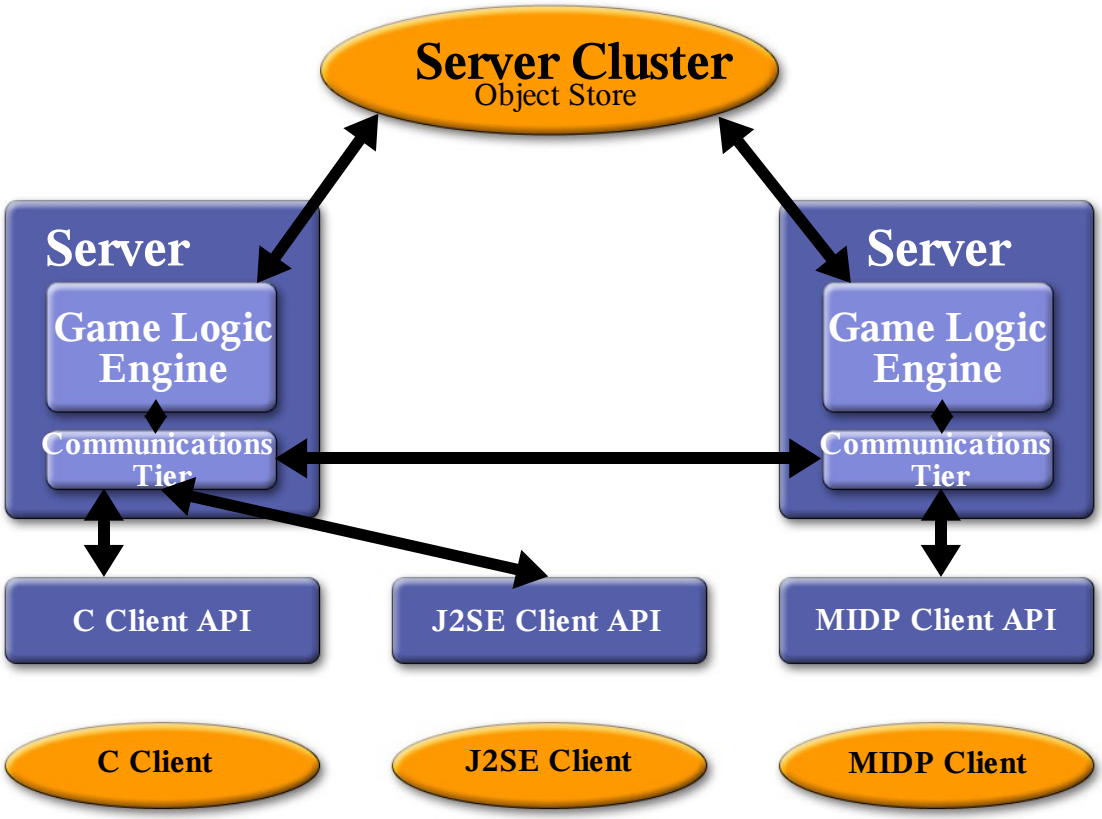
Sun Game Server
J2ME Client API
Programmer's Guide

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.

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 fail-over to a new server should the one it is currently connected to suddenly fail.

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

More on the Communications Tier

The communications tier contains the following main sections:

- 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 user-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 **JMEDiscoverer** 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 **JMEDiscoverer** is the only one supported by the SGS back end for J2ME applications. 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 user managers is user-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 J2ME client API. It assumes that there is a server-side application with which the client communicates.

Overview of the J2ME Client API to the Sun Game Server

Note: Reference information on the classes discussed here is located in the **J2ME API** Javadocs directory.

The J2ME 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 MIDlet implements the API.

The JMEClientManager and the JMEClientListenerInterface

A J2ME SGS client application is a remote program that communicates with the SGS back end to do interclient communication and to interface with SGS server applications. The client application communicates with the server application and with other clients through an instance of the **JMEClientManager** class. A single instance represents a single user of a single SGS server application. A client application can create a single **JMEClientManager** instance to log on to a single server application.

The client application performs outgoing communication (that is, login, data packets, and so on) by making calls on the **JMEClientManager**'s API.

Incoming data is translated to events that are sent to the client application via a **JMEClientListenerInterface.** The client application must register an instance of the **JMEClientListenerInterface** (most likely itself) with the **JMEClientManager**.

The User Manager

The **JMEClientManager** 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 J2ME applications, **JMEHttpUserManagerClient.** Others could be added in the future, or game developers could add their own. This is a connectionless, polling protocol. This means that client applications cannot assume a persistent connection to the server, and certain aspects of the API available in the J2SE version (e.g., reconnecting) are not available for J2ME clients.

A MIDlet that uses the SGS client library must specify the name of the user manager class it will be using by adding an attribute in its JAD file:

```
UsrMgrClassName: com.sun.gi.comm.users.client.impl.JMEHttpUserManagerClient
```

By default, **JMEHttpUserManagerClient** polls for events from the server, and sends client events to the server, every 500 milliseconds (that is, twice per second). This is configurable per game. Developers should be aware of polling frequency when designing client applications, since this will affect game performance.

Discovering Games

The client application must begin by finding its game and user manager on the server. As discussed in the previous section, this information is provided by a Discoverer. The one currently supported Discoverer for J2ME, **JMEDiscoverer**, gets this information from a *discovery file.*

A MIDlet that uses the SGS client library must specify the location of this file by adding an attribute in its JAD file. For example:

```
XML-Discovery-URL: http://localhost:8080/FakeDiscovery.xml
```

The Discovery File

The discovery file 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 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** would be used for J2SE and C++ clients, **J2MEHttpUserManagerClient** would be 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 in 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 HTTPUserManagers for different games all share a single "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 get a dynamically generated discovery file. The URL of this web server would be provided by the SGS host.

Parsing the Discovery Document

Client applications should use the **JMEDiscoverer** class provided in the API to parse the discovery document and return games, user managers, and parameters found in the document. Currently, XML-based discovery from a known URL is the only discovery strategy provided; however, the API is pluggable so that other strategies could be developed for different network architectures.

