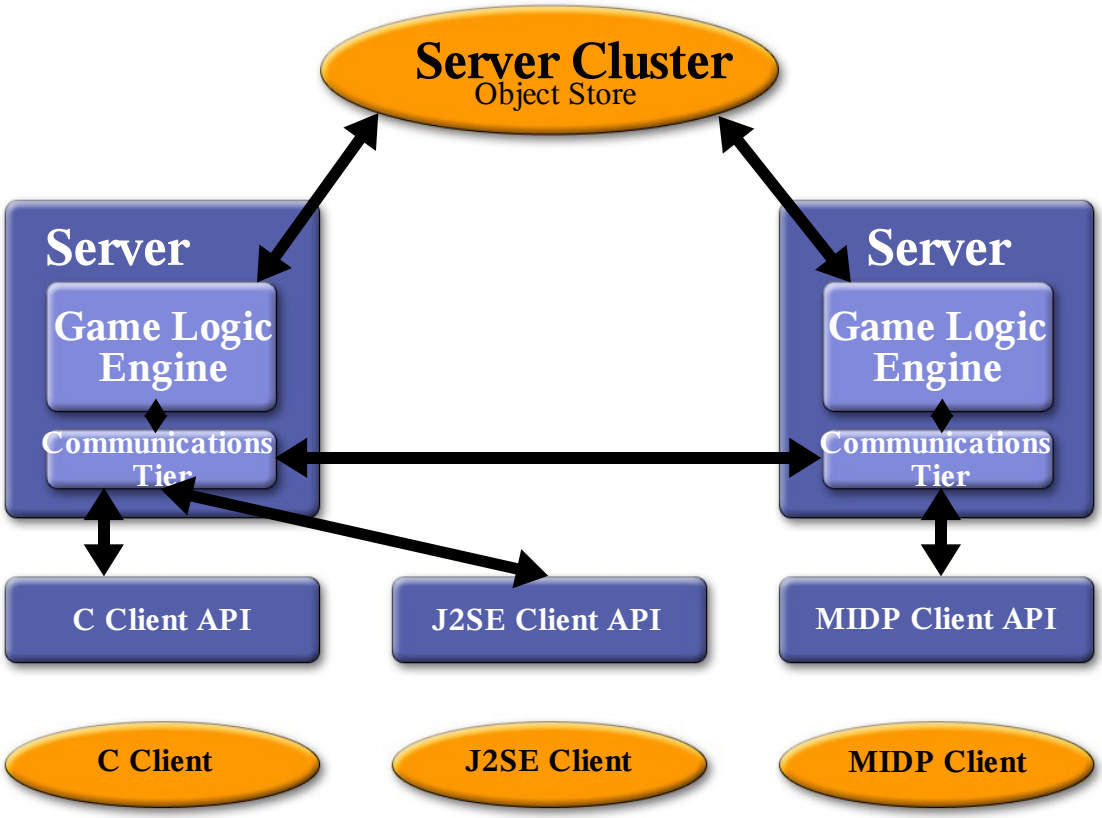
Sun Game Server
C++ Client API
Programmer's Guide

Introduction to the Sun Game System

The Sun Game System (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 users 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 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 C++ 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 C++ programming.

Overview of Communication between the Server Application and the C++ Client

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

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 an **IClientCommunicationManager** object (described below) via a **ClientCommunicationManager** interface. The **IClientCommunicationManager** 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 **IClientConnectionManager** instance is stateful and logged into a single game; however, the **IClientConnectionManager** 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 C++ 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 IClientConnectionManager and the IClientConnectionManagerListener

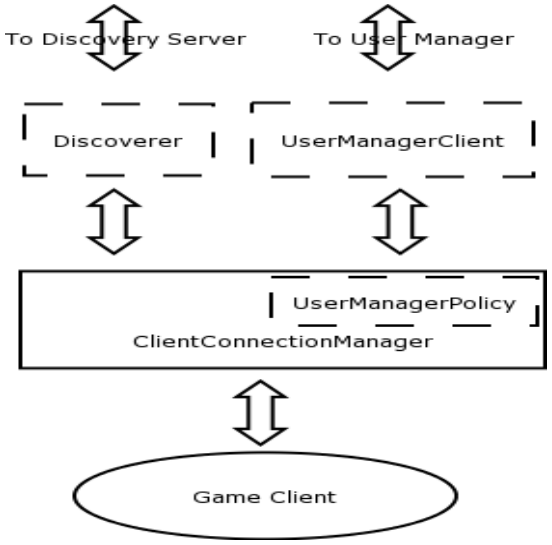
An SGS client application is a remote program that communicates with the SGS back end to do interclient communication and interface with SGS server applications. The client application communicates with the server application through an instance of the **IClientConnectionManager** class. A single instance 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. The **IClientConnectionManager** interface is implemented by the implementation class **ClientConnectionManager**.

