Ver. 20060815

Reference Server Documentation

Kyle Reed
Mark White

8/15/2006

CS390 Java Networking Research
Dr. James
Saginaw Valley State University

For project information:
http://simsa.sourceforge.net

The SIMSA (Single Interface for Multiple Services Architecture) began as a student research project. The goal of the project was to create a single threaded, server that handles multiple, stated, concurrent socket connections and also allows services to be started and stopped at runtime. The long term goal of the project as a whole is to be adopted by our university's Computer Science program as a teaching tool for network and architecture classes. Since development began, it became obvious that this project has a lot of applications in subjects like service-oriented problems, distributed computing, message brokering (which is really the whole point of the server).

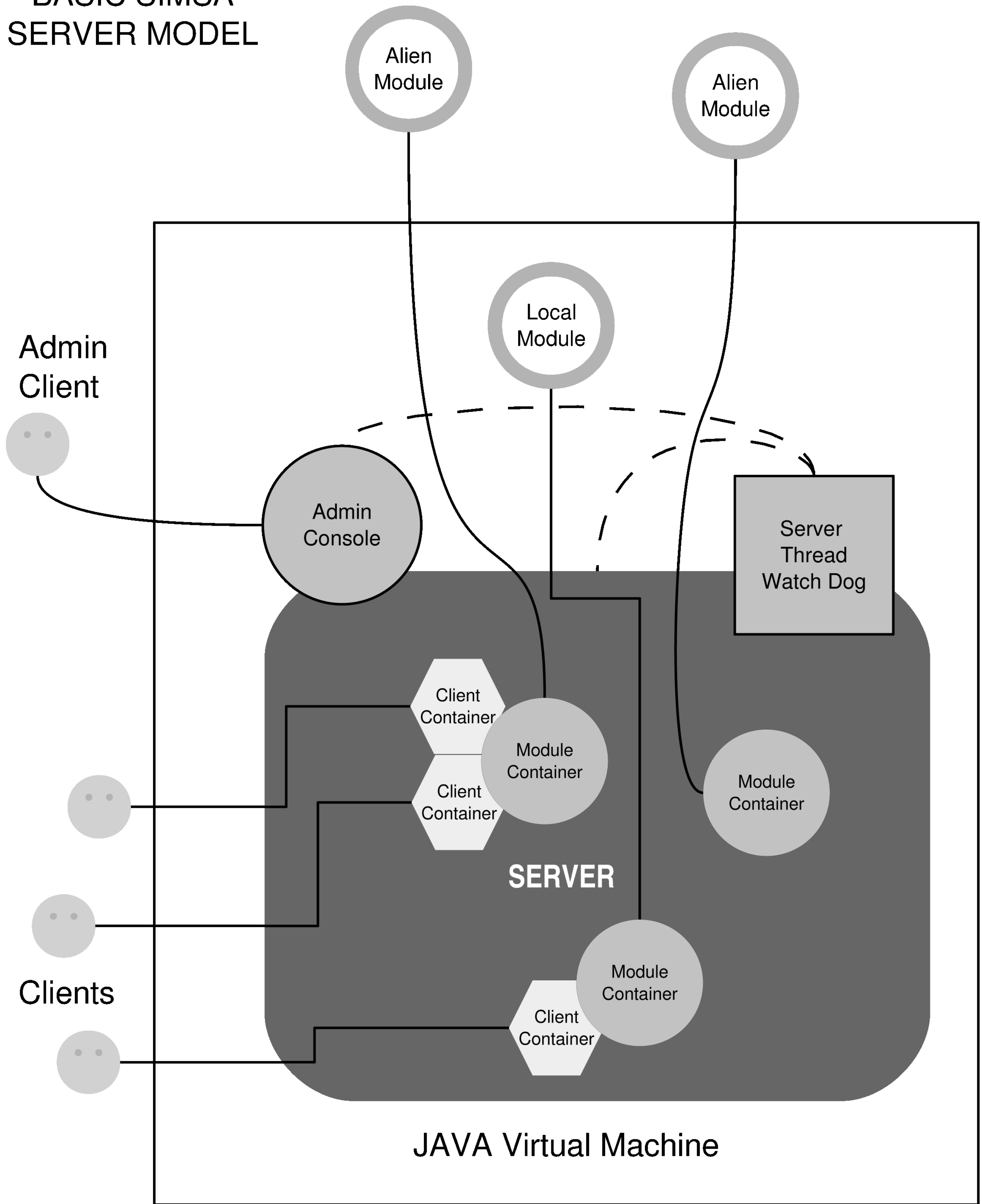
Before looking at the technical details, it's important to understand the terminology in the SIMSA project.
CLIENT: any program that connects to the server via a socket connection to port 12345 and speaks the SIMSA-client XML commands.
MODULE: any program that connects to the server via socket connection to port 12346 and speaks the SIMSA-modules XML commands. Modules that are written in the same language as the server should be able to be dynamically loaded.
ALIEN MODULE: a module that connects to the server without being explicitly instantiated by the server. In our Java server, any module that is loaded outside the VM that the server is running in is an alien. Alien modules can be running on remote hosts.
SERVER: a program that provides two socket connections for modules and clients to connect on. It handles the registration, message brokering and other functions.

