C2SIM Server Reference Implementation

User Instructions

Overview

GMU C4I-Cyber center is making available C2SIM Server release 4.4 and C2SIM Client release 2.4 as an initial version of the C2SIM Reference Implementation. These packages support the following:

- Legacy IBML09 message formats for basic orders and general status reports (GSR)
- Basic C2SIM message formats for basic orders and position reports
- Sending and receiving responses to C2SIM messages when necessary
- C2SIM based initialization procedures
- Basic session control for the simulation exercise
- Translation between IBML09 and C2SIM

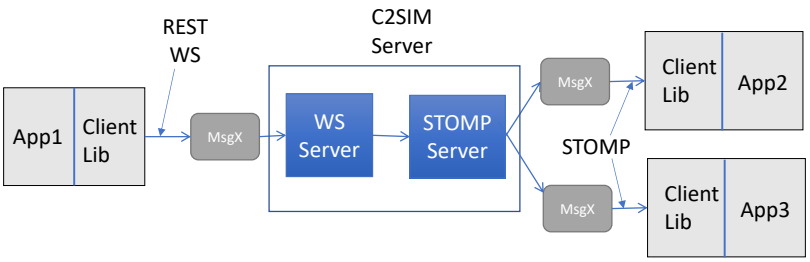
Note that the C2SIM formats used for this exercise are preliminary as the standard is still in work.

Server Configuration

The server(s) run as a VM using the following components

- Linux Centos 7
- Java Version 8
- JDOM 2.0.6
- Apache Tomcat 8.0.30 Web Services (RESTful WS)
- Apache Apollo 1.7.1 Messaging (STOMP)

The basic server configuration is shown below. Note that a single input message may be sent to several destinations.



Server Operation

Basic Message Processing

Sending – Messages are sent to the server using RESTful web services protocols via the BMLClientREST_Lib class in the C2SIM_Client Library. Based on information obtained from the XML schema, the server characterizes the message determining the type of message and the protocol used. The message is then sent to the STOMP server where it is published to all subscribed systems. The server adds STOMP headers to the outgoing message so that filtering may be done by the STOMP server or by receiving system.

In a REST transaction the client will disconnect after the message is sent and a response from the server has been received. Submitting a message consists of the following

- Instantiating a BMLClientREST_Lib object
- Setting parameters
- Sending the request
- Receiving the result
- Destroying the BMLClientREST_Lib object

Receiving – Messages sent to the server are sent to all systems that have subscribed to the STOMP server. Message receipt is via the BMLClientSTOMP_Lib class of the C2SIM_Client Library. Receiving systems (Most systems are both senders and receivers) establish a subscription via the C2SIM_Client library. The TCP connection is kept open as long as is needed and multiple messages will be received over this subscription. Both blocking and non-blocking calls to receive the next message are supported by the library. Receiving consists of the following:

- Instantiating a BMLClientSTOMP_lib object
- Set parameters (Topic should normally be /topic/BML
- Specifying optional subscription(s)
  Example: c.addAdvSubscription('protocol = 'C2SIM'");
- Connecting by executing the connect method of BMLClientSTOMP_lib.
- Loop and receive messages c.getNextBlock() or c.getNextNoBlock()
- Connection stays open until c.disconnect() is called

STOMP messages include a variable number of headers much like HTTP messages. These may be used for filtering after the message has been received insuring that only messages of interest are processed. In addition, the connect request may carry a subscribe string (Much like a SQL statement) that will cause filtering to be done at the server, delivering only messages that satisfy the subscribe string on that connection.

Headers that may be used for filtering include the following:

protocol BML, or C2SIM)

submitter - Identifier used when the message was sent to the server

message-selector Indicates the type of message. Possible message selectors are:

- MSDL
- IBML09_GSR
- CWIX_PositionReport
- IBML09_Order
- CBML_Order
- C2SIM_Order

message-type – Indicates general type of message

- Order
- UNIT (Position or General Status Report)

message-dialect – Specific BML version used

- IBML09
- CBML
- C2SIM

The server can generate all three dialects. This can be controlled by server startup options.