Connecting and Logging in to the Server

As noted above, the current J2ME API includes only one user manager, **JMEHttpUserManagerClient**. Since this is a connectionless, polling protocol, "connecting" is equivalent to logging in 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 **JMEClientManager**. The **JMEClientManager** 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 **JMEClient Manager** to connect to a User Manager of that type.

When the **JMEClientManager** connects to a game, it requests the 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 **JMEClientManager** 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.

Validation

The **JMEClientManager** starts the login process by sending a login request to the user manager.

A game typically requires validation in order to accept a connection from an instance of its **UserManagerClient** class.The user manager responds to a login request by returning an array of **Calllback** objects to the client application through one or more **validationDataRequest** messages that are delivered to the **JMEClientListenerInterface** object.

A **JMEClientListener** object responds to this event by calling the **JMEClientManager** method **sendValidationResponse** with the appropriately filled-in callbacks.

This process continues until either the response is rejected by the user manager (in which case the listener receives a **connectionRejected** message containing a string giving the reason for refusal), or all responses have been accepted, in which case the listener receives a **connectionAccepted** message containing the assigned userID 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 the server doesn't require validation, the validation process is skipped and the listener immediately receives the **connectionAccepted** message.

See the discussion of the sample client, **JMEChatClient**, for an example of how validation works.

Logoff

A client application requests that a **JMEClientManager** disconnect from the server by calling the **disconnect** method on the **JMEClientManager**'s API. At this point it stops polling and will receive no more messages from the server. Other clients connected to the game receive a message that the user logged out, as described below.

Game Events

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

- **dataReceived**
  This listener method gets called whenever an inbound data packet arrives that was sent to a channel to which this user subscribes. See below for a discussion of channels.
- **userLoggedIn**
  This method gets called once for every **JMEClientManager** (that is, every user) that is connected to the game that this **JMEClientManager** joined. It then gets called periodically as new users join the game.

- **userLoggedOut**
  This method gets called when any **JMEClientManager** whose connection was reported by **userLoggedIn** logs off.

The **JMEClientManager** sends a message to the server via the **sendServerMessage** method.

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 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.

The **JMEClientManager** defines SGS channels for use by a client application, and the **JMEClientListenerInterface** defines how channel events are reported back to the client.

Clients can send messages to a specific user of the channel, multicast messages to multiple users, and broadcast messages to all users of the channel. In addition, the user receives **userJoinedChannel** and **userLeftChannel** messages when other users join or leave a channel.

As discussed in the first section, messages sent between clients only go up through Tier 1 (the 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 J2ME Client Application Needs to Do

A J2ME SGS client application at a minimum needs to do the following to interact with the server:

- Implement the **JMEClientListenerInterface** so that it can receive events from the server.

- Create a Discoverer to find the discovery information. In practice today this means instancing the **JMEDiscoverer** and giving it the URL location of an XML discovery file.

- Create a UserManagerPolicy to help the **JMEClientManager** decide which of the various possible connection points to user. Currently the API provides one UserManagerPolicy, the **DefaultUserManagerPolicy**.

- Request a list of the types of connections available for the game we wish to log in 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.
- Sending data and respond to data that is received.
- Provide a means of logging off from the server application.

The following section walks through a very simple client application, **JMEChatClient**, that does these things.

The JMEChatClient Example Client

The example program **ChatTest** is a basic chat program implemented using the Sun Game Server and provided with the SGS distribution. It allows users to log in to the game, send messages to the server and to each other over a DCC (Direct Client to Client) channel, and choose other channels over which to send and receive messages.

JMEChatClient is a J2ME MIDlet for this server application. It shows, in simplified fashion, how to use the SGS J2ME client API. In summary:

- It uses the **JMEDiscoverer** constructor to parse the discovery file and provide discovery information back to the rest of the API
- It uses the **JMEDiscoverer** class to discover the available games and their user managers.
- It calls methods in the **JMEClientManager** class to communicate with the user manager (for example, to log into and out of the game, to set up channels, and to send messages to the server and to other game users).
- It implements the **JMEClientListenerInterface** to receive events and messages from the server. These are sent to the client by the user manager via the **JMEClientManager**.
- It uses various utilities provided in the API (for example, **NameCallback** and **PasswordCallback**) to handle name and password callbacks.

Source code for **JMEChatClient** is included in the appendix.

In our discussion of **JMEChatClient**, we focus on the interactions with the server via the API; we assume the reader is familiar with the details of Java programming and, in particular, with programming a J2ME user interface.

Command Actions

This code in **JMEChatClient** shows the methods called in response to various user actions on the displayable. Following sections will go into more detail about these methods.

```
public void commandAction(Command command, Displayable displayable) {
        if (displayable == loginForm) {
            if (command == exitCommand) {
                doLogout();
                exitMIDlet();
            } else if (command == loginCommand) {
                initializeClientManager();
            }
        } else if (displayable == postLoginForm) {
            if (command == channelOk) {
                clientMgr.openChannel(channelNameField.getString());
            } else if (command == logoutCommand) {
                doLogout();
                getDisplay().setCurrent(loginForm);
            } else if (command == sendCommand) {
                sendMessageToUser();
```

```
            } else if (command == channelSend) {
                sendChannelMessage();
            } else if (command == serverMessage) {
                sendServerMessage();
            }
        }
    }
```

Logging in to the Server Application

As discussed above, a client must begin by connecting to the server application. Since **JMEChatClient** is using a connectionless, polling protocol (**JMEHttpUserManagerClient**), this is equivalent to logging in to the server.

Launching the Application

This process begins when the user presses **Launch** to launch **JMEChatClient**, as shown below.

Logging In

JMEChatClient then displays the login screen, as shown below.