SIMSA is an architecture and it defines a communication interface/protocol rather than an actual program. It defines the language and program requirements that define how clients and modules must behave in order to be used with a single interface that connects to multiple services. The API reference is under development and we have a reference implementation of the server written in Java. There are also reference implementations of modules and clients written in various languages proving that the interaction is completely language independent.

The reference server is written in Java because Java has the unique ability to load byte code dynamically without a lot of pain. Java is also a natural choice for platform portability. The final reason we chose Java is the new java.nio package that provides ByteBuffer based SocketChannel and the Selector object that allows multiple concurrent socket connections on the same thread. This project requires Java 1.5 or greater to run. The server is meant to be run as a daemon and provides another socket connection (port 12344) for an administration console.

Communication with the server uses very simple XML and allows for service lookup, name resolution, connection and disconnection. Modules and clients may define their own tags as well as long as they aren't the same as the server tags. Conceptually it is useful to think of the server as a post office. Incoming messages are contained in an XML "envelope", that envelope is read and removed. The message is then passed to the final recipient who actually reads the message and processes accordingly.

Some possible projects include:
1) Game back ends(Module)/networking layer(Server)
2) Chat servers
3) Distributed computing(modules that bridge multiple servers)
4) MIME encoding could provide file transfer
5) Service Brokering
6) etc...


BASIC SIMSA
SERVER MODEL
Alien Module
Alien Module
Local Module
Admin Client
Admin Console
Server Thread Watch Dog
Client Container
Client Container
Module Container
Module Container
SERVER
Module Container
Client Container
Clients
JAVA Virtual Machine

QUICK START

Section 1 - Project Directory

In order to get started, you'll need to download the latest "dist" package from the source tree at http://simsa.sourcefourge.net. We suggest that you create a folder in your root directory called "SIMSA_ROOT" and unzip the downloaded "dist" file into this directory. In the "dist" directory there are a few directories that you need to know about:

- MOD – contains the SIMSA-Modules.jar
- LIB – contains the SIMSA-Lib.jar
- CLIENTS – contains various compiled clients

