire >> "0[\r][\n]"
[2013-09-22 19:47:58,018] DEBUG - wire >> "[\r][\n]"
[2013-09-22 19:47:58,021] DEBUG - wire << "4d8[\r][\n]"
[2013-09-22 19:47:58,022] DEBUG - wire << "urn:getQuoteResponseurn:uuid:9e1b0def-a24b-4fa2-8016-86cf3b458f673.827143922330303-8.819296796724336-
170.50810412063595170.732189445609445un Sep 22 19:47:57 IST 2013-170.472077024782785.562077973231586E7IBM
Company178.0616712932281324.9438904049222641.9564266653777567195.619084019760041BM6216[\r][\n]"
[2013-09-22 19:47:58,022] DEBUG - wire << "0[\r][\n]"
[2013-09-22 19:47:58,022] DEBUG - wire << "[\r][\n]"
```

To read a wire log, first we have to identify the message direction.

DEBUG - wire >> - This represent the message coming into the ESB from the wire (outside)
DEBUG - wire << - This represent the message going into the wire from the ESB

As you can see there are two incoming messages and two outgoing messages in the above log.
Logs highlighted in yellow and purple indicate the two messages coming into ESB. One of them is the incoming request from the client and the other one is the response coming from the backend service. Since only a response contains a status line, we can use the HTTP status line to differentiate them. In the above log, the message highlighted in purple has the following status line. So it is the response coming from the backend service.

```
[2013-09-22 19:47:58,002] DEBUG - wire >> "HTTP/1.1 200 OK[\r][\n]"
```

The first part of the log of a message contains the http headers and it is followed by the message payload. Due to asynchronous and concurrent request/response handling nature and the time it takes to print the logs, we might see overlapping of messages when it is get printed in the log file.

Please note that wire logs should be enabled for troubleshooting purposes only. Running productions systems with wire logs constantly enabled is not recommended.

Wire Logs for Callout Mediator/Forwarding Message Processor

The callout mediator and the forwarding message processor uses the Axis2 CommonsHTTPSender to invoke services. It does not use the non-blocking nhttp/passthrough transports. Hence, to enable wire logs we have to follow a different approach as follows;

- Shutdown the ESB server
- Open log4j.properties file from a text editor.
- Put the following two entries into the log4j.properties.

```
log4j.logger.httpclient.wire.header=DEBUG
log4j.logger.httpclient.wire.content=DEBUG
```

- Start the ESB server

Solutions for Commonly Occurring Exceptions

In this section we discuss about some of the common issues which can occur due to misconfigurations. Solutions for those issues are also provided.

Error/Warning on Log	Reason	Solution
ERROR - Axis2Sender unexpected error while sending message out org.apache.axis2.AxisFault: **The system cannot infer the transport information** from the vfs:file:///home/user/test/out URL.	ESB has failed to extract the transport protocol from the given url. This happens when transport defined in the url is not enabled in the system. In this case vfs transport sender is not enabled.	Enable relevant transport sender from axis2.xml. If you are using the Callout Mediator or the MessageProcessor, transport senders should be enabled in the axis2_blocking_sender.xml.
ERROR - RelayUtils error while building Passthrough stream org.apache.axiom.soap.SOAPProcessingException: **Transport level information does not match with SOAP Message namespace URI**	SOAP 1.2 message has been sent to a SOAP 1.1 binding endpoint or vice versa.	Correct the soap message namespace.
ERROR - RelayUtils error while building Passthrough stream org.apache.axiom.soap.SOAPProcessingException: **First Element must contain the local name, Envelope, but found getQuote**	- Message does not contain the SOAP envelope. - Message builder for a particular Content-Type is not set properly	If the message is supposed to be a SOAP message, correct the message format to have a envelope. If the message is in some format other than soap, define the correct Message Builder at the axis2.xml.When a builder is not defined for a particular Content-Type, by default SOAPBuilder is used to build the message. If the message is not in soap format we will get this error. Setting the correct builder for the Content-Type at axis2.xml will resolve this issue.
WARN - SourceHandler connection time out after request is read: http-incoming-1	Connection between the client and the ESB timeouts. Socket timeout of the http listener has been exceeded. Default socket timeout is 60 seconds.	Increase the socket timeout of the passthrough http transport. To do that add the following line to the passthru-http.properties file. http.socket.timeout=120000 Here socket timeout is set to 120 seconds.