C2SIM Message Parameters

- sender
- receiver
- conversationid

Other Server Functions

Translation - Messages can be translated between IBML09 and C2SIM. This includes orders and Position/General Status reports.

Initialization – Exercise initialization in the past has been done by MSDL. MSDL is in the process of being folded into C2SIM. Before the simulation exercise starts initialization will be performed by processing C2SIM_MilitaryOrganization transactions that define the characteristics and initial positions of Units. Multiple sets of these transactions may be submitted. If a C2SIM_MilitaryOrganization transaction is received for a Unit that was already defined the later one will be kept.

Upon receiving a SHARE command, the accumulated set of Unit definitions will be distributed to all participants and the simulation will be started. No additional C2SIM_MilitaryOrganization transactions will be accepted after the submission of a SHARE command which signals the start of the exercise. At this point the server is prepared to accept tasking and reporting messages. When clients receive a C2SIM_MilitaryOrganization it will contain all units that have been provided through initialization and indicate that the server is ready to start the exercise.

The set of accumulated Unit definitions will be treated as a database. Before starting the exercise, the set of Units may be saved to external storage and may be reloaded for a subsequent exercise without reprocessing the transactions used to build up the file.

Initialization Commands – A set of server commands is supported and used to manipulate the Unit database and implement minimal control of the exercise. These commands are listed in the C2SIM Server Initialization Commands table later in this document.

Unit Status Tracking – The Unit Database establishes the initial position and other properties of the Units in the simulation. After that, C2SIM makes no effort to maintain absolute position; only position in the last report from the simulation. Therefore, two queries have been implemented: *QueryUnit* will retrieve the last report received on this unit in the format specified. *QueryInit* will retrieve the initialization data distributed at the beginning of the exercise in C2SIM_MilitaryOrganization format.

C2SIM Message Envelope Support – The new C2SIM standard uses an XML message header separated from the actual data. This header contains a number of fields used to identify the sender, specific receivers, command indicating the type of message (Performative), unique identification for this specific message and for a series of messages known as a conversation.

The C2SIM Client Library does most of the processing of the C2SIM message header including header creation, stripping off the header before delivering the original message, sending a response where required and other functions.

Header Creation – When a BMLClientREST_Lib object is created with a parameter list that includes sender, receiver, and performative indicating that the transaction is to be encapsulated In a C2SIM message the Client Library creates the C2SIM header in preparation for sending the message. The C2SIM standard doesn't describe the format of the Sender and Receiver fields. A coalition planning on using C2SIM should establish a plan so that all Sender and Receiver entities have unique names. The messageID and convesationID are created as new UUIDs. These can be accessed and/or changed before the message is transmitted.

Header Access - A getC2SIMHeader() method against the BMLClientREST_Lib object will return a reference to the C2SIMHeader which may be queried and/or modified before the actual message is sent. On receipt of a message the BMLClientSTOMP_Lib class returns a BMLSTOMPMessage object. This object also supports a getC2SIMHeader() method which returns the C2SIM header on the receiver end.

C2SIM Client Library

The ClientLib is available in both Java and C++. The details of the Java C2SIM Client Library are contained in the JavaDoc file which accompanies this document.

Sending a message – Sample Code

```
import edu.gmu.c4i.c2simclientlib2.*;

String xmlMsg = "xxxxx";
String response = "";
BMLClientREST_lib c2s;

// Create new BMLClientREST_Lib object
c2s = new BMLClientREST_Lib();

// Set parameters
c2s.setHost("localhost");
c2s.setSubmitter("myID");

// Send the message
try {
    response = c2s.bmlRequest(xmlMsg);
    }
catch (BMLClientException e)
{System.out.println("BMLException: " + e.getMessage() + " Cause:"
    + e.getCauseMessage());
 return;
}

// Print the result
System.out.println(response);
```