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

Incoming data is translated to events that are sent to the client application via a listener interface that the client application must implement and register with the **IClientConnectionManager**. The client must periodically pull these events down from the server by calling the **Update** function; the interval at which **Update** is called (for example, every *N* frames) is up to the client.

The C++ API includes an **IClientConnectionManagerListener** implementation class for listening to events from **IClientConnectionManager**.

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



The User Manager

The **IClientConnectionManager** 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 C++ applications, **TCPIPUserManagerClient**; others may be added in the future or can be added by the game developers 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

An **IClientConnectionManager** contains a reference to an instance of a class that implements an **IUserManagerPolicy** interface. This object is responsible for deciding to which of all possible connection points (for example, on different hosts) the **IClientConnectionManager** will actually connect. The default **IUserManagerPolicy** 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 used in this distribution. 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 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 **IClientConnectionManager**. The **IClientConnectionManager** 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 **IClientConnectionManager** to connect to a User Manager of that type.

When the **IClientConnectionManager** needs to make a connection to the game (either a login connection or a reconnect connection, as described below) it requests the list of communication connection points from the Discoverer. It then asks the **IUserManagerPolicy** to choose one.

Once the **IClientConnectionManager** has a communication connection point, it creates an instance of the **IUserManagerClient** 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 **IUserManagerClient** to complete the connection.

Client Login and Validation

A game typically requires validation in order to accept a connection from an instance of its **IUserManagerClient** class. Validation consists of the client application filling in the required fields in one or more arrays of callback objects.
In such a case, an array of **Calllback** objects is returned to the client application through an **OnValidationRequest** message that is delivered to the **IClientConnectionManagerListener** object.

An **IClientConnectionManagerListener** object responds to this event by calling the S**endValidationResponse** method on the **IClientConnectionManager**'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 an **OnConnectionRefused** message containing a string giving a textual

reason for refusal), or all responses have been accepted (in which case the listener receives an **OnConnected** message containing the assigned user ID for this connection).
It is possible that more than one **OnValidationRequest** callback may occur. A client application should be written so that it can handle any number of these callbacks. The sequence of **OnValidationRequest** callbacks for any given connection attempt will be terminated by either the **OnConnected** or **OnConnectionRefused** 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 **OnConnected** 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 **IClientConnectionManager** notifies its registered **IClientConnectionManagerListener** objects with an **OnFailOverInProgress** event. It then restarts the connection process, beginning with the discovery and choice of a new communication connection point.

While an **IClientConnectionManager** 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 **IClientConnectionManager** submits this key as its validation. If they 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 **IClientConnectionManager** sends its listener an **OnConnected** event, after which communication can continue normally.

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

Logging Off

A client application requests that an **IClientConnectionManager** disconnect from the back end by calling the **disconnect** method on the **IClientConnectionManager**'s API. The **IClientConnectionManagerListener** will receive an **OnDisconnected** message once logout has been accomplished.

Game Events

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

- **OnDataReceived**
  This listener method gets called whenever an inbound data packet arrives that was sent to the specific userID associated with this **IClientConnectionManager**.

- **OnUserJoined**
  This listener method gets called once for every **IClientConnectionManager** that is connected to the game that this **IClientConnectionManager** joined (connected to). It then gets called periodically as new user join the game.

- **OnUserLeft**
  This listener method gets called when any **IClientConnectionManager** whose connection was reported by **OnUserJoined** 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. The C++ API includes **IClientChannel** implements this interface, and **IClientChannelListener** listens for events reported by **IClientChannel**.

A player can send data to one, many, or all players on a channel. In addition, channel joiners receive **OnPlayerJoined** and **OnPlayerLeft** 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 **IClientConnectionManagerListener** so that it can receive events from the server. Call the **Update** function periodically to pull these events down 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 **IClientConnectionManager** decide which of the various possible connection points to user. Currently the API provides one **UserManagerPolicy**, the **DefaultUserManagerPolicy**,.