Finding the Game

When the user enters the **Userid** and **Password** and presses **Ok**, **JMEChatClient** executes the **initializeClientManager** method to start an instance of **JMEClientManager** for this user and begin the process of game discovery:

```
private void initializeClientManager() {
      clientMgr = new JMEClientManager("ChatTest",
             new JMEDiscoverer(this.getAppProperty("XML-Discovery-URL")),
             this.getAppProperty("UsrMgrClassName"));
      clientMgr.setListener(this);
      clientMgr.discoverGames();
```

JMEChatClient communicates with the server application via the **JMEClientManager** interface. It needs to tell **JMEClientManager** the name of the server application and how to find it. So it passes the following:

- The name of the server application, **ChatTest.**

- A **JMEDiscoverer** that is set to parse the local discovery file (whose location is specified by the attribute **XML-Discovery-URL**) that lists all available games and the user managers for them. **JMEDiscoverer** parses the file to determine if **ChatTest** is there and, if so, what User Managers are supported at what connection points.

- The user manager to use with the game; its class is specifed by the attribute **UsrMgrClassName**.

The actual values of **XML-Discovery-URL** and **UsrMgrClassName** are specified as attributes in the JAD file for the MIDlet, as mentioned in the previous section. Here are the contents of the **JMEChatClient** JAD file:

```
MIDlet-1: JMEChatClient, , com.sun.gi.apps.commtest.client.JMEChatClient
MIDlet-Jar-Size: 42804
MIDlet-Jar-URL: JMEChatClient.jar
MIDlet-Name: JMEChatClient
MIDlet-Vendor: Vendor
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.0
UsrMgrClassName: com.sun.gi.comm.users.client.impl.JMEHttpUserManagerClient
XML-Discovery-URL: http://localhost:8080/FakeDiscovery.xml
```

See the previous section for sample contents of the **FakeDiscovery.xml** document.

As part of establishing the connection, **JMEChatClient** uses the **setListener** method from **JMEClientManager** to establish itself as the listener for events that the server sends to **JMEClientListenerInterface** via **JMEClientManager**.

Finally, it uses the **JMEClientManager** method **discoverGames** to determine if any of the discovered games is **ChatTest**.

JMEChatClient uses the callback **discoveredGames** from **JMEClientListenerInterface** to find out if **JMEDiscoverer** was able to find the **ChatTest** game. If so, it uses the **JMEClientManager** method **connect** to connect to the game:

```
public void discoveredGames() {
        try {
            clientMgr.connect();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
```

Specifying User Name and Password

The validation strategy is up to the server application; **ChatTest** requires a name and password. The user manager informs the **JMEClientListenerInterface** via the **validationDataRequest** callback that it needs validation information.

JMEChatClient uses the API utilities **NameCallback** and **PasswordCallback** to package up the login information provided by the user:

```
   private void dealWithCallbacks(Callback[] callbacks) {
        //for now hard code dealing with the login callback
        NameCallback nm = (NameCallback)callbacks[0];
        nm.setName(userNameField.getString());
        PasswordCallback pb = (PasswordCallback)callbacks[1];
        pb.setPassword(passwordField.getString().toCharArray());
    }
```

JMEChatClient sends this information to to the server application via the **JMEClientManager** method **sendValidationResponse**, sent in response to the **validationDataRequest** callback:

```
    public void validationDataRequest(Callback[] callbacks) {
        dealWithCallbacks(callbacks);
        try {
            clientMgr.sendValidationResponse(callbacks);
        } catch (UnsupportedCallbackException ex) {
            exceptionOccurred(ex);
        }
    }
```

JMEChatClient uses the API class **UnsupportedCallbackException** to handle the case in which it doesn't recognize the callback from the server. This causes it to call the **exceptionOccurred** method to display a message on the displayable:

```
    public void exceptionOccurred(Exception ex) {
        Displayable newScreen = changeState(EXCEPTION_OCCURRED);
        getDisplay().setCurrent(getGenericAlert("Exception occurred " + ex.getMessage()), newScreen);
    }
```

If the Login Is Accepted

If the server accepts the login, it returns a userID for this user via the **loginAccepted** callback in the **JMEClientListenerInterface**:

```
    public void loginAccepted(byte[] userID) {
        clientMgr.openChannel(DCC_CHAN_NAME);
        Displayable newScreen = changeState(LOGON);
        myUserIDItem.setText(StringUtils.bytesToHex(userID));
        getDisplay().setCurrent(getGenericAlert("Login Successful for " +
StringUtils.bytesToHex(userID)), newScreen);
    }
```

Note that, as part of **loginAccepted**, **JMEChatClient** calls the **JMEClientManager** method **openChannel** to open **DCC_CHAN_NAME**. This opens a DCC (Direct Client-to-Client) channel for communication between users. A user can also open other channels, as described in the next section.

JMEChatClient also displays a new screen, showing the user's userID for the game, as shown below.

Note that user IDs are really session IDs in that they change from login to login. They stay the same during failover, however.

If the Login Is Rejected

If the login is rejected, the server sends a **loginRejected** event to the **JMEClientListernerInterface**, along with a message explaining the rejection. **JMEChatClient** updates the user interface accordingly:

```
public void loginRejected(String message) {
    Displayable newScreen = changeState(LOGON_REJECT);
    getDisplay().setCurrent(getGenericAlert(message), newScreen);
}
```

After Login

When the user presses **Done** after receiving the successful login message, **JMEChatClient** displays the post-login form. Pressing **Menu** from this screen displays a menu that offers three choices:

- **OpenChannel** – This opens a channel for communication with other users.

- **Send message** – This lets the user send a message to one or more other users.
- **Server Message** – This lets the user send a message to the server.

See the figure below.

Displaying Users

Another User Logs in to the Game

When a user first logs in to **ChatTest**, the server sends **userLoggedIn** callbacks to **JMEClientListenerInterface** for every other user currently logged in to the game.

JMEChatClient then adds the userIDs of each of these users to the list on the screen. As other users join, additional callbacks are issued for them.