WARN - TargetHandler http-outgoing-1: connection timeout while in state: REQUEST_DONE	Connection between the ESB and the Backend times out. Socket timeout of the http sender has been exceeded. Default socket timeout is 60 seconds.	Increase the socket timeout of the passthrough http transport. To do that add the following line to the passthru-http.properties file. http.socket.timeout=120000 Here socket timeout is set to 120 seconds.
WARN - TargetHandler connection closed by target host before receiving the request	Backend server has closed the connection after reading the request without sending a response back.	Troubleshoot at the backend server side. Increase backend server connection timeout.
WARN - TargetHandler connection closed by target host while receiving the response	Backend server has closed the connection while sending the response back to the ESB.	Troubleshoot at the backend server side.
WARN - SourceHandler I/O error (The connection was probably closed by the remote party):Broken pipe	Client has closed the connection between ESB and the client. This gets printed when ESB tries to send a response to client.	Troubleshoot at the client side.
WARN - TimeoutHandler expiring message ID : urn:uuid:bbd048bd-f504-48d6-9e0f-8146f6849eb0; dropping message after global timeout of : 120 seconds	ESB has not received a response within the endpoint timeout duration. If timeout is not configured at the endpoint, global timeout value is considered as the timeout value.	Troubleshoot the backend server side to find why response is not sent. If server takes more time than the endpoint timeout duration, increase the endpoint timeout value. If service invocation is one way, set the OUT_ONLY property to 'true' before sending the message using an endpoint.
WARN - SynapseCallbackReceiver synapse received a response for the request with message Id : urn:uuid:bbd048bd-f504-48d6-9e0f-8146f6849eb0 but a callback is not registered (anymore) to process this response	ESB has received a response from backend server after exceeding the endpoint timeout duration.	Increase the endpoint timeout value.
WARN - SourceHandler illegal incoming connection state: REQUEST_READY. Two send backs are possibly happening for the same request	ESB is trying to send a response back to client while it has already sent a response to the client for a particular service invocation.	Inspect the synapse configuration and make required changes to avoid two or more send backs.

WARN - SourceHandler trying to write response body while the handler is in an inconsistent state REQUEST_READY		
411 content length required	Chunked transfer encoding is a data transfer mechanism introduced in Http 1.1 protocol version. Some of the servers (mostly old servers which only support http 1.0) might not be able to handle chunked messages.	To get rid of this issue we need to disable chunking at the ESB for that particular service invocation. We can disable chunking by adding the following configuration. *<property name="DISABLE_CHUNKING" value="true" scope="axis2"/>* Above configuration segment makes the ESB to send the message with a content length header instead of sending chunking messages with a transfer-encoding header.
Out of several consecutive service invocations, only one service invocation succeeds while the rest of them fail.	Some of the backend servers or the way through a loadbalancer to the backend server might not able to handle http persistent connections. If out of several consecutive service invocations, only one service invocation succeeds while the rest of them fail, we can suspect that it is something to do with persistent connection handling.	As a solution to cope with those situations we can avoid making persistent connections by adding the following configuration for the message follow. *<property name="NO_KEEPALIVE" value="true" scope="axis2"/>*
WARN - PassThroughHttpSSLListener system may be unstable: HTTPS ListeningIOReactor encountered a checked exception : **Address already in use** java.net.BindException: Address already in use	This means the https Listener port is already used by some other application. There may be another ESB instance already running on the same machine.	Shutdown other instances running in the same machine which uses the same ports. If you want to run two instances of WSO2 servers in the same machine, configure port offset correctly in carbon.xml

Troubleshooting Timeout Issues

Since the above discussed issues contain a significant number of problems related to timeouts, it is good to get a clear understanding of different timeout configurations available in the ESB. Having a good understanding in different timeout configurations will certainly help with troubleshoot issues.

