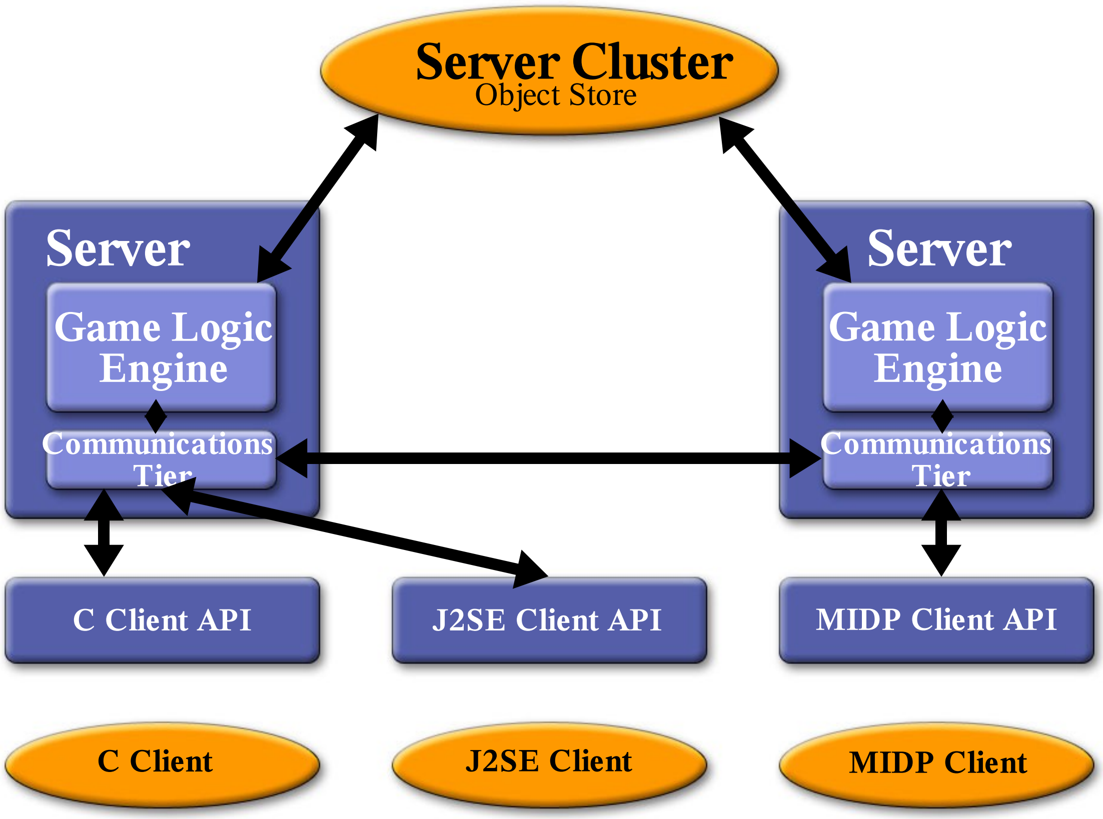
Sun Game Server J2SE Client API Programmer's Guide

Introduction to the Sun Game Server

The Sun Game Server (SGS) is a distributed software system built specifically for the needs of game developers. It is an efficient, scalable, reliable, and fault-tolerant architecture that simplifies the coding model for the game developer.

The SGS is a distributed system. Each host contains an SGS *slice* that consists of a single stack of software running in a single process space. Each slice can handle 200-500 users, depending on the game and the hardware.

The figure below shows the architecture of the Sun Game Server:



This architecture consists of four tiers.

The top tier (Tier 3) is a common data repository called the *Object Store*, which contains the entire game state and logic, implemented as a set of persistent *Game Logic Objects* (GLOs). Each SGS slice talks to this repository in order to save and retrieve the game state. .

Tier 2, on the Servers, is the *Game Logic Engine.* The Game Logic Engine is a container that executes event-driven code provided by the game developer. This event handling code is farmed out across the slices such that the entire set of active Game Logic Engines looks to the developer like a single execution environment. Race conditions and deadlock avoidance are handled automatically by the system so the user can write their code as if it were a single-threaded system.

Tier 1, the communications tier, provides a highly efficient publish/subscribe channel system to the game clients. It also handles passing data from the clients up to the servers and from the servers back to the clients.

Tier 0 is the Client API. This is a code library that client applications (game clients) use to communicate with each other and with the communications tier. As part of the SGS SDK we make Tier 0 available for J2SE, J2ME, and C++ applications. The C++ code is portable C++ and has been ported by us to two targets—Win32 and the Sony Playstation Portable (PSP).

Tier 0 takes care of discovering and connecting to SGS servers, as well as handling failover to a new server should the one to which it is currently connected suddenly fail.

For peer-to-peer communications, data passes only through Tier 0 and Tier 1.

More on the Communications Tier

- The *Router* does user authentication through pluggable Validators. Once a user has been authenticated, the Router provides a publish/subscribe channel mechanism to facilitate both peer-to-peer communications among clients and client/server communications with the Game Logic Engine. Validators are developer-extensible (see the document **Extending the SGS**.)
- The *Discoverer* collects information on available game connection points**.** (A *connection point* is a well-defined place where a client application can connect to a server application—for example, a particular *user manager* (discussed below) on a particular host.) The SGS back end makes this information available in the form of a regularly updated XML document that is accessible at a well-known URL. This document is the starting point for all client communications. From the XML document, the **URLDiscoverer** extracts all the possible connection points for a given game, examines included parameters (such as the host and port), and chooses a connection point. Discoverers are also extensible. At the moment the **URLDiscoverer** is the only one supported by the SGS back end. In the future, however, other means of obtaining the discovery information may be provided.
- The *User Manager* encapsulates all knowledge of how a client connects to and communicates with the SGS. The user manager is defined by interfaces and encapsulate a particular manner of communication (protocol, connection type, and so on). The SGS software distribution provides a TCP/IP-based user manager implementation, suitable for J2SE and C++ clients, and an HTTP-based implementation for J2ME clients. The set of available UserManagers is developer extensible (see the document **Extending the SGS**.)

There is an API wrapper that the client side of the user manager plugs into. This API wrapper knows about protocols defined by the Router for user authentication and failover and how to use the Discoverer to obtain discovery information.

The SGS provides the following client-side APIs:

- J2SE
- J2ME (MIDP2.0)
- C++

This Document

This document focuses on the J2SE client API. It assumes that there is a server-side application with which the client communicates.

The document gives an overview of the interfaces, then walks through a sample program, **ChatTest**, that makes use of these interfaces. It assumes you are familiar with basic concepts of Java programming.

Overview of the J2SE Client API to the Sun Game Server

Note: Reference information on the classes discussed here is located in the **J2SE API** Javadocs directory. Most of the classes are in the package **com.sun.gi.comm.users.client**.

The client interface is the level at which game clients operating on remote machines access the system. It hides all the details of game discovery, login, data transport, and failover from the game client itself. The game client instances and interacts with a **ClientCommunicationManager** object (described below) that handles all the details for the client.

The model presented to the game client is of a "pseudo-lan" where communication can take place with all other users logged into the same game. Other users of the system are invisible to the game client. Each **ClientConnectionManager** instance is stateful and logged into a single game; however, the **ClientConnectionManager** can be instanced multiple times in a single client program in order to log multiple clients into a single game, or a single client into multiple games.

The J2SE client interface to the Sun Game Server is an event/callback-based API. All server events are returned to the client as events/callbacks. This section goes through the major interfaces and events. The next section shows how a simple client application implements the API.

The ClientConnectionManager and the ClientConnectionManagerListener

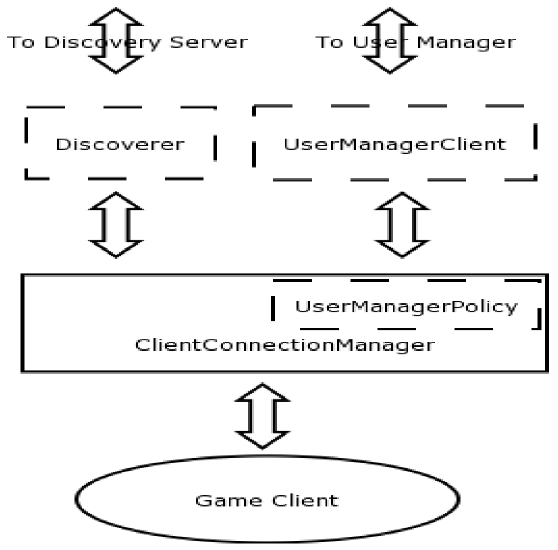
An SGS client application is a remote program that communicates with the SGS back end to do interclient communication and interface with SGS server applications. As discussed above, the client application communicates with the server application and other clients through an instance of the **ClientConnectionManager** class. A single instance represents a single user of a single SGS server application. One client application can create many **ClientConnectionManager** instances in order to create multiple user logons to one or more server applications.

Outgoing communications (that is, login, data packets, and so on) are sent by the client application making calls on the **ClientConnectionManager**'s API.

Incoming data is translated to events that are sent to the client application via a **ClientConnectionManagerListener** interface that the client application must implement and register with the **ClientConnectionManager**.

The SGS software distribution provides an implementation of the **ClientConnectionManager** interface, called **ClientConnectionManagerImpl**.

The **ClientConnectionManager** utilizes a number of pluggable objects, as shown in the figure and explained below.



The User Manager

The **ClientConnectionManager** talks to the server via a user manager. As discussed in the previous section, a game can have multiple user managers. Developers must determine which user manager their client application should use, based on its requirements. In the current SGS distribution, we provide only one user manager for J2SE and C++ applications, **TCPIPUserManagerClient**; others may be added in the future or can be added by the game developer themselves. The **TCPIPUserManagerClient** can be used as is, or as a template for developing your own communications protocol plug-ins.

