Sun Game Server
Client Tutorial

03/05/07

Contents

Introduction

This document is a short tutorial for programming clients that connect to the Sun Game Server (SGS).

Programming game clients that use servers provided by the Sun Game Server is not much different from coding any other sort of game client. The big difference between SGS-based games and others is how they connect to and communicate with the server and with each other. This tutorial focuses on the "plumbing" necessary to make those connections.

It does *not* attempt to address the design issues of how to design a good client/server architecture for an online game, since this is very game-specific and well beyond the scope of what can be covered here. There are many good books and articles on the subject already available.

Lesson 1: HelloUserClient

This lesson shows how to connect to the SGS server and then communicate with it in the basic client/server mode. It is designed to go along with Lesson 5 of the *Sun Game Server Application Tutorial* and the **HelloUser** and **HelloEcho** examples in those lessons. It can also be used as a client for the **SwordWorld** sample application in Appendix A of the *Sun Game Server Application Tutorial*.

To test this client, you will need to start either **HelloUser** or **HelloEcho** on the server (**HelloEcho** has more functionality). See the application tutorial for details on how to do this.

SimpleClient

The class **com.sun.sgs.client.simple.SimpleClient** is your gateway to the SGS. You create an instance of **SimpleClient** and then use it and its associated listener interface **SimpleClientListener** to communicate with the server.

Connecting

The first thing an SGS Client needs to do is connect to its server application. Connecting is done in four steps:

1. **Create an instance of SimpleClient.**
   The first thing you need to do is create an instance of the **SimpleClient** class. **SimpleClient**'s constructor takes one parameter: a **SimpleClientListener** to call for communication events. In most basic clients, this is likely the client's main class, in which case the code would look something like this:

   ```
   public class MyClient implements SimpleClientListener {

       // ...

       simpleClient = new SimpleClient(this);
   ```

2. **Create the login properties.**
   The **login** method of **SimpleClient** expects one parameter as well, a **Properties** object. The **SimpleClient** implementation code expects two properties to be set: **host** and **port**. Below is an example of how to set these:

   ```
   Properties connectProps = new Properties();
   connectProps.put("host", "localhost");
   connectProps.put("port", "1139");
   ```

3. **Call the login method.**
   To actually start the login process, you call the **login** method. It is possible for **login** to throw an **IOException**, so you should surround the call with a try/catch block:

   ```
   try {
         simpleClient.login(connectProps);
   } catch (IOException e) {
         e.printStackTrace();
         // ... try again, or try a different hostname
   }
   ```

4. **Handle the authentication callback.**
   In response to your request to log in, the API will call the **getPasswordAuthentication** callback on your **SimpleClientListener** to request the user name and password it will use to log in. “Password” in this case is a general term for any authentication information returned to the API in a byte array. Exactly

what form this must take will depend on the authenticator installed on the server side. The default authenticator ignores password completely and lets anyone in. The SDK also ships with a sample encrypted string-password-file-based authenticator. Other authenticators can be written to support specific user validation infrastructures.[1]

A sample **getPasswordAuthentication** callback implementation might look like this:

```
/*
 * Returns dummy credentials where user is "guest-<random>"
 * and the password is "guest".  Real-world clients are likely
 * to pop up a login dialog to get these fields from the player.
 */
public PasswordAuthentication getPasswordAuthentication() {
    String player = "guest-" + random.nextInt(1000);
    setStatus("Logging in as " + player);
    String password = "guest";
    return new PasswordAuthentication(player, password.toCharArray());
}
```

At the completion of these steps, the API will attempt to log in to the server. If it is successful, it will call the **loggedIn** callback on the **SimpleClientListener**. At this point, the client is connected to the server and can begin communicating with it. If login fails, the **loginFailed** callback will be called instead and passed a string that contains a description of why login failed.

Client/Server Communication

Once we are connected to the server application, we can begin communicating. There are two forms of communication in the Sun Game Server API: *client/server* communication and *publish/subscribe channel* communication. Our first example uses client/server, which is the simpler of the two.