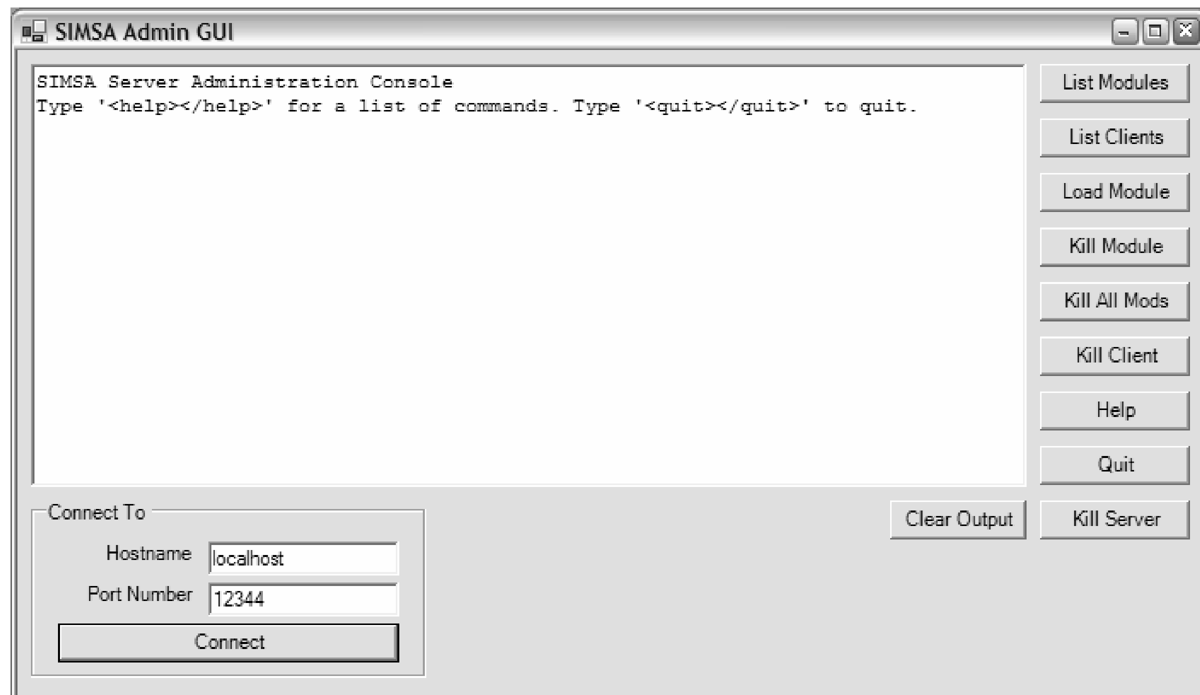
If your directory has these folders, you're ready to move on to the next step.

Section 2 – Starting the Server

To start the server, simply double click on "SIMSA-Server.jar" (provided that you have JRE 1.5 or greater configured on your machine). This will launch a daemon server that is ready to accept new connections.

Section 3 – Connecting to the Admin Console

The server doesn't really do a whole lot until we can see what it's doing and even better, load a back end service (a.k.a. module). To do this, run "AdminGUI" from the "clients" folder. It should be ready to go so just click "Connect". If all is well you should see a screen similar to the following:



This means that the server is up and running so we are ready for the next step.

Section 4 – Loading a Module

Now we need to start a module to make something interesting happen. To load a module, simply click on “Load Module”. An input prompt will ask you for the name of the module you would like to load. In our case we are going to load a chat module so enter “Modules.SmartChat” into the input box. You’ll notice that there’s a name qualifier because we are running the module from the “SIMSA-Modules.jar”. If the module is loaded successfully you should see the message:

“Module loaded: Modules.SmartChat”

Now that we have a module loaded, it’s time to connect with a client.

Section 5 – Connecting with a Raw Client

Now run telnet or some other raw client and connect to “localhost 12345”. Now you’re connected to the server and you’re ready to send commands. We are going to connect to the chat module so type:

“<connect><name>TEST</name><mid>1</mid></connect>”

We’ve only loaded one module so its ID is 1 which is what we used for “<mid>”. If there were other modules loaded, you should make a “<whereis>” query which is explained in the message specification at the end of this document. We also connected using the name “TEST” – our unique name in the server. You should receive a “<conack>” message acknowledging your connection. Now that you’re connected, you can send a message to the chat module by typing the following:

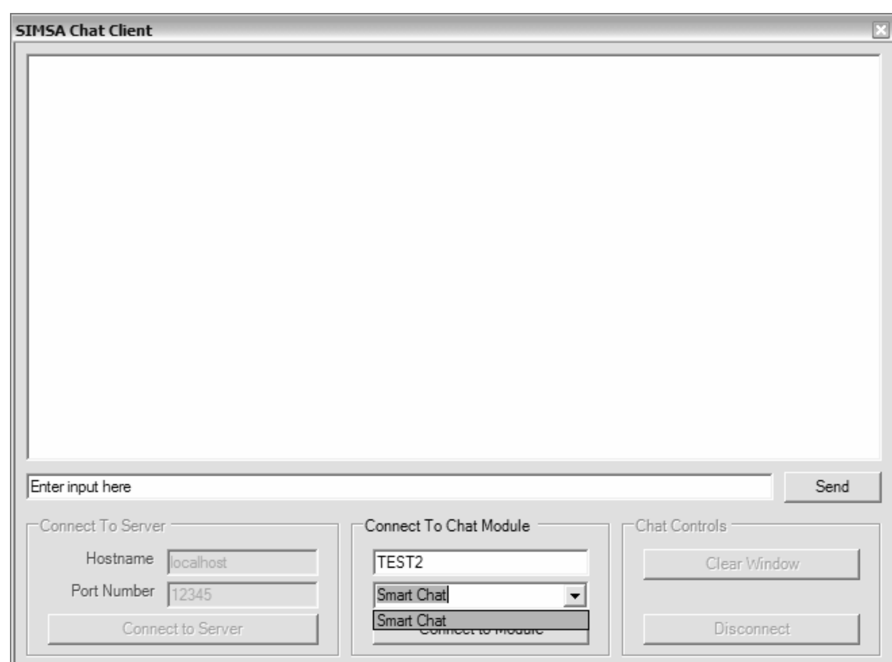
“<_msg>Hello World</_msg>”

What’s that? You didn’t see anything? The chat module doesn’t echo your input so you’re going to need to load another chat client to see anything worth while. *(Leave the telnet client open)*

Section 6 – Connecting with VBChat Client

In the “clients” folder, there is a program called “VBChat”. This client manages all of the lookup and connection overhead (like the connect string above) for you so all you need to do is press connect, select a chat module to connect to and you’re ready to chat.

As you can see, we are connecting with the name TEST2 and have selected a chat module to connect to from the drop down box. Next press the “Connect to Module” button and you’re ready to chat. For testing we’ve entered “Wow, SIMSA is cool” into the VBChat client and sent it. You should now see a message in your telnet window. Notice that the message contains an “ID” and a “SAID” tag. These are the module defined tags common to all chat modules.



```
Telnet localhost
                                                    <conack><id>4</id><key>294</key>
</conack><n
<_msg>Hello World</_msg><_msg><id>3</id><said>TEST2: Wow, SIMSA is cool</said></
_msg>_
```

Now these two clients can chat back and forth. Remember that the outgoing messages from the telnet client MUST be wrapped in “<_msg>” and “</_msg>”.

Section 7 – When you’re done playing

The server can load other modules and handle more clients from here. It’s just a matter of exploring the other clients and modules included in the reference implementation. One fun module to check out is “Modules.Spong” with the related “SPong.exe” client.

When you’re done playing, you can either close the clients individually and kill the module that was loaded. This will allow the server to continue running leaving it available to load other modules and be connected to by more clients. If you’re totally finished with the server, you must kill it with the AdminGUI. Simply click the “Kill Server” button on the bottom left and it will tell the server to begin its shutdown routine. The server will kill all of its loaded modules which in turn will close all of their clients. Finally it closes the connection to the Admin Console and the AdminGUI will say “Connection Closed, please reconnect...” indicating that the server has been shutdown.

Section 8 – For more information

The best way to learn about the modules that are available is to look at the source for the SIMSA-Modules. For more information on the rest of the project continue reading about the SIMSA Message Specification which is the protocol used to speak to the SIMSA server. Now you’re ready to start using SIMSA.

CHECK OUT THE PROJECT WEBSITE @ http://simsa.sourceforge.net

SIMSA MESSAGE SPECIFICATION

These are used by all connections via port 12345:

<WHOIS>ID[,ID]|0</WHOIS>

Returns Name of numeric ID or a list of IDs, or a list of everyone when the ID requested is '0'.