Here is the code that handles this in **JMEChatClient**:

```
public void userLoggedIn(byte[] userID) {
        String userName = StringUtils.bytesToHex(userID);
        users.put(userName,userID);
        usersList.append(userName,null);
        Displayable newScreen = changeState(USER_LOGGED_IN);
        getDisplay().setCurrent(getGenericAlert("User added " + userName), newScreen);
    }
```

A User Logs out of the Game

If a user leaves the game, the server sends an event to the **userLoggedOut** callback in **JMEClientListenerInterface**. In response, **JMEChatClient** removes this user's userID from the list on the screen:

```
public void userLoggedOut(byte[] userID) {
        Displayable newScreen = changeState(USER_LOGGED_OUT);
        usersList.deleteAll();
        String userName = StringUtils.bytesToHex(userID);
        users.remove(userName);
        Enumeration userIds = users.keys();
        while (userIds.hasMoreElements()) {
            usersList.append((String) userIds.nextElement(), null);
        }
        getDisplay().setCurrent(getGenericAlert("User dropped " + StringUtils.bytesToHex(userID)),
newScreen);
}
```

Opening a Channel

There are two kinds of channels in **ChatTest**: the DCC (Direct Client-to-Client) channel and user-specified channels.

The DCC channel is defined by the application. It is used for sending messages to individual users or groups of users, and it was opened when the user's login was accepted, as discussed above.

User-specified channels can have any name. They can be used to send messages to any other users who have joined the same channel. In **ChatTest**, a user can open a channel by choosing **1** from the menu.

This code in **JMEChatClient**'s **commandAction** method invokes the **openChannel** method from **JMEClientManager** to open the channel that the user specified:

```
            if (command == channelOk) {
               clientMgr.openChannel(channelNameField.getString());
```

The server sends a **joinedChannel** event to **JMEClientListenerInterface** to confirm that the user has joined the specified channel. In response, **JMEChatClient** updates the screen and displays a “Connected to Channel” message:

```
public void joinedChannel(String name, byte[] channelID) {
       if (name.equals(DCC_CHAN_NAME)) {
           dccChannelId = channelID;
       } else {
           channels.put(name,channelID);
```

```
            Displayable newScreen = changeState(JOINED_CHAN);
            channelList.append(name,null);
            getDisplay().setCurrent(getGenericAlert("Connected to Channel " + name), newScreen);
            this.channelID = channelID;
        }
    }
```

Once a user has joined a channel, the server sends events concerning that channel to this client instance.

Another User Joins an Open Channel

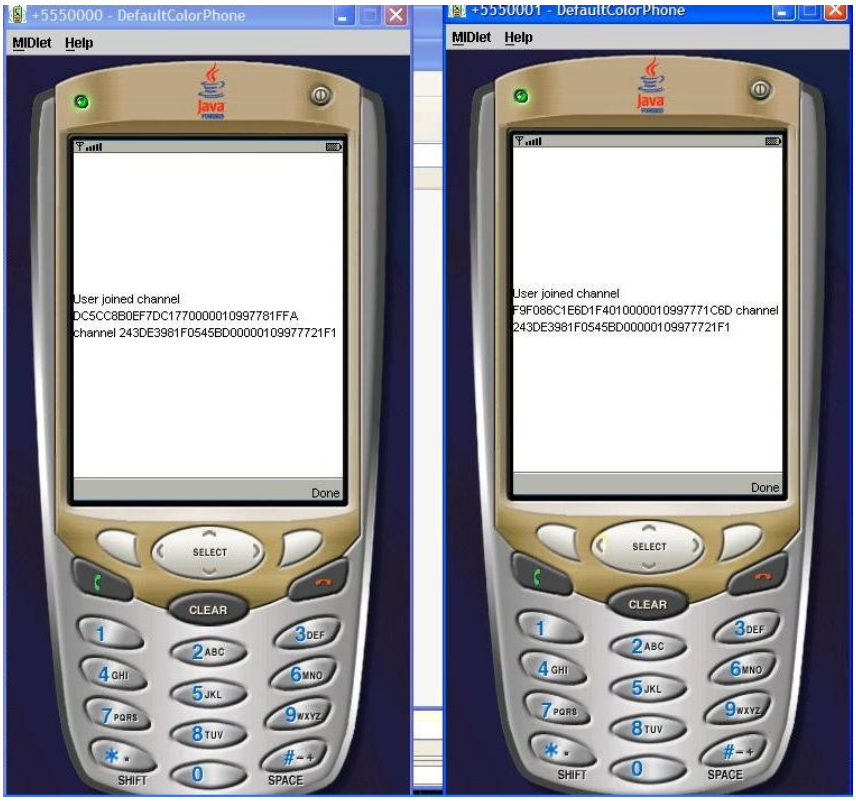
When a user first opens a channel, the server sends an event to the **JMEClientListenerInterface** callback **userJoinedChannel** for every other user who currently has that channel open. In response, **JMEChatClient** displays a message and adds the userID of each of these users to the list on the screen. As other users join, additional callbacks are issued for them.

Here is the code in **JMEChatClient**:

```
public void userJoinedChannel(byte[] channelID, byte[] userID) {
        Displayable newScreen = changeState(USER_JOINED_CHAN);
        getDisplay().setCurrent(getGenericAlert("User joined channel " +
StringUtils.bytesToHex(userID) + " channel " + StringUtils.bytesToHex(channelID)), newScreen);

    }
```

And here is the screen display:



Another User Leaves an Open Channel

If another user leaves a channel that this user has open, the server sends an event to the **JMEClientListenerInterface** callback **userLeftChannel**. **JMEChatClient** displays a message and removes this user's userID from the list on the screen:

```
public void userLeftChannel(byte[] channelID, byte[] userID) {
       Displayable newScreen = changeState(USER_LEFT_CHAN);
       getDisplay().setCurrent(getGenericAlert("User left channel " + StringUtils.bytesToHex(userID)
+ " channel " + StringUtils.bytesToHex(channelID)), newScreen);

    }
```

This User Leaves an Open Channel

If the server removes this user from a channel, it sends an event to the **JMEClientListenerInterface** callback **leftChannel**. **JMEChatClient** displays a message and updates the screen:

```
public void leftChannel(byte[] channelID) {
       Displayable newScreen = changeState(JOINED_CHAN);
       getDisplay().setCurrent(getGenericAlert("Left channel "+ StringUtils.bytesToHex(channelID)),
newScreen);
    }
```

Sending and Receiving Messages

As discussed above, **JMEChatClient** provides ways of sending messages to and receiving messages from the server and other users.