All communication in the SGS is done by sending and receiving byte arrays. To a Java coder this may seem a somewhat old-fashioned way of doing things. The SGS, however, is a client-agnostic system. The SGS team intends to deliver client APIs for J2SE, J2ME, and C/C++. Other platforms may also be identified as later client targets by the SGS team or by the community.

While a few cross-language object systems exist, they are either platform-specific (for example, DCOM) or very complex, with serious overhead (for example, CORBA). The conclusion of the team was that it was best for the system to provide the common and efficient base of sending and receiving arrays of bytes, and to let specific applications build on top of that according to their needs.

To send a packet to the server, all we have to do is pass the packet in a byte array to the **send** method on the **SimpleClient** object. Since this call can throw an **IOException**, it too needs to be in a try/catch block. Here is a simple example from the **HelloUserClient** program:

```
try {
    simpleClient.send(encodeString(getInputText()));
} catch (IOException e) {
    e.printStackTrace();
}
```

When the server sends a packet back to the client, it gets delivered to the application via the **receivedMessage** callback on the **SimpleClientListener**. Here again is a simple example from **HelloUserClient**:

```
public void receivedMessage(byte[] message) {
    appendOutput("Server: " + decodeString(message));
}
```

1 Writing and using custom authenticators will be covered in the *SGS Stack Extension Manual*.

That's all there is to getting client/server communication working with the Sun Game Server. Below is the complete code to **HelloUserClient**. If you start the **HelloEcho** server application from the *Sun Game Server Application Tutorial* and then run **HelloUserClient**, you should see the user log in on the server; anything typed into the input field at the bottom of **HelloUserClient's** window will be sent to the server application. The server application will echo it back, and you should see it appear in the large text output space in the top of the client's window.

Running HelloUserClient

The tutorial comes with all the examples pre-compiled in the **tutorial.jar** file. To run **HelloUserClient** from the jar file:

1. Change your working directory to the **tutorial** directory in the SGS SDK.
2. Type the following command line:

   For Win32:

   java -cp tutorial.jar;..\lib\sgs-client.jar com.sun.sgs.tutorial.client.lesson1.HelloUserClient

   For Unix:

   java -cp tutorial.jar:../lib/sgs-client.jar com.sun.sgs.tutorial.client.lesson1.HelloUserClient