We also provide a special user manager for J2ME clients called the **JMEHTTPUserManager**. It does not have the performance of the **TCPIPUserManager** but can be used by J2ME CLDC1.0/MIDP devices such as cell phones.

The User Manager Policy

A **ClientConnectionManager** contains a reference to an instance of a class that implements a **UserManagerPolicy** interface. This object is responsible for deciding to which of all possible connection points (for example, on different hosts) the **ClientConnectionManager** will actually connect. The default **UserManagerPolicy** is a simple random choice, which provides stochastic load balancing. More sophisticated mechanisms are certainly possible and can be substituted for the default with an optional **ClientConnectionManager** constructor parameter.

The Discoverer

The client application must begin by finding its game and the connection points to it on the server. As discussed in the previous section, this information is provided by a Discoverer.

The one currently supported Discoverer, **URLDiscoverer**, gets this information from a *discovery file.*

The Discovery File

The discovery file is an XML document that contains all the information the client needs to know to connect to one of the user managers in the available server applications.

Here are sample contents of **FakeDiscovery.xml**, which is provided in this distribution.for use during early client development. It contains entries for two games: **ChatTest** (which we discuss in the next section) and **BattleBoard**. There are two user managers for both **ChatTest** and **BattleBoard**.

```
<DISCOVERY>
-
        <GAME id="1" name="ChatTest">
-
        <USERMANAGER clientclass="com.sun.gi.comm.users.client.impl.TCPIPUserManagerClient">
<PARAMETER tag="host" value="127.0.0.1"/>
<PARAMETER tag="port" value="1150"/>
</USERMANAGER>
        <USERMANAGER  clientclass=”com.sun.gi.comm.users.client.impl.JMEHttpUserManagerClient”>
<PARAMETER tag="host" value="127.0.0.1"/>
<PARAMETER tag=”port” value=”8080”/>
<PARAMETER tag=”pollinterval” value=”500”/>
</USERMANAGER>
</GAME>
-
        <GAME id="5" name="BattleBoard">
-
        <USERMANAGER clientclass="com.sun.gi.comm.users.client.impl.TCPIPUserManagerClient">
<PARAMETER tag="host" value="127.0.0.1"/>
<PARAMETER tag="port" value="1180"/>
</USERMANAGER>
        <USERMANAGER  clientclass=”com.sun.gi.comm.users.client.impl.JMEHttpUserManagerClient”>
<PARAMETER tag="host" value="127.0.0.1"/>
<PARAMETER tag=”port” value=”8080”/>
<PARAMETER tag=”pollinterval” value=”500”/>
</USERMANAGER>
</GAME>
</DISCOVERY>
```

Note these points about the file:

- **TCPIPUserManagerClient** is used for J2SE and C++ clients, **J2MEHttpUserManagerClient** is used for J2ME clients only.

- **JMEHttpUserManagerClient** has a **pollinterval** parameter, because it is a polling protocol. **TCPIPUserManagerClient** doesn't require this parameter.

- All user managers for both games use the same host. In a production environment, of course, there would be multiple hosts. These would appear as other **<USERMANAGER>** blocks in the same discovery file.

- **TCPIPUserManagerClient** uses different ports on the host for different games. All communication for **ChatTest** goes through port 1150, and all communication for **BattleBoard** goes through port 1180.

- **JMEHttpUserManagerClient** uses the same port, 8080, for both games. This is because the **HTTPUserManager**s for different games all share a single internal “web browser” handling all HTTP communications for all games on a host. It listens on a single port, and the HTTP traffic for all games must arrive on that port.

A static local file such as **FakeDiscovery.xml** is useful when testing an application. In a production environment, however, the client application would point to a web server to obtain a dynamically generated discovery file. The URL of this web server would be provided by the SGS host.

Parsing the Discovery Document

Client applications can use the constructor **URLDiscoverer**, from the **com.sun.gi.comm.discovery.impl** package in the SGS distribution, to parse the specified discovery file. Currently, XML based discovery from a known URL is the only discovery strategy provided, however the API is pluggable so that other strategies could also be developed for different networking architectures.

Connecting to the Server

A client begins by requesting a list of the kinds of connection points (the user manager client class) available for their game from the **ClientConnectionManager**. The **ClientConnectionManager** gets this information from the Discoverer and returns it to the client.

The client compares that list with its own preferences and selects one user manager client class name and requests the **ClientConnectionManager** to connect to a User Manager of that type.

When the **ClientConnectionManager** connects to a game (either a login connection or a reconnect connection) it requests the list of connection points for the game that match the chosen User Manager type from the Discoverer. It then asks the **UserManagerPolicy** to choose one.

Once the **ClientConnectionManager** has a communication connection point, it creates an instance of the **UserManagerClient** class specified by that connection point and passes it the list of initialization parameter tuples that are also defined by the connection point. It then asks the new **UserManagerClient** to complete the connection.

Client Login and Validation

A game typically requires validation in order to accept a connection from an instance of its **UserManagerClient** class. Validation consists of the client application filling in the required fields in one or more arrays of **javax.security.auth.callback**.**Callback** objects.
In such a case, an array of **Calllback** objects is returned to the client application through one or more v**alidationRequest** messages that are delivered to the **ClientConnectionManagerListener** object.

A **ClientConnectionManagerListener** object responds to this event by calling the **sendValidationResponse** method on the **ClientConnectionManager**'s API with the appropriately filled-in callbacks.

This process continues until either the response is rejected by the connection point (in which case the listener receives a **connectionRejected** message containing a string giving a reason for refusal), or all responses have been accepted (in which case the listener receives a **connectionAccepted** message containing the assigned user ID for this connection).

It is possible that more than one **validationRequest** callback may occur. A client application should be written so that it can handle any number of these callbacks. The sequence of **validationRequest** callbacks for any given connection attempt will be terminated by either the **connectionAccepted** or **connectionRejected** callback.

Validation is entirely optional and specified at server-side game-install time. In the trivial case where no validation is specified at the server, the validation process is skipped and the listener immediately receives the **connectionAccepted** message.

Reconnecting

During the course of a client application's interaction with other clients and the (optional) server application that make up its game session, it is possible that it might lose connection to the SGS back end. This could be because of the failure of the SGS slice to which it is connected, or a failure of any of the systems and software processes that connect them.

In such a case the **ClientConnectionManager** notifies its registered **ClientConnectionManagerListener** objects with a **failoverInProgress** message. It then restarts the connection process, beginning with the discovery and choice of a new communication connection point.

While a **ClientConnectionManager** is connected to the SGS back end, it periodically receives a time-limited reconnection key from the back end. When it attempts a reconnection, the **ClientConnectionManager** submits this key as its validation. If the key is valid and has not expired, the SGS back end skips the connection validation phase and just returns the user ID that is associated with this key (the same user ID that was returned in the initial login). In this case the **ClientConnectionManager** sends its listener a **connected** message, after which communication can continue normally.

If the **ClientConnectionManager** cannot find any valid new communication connection point within the lifetime of the key, it sends its listener a **disconnected** message and gives up.

Logging Off

A client application requests that a **ClientConnectionManager** disconnect from the back end by calling the **disconnect** method on the **ClientConnectionManager**'s API. The **ClientConnectionManagerListener** will receive a **disconnected** message once logout has been accomplished.

Game Events

While the **ClientConnectionManager** is connected to the SGS back end, it can report the following events to its listener:

- **dataReceived**
  This listener method gets called whenever an inbound data packet arrives that was sent to the specific userID associated with this **ClientConnectionManager**.
- **userJoined**
  This listener method gets called once for every **ClientConnectionManager** that is connected to the game that this **ClientConnectionManager** joined (connected to). It then gets called periodically as new user join the game.
- **userLeft**
  This listener method gets called when any **ClientConnectionManager** whose connection was reported by **userJoined** logs off. (This can be either a purposeful logoff, or a communication failure and the timeout of its reconnection key.)

Channels

In addition to client-to-server communications, the client API makes a publish/subscribe channel system available to the client application. Client to client communication and server to client communications always happen via a channel. Channels are scoped within a single server application so that only that server application and the clients logged in to it may communicate with each other over a channel.

ClientChannel defines an SGS channel for use by a client program, and the **ClientChannelListener** interface defines how events are reported back to the client from a **ClientChannel**.

A player can send data to one, many, or all players on a channel. In addition, channel joiners receive **playerJoined** and **playerLeft** messages.

As discussed in the first section, messages sent between clients only go up through Tier 1 (Communications Tier) of the SGS architecture. They are not processed by the Game Logic Engine. (There is one exception, which is when the server application specifically requests to "eavesdrop" on one or more players on one or more channels. In that case, the eavesdropping data is propagated up to the Game Logic Engine. See the **SGS Server Application Developer's Guide** for more information on this feature.)

What a Client Application Needs to Do

An SGS client application at a minimum needs to do the following to interact with the server:

- Implement the **ClientConnectionManagerListener** so that it can receive events from the server.
- Create a Discoverer to find the discovery information. In practice today this means instancing the **URLDiscoverer** and giving it the URL location of an XML discovery file.
- Create a **UserManagerPolicy** to help the **ClientConnectionManager** decide which of the various possible connection points to user. Currently the API provides one **UserManagerPolicy**, the **DefaultUserManagerPolicy**,.

- Create an instance of **ClientConnectionManagerImpl** using the Discoverer and the user manager policy created above.
- Request a list of the types of connections available for the game we wish to log in to.
- Ask the **ClientConnectionManagerImpl** to connect to a connection point of the chosen type. Handle authentication callbacks as requested.
- Create and manage channels by which users can communicate with each other.and/o receive data from the sever.
- Send data, and respond to data received.
- Provide a means of disconnecting from the server application.

