C2SIM Server Reference Implementation Documentation– Version 4.7.7.X

User Instructions

Overview

GMU C4I and Cyber center is making available C2SIM Server release 4.7.7.x and C2SIM Client release 4.7.7.x as the current version of the C2SIM Reference Implementation. Note that the Server and Client release numbers are now the same, though the last digit may not match if changes to one do not require changes to the other. These packages support the following:

- C2SIM orders and position reports for C2SIM Version 11 the version that was submitted to SISO for standardization.
- Legacy protocols IBML09, CBML-Light and an early version of C2SIM used at CWIX2018 for orders and position reports.
- Translation among all four dialects for orders and reports
- Processing and marshalling of C2SIM initialization messages.
- Translation between C2SIM initialization messages and MSDL
- Operator commands and session state messages.
- Simulated Cyber attacks
- Collection of response time statistics by the server
- Verification that the version numbers of the client meets the minimum required for a particular version of the server.

The C2SIM formats used by the current version of the server are in compliance with C2SIM draft Version 9 with some extensions made for experiments during June and July 2019 by NATO MSG-145.

Server Configuration

The server(s) usually 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 Web Services)
- 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. Also that the same client may submit REST messages and receive STOMP messages.

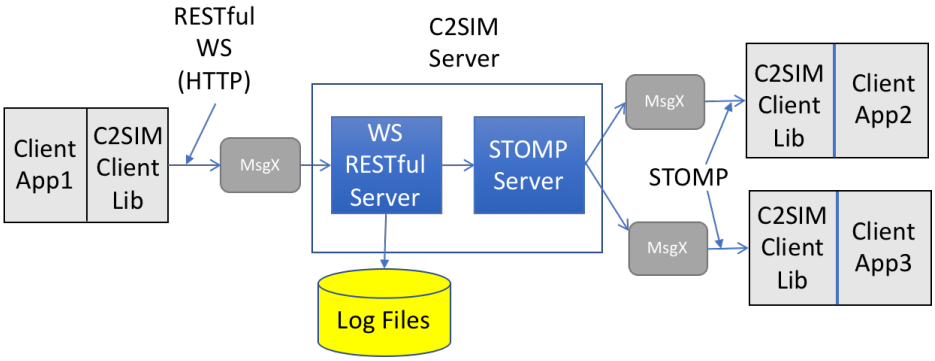


Figure 1: Server Operation

Basic Message Processing

Sending – Messages are sent to the server using RESTful Web Services protocols via the C2SIMClientREST_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 BML/C2SIM version (also known as dialect) used. Messages are translated to the other three dialects, if that feature is turned on in the server. The message is then sent to the STOMP server where it is published to all subscribers. The server adds a number of 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 C2SIMClientREST_Lib object
- Setting parameters
- Sending the request
- Receiving the result
- Destroying the C2SIMClientREST_Lib object

Under control of a switch in the C2SIMServer.properties file, the server may request that the response time for the transaction be sent by the client to the server as a separate transaction for use in testing server performance. This is completely handled by the C2SIM Server and the C2SIMClientREST_Lib class of the C2SIM client library; the client application is not required to participate. The server records the response time in the C2SIM Debug log.

When a message is sent to the server, a message number is generated. This message number is returned in the response, echoed back by the client library with response time statistics, and added to the STOMP message as one of the headers. This enables tracking of all messages from end to end.

Receiving – Messages sent to the C2SIM server are sent to all systems that have subscribed to the STOMP server. Message receipt is via the C2SIMClientSTOMP_Lib class of the C2SIM_Client Library. Most client systems are both senders and receivers. Receiving systems establish a STOMP subscription, using TCP, via the C2SIM_Client library. The TCP connection is kept open as long as the client holds a STOMP connection; 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 STOMPClientSTOMP_lib object
- Set parameters (Topic should normally be /topic/STINO)
- Specifying optional subscription(s)
  Example: c. addAdvSubscription(“ protocol = ‘C2SIM’ ”);
- Connecting by executing the connect() method of C2SIMClientSTOMP_lib.
- Loop and receive messages c.getNextBlock() or c.getNextNoBlock() indefinitely
- Connection stays open until c.disconnect() is called

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

Headers (underlined) 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_Report
CWIX_Report
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
CWIX (CWIX2018 version of C2SIM)

The server can translate among all four dialects. Which translations are performed is controlled by server startup options.