Sending a Message to the Server

In the **JMEChatClient** application, the user sends a message to the server application by typing the message in the message field and choosing **3** from the menu. **JMEChatClient** then calls the **JMEClientManager** method **SendServerMessage** to send the message to the server application:

```
private void sendServerMessage() {
       String dataString = messageField.getString();
       ByteBuffer data = ByteBuffer.wrap(dataString.getBytes());
       clientMgr.sendServerMessage(data);
       messageField.setString("");
    }
```

In **ChatTest**, the server simply prints the message sent to it. The functionality is included in this program to show how a client application would send data to the server.

Sending a Message to Specific Users

If the user types a message in the message field and chooses **2** from the menu, the message will be sent to one or more users, or to all users of a channel. Messages for specific users are sent over the special DCC channel. The user selects the users by choosing their checkboxes on the screen.

JMEChatClient calls the **sendMessageToUser** method to determine how many users have been selected. If there is a single user, this method calls the **JMEClientManager** method **sendUnicastData**. If there are multiple users, it calls the method **sendMulticastData**.

```
 private void sendMessageToUser() {
        boolean [] selected = new boolean[users.size()];
        usersList.getSelectedFlags(selected);
        Vector selectedUsers = new Vector();
        for (int i = 0;i < selected.length;i++) {
            if (selected[i]) {
                selectedUsers.addElement(usersList.getString(i));
            }
        }
        if (selectedUsers.size() == 1) {
            String message = messageField.getString();
            //byte[] user = getSelectedUser((String) selectedUsers.firstElement());
            byte[] user = (byte[]) users.get(selectedUsers.firstElement());
            clientMgr.sendUnicastMsg(dccChannelId,user, ByteBuffer.wrap(message.getBytes()));
        } else if (selectedUsers.size() > 1) {
            byte[][] targetList = getSelectedUsers(selectedUsers);
            clientMgr.sendMulticastData(dccChannelId,targetList,ByteBuffer.wrap(messageField.getString
().getBytes()));
        }
        messageField.setString("");
    }
```

Sending a Message to All Users of a Channel

If the message is to go to all users of a channel, **JMEChatClient** calls the method **sendChannelMessage** to get the text of the message field and the name of the channel from the user input. The method then calls the **JMEClientManager** method **sendBroadcastMessage** to actually send the data on the specified channel. Here is the code from **JMEChatClient**:

```
private void sendChannelMessage() {
        String dataString = messageField.getString();
        String channelName = channelList.getString(channelList.getSelectedIndex());
        ByteBuffer data = ByteBuffer.wrap(dataString.getBytes());
        clientMgr.sendBroadcastMessage((byte[]) channels.get(channelName), data);
        messageField.setString("");
    }
```

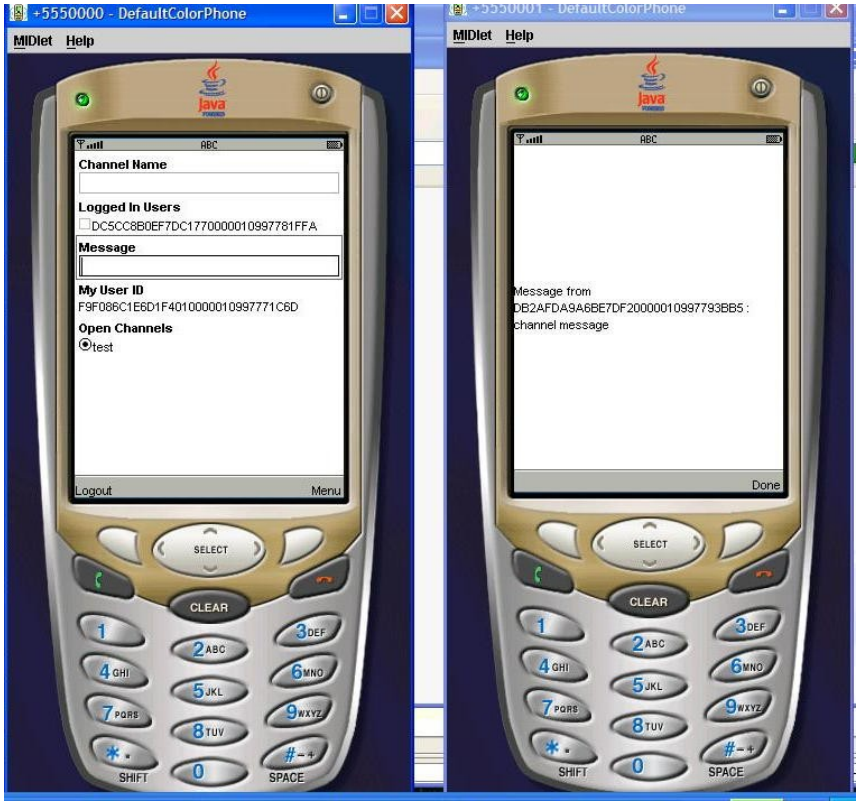
Receiving Data

When data is received for this user, the server informs **JMEClientListenerInterface** via the **dataReceived** callback, specifying the channel it arrived on, the userID of the user who sent it, and the data itself. In response, **JMEChatClient** updates the screen and displays the data.