Receiving a message – Sample Code

```
import edu.gmu.c4i.c2simclientlib2.*;

// Create the Client Object
BMLClientSTOMP_Lib c = new BMLClientSTOMP_Lib();

// Set parameters
c.setHost("localhost");
c.setDestination("/topic/BML");

BMLSTOMPMessage resp;

try {
   resp = c.connect();
}
catch (BMLClientException e)
{
   // Error during connect print message and return
   System.out.println("Error during connection to STOMP host"
       + c.getHost() + " " + e.getMessage() + " - " +  e.getCauseMessage());
   return;
}

// Start listening and loop forever
while (true) {
   try {
      resp = c.getNext_Block();
   }
   catch (BMLClientException e)
   {
      System.out.println("Exception while reading STOMP message "
         + e.getMessage() + " - " + e.getCauseMessage());
      return;
   }
   // Print received message
   System.out.println(resp.getMessageBody());
}
```

Sending a C2SIM Message requesting a response

```
import edu.gmu.c4i.bmlclientlib2.*;

BMLClientREST_Lib c2s;
String xml = "xml xml xml xml";
String convID = "";

// Instantiate BMLClientREST object for C2SIM message
c2s = new BMLClientREST_Lib("C2_Host","SIM_Host", "Request");

// Remember the conversationID for the C2SIM message we are sending
convID = c2s.getC2SIMHeader().getConversationID();

// Set parameters
c2s.setHost("localhost");
c2s.setSubmitter("C2Tester");
c2s.setPath("BMLServer/bml"

// Send the message
try {
   response = c2s.bmlRequest(xml);
}
catch (BMLClientException e) {
   System.out.println("BMLException: " + e.getMessage() + " Cause:"
       + e.getCauseMessage());
}

// Received Web Services response.  Print it
System.out.println("Response to WS request: " + response);

// Open up STOMP connection to receive response from C2SIM_SIM
BMLClientSTOMP_Lib c = new BMLClientSTOMP_Lib();

// Set parameters
c.setHost("localhost");
c.setDestination("/topic/BML");

// Add subscription to listen for same conversationID we just used to send
c.addAdvSubscription("conversationid = '" + convID + "'");

try {
   BMLStompMessage sm = c.connect();
   // Print response to connect
   System.out.println(sm.getMessageType().toString());

   // Get next message - Should be a response to the order sent via WS
   sm = c.getNext_Block();

   if (!sm.getC2SIMHeader().getPerformative().equals("Accept"))
      System.out.println("C2SIM Message not accepted");
}
catch (BMLClientException e) {
   System.out.println("Exception while communicating with STOMP server" + e);
}
Continue . . . .
```

Responding to a C2SIM message

```
import edu.gmu.c4i.bmlclientlib2.BMLClientException;
String conversationID = "";
String order = "";
C2SIMHeader c2s;

// Create the STOMP Client Object
BMLClientSTOMP_Lib c = new BMLClientSTOMP_Lib();

// Set host
c.setHost("localhost");

// Set the topic
c.setDestination("/topic/BML");

// Subscribe to get C2SIM messages
c.addAdvSubscription("protocol = 'C2SIM'");

// Connect to the STOMP server
try {
   System.out.println("Connecing to STOMP host");
   resp = c.connect();
}
catch (BMLClientException e) {
   System.out.println("Error during connection to STOMP server " +
       e.getMessage() + " - " + e.getCauseMessage());
   return;
}
// Start listening for an order
try {
   System.out.println("Waiting for order");
   resp = c.getNext_Block();
}
catch (BMLClientException e) {
   System.out.println("Exception while reading STOMP message "
      + e.getMessage() + " - " + e.getCauseMessage());
   return;
}

// Did we get a request?
if (resp.getC2SIMHeader().getPerformative().equals("Request")) {


// Get the xml order without the C2SIM Header
String order = resp.getMessageBody();

// Save the incoming C2SIM Header
C2s = resp.getC2SIMHeader();
```

```
// Send an Accept response
try {
   c.sendC2SIM_Response(resp, "Accept", "ACK");

   // Close STOMP circuit
   c.disconnect();
}
catch (BMLClientException e) {
   System.out.println("Exception while sending response to C2SI message"
   + e);
}
```

Note – These samples were taken from a pair of reference applications C2SIM_C2 and C2SIM_SIM. The source for these applications are available as separate files with this document.