C2SIM Message Parameters

sender
receiver
communicativeActTypeCode
conversationid

Other Server Functions

Translation - Messages can be translated among four protocols:

- C2SIMv9
- CWIX version of C2SIM
- IBML09
- CBML-Light

This includes orders and Position/General Status reports.

Initialization data can be translated between:

- C2SIMv9
- MSDL

Initialization – Exercise initialization in the past has been done using MSDL. C2SIM is intended to combine the functions of CBML and MSDL.. Before the simulation exercise starts initialization will be performed by processing C2SIM ObjectInitialization transactions that define the characteristics and initial positions of Units and other items. Multiple sets of these transactions may be submitted by different sources. An alternative method for entering initialization information is to position a file on the server containing the same information as is submitted in ObjectInitialization transactions and then executing a LOAD command.

When the server receives a SHARE command, the accumulated set of definitions in C2SIMInitializationBody format will be published to all participants and the simulation will be started. Additionally, the accumulated C2SIM initialization information will be translated to MSDL and published. No additional initialization transactions will be accepted after the execution of a SHARE command. A START command starts the scenario and at this point the server is prepared to accept orders and reports. When clients receive a C2SIMInitializationBody message it will contain all units and other items that have been provided through initialization.

Unit Status Tracking – The initialization database establishes the initial position and other properties of the units in the simulation. After that, the server will maintain the last position received via position report for each unit. A query has been implemented to access this data. Q*UERYINIT* will return the initialization data distributed at the beginning of the exercise, in C2SIMInitializationBody format with updated positions. It also will return the same data in MSDL format. This response can support join of clients after an exercise has been started by SHARE and RUN(late join).

C2SIM Message Envelope Support – The new C2SIM standard specifies an XML message header separate from the actual data. This header contains a number of fields used to identify the sender, specific receivers, command indicating the type of message (CommunicativeActTypeCode), unique identification for this specific message (MessageID) and for a series of messages (ConversationID).

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, and sending a response where required and other functions. See the C2SIM_ClientLib Javadoc file for more information.

Header Creation – When a C2SIMClientREST_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 conversationID are created as new UUIDs. These can be accessed and/or changed before the message is transmitted.

Header Access - A getC2SIMHeader() method on the C2SIMClientREST_Lib object will return a reference to the C2SIMHeader, which then may be queried and/or modified before the actual message is sent. On receipt of a message the C2SIMClientSTOMP_Lib class returns a C2SIMSTOMPMessage object. This object also supports a getC2SIMHeader() method which returns the C2SIM header of the message that was just received.

Interfacing without the use of the Client Library

There are two client libraries provided by this project, Java and C++. The Java library consists of a single jar file and was built on Java 1.8.0_65 and NetBeans 8.2. The C2SIM Server components do conform to industry standards and may be used without the supplied client libraries although **this is not recommended**. The library routines are straightforward to use, are provided with source code, and are well documented.

The C2SIM server is implemented using REST procedures. The following URLs are used to access the server:

URL: http://hostname:8080/C2SIMServer/c2sim Submission of BML and C2SIM documents

REST Parameters:

protocol	C2SIM or BML
submitterID	Identification of the submitter
sender	Identifier of sending process (C2SIM Only)
receiver	Identifier of intended recipient (C2SIM Only)
communicativeActTypeCode	Type of communication (C2SIM Only)
conversationID	Identity of a series of messages
version	Version of client software – Currently 4.6.3

URL: http://hostname:8080/C2SIMServer/command Submission of session commands

REST Parameters:

command	Command as described later in this document
parm1, parm2	Parameters used with above command
submitter	Identification of the submitter
version	Version of client software – Currently 4.6.3

The STOMP server is an off-the-shelf copy of Apache Apollo 1.7.1 and implements standard STOMP version 1.2.

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. The C++ version provides equivalent implementations of all Java methods. Appendix A below gives sample code using the Java ClientLib.

C2SIM Client Utilities

In the following, xxx indicates the version number of the program being described.

Several standalone utilities are provided primarily as examples of how to program the C2SIM Client library. Most of their functions also are available under a user-friendly interface in the open source BMLC2GUI, available at http://c4i.gmu.edu/OpenBML.

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  C2SIM_WSClient2-xxx_ALL.jar
```

C2SIM_WSClient2-xxx_All – Submit an xml document to the C2SIM server via RESTful Web Services.

```
java -jar C2SIM_WSClient2-xxx_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 or Cyber

If the protocol is C2SIM, a C2SIM header will be generated using "ALL" for sender and receiver and "Inform" for the C2SIM performative. An experimental module has been added to emulate Cyber attacks has been added to the server and to the WS Client. This capability attacks incoming messages according to parameters in a control file. This control file is submitted to the server using the WSClient specifying "Cyber" as the protocol. The full capability is described in a separate document.