Here is the code:

```
public void dataReceived(byte[] chanID, byte[] from, ByteBuffer data) {
        Displayable newScreen = changeState(USER_LEFT_CHAN);
        byte[] textb = new byte[data.remaining()];
        data.get(textb);
        getDisplay().setCurrent(getGenericAlert("Message from " + StringUtils.bytesToHex(chanID) + " :
" + new String(textb)), newScreen);
    }
```

The screenshot below shows the message “channel message” arriving on a channel:



Logging Out

When the user presses **Logout**, **JMEChatClient** calls the **doLogout** method. This method calls the **JavaClientManager** method **disconnect** to log out from the server. (Since we are using a connectionless protocol, logging out is equivalent to disconnecting.) Here is **doLogout**:

```
private void doLogout() {
        clientMgr.disconnect();
        users.clear();
        createPostLogonScreen();
        channels = new Hashtable();
        currentState = LOGGED_OUT;
    }
```

When the user has logged out, **JMEChatClient** can exit the MIDlet:

```
public void exitMIDlet() {
        getDisplay().setCurrent(null);
        destroyApp(true);
        notifyDestroyed();
    }
```

Appendix: Source Code for JMEChatClient

```
/*
 * JMEChatClient.java
 *
 * Created on February 14, 2006, 8:25 AM
 */

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

import com.sun.gi.comm.users.client.JMEClientListenerInterface;
import com.sun.gi.comm.users.client.impl.JMEJMEClientManager;
import com.sun.gi.comm.discovery.impl.JMEDiscoverer;
import com.sun.gi.utils.jme.ByteBuffer;
import com.sun.gi.utils.jme.Callback;
import com.sun.gi.utils.jme.NameCallback;
import com.sun.gi.utils.jme.PasswordCallback;
import com.sun.gi.utils.jme.StringUtils;
import com.sun.gi.utils.jme.UnsupportedCallbackException;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

/**
 *
 * @author as93050
 */
public class JMEChatClient extends MIDlet implements CommandListener, JMEClientListenerInterface {

    private JMEJMEClientManager clientMgr;
    private Command logoutCommand;
    private Form loginForm;
    private Command exitCommand;
    private TextField userNameField;
    private TextField passwordField;
    private Command loginCommand;
    private Form postLoginForm;
    private TextField channelNameField;
    private StringItem myUserIDItem;
    private Command channelOk;
    private Command channelSend;
    private Command serverMessage;
    private Hashtable users;
    private Hashtable channels;
    private Alert genericAlert;
    private Command sendCommand;
    private Displayable shouldBeCurrent;
    private ChoiceGroup usersList;
    private TextField messageField;
    private ChoiceGroup channelList;

    private byte[] channelID;
    private String userid;
    private static String DCC_CHAN_NAME = "__DCC_Chan";
    private byte[] dccChannelId;
    private int currentState = LOGGED_OUT;

    private final static int LOGGED_OUT = 0;
    private final static int LOGGED_IN = 1;
    private final static int OPENED_CHANNEL = 2;

    private final static int LOGON = 0;
    private final static int LOGON_REJECT = 1;
    private final static int CHANNEL_OPENED = 2;
    private final static int SENT_MESSAGE = 3;
    private final static int LOGOUT = 4;
    private final static int USER_LOGGED_IN = 5;
    private final static int USER_LOGGED_OUT = 6;
    private final static int USER_JOINED_CHAN = 7;
    private final static int USER_LEFT_CHAN = 8;
    private final static int JOINED_CHAN = 9;
    private final static int EXCEPTION_OCCURRED = 10;
    private final static int START_APP = 11;

    /** Creates a new instance of HelloMidlet */
    public JMEChatClient() {
        users = new Hashtable();
        channels = new Hashtable();
```

```
    }

    /** This method initializes UI of the application.
     */
    private void initialize() {
        createGuiComponents();
        getDisplay().setCurrent(changeState(START_APP));
        shouldBeCurrent = getDisplay().getCurrent();
    }

    /** Called by the system to indicate that a command has been invoked on a particular displayable.
     * @param command the Command that ws invoked
     * @param displayable the Displayable on which the command was invoked
     */
    public void commandAction(Command command, Displayable displayable) {
        if (displayable == loginForm) {
            if (command == exitCommand) {
                doLogout();
                exitMIDlet();
            } else if (command == loginCommand) {
                initializeJMEClientManager();
            }
        } else if (displayable == postLoginForm) {
            if (command == channelOk) {
                clientMgr.openChannel(channelNameField.getString());
            } else if (command == logoutCommand) {
                doLogout();
                getDisplay().setCurrent(loginForm);
            } else if (command == sendCommand) {
                sendMessageToUser();
            } else if (command == channelSend) {
                sendChannelMessage();
            } else if (command == serverMessage) {
                sendServerMessage();
            }
        }
    }

    /**
     * This method should return an instance of the display.
     */
    public Display getDisplay() {
        return Display.getDisplay(this);
    }

    private void createGuiComponents() {
        createLoginForm();
        createPostLogonScreen();

    }
    /**
     * This method should exit the midlet.
     */
    public void exitMIDlet() {
        getDisplay().setCurrent(null);
        destroyApp(true);
        notifyDestroyed();
    }

    /** This method returns instance for LoginForm component and should be called instead of accessing
LoginForm field directly.
     * @return Instance for LoginForm component
     */
    public void createLoginForm() {
        createLoginFormComponents();
        loginForm = new Form("Login Screen", new Item[] {
            userNameField,
            passwordField
        });
        loginForm.addCommand(exitCommand);
        loginForm.addCommand(loginCommand);
        loginForm.setCommandListener(this);
    }

    private void createLoginFormComponents() {
        exitCommand = new Command("Exit", Command.EXIT, 1);
        userNameField = new TextField("Userid", "Ari", 16, TextField.ANY);
        passwordField = new TextField("Password", "password", 16, TextField.PASSWORD);
        loginCommand = new Command("Ok", Command.OK, 1);
    }

    private void createPostLogonScreen() {
        createPostLogonComponents();
        postLoginForm = new Form(null, new Item[] {channelNameField,