Code: HelloUserClient

HelloUserClient

```
/*
 * Copyright 2007 Sun Microsystems, Inc. All rights reserved
 */

package com.sun.sgs.tutorial.client.lesson1;

import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.UnsupportedEncodingException;
import java.net.PasswordAuthentication;
import java.util.Properties;
import java.util.Random;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

import com.sun.sgs.client.ClientChannel;
import com.sun.sgs.client.ClientChannelListener;
import com.sun.sgs.client.simple.SimpleClient;
import com.sun.sgs.client.simple.SimpleClientListener;

/**
 * A simple GUI client that interacts with an SGS server-side app.
 * It presents a basic chat interface with an output area and input
```

```
 * field.
 * <p>
 * The client understands the following properties:
 * <ul>
 * <li><code>{@value #HOST_PROPERTY}</code> <br>
 *     <i>Default:</i> {@value #DEFAULT_HOST} <br>
 *     The hostname of the server.<p>
 *
 * <li><code>{@value #PORT_PROPERTY}</code> <br>
 *     <i>Default:</i> {@value #DEFAULT_PORT} <br>
 *     The port that the server is listening on.<p>
 *
 * </ul>
 */
public class HelloUserClient extends JFrame
    implements SimpleClientListener, ActionListener
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The name of the host property. */
    public static final String HOST_PROPERTY = "tutorial.host";

    /** The default hostname. */
    public static final String DEFAULT_HOST = "localhost";

    /** The name of the port property. */
    public static final String PORT_PROPERTY = "tutorial.port";

    /** The default port. */
    public static final String DEFAULT_PORT = "1139";

    /** The message encoding. */
    public static final String MESSAGE_CHARSET = "UTF-8";

    /** The output area for chat messages. */
    protected final JTextArea outputArea;

    /** The input field for the user to enter a chat message. */
    protected final JTextField inputField;

    /** The panel that wraps the input field and any other UI. */
    protected final JPanel inputPanel;

    /** The status indicator. */
    protected final JLabel statusLabel;

    /** The {@link SimpleClient} instance for this client. */
    protected final SimpleClient simpleClient;

    /** The random number generator for login names. */
    private final Random random = new Random();

    // Main

    /**
     * Runs an instance of this client.
     *
     * @param args the command-line arguments (unused)
     */
    public static void main(String[] args) {
        new HelloUserClient().login();
    }
```

```
// HelloUserClient methods

/**
 * Creates a new client UI.
 */
public HelloUserClient() {
    this(HelloUserClient.class.getSimpleName());
}

/**
 * Creates a new client UI with the given window title.
 *
 * @param title the title for the client's window
 */
protected HelloUserClient(String title) {
    super(title);
    Container c = getContentPane();
    JPanel appPanel = new JPanel();
    appPanel.setFocusable(false);
    c.setLayout(new BorderLayout());
    appPanel.setLayout(new BorderLayout());
    outputArea = new JTextArea();
    outputArea.setEditable(false);
    outputArea.setFocusable(false);
    appPanel.add(new JScrollPane(outputArea), BorderLayout.CENTER);
    inputField = new JTextField();
    inputField.addActionListener(this);
    inputPanel = new JPanel();
    inputPanel.setLayout(new BorderLayout());
    populateInputPanel(inputPanel);
    inputPanel.setEnabled(false);
    appPanel.add(inputPanel, BorderLayout.SOUTH);
    c.add(appPanel, BorderLayout.CENTER);
    statusLabel = new JLabel();
    statusLabel.setFocusable(false);
    setStatus("Not Started");
    c.add(statusLabel, BorderLayout.SOUTH);
    setSize(640, 480);
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    setVisible(true);
    simpleClient = new SimpleClient(this);
}

/**
 * Allows subclasses to populate the input panel with
 * additional UI elements.  The base implementation
 * simply adds the input text field to the center of the panel.
 *
 * @param panel the panel to populate
 */
protected void populateInputPanel(JPanel panel) {
    panel.add(inputField, BorderLayout.CENTER);
}

/**
 * Appends the given message to the output text pane.
 *
 * @param x the message to append to the output text pane
 */
protected void appendOutput(String x) {
    outputArea.append(x + "\n");
}
```

```
/**
 * Initiates asynchronous login to the SGS server specified by
 * the host and port properties.
 */
protected void login() {
    String host = System.getProperty(HOST_PROPERTY, DEFAULT_HOST);
    String port = System.getProperty(PORT_PROPERTY, DEFAULT_PORT);

    try {
        Properties connectProps = new Properties();
        connectProps.put("host", host);
        connectProps.put("port", port);
        simpleClient.login(connectProps);
    } catch (Exception e) {
        e.printStackTrace();
        disconnected(false, e.getMessage());
    }
}

/**
 * Displays the given string in this client's status bar.
 *
 * @param status the status message to set
 */
protected void setStatus(String status) {
    appendOutput("Status Set: " + status);
    statusLabel.setText("Status: " + status);
}

/**
 * Encodes a {@code String} into an array of bytes.
 *
 * @param s the string to encode
 * @return the byte array which encodes the given string
 */
protected static byte[] encodeString(String s) {
    try {
        return s.getBytes(MESSAGE_CHARSET);
    } catch (UnsupportedEncodingException e) {
        throw new Error("Required character set " + MESSAGE_CHARSET +
            " not found", e);
    }
}

/**
 * Decodes an array of bytes into a {@code String}.
 *
 * @param bytes the bytes to decode
 * @return the decoded string
 */
protected static String decodeString(byte[] bytes) {
    try {
        return new String(bytes, MESSAGE_CHARSET);
    } catch (UnsupportedEncodingException e) {
        throw new Error("Required character set " + MESSAGE_CHARSET +
            " not found", e);
    }
}

/**
 * Returns the user-supplied text from the input field, and clears
 * the field to prepare for more input.
```

```
     *
     * @return the user-supplied text from the input field
     */
    protected String getInputText() {
        try {
            return inputField.getText();