C2SIM_StompClient2-xxx_ALL – Connect to a STOMP server, receive all published messages and print them via System.out.println()

```
java -jar C2SIM_StompClient2-xxx_ALL hostname
```

hostname Name or IP address of the STOMP server.

C2SIM_Replay Client

The C2SIM server creates a log file containing all incoming and outgoing messages. Each log file contains all messages for a particular day. The replay log files are named as "yyyy-mm-dd.replay.log". The log file for the current day is simply called replay.log. These messages may be "replayed", resubmitting them to the server. The rate of submission may be on the same timescale as the original messages were received or at an accelerated rate.

The Replay Client is executed as follows:

java -jar C2SIM_Replay-xxx_ALL.jar hostname logFileName [source] [protocol] [submitter] [timescale]

hostname – Name of host to submit transactions to
logFileName – Name of file containing logged messages
source – Source of the log entry. Possible values are:
Client – Messages was received directly from client
Command – Command received from client
Generated – Messages was generated by the server (Used during initialization)
Translated – Message was translated from a different BML/C2SIM dialect
protocol – Base protocol of the message (C2SIM or BML)
submitter – Identification of entity submitting the transaction

timescale – Time difference between two messages is first computer then divided by the value of timescale. The default is 1 which causes no scaling of time. Larger values can be used to speed up the playback. If timescale = 0 then messages will be played back as quickly as possible, i.e. no delay between messages.

C2SIM_StompCollector

This client utility is used to collect messages as received from the STOMP server. The messages are recorded along with the time of receipt and in a form such that the received massages can be replayed through the C2SIM_Replay client. A number of filtering options are provided to control the collection process.

Execution:

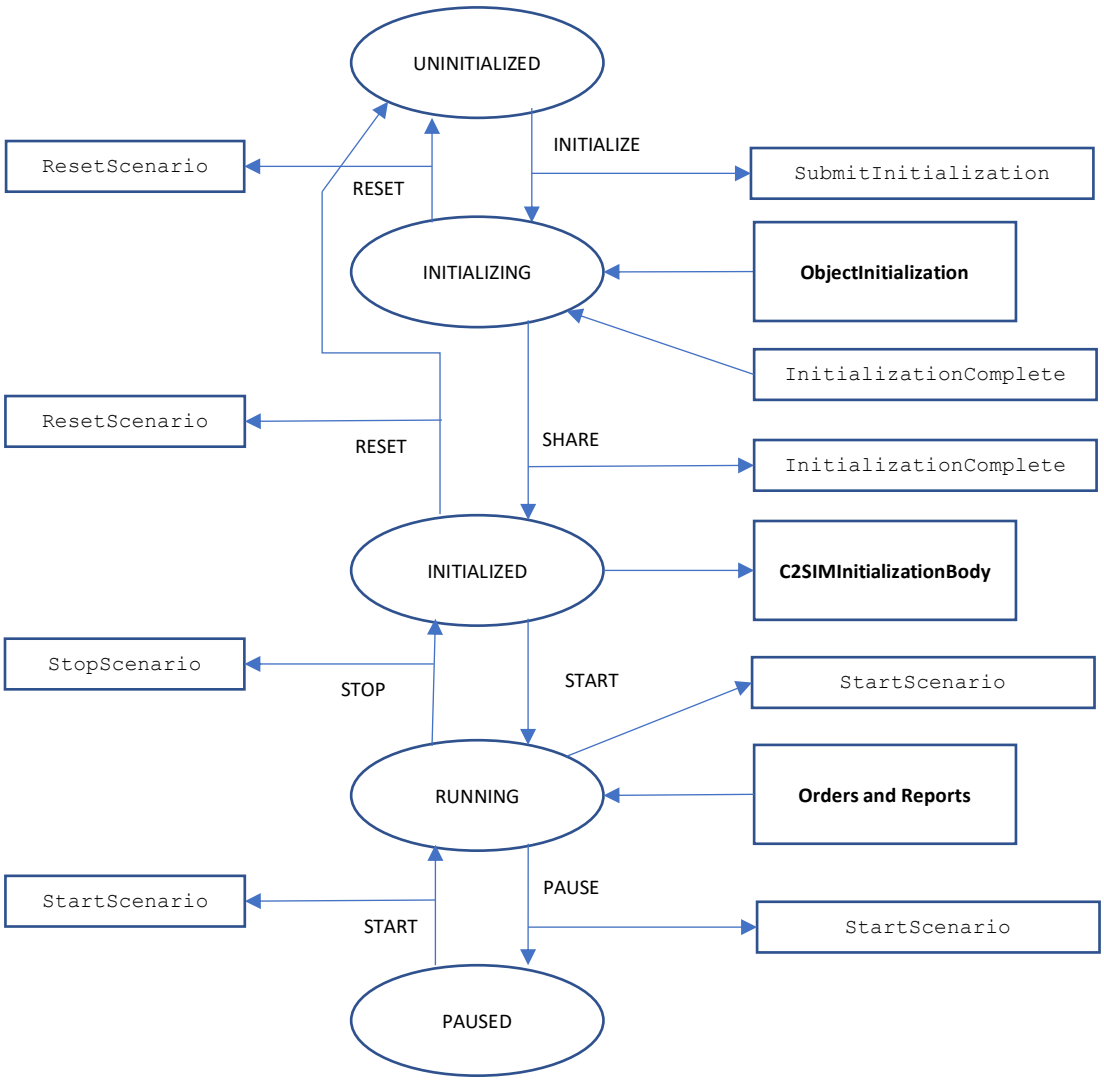
java -jar C2SIM_StompCollector-xxx.jar hostname captureFile [protocol] [submitter]
hostname – Name of host to collect from
captureFile – File to record captured messages (may be used as input to the Replay Client)
protocol – Underlying protocol of the messages to be collected (C2SIM or BML)
submitter – Entity that submitted the original message