The ChatTest Example Program

The example program **ChatTest** is a basic chat program implemented using the Sun Game Server. The client application, **ChatTestClient**, uses a simple Java Swing user interface by which the user can open channels and chat over them with other **ChatTest** users. It shows, in simplified fashion, how to use the SGS client API.

The client application consists of three classes. The main class, **ChatTestClient**, allows users to log on to the server application, send messages to the server and to each other over a DCC (Direct Client to Client) channel, and choose other channels by which to send messages. The other two classes do the following:

- **ChatChannelFrame** provides a GUI by which the user can interact with a specific channel other than DCC.

- **ValidatorDialog** implements validation for users of **ChatTest**.

Source code for the three client classes is shown in the appendix. In this section, we focus on the interactions with the server; we don't discuss details of how to program the Swing user interface.

Summary of the Chat Client's Use of the API

The sample client application uses these components of the API:

- **ChatTestClient** implements the **ClientConnectionManager** interface via the ClientConnectionManagerImpl implementation provided in the SGS software distribution.

- It establishes itself as the **ClientConnectionManagerListener**, to listen for events from the server. It does not implement the failover callbacks because there is really nothing an application this simple needs to do during failover. (An extension might be to change the status message to "failing-over" and then "reconnected" just for information purposes.)

- It uses the default **UserManagerPolicy**.

- It uses the **URLDiscoverer** constructor to parse the discovery file and provide discovery information back to the rest of the API

- It sets itself as the **ClientChannelListener** for the DCC channel. The **ChatTestFrame** class becomes the **ClientChannelListener** for user-specified channels.

Here are the **import** statements:

```
import com.sun.gi.comm.discovery.impl.URLDiscoverer;
import com.sun.gi.comm.users.client.ClientAlreadyConnectedException;
import com.sun.gi.comm.users.client.ClientChannel;
import com.sun.gi.comm.users.client.ClientChannelListener;
import com.sun.gi.comm.users.client.ClientConnectionManager;
import com.sun.gi.comm.users.client.ClientConnectionManagerListener;
import com.sun.gi.comm.users.client.impl.ClientConnectionManagerImpl;
import com.sun.gi.utils.types.BYTEARRAY;
import com.sun.gi.utils.types.StringUtils;
```

Connecting to the Server Application

As discussed above, a client must begin by finding the game it wants and connecting to the server application.

The client connects to the server application via the **ClientConnectionManager** interface. It needs to tell **ClientConnectionManager** the name of the application to connect to and how to find it. **ChatTestClient** does this by using the **ClientConnectionManagerImpl** implementation provided in the SGS software distribution. It passes the name of the server application, **ChatTest**, and a **URLDiscoverer** that is set to parse the local discovery file, **FakeDiscovery.xml**. (See the previous section for more information about this discovery file.)

ClientConnectionManagerImpl uses the discoverer **URLDiscoverer**, from the **com.sun.gi.comm.discovery.impl** package, to parse the specified discovery file and determine if **ChatTest** is there and if so what **UserManagerClients** are supported at what connection points.

If **ChatTestClient** had a **UserManagerPolicy**, it would specify it here. Since it doesn't, it uses the default policy, which means that is will use random assignment to game connection points.

As part of establishing the connection, **ChatTestClient** also uses the **setListener** function from **ClientConnectionManager** to establish itself as the **ClientConnectionManagerListener**. The **ClientConnectionManagerListener** listens for events coming back from the **ClientConnectionManager**.

Here is the code that does this:

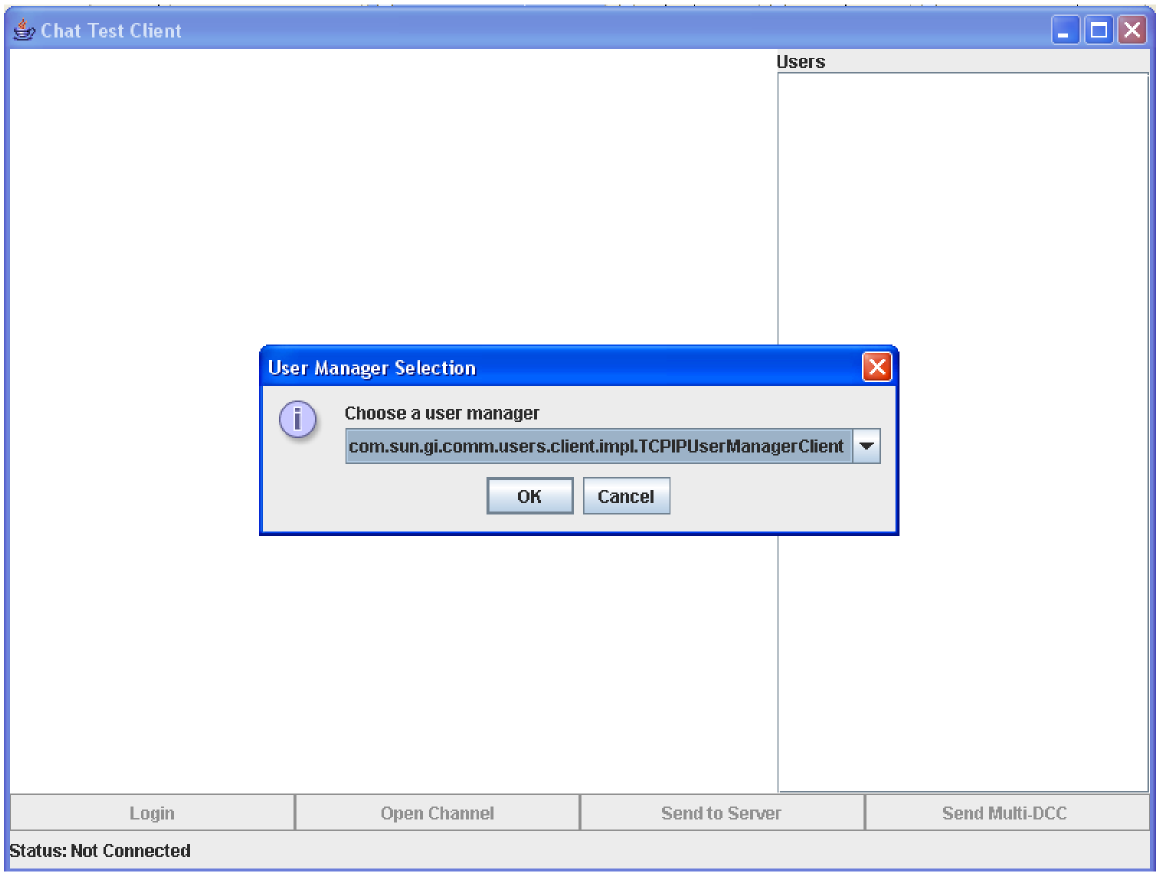
```
                 try {
                        mgr = new ClientConnectionManagerImpl("ChatTest",
                                        new URLDiscoverer(new File("FakeDiscovery.xml").toURI()
                                                        .toURL()));
                        mgr.setListener(this);

                } catch (MalformedURLException e) {
                        e.printStackTrace();
                        System.exit(2);
                }
```

Choosing a User Manager

The client application must choose the user manager by which it will communicate with the server. **ChatTestClient** does this as part of the login process. No real choice is offered, since the SGS software distribution provides only a TCP/IP User Manager for J2SE applications, but this example shows one mechanism that the client application can use. Note, however, that this is not a typical way of determining the user manager. The client application should know the user manager it wants to use, based on its requirements. For instance, a fast-action racing game would never ask for the **HTTPUserManagerClient** because that UserManager cannot handle the game's latency requirements. On the other hand, our chat application might prefer the **TCPIPUserManagerClient** but be willing to use the **HTTPUserManagerClient** (if one were available) because latency is not a big issue for chatting.

When the user presses the **Login** button in the dialog box (see the figure below), **ChatTestClient** gets the names of available user managers; it does this via the **getUserManagerClassName** method in the **ClientConnectionManager** interface. It then presents these user managers to the user in a dialog box and asks the user to choose one.



Choose a user manager

com.sun.gi.comm.users.client.impl.TCPIPUserManagerClient

Status: Not Connected

When the user clicks **OK**, **ChatTestClient** reads the user's choice and attempts to connect, using the **connect** function from **ClientConnectionManager**. If the user is already connected, it throws **ClientAlreadyConnectedException**.

Here is the connection code from **ChatTestClient**:

```
public void actionPerformed(ActionEvent e) {
				if (loginButton.getText().equals("Login")) {
					loginButton.setEnabled(false);
					String[] classNames = mgr.getUserManagerClassNames();
					String choice = (String) JOptionPane.showInputDialog(
							ChatTestClient.this, "Choose a user manager",
						"User Manager Selection",
						JOptionPane.INFORMATION_MESSAGE, null, classNames,
							classNames[0]);
					try {
						mgr.connect(choice);
					} catch (ClientAlreadyConnectedException e1) {
						// TODO Auto-generated catch block
						e1.printStackTrace();
						System.exit(1);
					}
				} else {
					mgr.disconnect();
				}

			}
```

Connection Failover

The **ClientConnectionManagerListener** provides callbacks that are called when there is a connection failure. **ChatTestClient** does not implement these callbacks.

The callback **failOverinProgress** informs the client that it is temporarily disconnected. The client might then want to take action to avoid piling up a backlog of messages.