<WHO>
<NAME>Kyle</NAME><ID>2</ID>
</WHO>
<WHO>
<NAME>Mark</NAME><ID>1</ID>
</WHO>

Disconnects.

<COMMAND>QUIT</COMMAND>

<WHEREIS>module-type[,module-type]|ALL</WHEREIS>

Returns Name, MID (Module ID), and type for modules with matching module-type, or all modules then the module type is 'ALL'.

<WHERE>
<NAME>Chat</NAME><MID>11</MID><MTYPE>CHAT</MTYPE>
</WHERE>
<WHERE>
<NAME>EchoUpper</NAME><MID>12</MID><MTYPE>ECHO</MTYPE>
</WHERE>

Disconnects.

<COMMAND>QUIT</COMMAND>

These are used by CLIENT connections via port 12345:

<CONNECT>
<NAME>name</NAME> <MID>ModuleID</MID> [<KEY>key</KEY>]
</CONNECT>

Returns ID, and numeric challenge-key on success. All further messages are sent to the connected module. (If multiple connections on a single NAME are used, the key is required)

<CONACK>
<ID>3</ID><KEY>10193</KEY>
</CONACK>

Connects to module.
OR
Returns reason for no connect.

<ERROR><NAME>Mike</NAME><MID>14</MID></ERROR>

(IE: NAME 'Mike' already exists, and no module with MID 14 exists.)

Disconnects.

<COMMAND>QUIT</COMMAND>

<DISCONNECT>
<ID>ID</ID> <KEY>key</KEY>
</DISCONNECT>

Disconnects connected ID, when ID and challenge-key match.

<DISACK>
<ID>2</ID>
</DISACK>

Disconnects specified client.
OR
Returns reason for no disconnect.

<ERROR><ID>1</ID></ERROR>
(IE: ID '1' does not exist.)

Disconnects.

<COMMAND>QUIT</COMMAND>

These are used by MODULE connections via port 12346:

<CONNECT>
<Name>name</NAME> <MTYPE>Module-Type</MTYPE> [<KEY>key</KEY>]
</CONNECT>

Returns MID, and numeric challenge-key on success. (If multiple connections on a single NAME are used, the key is required)

<CONACK>
<MID>13</ID><KEY>2648</KEY>
</CONACK>

MODULE is now available for CLIENT connections.
OR
Returns reason for no connect.

<ERROR><NAME>EchoUpper</NAME></ERROR>
(IE: NAME 'EchoUpper' already exists.)

Disconnects.

<COMMAND>QUIT</COMMAND>

```
<DISCONNECT>
<MID>MID</MID> <KEY>key</KEY>
</DISCONNECT>
```

Disconnects connected MID, when MID and challenge-key match.

```
<DISACK>
<MID>12</MID>
</DISACK>
```

Disconnects specified client.
OR
Returns reason for no disconnect.

```
<ERROR><ID>1</ID></ERROR>
```

(IE: ID '1' does not exist.)

Disconnects.

```
<COMMAND>QUIT</COMMAND>
```

A client's connection is known to be to a specific module. Once connected, all a client's message data is sent to that module. SIMSA wraps the data to tell the module who it is from.

```
<_MSG>text data in XML from client</_MSG>
```

The SIMSA receives a wrapped message
(Clients must send message wrapped in a <_MSG> tag. MSG and PKT both have a leading '_' to prevent tag name collision in module defined tags.)

```
<_PKT>
<FROM>client's ID</FROM>
<_MSG>text data in XML from client</_MSG>
</_PKT>
```

The module sends data out in a Packet wrapper, that SIMSA strips off, and the data is relayed to the client[s].

```
<_PKT>
<TO>id[,id]|!id|0</TO>
<_MSG>text data in XML to client</_MSG>
</_PKT>
```

(<TO> can be a single id or a comma separated list of ids. <TO> can also be '0' to send to all connected clients or an id preceded by '!' to send to all clients excluding the specified id.)

The message will be sent to the listed clients as:

```
<_MSG>text data in XML to client</_MSG>
```