C2SIM_Command-xxx_ALL – Submit initialization commands via a command line interface. The currently supported commands are given later in this document.

java -jar C2SIM_Command-xxx_ALL.jar hostname command parm1 parm2

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

C2SIM Commands – A number of commands are used to submit initialization data, manipulate the database of Units and to control the simulation. The diagram below shows the server states (Ovals), Transitions (Arrows) and allowed C2SIM/BML Transactions (Horizontal arrows). Commands are in all caps, C2SIM XML initialization messages in boxes in **bold** and System Command messages showing status and sent to all systems are in boxes (non bold)



C2SIM Server Commands

Command	Parm1	Parm2	Required Simulation State	Actions
Unit Database Manipulation Commands				
LOAD	filename		UNINITIALIZED	Load contents of named file process the initialization data found there as if received over network..
Simulation State Commands Initial state when server starts is UNINITIALIZED				
RESET	Password		INITIALIZING	Reset database and state back to "UNINITIALIZED". Delete initialization data. Reset emulated Cyber attacks
SHARE	Password		INITIALIZING	Publish existing database Terminate initialization phase Format is C2SIM_MilitaryOrganization Set simulation state to "INITIALIZED" Save Unit DB for late joiners
START	Password		INITIALIZED Or PAUSED	Start simulation Set simulation state to "RUNNING"
STOP	Password		RUNNING	Stop simulation Don't delete initialization data. Set simulation state to "INITIALIZED"
PAUSE	Password		RUNNING	Pause the simulation. Set simulation state to "PAUSED"
STATUS			ANY	Return state of server using the same format returned to submitters of XML documents.

Command	Parm1	Parm2	Required Simulation State	Actions
Initialization Information Query Commands				
QUERYINIT			INITIALIZED or RUNNING	Return all initialization data as originally specified for initialization in C2SIMInitializationBody format. Also translate to MSDL and return that as well.
RESTful POST of C2SIM_ObjectInitialization Document			INITIALIZING	Save in local initialization database.
RESTful POST of MSDL Document			N/A	Translate to C2SIM and store in initialization database
RESTful POST of Position or General Status Report			RUNNING	Update unit position from report

RESTful POST of documents is done via the c2simRequest method in the C2SIMClientREST_Lib class in the C2SIM Client Library.

Commands are submitted via the c2simCommand method in the same class.

There are also command line utilities for submission of C2SIM/BML documents (C2SIM_WSClient2_xxx_ALL.jar) and for commands (C2SIM_Command-xxx_ALL.jar) described earlier in this document.

Accessing Log Files

The server collects information critical for problem analysis in two files, replayLog and debugLog. The replay log contains a record for each message received, each server command received, and each message published via STOMP. A critical piece of data is the message number which is assigned as the message is received, appears in most debug log messages and appears in the response returned to the client.

The debug log contains miscellaneous messages showing the progress of each message through the server and errors detected either via checks in the server code or as the result of an exception that might be thrown.