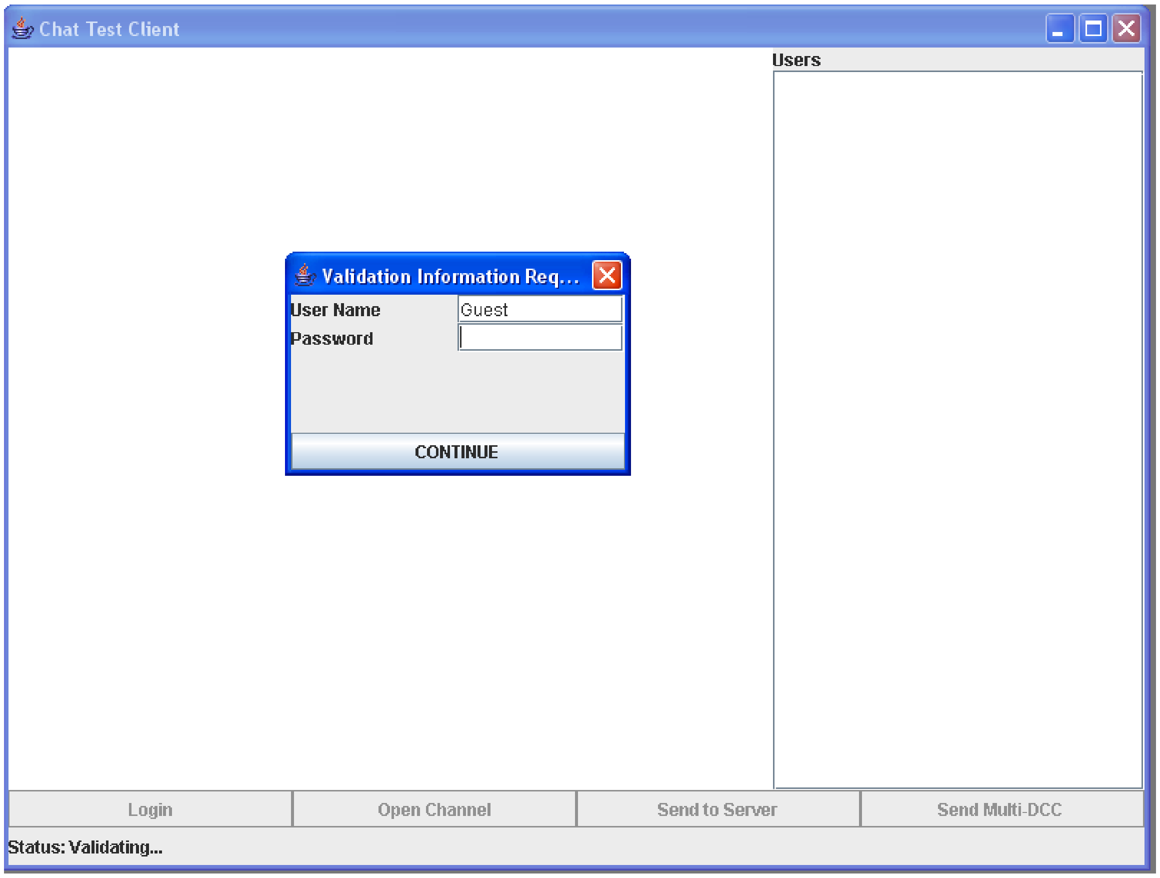
The callback **reconnected** informs the client that it has been reconnected, so it should start communicating again.

Logging In

As noted above, when the user presses the **Login** button, **ChatTestClient** begins by asking the user to choose a user manager. Once this has been done, the next step is to provide validation information.

Specifying User Name and Password

Once the user has chosen a user manager, the **ClientConnectionManager** requests validation for the user via the **validationRequest** callback. In response, **ChatTestClient** calls the **ValidatorDialog** method, which displays a dialog box (shown below) that prompts the user for any validation information requested by the validator;. In the case of the simple example validator FlatFileValidator, this is simply name and password.; The validation strategy is completely up to the server application's installed Validator or Validators; in **ChatTest**, the user name of **Guest** with any (or no) password works as the default login. If you look in the **passwd.txt** file you will see that Guest is defined with a password of “*”. The **FlatFileValidator** understands this to mean “match any password.”



When the user fills in the fields and presses the **CONTINUE** button, **ValidatorDialog** packages up the responses and sends them to the server application via the **sendValidationResponse** function from the **ClientConnectionManager** interface.

Here is the validation code from **ChatTestClient**:

```
public void validationRequest(Callback[] callbacks) {
                statusMessage.setText("Status: Validating...");
                new ValidatorDialog(this, callbacks);
                mgr.sendValidationResponse(callbacks);
        }
```

If the login is successful, **ClientConnectionManager** sends an event to the callback **connected** (part of the **ConnectionManagerListenerInterface**) in **ChatTestClient,** returning a userID for the user on this connection. **ChatTestClient** then enables the appropriate buttons in the GUI and opens a DCC (Direct Client-to-Client) channel. (See **Opening a Channel**, below, for more information about what happens with the channel.) UserIDs are really session IDs in that they change from login to login. (They stay the same during a failover, though.)

Here is the code that does this:

```
 public void connected(byte[] myID) {
                statusMessage.setText("Status: Connected");
                setTitle("Chat Test Client: " + StringUtils.bytesToHex(myID));
                loginButton.setText("Logout");
                loginButton.setEnabled(true);
                openChannelButton.setEnabled(true);
                dccButton.setEnabled(true);
                serverSendButton.setEnabled(true);
                mgr.openChannel(DCC_CHAN_NAME);
        }
```

If the connection is refused (for example, due to an incorrect password), the **ClientConnectionManager** sends back a message containing the reason via the **connectionRefused** callback in the **ClientConnectionManagerInterface**. **ChatTestClient** updates its GUI and displays the message. Here is the code that does this:

```
public void connectionRefused(String message) {
                statusMessage.setText("Status: Connection refused. (" + message + ")");
                loginButton.setText("Login");
                loginButton.setEnabled(true);
        }
```

Opening a Channel

The DCC Channel

The DCC (Direct Client-to-Client) channel (defined as **dccChannel**) is made available upon login. **ChatTestClient** sets up a **ClientChannelListener** interface to receive messages from it.

Here is the code that does this:

```
public void joinedChannel(ClientChannel channel) {
                if (channel.getName().equals(DCC_CHAN_NAME)) {
                        dccChannel = channel;
                        dccChannel.setListener(new ClientChannelListener() {

                                public void playerJoined(byte[] playerID) {
                                }

                                public void playerLeft(byte[] playerID) {
                                }

                                public void dataArrived(byte[] from, ByteBuffer data,
                                                boolean reliable) {
                                        byte[] bytes = new byte[data.remaining()];
                                        data.get(bytes);
                                        JOptionPane.showMessageDialog(ChatTestClient.this,
                                                        new String(bytes), "Message from "
                                                                        + StringUtils.bytesToHex(from,
                                                                                        from.length - 4),
                                                        JOptionPane.INFORMATION_MESSAGE);

                                }

                                public void channelClosed() {
                                }
                        });
                } else {
                        ChatChannelFrame cframe = new ChatChannelFrame(channel);
                        desktop.add(cframe);
                        desktop.repaint();
                }
        }
```

Note that, if the channel is not DCC, the code calls the **ChatChannelFrame** class to handle it, as discussed below.

The **setListener** method is a **ClientChannel** interface. It sets up the one and only listener on this channel. Thus, **ChatTestClient** is a listener on both the **ClientConnectionManager** and the **ClientChannel** for the DCC channel.

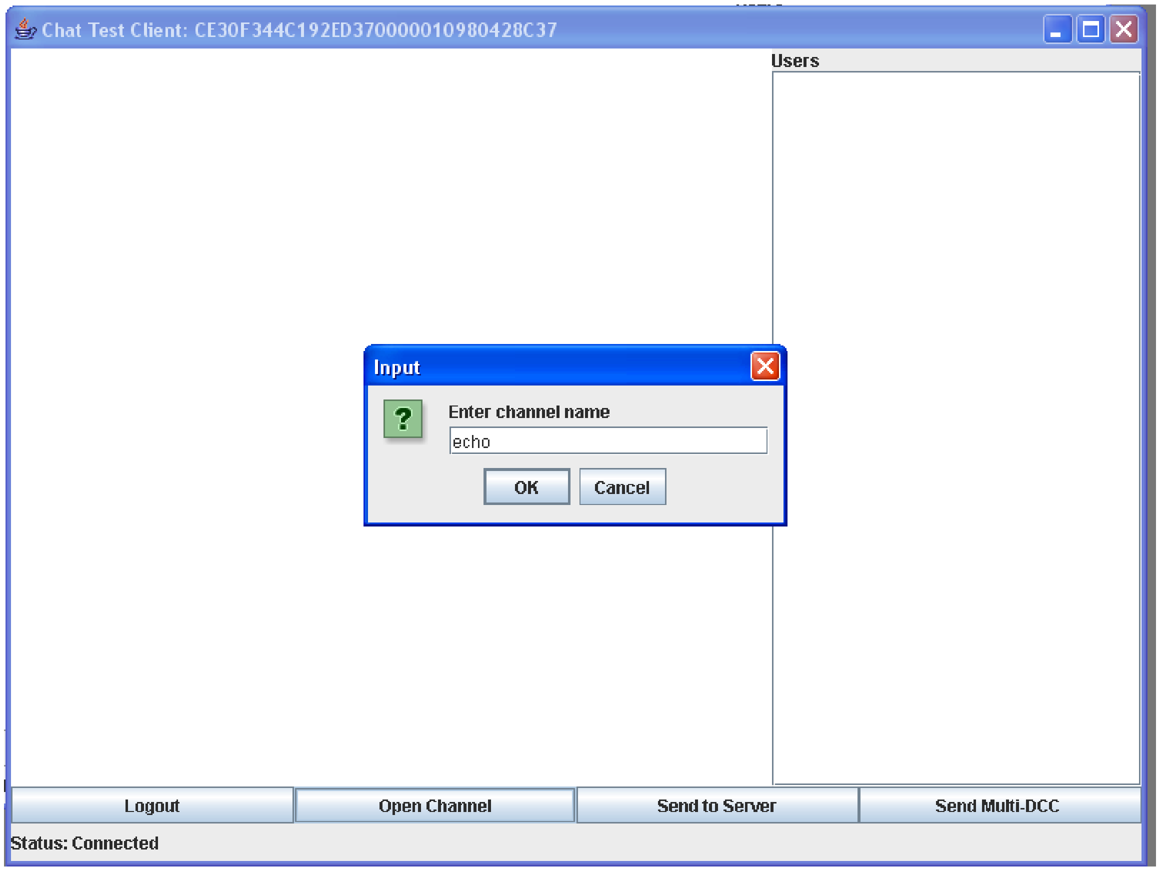
The **playerJoined** and **playLeft** functions are **ClientChannelListener** interfaces. **ChatTestClient** doesn't do anything with them. Since everyone who joins the application is automatically added to the DCC channel, it can use the **userJoined** and **userLeft** functions from the **ClientConnectionManagerListener** interface instead. See **Responding to Events**, below.