        } finally {
            inputField.setText("");
        }
    }

    // Implement SimpleClientListener

    /**
     * {@inheritDoc}
     * <p>
     * Returns dummy credentials where user is "guest-&lt;random&gt;"
     * and the password is "guest."  Real-world clients are likely
     * to pop up a login dialog to get these fields from the player.
     */
    public PasswordAuthentication getPasswordAuthentication() {
        String player = "guest-" + random.nextInt(1000);
        setStatus("Logging in as " + player);
        String password = "guest";
        return new PasswordAuthentication(player, password.toCharArray());
    }

    /**
     * {@inheritDoc}
     * <p>
     * Enables input and updates the status message on successful login.
     */
    public void loggedIn() {
        inputPanel.setEnabled(true);
        setStatus("Logged in");
    }

    /**
     * {@inheritDoc}
     * <p>
     * Updates the status message on failed login.
     */
    public void loginFailed(String reason) {
        setStatus("Login failed: " + reason);
    }

    /**
     * {@inheritDoc}
     * <p>
     * Disables input and updates the status message on disconnect.
     */
    public void disconnected(boolean graceful, String reason) {
        inputPanel.setEnabled(false);
        setStatus("Disconnected: " + reason);
    }

    /**
     * {@inheritDoc}
     * <p>
     * Returns {@code null} since this basic client doesn't support channels.
     */
    public ClientChannelListener joinedChannel(ClientChannel channel) {
        return null;
```

```
    }

    /**
     * {@inheritDoc}
     * <p>
     * Decodes the message data and adds it to the display.
     */
    public void receivedMessage(byte[] message) {
        appendOutput("Server: " + decodeString(message));
    }

    /**
     * {@inheritDoc}
     * <p>
     * Updates the status message on successful reconnect.
     */
    public void reconnected() {
        setStatus("reconnected");
    }

    /**
     * {@inheritDoc}
     * <p>
     * Updates the status message when reconnection is attempted.
     */
    public void reconnecting() {
        setStatus("reconnecting");
    }

    // Implement ActionListener

    /**
     * {@inheritDoc}
     * <p>
     * Encodes the string entered by the user and sends it to the server.
     */
    public void actionPerformed(ActionEvent event) {
        if (! simpleClient.isConnected())
            return;

        try {
            String text = getInputText();
            simpleClient.send(encodeString(text));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Lesson 2: HelloChannelClient

Lesson 1 illustrated how to connect to an SGS server application from a client and do client/server communication between them. This lesson adds the ability to handle sending and receiving data on publish/subscribe channels.

Publish/Subscribe Channels

As mentioned in Lesson 1, in addition to client/server communication, the SGS also supports publish/subscribe channels. Client/server communication always has the server and one client on either end of the message; in contrast, the channel system has *N* participating clients who can send and receive messages. Clients are made participants of the channel by being joined to it by the server. A client may be joined to many channels at once. The channel system supports both targeted messages to specific participants and broadcast messages that any participating client will receive.

Server applications are in control of channels. They are the ones who create the channel, add users to it, or remove users from it. Server applications can also listen to packets sent to any one user, or all packets on the channel, as well as send packets to users on the channel.

One big difference between channel communication and client/server communication is that the server application does not have to involve itself in sending or receiving the packets. Once a set of users have been joined to a channel, they can send and receive packets among themselves. Such client-to-client communication can be more efficient, since it does not involve the server application's task or persistence mechanisms. In the case of broadcast communications, it also removes the need to manage a list of users.

This is a common pattern for fast action games that have to update other users very rapidly, but have to update the server at a much lower frequency, if at all.

Another common pattern is for clients to request to be joined to channels, either explicitly or implicitly (by asking to be joined to a game session). In this case, the client sends a request to the server application via the game's own packet protocol, and the server application does the actual joining of the client to the channel.

Joining and Leaving a Channel

As mentioned above, adding and removing users from the channel is under the control of the server application. How then does a client know what channels it is a part of and how to communicate on them? The answer is another callback on the **SimpleClientListener** interface: **joinedChannel**. The **joinedChannel** callback receives a **ClientChannel** object as its one parameter. The client application should save this object, since this is its interface for sending messages on the channel.

The **SimpleClient** expects this callback to return an object that implements the **ClientChannelListener** interface. This interface has two methods on it, **receivedMessage** and **leftChannel**. The first is called on a callback whenever a packet for this user is received from the channel. The second callback is called if the server removes this user from the channel.

Below is an implementation of **joinedChannel** from our second sample client application, **HelloChannelClient**. This client adds a combo box to the left of the input field in **HelloUserClient** so you can select if the input is sent directly to the server or to one of the channels the server has joined you to. You can use the server application called **HelloChannels** (described in Lesson 6 of the *Sun Game Server Application Tutorial*) to test this client. This server application creates two channels, **foo** and **bar**, and automatically joins all users to them.