C2SIM Client Utilities

Several standalone utilities are provided primarily as examples of how to program the C2SIM Client library. The use of these utilities is documented below. The "_ALL" suffix is in indication that all dependencies are included in the jar file and the jar file is executable as is. Note that the source code is also included in the jar file. The source code may be obtained by completely unzipping the jar file as follows:

```
jar -xvf BML_WSClient2-2.4_ALL.jar
```

BML_WSClient2-2.4_All – Submit an xml document to the C2SIM server via RESTful Web Services.

```
java -jar BML_WSClient2-2.4_ALL.jar hostname xml_file submitterID protocol
```

hostname	Name or IP address of the C2SIM Server
xml_file	File containing the xml data to be submitted
submitterID	Name or initials identifying the submitter.
protocol	BML or C2SIM

If the protocol is C2SIM, a C2SIM header will be generated using "ALL" for sender and receiver and "Inform" for the C2SIM performative.

BML_StompClient2-2.4_ALL – Connect to a STOMP server, receive all published messages and print them via System.out.println()

```
java -jar BML_StompClient2-2.4_ALL hostname
```

hostname	Name or IP address of the STOMP server.

C2SIM_Unit_Initialization-2.4_ALL – Submit initialization commands via a command line interface. The currently supported commands are given later in this document.

```
java -jar C2SIM_Initialization-2.4_ALL.jar hostname command parm1 parm2
```

hostname	Name or IP address of the C2SIM Server (RESTful)
command	Command in accordance with the C2SIM Server Initialization Commands table
parm1	First parameter (See table)
parm2	Second parameter (See table)

C2SIM Server Initialization Commands

Command	Parm1	Parm2	Required Simulation State	Actions
NEW	filename		UNINITIALIZED	Create new empty local unit database. Save name for use if database is saved to permanent storage.
LOAD	filename		UNINITIALIZED	Load contents of named file and make it the active Unit database.
SAVE			UNINITIALIZED	Save existing database to permanent storage. If NEW had not been executed the name “defaultDB” will be used.
SAVEAS	fileName		UNINITIALIZED	Save existing database to permanent storage as fileName
DELETE	fileName		UNINITIALIZED	Delete named file from permanent storage
SHARE			UNINITIALIZED	Publish existing database Format is C2SIM_MilitaryOrganization Set simulation state to “INITIALIZED" Save Unit DB for late joiners
RESET			Either	Reset database and state back to unitialized. This command requires a password to be sent as parm1.
QUERY_UNIT	unitName	Format “C2SIM” or “IBML”	INITIALIZED	Return latest Position Report in format named in Parm2
QUERY_UNIT	“ALL”	Format “C2SIM” or “IBML”	INITIALIZED	Return latest Position Reports for each Unit giving current positions in format named in Parm2
QUERY_INIT			INITIALIZED	Return all UNITs as originally specified for initialization in C2SIM_MilitaryOrganization format
RESTful POST of C2SIM_MilitaryOrganization Document			UNINITIALIZED	Save in local UNIT database replacing any UNITs with same name.
RESTful POST of MSDL Document			UNINITIALIZED	Extract UNIT information Save in local unit database replacing any UNIT objects with same name
RESTful POST of PositionReport (IBML or C2SIM)			INITIALIZED	Update unit information from report

RESTful POST of documents is done via the bmlRequest method in the BMLClientREST_Lib class in the C2SIM Client Library.
Other commands are submitted via the c2simCommand method in same class.