The **dataArrived** function is also a **ClientChannelListener** interface. If data arrives on this listener, **ChatTestClient** displays it on the left side of the dialog box.

As users join the game, their IDs are added to the list on the right side of the dialog box, as discussed in **Responding to Events**, below.

A User-Specified Channel

In addition to the DCC channel, the user can click the **Open Channel** button to open a channel by name. The channel can have any name. See the figure below.



The code in **ChatTestClient** uses the **openChannel** method from the **ClientConnectionManager** interface. Here is the code that does this:

```
 openChannelButton = new JButton("Open Channel");
                openChannelButton.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent e) {
                                String channelName = JOptionPane.showInputDialog(
                                                ChatTestClient.this, "Enter channel name");
                                mgr.openChannel(channelName);
                 }
}};
```

ChatTestClient includes a callback to the **ClientConnectionManagerListener** method **joinedChannel**. **ClientConnectionManager** sends an event to **joinedChannel** to confirm that the user joined the specified channel.

ChatTestClient calls the **ChatChannelFrame** class to create a new dialog box for the specified channel and to attach a **ClientChannelListener** to the channel. The code in **ChatChannelFrame** that does this is:

```
public ChatChannelFrame(ClientChannel channel){
                super("Channel: "+channel.getName());
                outbuff = ByteBuffer.allocate(2048);
                chan = channel;
                chan.setListener(this);
```

Sending and Receiving Data

ChatTestClient provides ways of sending messages directly to the server application, to specified users logged in on the DCC channel, or a multicast message to all DCC users. The section **The DCC Channel**, above, covered how it receives a message on the DCC channel.

ChatChannelFrame sends and receives data for a user-specified channel.

Sending a Message to the Server

In **ChatTestClient** , the user sends a message to the server application by clicking the **Send to Server** button in the dialog box. This causes **ChatTestClient** to display a dialog box asking the user to enter the message. The message is then packaged up in the **doServerMessage** method. Here is the code that does this:

```
serverSendButton = new JButton("Send to Server");
                serverSendButton.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent arg0) {
                                Object[] targets = userList.getSelectedValues();
                                if (targets != null) {
                                        String message = JOptionPane.showInputDialog(
                                                        ChatTestClient.this,
                                                        "Enter server message:");
                                        doServerMessage(message);
```

The **doServerMessage** method then sends this message to the server via the **ClientConnectionManager**'s **sendToServer** method:

```
 protected void doServerMessage(String message) {
                ByteBuffer out = ByteBuffer.allocate(message.length());
                out.put(message.getBytes());
                mgr.sendToServer(out,true);

        }
```

In **ChatTest**, the server simply prints the message sent to it. The functionality is included to show how a client application sends data to the server.

Sending a Message to a Single DCC User

The user clicks on the User ID in the right-hand column of the dialog box to choose the user to whom the message is to be sent. **ChatTestClient** then prompts for the message. When the user enters the message, it is packaged up in the **doDCCMessage** method:

```
		userList.addMouseListener(new MouseAdapter() {
			public void mouseClicked(MouseEvent evt) {
				if (evt.getClickCount() > 1) {
					BYTEARRAY ba = (BYTEARRAY) userList.getSelectedValue();
					if (ba != null) {
						String message = JOptionPane.showInputDialog(
								ChatTestClient.this,
								"Enter private message:");
						doDCCMessage(ba.data(), message);
					}
				}

			}
```

The **doDCCMessage** method then sends the message over the channel via the **ClientChannel** method **sendUnicastData**:

```
 protected void doDCCMessage(byte[] target, String message) {
		ByteBuffer out = ByteBuffer.allocate(message.length());
		out.put(message.getBytes());
		dccChannel.sendUnicastData(target, out, true);
```

Sending a Multicast Message to the DCC Channel

If the user clicks the **Send Multi-DCC** button, **ChatTestClient** prompts for the message and builds an array of the users that are listed in the right-hand side of the dialog box. It packages up this list of userIDs and the message in the **doMultiDCCMessage** method:

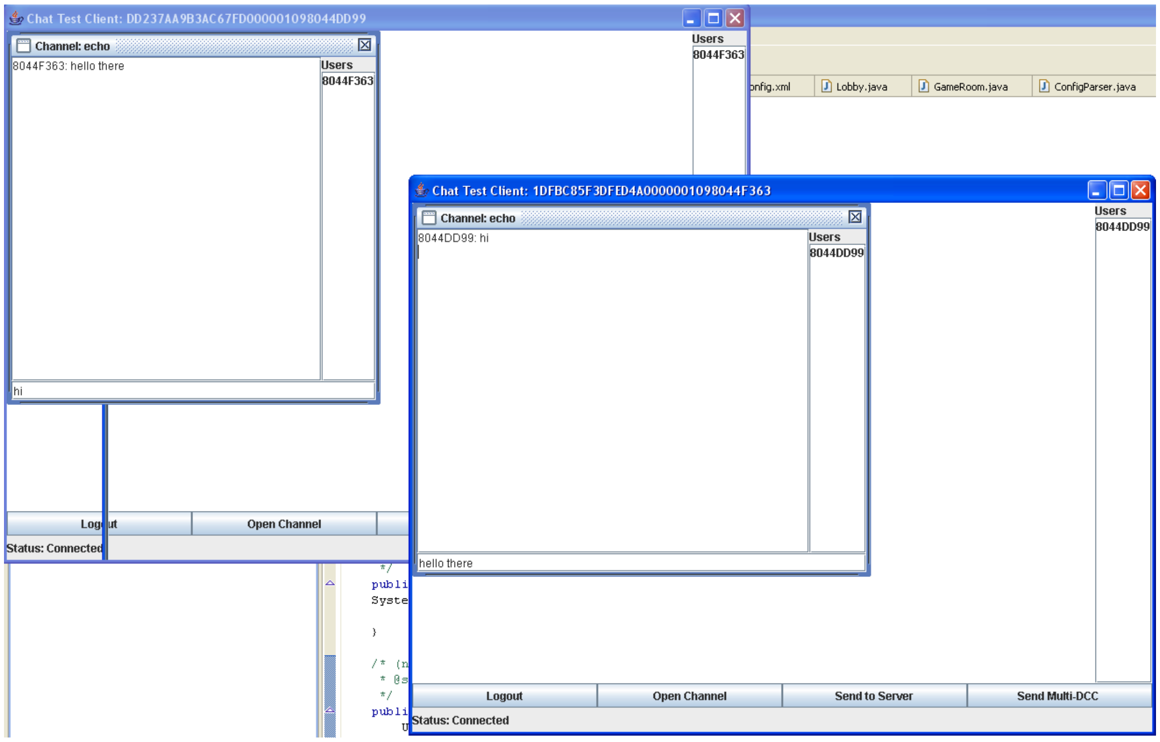
```
dccButton = new JButton("Send Multi-DCC ");
		dccButton.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent arg0) {
				Object[] targets = userList.getSelectedValues();
				if (targets != null) {
					String message = JOptionPane.showInputDialog(
							ChatTestClient.this,
							"Enter private message:");
					byte[][] targetList = new byte[targets.length][];
					for(int i=0;i<targets.length;i++){
						targetList[i] = ((BYTEARRAY)targets[i]).data();
					}
					doMultiDCCMessage(targetList,message);
```

The **doMultiDCCMessage** method then sends the message over the channel via the **ClientChannel** method **sendMulticastData**:

```
	protected void doMultiDCCMessage(byte[][] targetList, String message) {
		ByteBuffer out = ByteBuffer.allocate(message.length());
		out.put(message.getBytes());
		dccChannel.sendMulticastData(targetList, out, true);
	 }
```

Sending a Message to All Users of a User-Specified Channel

Users of a non-DCC channel can send a message by typing it in the space at the bottom of the dialog box set up by **ChatChannelFrame**. The figure below shows messages sent and received on the channel **Echo**.



8044F363: hello there

hi

8044DD99: hi

hello there

ChatChannelFrame uses the following constructor to read the text, send it to all users of the channel via the **ClientChannel** method **sendBroadcastData**, and then clear the field:

```
inputField.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e) {
                                outbuff.clear();
                                outbuff.put(inputField.getText().getBytes());
                                chan.sendBroadcastData(outbuff,true);
                                inputField.setText("");
                        }});
```

Receiving Data on a User-Specified Channel

ChatChannelFrame uses the **ClientChannelListener** callback **dataArrived** to listen for data coming on in the user-specified channel. When data does arrive, it appends the data to the output area on the left side of the dialog box, as shown above. Here is the code that does this:

```
        public void dataArrived(byte[] from, ByteBuffer data, boolean reliable){
                byte[] textb =new byte[data.remaining()];
                data.get(textb);
                outputArea.append(StringUtils.bytesToHex(from,from.length-4)+": "+ new
                 String(textb)+"\n");
        }
```

Responding to Events

Callbacks in **ChatTestClient** and **ChatTestFrame** determine what happens when various connection events occur.