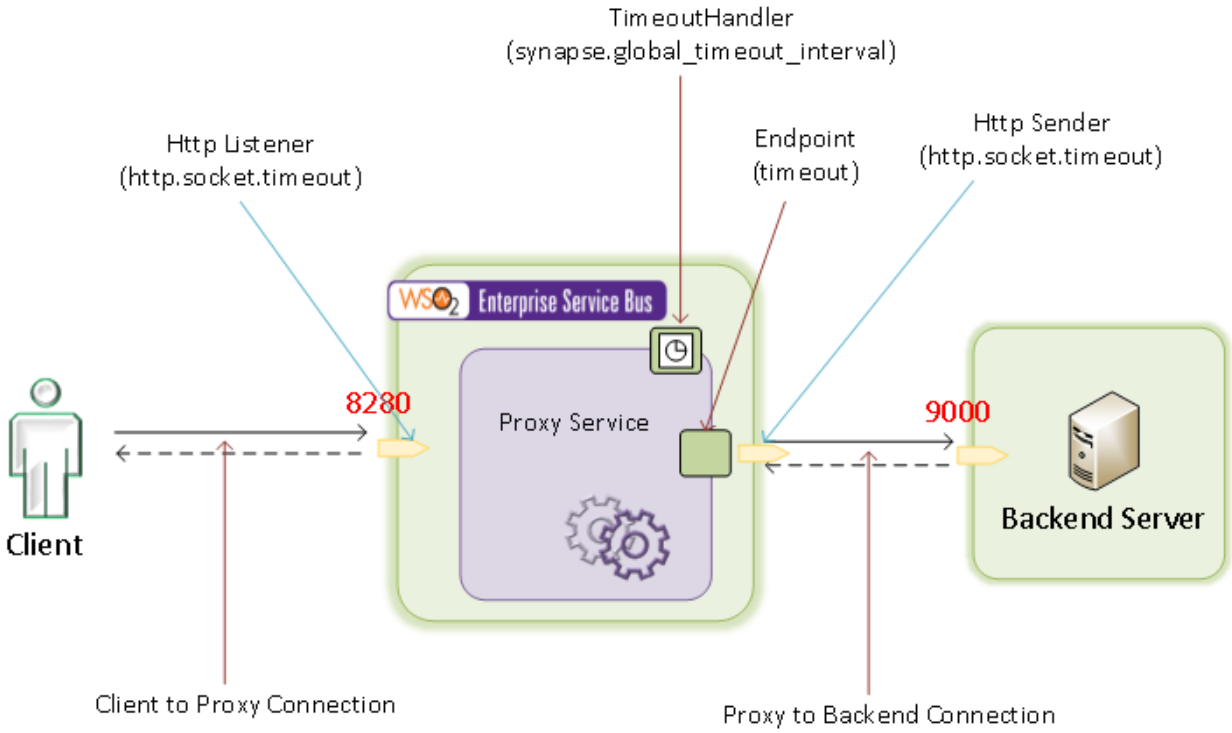

Fig. 5

The above diagram shows the typical message flow when an ESB Proxy service is involved in a client server communication. There are two connections involved in the scenario; 'Client-to-Proxy connection' and 'Proxy-to-Backend connection'. They are two separate connections that do not depend on each other. This means that even if one connection times out, the other connection won't time out at the same time.

Client-to-Proxy Connection Timeout Parameters

http.socket.timeout:
In the client-to-proxy connection we have to consider only one configuration parameter. That is http.socket.timeout which can be configured from the passthru-http.properties file. This file is located at the <ESB_HOME>/repository/conf directory. This represents the socket timeout value of the passthrough http/https transport listener.

If we use NIO transport instead of the default passthrough transport we need to use the parameter below;

ESB Version	Parameter	Configuration file
4.7.0 or later	http.socket.timeout.receiver	nhttp.properties
4.6.0 or before	http.socket.timeout	nhttp.properties

Proxy-to-Backend Connection Timeout Parameters

Timeout configuration for proxy-to-backend connection is somewhat advanced compared to the client-to-proxy connection parameters.

http.socket.timeout:
This represents the socket timeout value of the passthrough http/https transport sender. This is the same parameter used in transport listener.

If we use NIO transport instead of the default Passthrough transport, we need to use the parameter below;

ESB Version	Parameter	Configuration file
4.7.0 or later	http.socket.timeout.sender	nhttp.properties
4.6.0 or before	http.socket.timeout	nhttp.properties

Endpoint timeout:
This is the timeout configuration parameter which can be configured at the endpoint level. It allows us to configure different timeout values for different endpoints.

Below is a sample endpoint configuration which is configured with timeout parameters. The 'duration' denotes the timeout value and the 'responseAction' denotes the action which should be taken for the timeout message. Here, fault sequence will be invoked for timeouts;

```
<endpoint>
      <address uri="http://localhost:8281/services/SimpleStockQuoteService">
        <timeout>
          <duration>120000</duration>
          <responseAction>fault</responseAction>
        </timeout>
      </address>
 </endpoint>
```

It is a must to configure the http.socket.timeout to a higher value than all the endpoint timeout values.