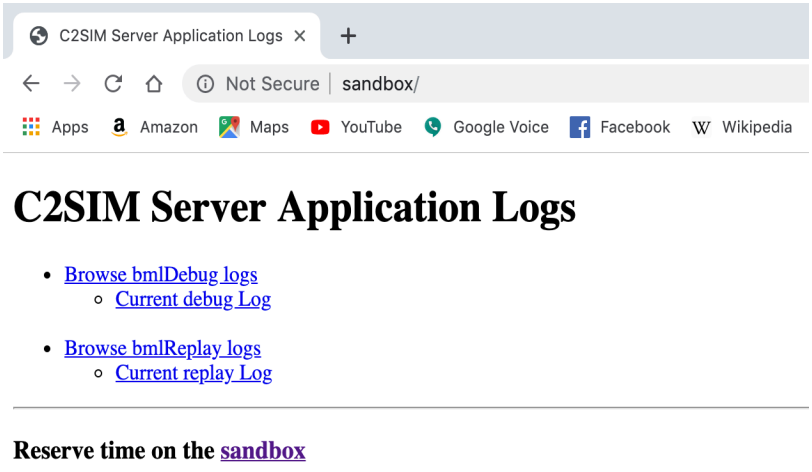
Both log files are collected during the day and then rolled over at midnight (local to the server). The names and locations for the two log files are for the current day:

/home/bmluser/c2simFiles/c2simDebug/debug.log

/home/bmluser/c2simFiles/c2simReplay/replay.log

Log files from previous days are located with the daily log and are named yyyy-mm-dd.debug.log and yyyy-mm-dd.replay.log.

The log files can be accessed by accessing port 80 of the C2SIM Server platform using a web browser (port 80).



Server initialization and control

Server initialization and operation is controlled with a (java) properties file. This file contains a number settings that may be used to select a number of options. The file named c2simServer.properties is located at:

/home/bmluser/NetBeansProjects/Server/C2SIMServer/src/main/resources.

During the build process the properties file is added to the web archive file, C2SIMServer.war, used to deploy the server application to Tomcat. To deploy a war file it is copied to /opt/tomcat/apache-tomcat-8.0.30/webapps. When Tomcat starts and detects a new war file it unpacks in the webapps folder. After Tomcat initialization the webapps folder will contain a folder named, C2SIMServer##a.b.c.d and the original war file C2SIMServer##a.b.c.d.war. The a.b.c.d is the release number of the server application. After unpacking the properties file will be located at

/opt/tomcat/apache-tomcat-8.0.30/webapps/C2SIMServer##a.b.c.d/WEB-INF/classes

To change one or more properties in the properties, file modify it with a text editor and restart tomcat by executing stop-all and then start-all. These script files are in the bmluser home directory.

The current properties file is displayed in Appendix B below. Note that the file contains initialization information for starting the server, location of various files on the server, the password required for command submission, switches that control what BML/C2SIM dialects are to be translated and other miscellaneous values.

Note that there is a property referencing a “Cyber Attack”. This is a capability that simulates an attack on incoming messages and is controlled by a separate configuration file. A separate document will be published describing this capability.

```
# Implement simulated cyber attack
server.cyberAttack = T
```

Capturing Response Time Statistics

The server.collectResponsTime property indicates that server response time statistics are to be captures. The C2SIM REST client measures the response time of each transaction as it is submitted to the server. If the response from the server, e.g.

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
      <status>OK</status>
```

```
      <message>Message processed successfully</message>
      <serverInitialized>true</serverInitialized>
      <serverVersion>4.6.3.6</serverVersion>
      <sessionState>INITIALIZING</sessionState>
      <unitDatabaseName>default</unitDatabaseName>
      <unitDatabaseSize>2</unitDatabaseSize>
      <msgNumber>2</msgNumber>
      <time> 0.033</time>
      <collectResponseTime>T</collectResponseTime>
</result>
```

Contains <collectResponseTime>T</collectReonseTime> the client code will submit an additional message to the C2SIMServer/Stats URL containing the measured response time from the previous transaction. An example is shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<C2SIM_Statistics xmlns="http://www.sisostds.org/schemas/c2sim/1.0">
    <REST_ResponseTime>
        <submitterID>dsc</submitterID>
        <msgNumber>3</msgNumber>
        <startTime>2019-01-29 18:11:51,212</startTime>
        <endTime>2019-01-29 18:11:55,968</endTime>
        <elapsedTime>4.756</elapsedTime>
        <serverTime> 0.010</serverTime>
    </REST_ResponseTime>
</C2SIM_Statistics>
```

The server writes each response time message preceeded with the word “Stats” to the debug log located at /home/bmluser/c2simFiles/c2simDebug. These messages can be extracted from the debug log for a particular date and analyzed.