The User Is Disconnected

If the user is disconnected, the **ClientConnectionManager** sends an event to the **disconnected** callback in **ChatTestClient**. **ChatTestClient** displays a message and updates the dialog box:

```
        public void disconnected() {
                statusMessage.setText("Status: logged out");
                loginButton.setText("Login");
                loginButton.setEnabled(true);
                openChannelButton.setEnabled(false);
                dccButton.setEnabled(false);
                serverSendButton.setEnabled(false);
        }
```

Another User Joins the Game

If another user joins the game, the **ClientConnectionManager** sends an event to the **userJoined** callback in **ChatTestClient**. **ChatTestClient** then adds this user's User ID to the list on the right side of the dialog box:

```
        public void userJoined(byte[] userID) {
                DefaultListModel mdl = (DefaultListModel) userList.getModel();
                mdl.addElement(new BYTEARRAY(userID));
                userList.repaint();

        }
```

Another User Leaves the Game

If another user leaves the game, the **ClientConnectionManager** sends an event to the **userLeft** callback in **ChatTestClient**. **ChatTestClient** then removes this user's User ID from the list on the right side of the dialog box:

```
        public void userLeft(byte[] userID) {
                DefaultListModel mdl = (DefaultListModel) userList.getModel();
                mdl.removeElement(new BYTEARRAY(userID));
                userList.repaint();
        }
```

Another User Joins the User-Specified Channel

If another user joins the user-specified channel, the **ClientConnectionManager** sends an event to the **ClientChannelListener** callback **playerJoined** in **ChatTestFrame**. **ChatTestFrame** adds the user to the list on the right side of the dialog box:

```
        public void playerJoined(byte[] playerID) {
                DefaultListModel mdl = (DefaultListModel)userList.getModel();
                mdl.addElement(new BYTEARRAY(playerID));
        }
```

A User Leaves the User-Specified Channel

If a user leaves the user-specified channel, the **ClientChannelListener** callback **playerLeft** is called in **ChatTestFrame**, and **ChatTestFrame** removes the user from the list on the right side of the dialog box:

```
        public void playerLeft(byte[] playerID) {
                DefaultListModel mdl = (DefaultListModel)userList.getModel();
                mdl.removeElement(new BYTEARRAY(playerID));
        }
```

The User-Specified Channel Is Closed

If the user-specified channel is closed by the server, the **ClientChannelListener** callback **channelClosed** is called in **ChatTestFrame**, and **ChatTestFrame** simply removes the dialog box for that channel:

```
        public void channelClosed() {
                if (getDesktopPane() != null) {
                        getDesktopPane().remove(this);
                }
```

Disconnecting

There is no formal logout in **ChatTestClient**. Closing the main window causes **ChatTestClient** to simply disconnect from its user manager. It does this via the **disconnect** method from the **ClientConnectionManager** interface. Here is the code:

```
this.addWindowStateListener(new WindowStateListener() {
                        public void windowStateChanged(WindowEvent arg0) {
                                if (arg0.getNewState() == WindowEvent.WINDOW_CLOSED) {
                                        mgr.disconnect();
                                }
                        }
                 }
```