Endpoint timeout <= http.socket.timeout

synapse.global_timeout_interval:
Synapse which is the underlying mediation engine of WSO2 ESB is a complete asynchronous messaging engine that does not block its worker threads on network IO. Instead of waiting for responses, it registers a callback for a particular request and return. When the response is received, the registered callback is used to correlate it with the request and further processing is done. If the backend server does not respond back, it is required to clear the registered callbacks after some time to prevent possible memory leaks. It is done by a timer task called TimeoutHandler. The 'synapse.global_timeout_interval' parameter represents the time duration that a callback should be kept in the callback store.

If we have configured a timeout duration at the endpoint level, this global timeout value is not taken into consideration for that particular endpoint. For all the other endpoints that don't have a timeout configuration, this global parameter value is considered as the timeout duration.

synapse.global_timeout_interval can be configured from the synapse.properties file. This file is located at the <ESB_HOME>/repository/conf directory. The default value is 120 seconds.

TimeoutHandler is executed in an interval of 15 seconds. So the time the callbacks get cleared can be deviated up to 15 seconds from the configured value. The TimeoutHandler execution interval can be configured by defining the following property at synapse.properties file: *synapse.timeout_handler_interval*

Timeout Configuration for Forwarding Message Processor/Callout Mediator

Timeout configuration of the callout mediator and the forwarding message processor is completely different from the above described procedure.

Timeouts can be be configured at the axis2_blocking_client.xml which is located at the following directory; <ESB_HOME>/repository/conf/axis2

SO_TIMEOUT and the CONNECTION_TIMEOUT are the parameters which should be configured for the http transport senders. The following is a sample configuration http transport;

```
<transportSender name="http" class="org.apache.axis2.transport.http.CommonsHTTPTransportSender">
      <parameter name="PROTOCOL">HTTP/1.1</parameter>
      <parameter name="Transfer-Encoding">chunked</parameter>
      <parameter name="cacheHttpClient">true</parameter>
      <parameter name="defaultMaxConnectionsPerHost">200</parameter>
      <parameter name="SO_TIMEOUT">120000</parameter>
      <parameter name="CONNECTION_TIMEOUT">120000</parameter>
 </transportSender>
```

Mediation Fault Handling

Configuring synapse configuration properly to handle faults is a very important aspect of developing integration scenarios using WSO2 ESB. It makes it easier to identify and debug issues that may occur in development time as well as in production.

Proxy services, APIs and the main sequence are the fundamental starting points of a message flow. When something goes wrong while mediating the messages either in inSequence or in outSequence of the proxy service, the faultSequence gets invoked. If we haven't configured a fault sequence, we will not be able to capture the error and act upon it. Keep in mind that the default fault sequence will not be invoked for failures happening in proxy services. It gets invoked only for the errors that happen at the main sequence. So it is important that we define a fault sequence for each proxy service properly and handle errors there. The same applies to APIs.

When configuration becomes complex, we usually have several sequences branching from the proxy inSequence. A fault handling sequence can be assigned at the sequence level as well. It will allow us to have a more fine grained error handling capability. If you have several sequences in your configuration, it is recommended to have a fault handling sequence defined for each and every sequence. So when something goes wrong within a particular sequence, the fault handling sequence defined at the sequence level is executed while ignoring the faultSequence defined at the proxy level.

The following is a sample fault sequence that basically logs the message with the sequence name and data related to the error like Error Message, Error Code etc. and then a soap fault is sent back to the client.

```
<sequence xmlns="http://ws.apache.org/ns/synapse" name="SampleFaultSeq">
  <log level="full">
    <property name="SEQUENCE_NAME" value="SampleFaultSeq"/>
    <property xmlns:ns="http://org.apache.synapse/xsd"
            name="ERROR_MESSAGE"
            expression="get-property('ERROR_MESSAGE')"/>
    <property xmlns:ns="http://org.apache.synapse/xsd"
            name="ERROR_CODE"
            expression="get-property('ERROR_CODE')"/>
    <property xmlns:ns="http://org.apache.synapse/xsd"
            name="ERROR_DETAIL"
            expression="get-property('ERROR_DETAIL')"/>
    <property xmlns:ns="http://org.apache.synapse/xsd"
            name="ERROR_EXCEPTION"
            expression="get-property('ERROR_EXCEPTION')"/>
  </log>
  <makefault>
        <code value="tns:Receiver" xmlns:tns="http://www.w3.org/2003/05/soap-envelope"/>
        <reason expression="get-property('ERROR_MESSAGE')"/>
   </makefault>
   <respond/>
</sequence>
```

Printing the sequence name is useful to clearly identify the sequence that the error triggered from.