Appendix A: Sample Code Using Java ClientLib

Sending a message

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

String xmlMsg = "xxxxx";
String response = "";
C2SIMClientREST_lib c2s;

// Create new C2SIMClientREST_Lib object
c2s = new C2SIMClientREST_Lib();

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

// Send the message
try {
      response = c2s.c2simRequest(xmlMsg);
    }
catch (C2SIMClientException e)
      {System.out.println("C2SIMException: " + e.getMessage() + " Cause:"
    + e.getCauseMessage());
 return;
}

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

Receiving a message

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

// Create the Client Object
C2SIMClientSTOMP_Lib c = new C2SIMClientSTOMP_Lib();

// Set parameters
c.setHost("localhost");
c.setDestination("/topic/C2SIM");

C2SIMSTOMPMessage resp;

try {
   resp = c.connect();
}
catch (C2SIMClientException e)
{
   // Error during connect print message and return
   System.out.println("Error during connection to STOMP host"
       + c.getHost() + " " + e.getMessage() + " - " +  e.getCauseMessage());
   return;
}
System.out.println(resp.getMessageType();

// Start listening and loop forever
while (true) {
   try {
      resp = c.getNext_Block();
   }
   catch (C2SIMClientException 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.c2simclientlib2.*;

C2SIMClientREST_Lib c2s;
String xml = "xml xml xml xml";
String convID = "";

// Instantiate C2SIMClientREST object for C2SIM message
c2s = new C2SIMClientREST_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("C2SIMServer/c2sim"

// Send the message
try {
   response = c2s.c2simRequest(xml);
}
catch (C2SIMClientException e) {
   System.out.println("C2SIMException: " + 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
C2SIMClientSTOMP_Lib c = new C2SIMClientSTOMP_Lib();

// Set parameters
c.setHost("localhost");
c.setDestination("/topic/C2SIM");

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

try {
   C2SIMStompMessage 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 (C2SIMClientException e) {
   System.out.println("Exception while communicating with STOMP server" + e);
}
Continue . . . .
```

Responding to a C2SIM message

```
import edu.gmu.c4i.c2simclientlib2.C2SIMClientException;
String conversationID = "";
String order = "";
C2SIMHeader c2s;

// Create the STOMP Client Object
C2SIMClientSTOMP_Lib c = new C2SIMClientSTOMP_Lib();

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

// Set the topic
c.setDestination("/topic/C2SIM");

// 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 (C2SIMClientException 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 (C2SIMClientException 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 (C2SIMClientException 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 code for these applications is available as separate files.

Appendix B: C2SIMServer.properties file

```
# STOMP Interface setup (Apache Apollo Server)
stomp.serverHost=localhost
stomp.port=61613
stomp.topicName=/topic/C2SIM

#Location of C2SIM Files ($C2SIM_HOME)
server.c2simFiles = /home/bmluser/c2simFiles

# Location of C2SIM initialization file (ObjectInitialization)
relative to $C2SIM_HOME when using LOAD command
server.initDB = /InitializationFiles/

# Name of Schema Database
server.schema_db_name=C2SIMSchemaDB

# Password used for submission of commands controlling
initialization and server state
server.c2sim_password = v0lgenau

# Just publish the document without any other processing
server.justDocumentMode = F

# Just parse the document and publish it - This will catch
structural xml errors
server.justParseDocument = F

# Just determine what kind of message, e.g. IBMLReport and
publish it  This will be included in the STOMP header when the
message is published
server.justIdentifyMessage = F

# Which messages are to be translated to
server.TranslateC2SIM_Order = T
server.TranslateC2SIM_Report = T
server.TranslateIBML09_Order = T
server.TranslateIBML09_Report = T
server.TranslateCBML_Order = T
server.TranslateCBML_Report = T
server.TranslateCWIX_Order = T
server.TranslateCWIX_Report = T

# Is C2SIM to be translated to MSDL?
server.TranslateMSDL = T

# Capture and update Unit position from position reports
```

```
server.CaptureUnitPosition = T

# Implement simulated cyber attack
server.cyberAttack = T

# Request response time statistics from REST client
server.collectResponseTime = T

# Minimum client version to be accepted by this server version
server.enforceVersion = T
server.minimumClientVersion = 4.6.2
```