Appendix: Source Code for the commtest Sample Program

ChatTestClient

```
package com.sun.gi.apps.commtest.client;

import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowStateListener;
import java.io.File;
import java.net.MalformedURLException;
import java.nio.ByteBuffer;

import javax.security.auth.callback.Callback;
import javax.swing.AbstractButton;
import javax.swing.JButton;
import javax.swing.JDesktopPane;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.DefaultListModel;
import javax.swing.ListCellRenderer;
import javax.swing.ListSelectionModel;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

import com.sun.gi.comm.discovery.impl.URLDiscoverer;
import com.sun.gi.comm.users.client.ClientAlreadyConnectedException;
import com.sun.gi.comm.users.client.ClientChannel;
import com.sun.gi.comm.users.client.ClientChannelListener;
import com.sun.gi.comm.users.client.ClientConnectionManager;
import com.sun.gi.comm.users.client.ClientConnectionManagerListener;
import com.sun.gi.comm.users.client.impl.ClientConnectionManagerImpl;
import com.sun.gi.utils.types.BYTEARRAY;
import com.sun.gi.utils.types.StringUtils;

//@SuppressWarnings("serial")

/**
 * <p>The ChatTestClient implements a simple chat program using a Swing UI, mainly for server channel testing purposes.
 * It allows the user to open arbitrary channels by name, or to use the Direct Client to Client (DCC)
 * channel.</p>
 *
 * <p>Before connecting to any channels, the user must login to the server via a UserManager.  When the Login button
 * is pressed, a list of available UserManagers is displayed. (Right now the TCPIPUserManager is the only implemented
 * option).  After selecting a UserManager, the user is prompted for credentials (currently "Guest" with a blank
 * password will do a default login).</p>
 *
 * <p>Once logged in, the user can open a new channel, or join an existing channel by clicking the "Open Channel" button
 * and typing in the channel name.  Once connected to a channel, a separate window opens revealing the users connected
to
 * that channel as well as an area to type messages.  Each channel opened will appear in a new window.</p>
 *
 * <p>A user can send a message directly to the server via the "Send to Server" button.  Messages are sent to the server
 * via the ClientConnectionManager.</p>
 *
 * <p>A user can send a Direct Client to Client message via the "Send Multi-DCC" button.  This will send a multicast
 * message on the well-known DCC channel.</p>
 *
 * <p>This class implements ClientConnectionManagerListener so that it can receive and respond to connection events
from the server.</p>
 *
 * <p>The ChatTestClient is designed to work with the server application, ChatTest.  This application must be installed
and
 * running on the server for the ChatTestClient to successfully login.  Multiple instances of this program can be run
so that
 * multiple users will be shown in the client.</p>
 */
public class ChatTestClient extends JFrame implements ClientConnectionManagerListener {

        JButton loginButton;

        JButton openChannelButton;
        JLabel statusMessage;

        JDesktopPane desktop;

        JList userList;                       // a list of users currently connected to the ChatTest app on the server.

        ClientConnectionManager mgr;            // used for communication with the server.

        ClientChannel dccChannel;               // the well-known channel for Direct Client to Client communication.

        private JButton dccButton;

        private JButton serverSendButton;

        private static String DCC_CHAN_NAME = "__DCC_Chan";

        public ChatTestClient() {
                // build interface
                super();
                setTitle("Chat Test Client");
                Container c = getContentPane();
                c.setLayout(new BorderLayout());
                JPanel buttonPanel = new JPanel();
                JPanel southPanel = new JPanel();
                southPanel.setLayout(new GridLayout(0, 1));
                statusMessage = new JLabel("Status: Not Connected");
                southPanel.add(buttonPanel);
                southPanel.add(statusMessage);
                c.add(southPanel, BorderLayout.SOUTH);
                desktop = new JDesktopPane();
                c.add(desktop, BorderLayout.CENTER);
                JPanel eastPanel = new JPanel();
                eastPanel.setLayout(new BorderLayout());
                eastPanel.add(new JLabel("Users"), BorderLayout.NORTH);
                userList = new JList(new DefaultListModel());
                userList.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);


                userList.addMouseListener(new MouseAdapter() {
                        public void mouseClicked(MouseEvent evt) {
                                if (evt.getClickCount() > 1) {
                                        BYTEARRAY ba = (BYTEARRAY) userList.getSelectedValue();
                                        if (ba != null) {
                                                String message = JOptionPane.showInputDialog(
                                                                ChatTestClient.this,
                                                                "Enter private message:");
                                                doDCCMessage(ba.data(), message);
                                        }
                                }

                        }
                });

                userList.setCellRenderer(new ListCellRenderer(){
                        JLabel text = new JLabel();
                        public Component getListCellRendererComponent(JList arg0, Object arg1, int arg2, boolean arg3,
boolean arg4) {
                                byte[] data = ((BYTEARRAY)arg1).data();
                                text.setText(StringUtils.bytesToHex(data,data.length-4));
                                return text;
                        }});
                eastPanel.add(new JScrollPane(userList), BorderLayout.CENTER);
                c.add(eastPanel, BorderLayout.EAST);
                buttonPanel.setLayout(new GridLayout(1, 0));
                loginButton = new JButton("Login");

                // The login button will attempt to login to the ChatTest application on the server.
                // It must do this via an UserManager.  The ClientConnectionManager is used to
                // return an array of valid UserManager class names.  Once selected, the ClientConnectionManager
                // will attempt to connect via this UserManager.

                loginButton.addActionListener(new ActionListener() {

                        public void actionPerformed(ActionEvent e) {
                                if (loginButton.getText().equals("Login")) {
                                        loginButton.setEnabled(false);
                                        String[] classNames = mgr.getUserManagerClassNames();
                                        String choice = (String) JOptionPane.showInputDialog(
                                                        ChatTestClient.this, "Choose a user manager",
                                                        "User Manager Selection",
                                                        JOptionPane.INFORMATION_MESSAGE, null, classNames,
                                                        classNames[0]);
                                        try {
                                                mgr.connect(choice);
                                        } catch (ClientAlreadyConnectedException e1) {
                                                // TODO Auto-generated catch block
                                                e1.printStackTrace();
                                                System.exit(1);
                                        }
                                } else {
                                        mgr.disconnect();
                                }

                        }
                });

                // Opens a channel by name on the server.  If the channel doesn't exist,
                // it will be opened.  In either case, it will attempt to join the user
                // to the specified channel.
                openChannelButton = new JButton("Open Channel");
                openChannelButton.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent e) {
                                String channelName = JOptionPane.showInputDialog(
                                                ChatTestClient.this, "Enter channel name");
                                mgr.openChannel(channelName);
                        }
                });
                openChannelButton.setEnabled(false);

                // Sends a message directly to the server.
                serverSendButton = new JButton("Send to Server");
                serverSendButton.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent arg0) {
                                Object[] targets = userList.getSelectedValues();
                                if (targets != null) {
                                        String message = JOptionPane.showInputDialog(
                                                        ChatTestClient.this,
                                                        "Enter server message:");
                                        doServerMessage(message);
                                }
                        }
                });
                serverSendButton.setEnabled(false);

                // sends a mulicast message to the selected users on the DCC channel.
                dccButton = new JButton("Send Multi-DCC ");
                dccButton.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent arg0) {
                                Object[] targets = userList.getSelectedValues();
                                if (targets != null) {
                                        String message = JOptionPane.showInputDialog(
                                                        ChatTestClient.this,
                                                        "Enter private message:");
                                        byte[][] targetList = new byte[targets.length][];
                                        for(int i=0;i<targets.length;i++){
                                                targetList[i] = ((BYTEARRAY)targets[i]).data();
                                        }
                                        doMultiDCCMessage(targetList,message);
                                }
                        }
                });
                dccButton.setEnabled(false);
                buttonPanel.add(loginButton);
                buttonPanel.add(openChannelButton);
                buttonPanel.add(serverSendButton);
                buttonPanel.add(dccButton);
                pack();
                setSize(800, 600);
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                // start connection process.  The ClientConnectionManager is the central
                // point of server communication for the client.  The ClientConnectionManager
                // needs to know the application name to attempt to connect to on the server
                // and how to find it.  In this case the app name is "ChatTest" and FakeDiscovery.xml
                // lists the valid UserManagers.
                try {
                        mgr = new ClientConnectionManagerImpl("ChatTest",
                                        new URLDiscoverer(new File("FakeDiscovery.xml").toURI()
                                                        .toURL()));
                        mgr.setListener(this);

                } catch (MalformedURLException e) {
                        e.printStackTrace();
                        System.exit(2);
                }

                // When the window closes, disconnect from the manager.
                this.addWindowStateListener(new WindowStateListener() {
                        public void windowStateChanged(WindowEvent arg0) {
                                if (arg0.getNewState() == WindowEvent.WINDOW_CLOSED) {
                                        mgr.disconnect();
                                }
                        }
                });
                setVisible(true);
        }

        /**
         * Sends a message to the server via the ClientConnectionManager.
         *
         * @param message               the message to send.
         */
        protected void doServerMessage(String message) {
                ByteBuffer out = ByteBuffer.allocate(message.length());
                out.put(message.getBytes());
                mgr.sendToServer(out,true);
```

```
}

/**
 * Sends a multicast message on the DCC channel.
 *
 * @param targetList            the list of users to send to
 * @param message                       the message to send
 */
protected void doMultiDCCMessage(byte[][] targetList, String message) {
        ByteBuffer out = ByteBuffer.allocate(message.length());
        out.put(message.getBytes());
        dccChannel.sendMulticastData(targetList, out, true);

}

/**
 * Sends a message to a single user on the DCC channel.
 *
 * @param target                the user to send to.
 * @param message               the message to send.
 */
protected void doDCCMessage(byte[] target, String message) {
        ByteBuffer out = ByteBuffer.allocate(message.length());
        out.put(message.getBytes());
        dccChannel.sendUnicastData(target, out, true);

}

/**
 * ClientConnectionManagerListener callback.
 *
 * Called by the ClientConnectionManager to request validation credentials.
 * The ValidatorDialog takes over from here to populate the CallBacks with
 * user data.
 *
 * In the case of ChatTest, both a username and password are required.
 *
 * @param callbacks      the array of javax.security.auth.callbacks.CallBacks to validate against.
 */
public void validationRequest(Callback[] callbacks) {
        statusMessage.setText("Status: Validating...");
        new ValidatorDialog(this, callbacks);
        mgr.sendValidationResponse(callbacks);
}

/**
 * ClientConnectionManagerListener callback.
 *
 * Called by the ClientConnectionManager when the login is successful.
 *
 * @param myID                  the user's id.  Used for future communication with the server.
 */
public void connected(byte[] myID) {
        statusMessage.setText("Status: Connected");
        setTitle("Chat Test Client: " + StringUtils.bytesToHex(myID));
        loginButton.setText("Logout");
        loginButton.setEnabled(true);
        openChannelButton.setEnabled(true);
        dccButton.setEnabled(true);
        serverSendButton.setEnabled(true);
        mgr.openChannel(DCC_CHAN_NAME);
}

/**
 * ClientConnectionManagerListener callback.
 *
 * Called by the ClientConnectionManager if the login attempt failed.
 *
 * @param message               the reason for failure.
 */
public void connectionRefused(String message) {
        statusMessage.setText("Status: Connection refused. (" + message + ")");
        loginButton.setText("Login");
        loginButton.setEnabled(true);
}

/**
 * ClientConnectionManagerListener callback.
 *
 * Called by ClientConnectionManager when the user is disconnected.
 *
 */
public void disconnected() {
        statusMessage.setText("Status: logged out");
        loginButton.setText("Login");
        loginButton.setEnabled(true);
        openChannelButton.setEnabled(false);
        dccButton.setEnabled(false);
        serverSendButton.setEnabled(false);
}
```

```
	/**
	 * ClientConnectionManagerListener callback.
	 *
	 * Called by the ClientConnectionManager when a new user joins the application.
	 * The new user is added to the user list on the right side.
	 *
	 * @parma userID		the new user
	 */
	public void userJoined(byte[] userID) {
		DefaultListModel mdl = (DefaultListModel) userList.getModel();
		mdl.addElement(new BYTEARRAY(userID));
		userList.repaint();

	}

	/**
	 * ClientConnectionManagerListener callback.
	 *
	 * Called by the ClientConnectionManager when a user leaves the application.
	 * The user is removed from the user list on the right side.
	 *
	 * @param userID		the user that left.
	 */
	public void userLeft(byte[] userID) {
		DefaultListModel mdl = (DefaultListModel) userList.getModel();
		mdl.removeElement(new BYTEARRAY(userID));
		userList.repaint();
	}


	/**
	 * ClientConnectionManagerListener callback.
	 *
	 * Called by the ClientConnectionManager as confirmation that the user
	 * joined the given channel.  If the channel is the DCC channel, a
	 * ClientChannelListener will be attached to receive data from it.
	 *
	 * @param channel		the channel to which the user was joined.
	 */
	public void joinedChannel(ClientChannel channel) {
		if (channel.getName().equals(DCC_CHAN_NAME)) {
			dccChannel = channel;
			dccChannel.setListener(new ClientChannelListener() {

				public void playerJoined(byte[] playerID) {
				}

				public void playerLeft(byte[] playerID) {
				}

				public void dataArrived(byte[] from, ByteBuffer data,
						boolean reliable) {
					byte[] bytes = new byte[data.remaining()];
					data.get(bytes);
					JOptionPane.showMessageDialog(ChatTestClient.this,
							new String(bytes), "Message from "
									+ StringUtils.bytesToHex(from,
											from.length - 4),
							JOptionPane.INFORMATION_MESSAGE);

				}

				public void channelClosed() {
				}
			});
		} else {
			ChatChannelFrame cframe = new ChatChannelFrame(channel);
			desktop.add(cframe);
			desktop.repaint();
		}
	}

	/**
	 * Starting point for the client app.
	 *
	 *
	 * @param args
	 */
	public static void main(String[] args) {
		new ChatTestClient();

	}

	/*
	 * (non-Javadoc)
	 *
	 * @see com.sun.gi.comm.users.client.ClientConnectionManagerListener#failOverInProgress()
	 */
	public void failOverInProgress() {
		// TODO Auto-generated method stub

	}

	/*
```

```
         * (non-Javadoc)
         *
         * @see com.sun.gi.comm.users.client.ClientConnectionManagerListener#reconnected()
         */
        public void reconnected() {
                // TODO Auto-generated method stub

        }

        /**
         * This method is called whenever an attempted join/leave fails due to
         * the target channel being locked.
         *
         * @param channelName           the name of the channel.
         * @param userID                        the ID of the user attemping to join/leave
         */
        public void channelLocked(String channelName, byte[] userID) {
                System.out.println("ChatTestClient received locked notification: " + channelName);
        }

}
```