        usersList,messageField,myUserIDItem});
        postLoginForm.addCommand(channelOk);
        postLoginForm.addCommand(sendCommand);
        postLoginForm.addCommand(logoutCommand);
```

```
    postLoginForm.addCommand(serverMessage);
    postLoginForm.setCommandListener(this);
}

private void createPostLogonComponents() {
    channelNameField = new TextField("Channel Name", null, 120, TextField.ANY);
    usersList = new ChoiceGroup("Logged In Users",ChoiceGroup.MULTIPLE);
    messageField = new TextField("Message", "", 120, TextField.ANY);
    channelOk = new Command("OpenChannel", Command.OK, 1);
    sendCommand = new Command("Send message", Command.OK, 2);
    logoutCommand = new Command("Logout",Command.CANCEL,2);
    channelSend = new Command("Send to Channel", Command.OK, 1);
    serverMessage = new Command("Server Message", Command.OK,3);
    channelList = new ChoiceGroup("Open Channels",ChoiceGroup.EXCLUSIVE);
    usersList = new ChoiceGroup("Logged In Users",ChoiceGroup.MULTIPLE);
    myUserIDItem = new StringItem("My User ID ","");
}

private void updatePostLoginScreen() {
    postLoginForm.addCommand(channelSend);
    postLoginForm.append(channelList);
}

private Alert getGenericAlert(String alertText) {
    if (genericAlert == null) {
        genericAlert = new Alert(null, alertText, null, AlertType.INFO);
        genericAlert.setTimeout(-2);
    } else {
        genericAlert.setString(alertText);
    }
    return genericAlert;
}

private Displayable changeState(int event) {
    Displayable newScreen = shouldBeCurrent;
    switch (currentState) {
        case LOGGED_OUT:
            if (event == LOGON) {
                currentState = LOGGED_IN;
                newScreen = postLoginForm;
                shouldBeCurrent = newScreen;
            } else if (event == LOGON_REJECT) {
                newScreen = loginForm;
                shouldBeCurrent = newScreen;
            } else if (event == START_APP) {
                newScreen = loginForm;
                shouldBeCurrent = newScreen;
            }
            break;
        case LOGGED_IN:
            if (event == LOGOUT) {
                currentState = LOGGED_OUT;
                newScreen = loginForm;
                shouldBeCurrent = newScreen;
            } else if (event == JOINED_CHAN) {
                currentState = CHANNEL_OPENED;
                updatePostLoginScreen();
                newScreen = postLoginForm;
                shouldBeCurrent = newScreen;
            } else if (event == START_APP) {
                newScreen = postLoginForm;
                shouldBeCurrent = newScreen;
            }
        case OPENED_CHANNEL:
            if (event == START_APP) {
                currentState = CHANNEL_OPENED;
                updatePostLoginScreen();
                newScreen = postLoginForm;
                shouldBeCurrent = newScreen;
            }
        default:

    }
    return newScreen;
}

public void startApp() {
    initialize();
}

public void pauseApp() {
    clientMgr.stopPolling();
}
```

```
    public void destroyApp(boolean unconditional) {
    }

    /**
     * This event informs us that further information is
     * needed in order to validate the user.
     * @param cbs An array of Callback structures to be filled out and sent back.
     */
    public void validationDataRequest(Callback[] callbacks) {
        dealWithCallbacks(callbacks);
        try {
            clientMgr.sendValidationResponse(callbacks);
        } catch (UnsupportedCallbackException ex) {
            exceptionOccurred(ex);
        }
    }

    /**
     * This event informs us that the user has successfully logged in
     *
     * @param userID An ID issued to represent this user.
     * <b>UserIDs are not guaranteed to remain the same between logons.</b>
     */
    public void loginAccepted(byte[] userID) {
        clientMgr.openChannel(DCC_CHAN_NAME);
        Displayable newScreen = changeState(LOGON);
        myUserIDItem.setText(StringUtils.bytesToHex(userID));
        getDisplay().setCurrent(getGenericAlert("Login Successful for " + StringUtils.bytesToHex(userID)),
newScreen);
    }

    /**
     * This event informs us that user validation has failed.
     * @param message A string containing an explaination of the failure.
     */
    public void loginRejected(String message) {
        Displayable newScreen = changeState(LOGON_REJECT);
        getDisplay().setCurrent(getGenericAlert(message), newScreen);
    }

    /**
     * This event informs us about other users of the game.
     * When logon first succeeds there will be one of these callabcks sent for every currently
     * logged-in user of this game.  As other users join, additional callabcks will be issued for them.
     * @param userID The ID of the other user
     */
    public void userLoggedIn(byte[] userID) {
        String userName = StringUtils.bytesToHex(userID);
        users.put(userName,userID);
        usersList.append(userName,null);
        Displayable newScreen = changeState(USER_LOGGED_IN);
        getDisplay().setCurrent(getGenericAlert("User added " + userName), newScreen);
    }

    /**
     * This event informs us that a user has logged out of the game.
     * @param userID The user that logged out.
     */
    public void userLoggedOut(byte[] userID) {
        Displayable newScreen = changeState(USER_LOGGED_OUT);
        usersList.deleteAll();
        String userName = StringUtils.bytesToHex(userID);
        users.remove(userName);
        Enumeration userIds = users.keys();
        while (userIds.hasMoreElements()) {
            usersList.append((String) userIds.nextElement(), null);
        }
        getDisplay().setCurrent(getGenericAlert("User dropped " + StringUtils.bytesToHex(userID)), newScreen);
    }

    /**
     * This event informs us of the successful opening of a channel.
     * @param channelID The ID of the newly joined channel
     */
    public void joinedChannel(String name, byte[] channelID) {
        if (name.equals(DCC_CHAN_NAME)) {
            dccChannelId = channelID;
        } else {
            channels.put(name,channelID);
            Displayable newScreen = changeState(JOINED_CHAN);
            channelList.append(name,null);
            getDisplay().setCurrent(getGenericAlert("Connected to Channel " + name), newScreen);
            this.channelID = channelID;
        }
    }

    /**
     * This callback informs the user that they have left or been removed from a channel.
```

```
     * @param channelID The ID of the channel left.
     */
    public void leftChannel(byte[] channelID) {