```
public ClientChannelListener joinedChannel(ClientChannel channel) {
    channelsByName.put(channel.getName(), channel);
    appendOutput("Joined to channel " + channel.getName());
```

```
        channelSelectorModel.addElement(channel.getName());
        return new HelloChannelListener();
    }
```

The complete implementation of **HelloChannelClient** can be found at the end of this lesson.

Sending and Receiving Channel Messages

Once you have processed the **joinedChannel** callback, sending and receiving channel messages is just about as easy as client/server messages. To send a message on a channel, you use one of three variant methods of the **send** method. The simplest of these just takes a byte buffer and broadcasts it to all users who are currently joined to the channel. This is how it is used in **HelloChannelClient**:

```
ClientChannel channel = channelsByName.get(channelName);
channel.send(message);
```

The second version adds a **SessionID** as a destination for a private message (a **SessionID** is how the server identifies a user). The third takes an array of **SessionIDs.**

An application obtains a **SessionID** for another user in one of two ways:

- The server knows the **SessionIDs** of all the users, so server applications may choose to transmit this information via client/server communication.
- Clients can do their own bookkeeping, since every channel message comes with a “return address”.

The **receivedMessage** callback on the **ChannelListener** has three parameters:

- The first is a **Channel** object that represents the channel on which the message was received.
- The next is the **SessionID** of the sender.
- The third is the actual data sent.

Below is the entire code for **HelloChannelClient**, including the **HelloChannelListener** implementation, which is implemented as an inner class of **HelloChannelClients**. You should use the **HelloChannels** application from the “Sun Game Server Application Tutorial” as the server for running this client.

Running HelloChannelClient

The tutorial comes with all the examples pre-compiled in the **tutorial.jar** file. To run **HelloChannelClient** from the jar file:

1. Change your working directory to the **tutorial** directory in the SGS SDK.
2. Type the following command line:

   For Win32:

   java -cp tutorial.jar;..\lib\sgs-client.jar com.sun.sgs.tutorial.client.lesson2.HelloChannelClient

   For Unix:

   java -cp tutorial.jar:../lib/sgs-client.jar com.sun.sgs.tutorial.client.lesson2.HelloChannelClient

Code: HelloChannelClient

HelloChannelClient

```
/*
 * Copyright 2007 Sun Microsystems, Inc. All rights reserved
 */

package com.sun.sgs.tutorial.client.lesson2;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.atomic.AtomicInteger;

import javax.swing.DefaultComboBoxModel;
import javax.swing.JComboBox;
import javax.swing.JPanel;

import com.sun.sgs.client.ClientChannel;
import com.sun.sgs.client.ClientChannelListener;
import com.sun.sgs.client.SessionId;
import com.sun.sgs.tutorial.client.lesson1.HelloUserClient;

/**
 * A simple GUI client that interacts with an SGS server-side app using
 * both direct messaging and channel broadcasts.
 * <p>
 * It presents a basic chat interface with an output area and input
 * field, and adds a channel selector to allow the user to choose which
 * method is used for sending data.
 *
 * @see HelloUserClient for a description of the properties understood
 *      by this client.
 */
public class HelloChannelClient extends HelloUserClient
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** Map that associates a channel name with a {@link ClientChannel}. */
    protected final Map<String, ClientChannel> channelsByName =
        new HashMap<String, ClientChannel>();