ChatChannelFrame

```
package com.sun.gi.apps.commtest.client;

import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.nio.ByteBuffer;

import javax.swing.DefaultListModel;
import javax.swing.JInternalFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.ListCellRenderer;
import javax.swing.event.InternalFrameEvent;
import javax.swing.event.InternalFrameListener;

import com.sun.gi.comm.users.client.ClientChannel;
import com.sun.gi.comm.users.client.ClientChannelListener;
import com.sun.gi.utils.types.StringUtils;
import com.sun.gi.utils.types.BYTEARRAY;


/**
 * <p>The ChatChannelFrame presents a GUI so that a user can interact with a channel.  The users connected
 * to the channel are displayed in a list on the right side.  Messages can be sent on the channel via an
 * input area on the left side.</p>
 *
 * <p>This class communicates with its channel by implementing ClientChannelListener, and signing up as
 * a listener on the channel.  As data arrives, and players leave or join, the appropriate call backs are
 * called.</p>
 */

//@SuppressWarnings("serial")
public class ChatChannelFrame extends JInternalFrame implements ClientChannelListener{
        final ClientChannel chan;
        JList userList;
        JTextField inputField;
        JTextArea outputArea;
        ByteBuffer outbuff;

        /**
         * Constructs a new ChatChannelFrame as a wrapper around the given channel.
         *
         * @param channel                the channel that this class will manage.
         */
        public ChatChannelFrame(ClientChannel channel){
                super("Channel: "+channel.getName());
                outbuff = ByteBuffer.allocate(2048);
                chan = channel;
                chan.setListener(this);
                Container c = getContentPane();
                c.setLayout(new BorderLayout());
```

```
		JPanel eastPanel = new JPanel();
		eastPanel.setLayout(new BorderLayout());
		c.add(eastPanel,BorderLayout.EAST);
		eastPanel.add(new JLabel("Users"),BorderLayout.NORTH);
		userList = new JList(new DefaultListModel());
		userList.setCellRenderer(new ListCellRenderer(){
			JLabel text = new JLabel();
			public Component getListCellRendererComponent(JList arg0, Object arg1, int arg2,
boolean arg3, boolean arg4) {
				byte[] data = ((BYTEARRAY)arg1).data();
				text.setText(StringUtils.bytesToHex(data,data.length-4));
				return text;
			}});
		eastPanel.add(new JScrollPane(userList),BorderLayout.CENTER);
		JPanel southPanel = new JPanel();
		c.add(southPanel,BorderLayout.SOUTH);
		southPanel.setLayout(new GridLayout(1,0));
		inputField = new JTextField();
		southPanel.add(inputField);
		outputArea = new JTextArea();
		c.add(new JScrollPane(outputArea),BorderLayout.CENTER);
		inputField.addActionListener(new ActionListener(){
			public void actionPerformed(ActionEvent e) {
				outbuff.clear();
				outbuff.put(inputField.getText().getBytes());
				chan.sendBroadcastData(outbuff,true);
				inputField.setText("");
			}});
		setSize(400,400);
		this.setClosable(true);
		this.setDefaultCloseOperation(JInternalFrame.DISPOSE_ON_CLOSE);
		this.addInternalFrameListener(new InternalFrameListener(){

			public void internalFrameOpened(InternalFrameEvent arg0) {
				// TODO Auto-generated method stub

			}

			public void internalFrameClosing(InternalFrameEvent arg0) {
				// TODO Auto-generated method stub

			}

			public void internalFrameClosed(InternalFrameEvent arg0) {
				chan.close();
			}

			public void internalFrameIconified(InternalFrameEvent arg0) {
				// TODO Auto-generated method stub

			}

			public void internalFrameDeiconified(InternalFrameEvent arg0) {
				// TODO Auto-generated method stub

			}

			public void internalFrameActivated(InternalFrameEvent arg0) {
				// TODO Auto-generated method stub

			}

			public void internalFrameDeactivated(InternalFrameEvent arg0) {
				// TODO Auto-generated method stub

			}});
		setResizable(true);
		setVisible(true);

	}


	/**
	 * A call back from ClientChannelListener.  Called when a player/user joins the channel.
	 * This implementation responds by adding the user to the list.
	 */
	public void playerJoined(byte[] playerID) {
		DefaultListModel mdl = (DefaultListModel)userList.getModel();
		mdl.addElement(new BYTEARRAY(playerID));

	}

	/**
	 * A call back from ClientChannelListener.  Called when a player/user leaves the channel.
	 * This implementation responds by removing the user from the user list.
	 */
	public void playerLeft(byte[] playerID) {
		DefaultListModel mdl = (DefaultListModel)userList.getModel();
```

```
                mdl.removeElement(new BYTEARRAY(playerID));
        }

        /**
         * A call back from ClientChannelListener.  Called when data arrives on the channel.
         * This implementation simply dumps the data to the output area as a String in the form of:
         *
         * <User who sent the message>: <Message>
         */
        public void dataArrived(byte[] from, ByteBuffer data, boolean reliable){
                byte[] textb =new byte[data.remaining()];
                data.get(textb);
                outputArea.append(StringUtils.bytesToHex(from,from.length-4)+": "+ new String(textb)+"\n");
        }

        /**
         * Called when the channel is closed.  The frame has no need to exist if the channel is closed,
         * so it removes itself from the parent.
         */
        public void channelClosed() {
                if (getDesktopPane() != null) {
                        getDesktopPane().remove(this);
                }
        }
}
```

ValidatorDialog

```
package com.sun.gi.apps.commtest.client;

import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.Container;
import java.awt.Frame;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.ChoiceCallback;
import javax.security.auth.callback.ConfirmationCallback;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.TextInputCallback;
import javax.security.auth.callback.TextOutputCallback;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JPasswordField;
import javax.swing.JTextField;

/**
 * This class provides a Swing GUI for server validation fulfillment.  When connecting to a server application
 *  via a UserManager, the UserManager will attempt to validate the user based on the applications validation
 *  settings as specified in its deployment descriptor.  In the case of the CommTest application, a name and a
 *  password are required.
 *
 */

//@SuppressWarnings("serial")
public class ValidatorDialog extends JDialog {

        Callback[] callbacks;                                                                   // The array of
javax.security.auth.callbacks.CallBacks

        List<Component> dataFields = new ArrayList<Component>();        // an array of UI components

// that parallel callbacks[] in size and order.

        /**
         * Constructs a new ValidatorDialog.  The dialog iterates through the CallBack array
         * and displays the appropriate UI based on the type of CallBack.  In the case of CommTest,
         * the CallBacks are of type NameCallBack and PasswordCallBack.  This causes both a "username"
         * textfield, and a "password" input field to be rendered on the dialog.
         *
         * @param parent                the dialog's parent frame
         * @param cbs                   an array of CallBacks
         */
```

```
public ValidatorDialog(Frame parent,Callback[] cbs){
        super(parent,"Validation Information Required",true);
        callbacks = cbs;
        Container c = getContentPane();
        JPanel validationPanel = new JPanel();
        validationPanel.setLayout(new GridLayout(2,0));
        c.add(validationPanel,BorderLayout.NORTH);
        JButton validateButton = new JButton("CONTINUE");
        c.add(validateButton,BorderLayout.SOUTH);

        // when pressed, set the data from the UI components to the matching CallBacks.
        validateButton.addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e) {
                        transcribeToCallbacks();
                        ValidatorDialog.this.setVisible(false);
                        ValidatorDialog.this.getParent().remove(ValidatorDialog.this);
                }
        });

        // Iterate through the javax.security.auth.callback.CallBacks
        // and render the appropriate UI accordingly.
        // For each CallBack, the matching UI Component is stored in
        // the dataFields array.  The order is important, as they will
        // be retrieved along side their matching CallBack.
        for(Callback cb : cbs){
                if (cb instanceof ChoiceCallback){
                        ChoiceCallback ccb = (ChoiceCallback)cb;
                        validationPanel.add(new JLabel(ccb.getPrompt()));
                        JComboBox combo = new JComboBox(ccb.getChoices());
                        combo.setSelectedItem(ccb.getDefaultChoice());
                        validationPanel.add(combo);
                        dataFields.add(combo);
                } else if (cb instanceof ConfirmationCallback) {
                        ConfirmationCallback ccb = (ConfirmationCallback)cb;
                        validationPanel.add(new JLabel(ccb.getPrompt()));
                        JComboBox combo = new JComboBox(ccb.getOptions());
                        combo.setSelectedItem(ccb.getDefaultOption());
                        validationPanel.add(combo);
                        dataFields.add(combo);
                } else if (cb instanceof NameCallback){
                        NameCallback ncb= (NameCallback)cb;
                        validationPanel.add(new JLabel(ncb.getPrompt()));
                        JTextField nameField = new JTextField(ncb.getDefaultName());
                        validationPanel.add(nameField);
                        dataFields.add(nameField);
                } else if (cb instanceof PasswordCallback){
                        PasswordCallback ncb= (PasswordCallback)cb;
                        validationPanel.add(new JLabel(ncb.getPrompt()));
                        JPasswordField passwordField = new JPasswordField();
                        validationPanel.add(passwordField);
                        dataFields.add(passwordField);
                } else if (cb instanceof TextInputCallback){
                        TextInputCallback tcb = (TextInputCallback)cb;
                        validationPanel.add(new JLabel(tcb.getPrompt()));
                        JTextField textField = new JTextField(tcb.getDefaultText());
                        validationPanel.add(textField);
                        dataFields.add(textField);
                } else if (cb instanceof TextOutputCallback ){
                        TextOutputCallback tcb = (TextOutputCallback)cb;
                        validationPanel.add(new JLabel(tcb.getMessage()));

                }
        }
        pack();
        setVisible(true);
}

/**
 * Called when the validation button is pressed.  It iterates through the array of CallBacks
 * and takes the data from the matching UI component and sets it in the CallBack.
 *
 */
protected void transcribeToCallbacks() {
        Iterator iter = dataFields.iterator();
        for(Callback cb : callbacks){
                if (cb instanceof ChoiceCallback){
                        //note this is really wrong, should allow for multiple select
                        ChoiceCallback ccb = (ChoiceCallback)cb;
                        JComboBox combo = (JComboBox) iter.next();
                        ccb.setSelectedIndex(combo.getSelectedIndex());
                } else if (cb instanceof ConfirmationCallback) {
                        ConfirmationCallback ccb = (ConfirmationCallback)cb;
                        JComboBox combo = (JComboBox) iter.next();
                        ccb.setSelectedIndex(combo.getSelectedIndex());
                } else if (cb instanceof NameCallback){
                        NameCallback ncb= (NameCallback)cb;
                        JTextField nameField = (JTextField)iter.next();
                        ncb.setName(nameField.getText());
                } else if (cb instanceof PasswordCallback){
                        PasswordCallback ncb= (PasswordCallback)cb;
                        JPasswordField passwordField = (JPasswordField)iter.next();
                        ncb.setPassword(passwordField.getPassword());
                } else if (cb instanceof TextInputCallback){
                        TextInputCallback tcb = (TextInputCallback)cb;
                        JTextField textField = (JTextField)iter.next();
```

```
                                tcb.setText(textField.getText());
                        } else if (cb instanceof TextOutputCallback ){
                                // no response required
                        }
                }

        }
}
```