        Displayable newScreen = changeState(JOINED_CHAN);
        getDisplay().setCurrent(getGenericAlert("Left channel "+ StringUtils.bytesToHex(channelID)),
newScreen);
    }

    /**
     * This method is called whenever a new user joins a channel that we have open.
     * A set of these events are sent when we first join a channel-- one for each pre-existing
     * channel mamber.  After that we get thsi event whenever someone new joins the channel.
     * @param channelID The ID of the channel joined
     * @param userID The ID of the user who joined the channel
     */
    public void userJoinedChannel(byte[] channelID, byte[] userID) {
        Displayable newScreen = changeState(USER_JOINED_CHAN);
        getDisplay().setCurrent(getGenericAlert("User joined channel " + StringUtils.bytesToHex(userID) + "
channel " + StringUtils.bytesToHex(channelID)), newScreen);

    }

    /**
     * This method is called whenever another user leaves a channel that we have open.
     * @param channelID The ID of the channel left
     * @param userID The ID of the user leaving the channel
     */
    public void userLeftChannel(byte[] channelID, byte[] userID) {
        Displayable newScreen = changeState(USER_LEFT_CHAN);
        getDisplay().setCurrent(getGenericAlert("User left channel " + StringUtils.bytesToHex(userID) + "
channel " + StringUtils.bytesToHex(channelID)), newScreen);

    }

    /**
     * This event informs the listener that data has arrived from the Darkstar server channels.
     * @param chanID The ID of the channel on which the data has been received
     * @param from The ID of the sender of the data
     * @param data The data itself
     */
    public void dataReceived(byte[] chanID, byte[] from, ByteBuffer data) {
        Displayable newScreen = changeState(USER_LEFT_CHAN);
        byte[] textb = new byte[data.remaining()];
        data.get(textb);
        getDisplay().setCurrent(getGenericAlert("Message from " + StringUtils.bytesToHex(chanID) + " : " + new
String(textb)), newScreen);
    }

    public void recvServerID(byte[] user) {
    }

    /**
     * Send a message to all the suers on a channel
     */
    private void sendChannelMessage() {
        String dataString = messageField.getString();
        String channelName = channelList.getString(channelList.getSelectedIndex());
        ByteBuffer data = ByteBuffer.wrap(dataString.getBytes());
        clientMgr.sendBroadcastMessage((byte[]) channels.get(channelName), data);
        messageField.setString("");
    }

    private byte[][] getSelectedUsers(Vector selectedUsers) {
        byte[][] targetList = new byte[selectedUsers.size()][];
        for (int i = 0;i < selectedUsers.size();i++) {
            targetList[i] = (byte[]) users.get(selectedUsers.elementAt(i));
        }


        return targetList;
    }
    /**
     * Send a message to a specific user or group of users
     * We use the special DCC_CHANNEL for this
     */
    private void sendMessageToUser() {
        boolean [] selected = new boolean[users.size()];
        usersList.getSelectedFlags(selected);
        Vector selectedUsers = new Vector();
        for (int i = 0;i < selected.length;i++) {
            if (selected[i]) {
                selectedUsers.addElement(usersList.getString(i));
            }
        }
```

```
        if (selectedUsers.size() == 1) {
            String message = messageField.getString();
            //byte[] user = getSelectedUser((String) selectedUsers.firstElement());
            byte[] user = (byte[]) users.get(selectedUsers.firstElement());
            clientMgr.sendUnicastMsg(dccChannelId,user, ByteBuffer.wrap(message.getBytes()));
        } else if (selectedUsers.size() > 1) {
            byte[][] targetList = getSelectedUsers(selectedUsers);
            clientMgr.sendMulticastData(dccChannelId,targetList,ByteBuffer.wrap(messageField.getString().getBy
tes()));
        }
        messageField.setString("");
    }
    /**
     * Log out of the Darkstar server
     */
    private void doLogout() {
        clientMgr.disconnect();
        users.clear();
        createPostLogonScreen();
        channels = new Hashtable();
        currentState = LOGGED_OUT;
    }

    /**
     * Initialize the client manager. This will start the process of game discovery.
     * This will go out to the server and discover all the available games and try to
     * match the game name that was provided to the client manager with an existing game
     * if successful you will receive a callback discoveredGames() and can then proceed to
     * log in
     *
     */
    private void initializeJMEClientManager() {
        clientMgr = new JMEJMEClientManager("ChatTest",
                new JMEDiscoverer(this.getAppProperty("XML-Discovery-URL")),
                this.getAppProperty("UsrMgrClassName"));
        clientMgr.setListener(this);
        clientMgr.discoverGames();
    }

    /**
     * Handle validation callbacks to login to the server
     */
    private void dealWithCallbacks(Callback[] callbacks) {
        //for now hard code dealing with the login callback
        NameCallback nm = (NameCallback)callbacks[0];
        nm.setName(userNameField.getString());
        PasswordCallback pb = (PasswordCallback)callbacks[1];
        pb.setPassword(passwordField.getString().toCharArray());
    }

    private boolean equals(byte[] a, byte[] a2) {
        if (a==a2)
            return true;
        if (a==null || a2==null)
            return false;

        int length = a.length;
        if (a2.length != length)
            return false;

        for (int i=0; i<length; i++)
            if (a[i] != a2[i])
                return false;

        return true;
    }

    /**
     * We have discovered all the games. We can now attempt to log on to the server with the
     * host, port, etc. that we have discovered.
     */
    public void discoveredGames() {
        try {
            clientMgr.connect();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    /**
     * Notifies us that an exception occurred when either sending or receiving data
     */
    public void exceptionOccurred(Exception ex) {
        Displayable newScreen = changeState(EXCEPTION_OCCURRED);
        getDisplay().setCurrent(getGenericAlert("Exception occurred " + ex.getMessage()), newScreen);
    }

    /**
```

```
     * Send a message to the server
     */
    private void sendServerMessage() {
        String dataString = messageField.getString();
        ByteBuffer data = ByteBuffer.wrap(dataString.getBytes());
        clientMgr.sendServerMessage(data);
        messageField.setString("");
    }
}
```