    /** The UI selector among direct messaging and different channels. */
    protected JComboBox channelSelector;

    /** The data model for the channel selector. */
    protected DefaultComboBoxModel channelSelectorModel;

    /** Sequence generator for counting channels. */
    protected final AtomicInteger channelNumberSequence =
        new AtomicInteger(1);

    // Main

    /**
     * Runs an instance of this client.
     *
     * @param args the command-line arguments (unused)
```

```
 */
public static void main(String[] args) {
    new HelloChannelClient().login();
}

// HelloChannelClient methods

/**
 * Creates a new client UI.
 */
public HelloChannelClient() {
    super(HelloChannelClient.class.getSimpleName());
}

/**
 * {@inheritDoc}
 * <p>
 * This implementation adds a channel selector component next
 * to the input text field to allow users to choose between
 * direct-to-server messages and channel broadcasts.
 */
@Override
protected void populateInputPanel(JPanel panel) {
    super.populateInputPanel(panel);
    channelSelectorModel = new DefaultComboBoxModel();
    channelSelectorModel.addElement("<DIRECT>");
    channelSelector = new JComboBox(channelSelectorModel);
    channelSelector.setFocusable(false);
    panel.add(channelSelector, BorderLayout.WEST);
}

/**
 * {@inheritDoc}
 * <p>
 * Returns a listener that formats and displays received channel
 * messages in the output text pane.
 */
@Override
public ClientChannelListener joinedChannel(ClientChannel channel) {
    channelsByName.put(channel.getName(), channel);
    appendOutput("Joined to channel " + channel.getName());
    channelSelectorModel.addElement(channel.getName());
    return new HelloChannelListener();
}

/**
 * {@inheritDoc}
 * <p>
 * Encodes the string entered by the user and sends it on a channel
 * or directly to the server, depending on the setting of the channel
 * selector.
 */
@Override
public void actionPerformed(ActionEvent event) {
    if (! simpleClient.isConnected())
        return;

    try {
        String text = getInputText();
        byte[] message = encodeString(text);
        String channelName =
            (String) channelSelector.getSelectedItem();
        if (channelName.equalsIgnoreCase("<DIRECT>")) {
```

```
                simpleClient.send(message);
            } else {
                ClientChannel channel = channelsByName.get(channelName);
                channel.send(message);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * A simple listener for channel events.
     */
    public class HelloChannelListener
        implements ClientChannelListener
    {
        /**
         * An example of per-channel state, recording the number of
         * channel joins when the client joined this channel.
         */
        private final int channelNumber;

        /**
         * Creates a new {@code HelloChannelListener}. Note that
         * the listener will be given the channel on its callback
         * methods, so it does not need to record the channel as
         * state during the join.
         */
        public HelloChannelListener() {
            channelNumber = channelNumberSequence.getAndIncrement();
        }

        /**
         * {@inheritDoc}
         * <p>
         * Displays a message when this client leaves a channel.
         */
        public void leftChannel(ClientChannel channel) {
            appendOutput("Removed from channel " + channel.getName());
        }

        /**
         * {@inheritDoc}
         * <p>
         * Formats and displays messages received on a channel.
         */
        public void receivedMessage(ClientChannel channel,
                SessionId sender, byte[] message)
        {
            appendOutput("[" + channel.getName() + "/ " + channelNumber +
                "] " + sender + ": " + decodeString(message));
        }
    }
}
```

Conclusion

That's all you need to know in order to write SGS client applications. The SGS team has made every effort to make this revision of the API simple and intuitive. We hope you find it pleasant and easy to work with. For more details and options, please see the SGS Client API Javadocs.

One thing not touched on in this introduction is the pluggable nature of the SGS communication APIs. The default implementation shipped with the SDK uses TCP/IP for reliable communication and UDP for unreliable communication. This can be changed, however, on an application-specific basis. This is an advanced topic that will be dealt with in the *SGS Stack Extension Manual*.