- Create an instance of **IClientConnectionManager** 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 **IClientConnectionManager** 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/or receive data from the sever.

- Send data, and respond to data received.

- Provide a means of disconnecting from the server application.

Unicode Strings

Wherever appropriate, the Sun Game Server C++ API uses Unicode for string handling. Unicode is an industry-standard character set representation that includes mappings for non-Western languages such as Japanese and Chinese, as well as for mathematical and other symbols. One key difference between Unicode and ASCII is that Unicode characters take two-bytes of storage, as opposed to one byte for ASCII.

Note that, in interpreting a string, an **L** before the string indicates that it is a string of Unicode characters.

To use the Unicode API, you must do the following:

- If you are using Visual Studio, you must set the option **Treat wchar_t as Built-in Type** to **Yes**.

- If you are using the command-line compiler instead of Visual Studio, you must use the option **/Zc:wchar_t**.

The WinMain Client Program

The example program **ChatTest** is a basic chat program implemented using the Sun Game Server. The client application, **WinMain.cpp**, is a very simple client for this server application. Source code for **WinMain.cpp** is shown in the appendix.

Summary of the Client's Use of the API

The sample client application uses these components of the API:

- It implements the **IClientConnectionManager** interface via the **ClientConnectionManager** implementation class provided in the SGS software distribution.
- It establishes itself as the **IClientConnectionManagerListener**, to listen for events from the server. It calls the **Update** function every 10 ms.
- It uses the **DefaultUserManagerPolicy**.
- It uses the **URLDiscoverer** constructor to parse the discovery file.
- It implements the **ClientChannel** interface for channel communication via **IClientChannel** and **IClientChannelListener**.

Here are the **#include** statements:

```
// SGS Core and Interfaces
#include "SGSClient.h"

// SGS Implementation Classes
#include "Discovery\URLDiscoverer.h"
#include "Client\ClientConnectionManager.h"
#include "Client\DefaultUserManagerPolicy.h"
```

The header file **SGSClient.h** in turn includes the following header files:

```
#include "Utilities\Callback.h"
#include "Discovery\IDiscoverer.h"
#include "Discovery\IDiscoveredGame.h"
#include "Discovery\IDiscoveredUserManager.h"
#include "Client\IClientConnectionManager.h"
#include "Client\IClientConnectionManagerListener.h"
#include "Client\IClientChannel.h"
#include "Client\IClientChannelListener.h"
#include "Client\IUserManagerPolicy.h"
```

Connecting to the Server Application

As discussed above, a client must begin by connecting to the server application.

WinMain connects to the server application via the **ClientConnectionManager** implementation class for I**ClientConnectionManager**. It needs to tell **ClientConnectionManager** the name of the application to connect to and how to find it. It passes the name of the server application, **ChatTest**, and a discovery file, **FakeDiscovery.xml**. See below for more information about the

discovery file. **IClientConnectionManager** uses the constructor **URLDiscoverer** to return the URL of this file.

WinMain specifies the **DefaultUserManagerPolicy**, which means that is will use random assignment to game connection points.

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

Finally, it calls the **Update** function periodically to pull events down from the server.

Here is the code that does this, along with the **Disconnect** method for disconnecting at the end of the session:

```
int wmain()
{
#ifdef _DEBUG
	_CrtSetDbgFlag(_CRTDBG_LEAK_CHECK_DF | _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG));
#endif

	std::auto_ptr<IUserManagerPolicy> pPolicy(new DefaultUserManagerPolicy());
	std::auto_ptr<IDiscoverer> pDiscoverer(new URLDiscoverer(L"http://FakeDiscovery.xml"));
	std::auto_ptr<IClientConnectionManager> pCCM(new ClientConnectionManager(L"ChatTest",
pDiscoverer.get(), pPolicy.get()));

	std::auto_ptr<IClientConnectionManagerListener> pCCMListener(new ConsoleListener(pCCM.get()));
	pCCM->SetListener(pCCMListener.get());

	std::vector<std::wstring> classNames = pCCM->GetUserManagerClassNames();
	for (size_t i = 0; i < classNames.size(); ++i)
	{
		if (pCCM->Connect(classNames[i]))
		{
			// Keep the connection up for 60s, allowing the network to update every 10ms
			for (int i = 0; i < 6000; ++i)
			{
				pCCM->Update();
				Sleep(10);
			}
			pCCM->Disconnect();
		}
	}
	pCCM->SetListener(NULL);

	return 0;
}
```

Connection Failover

The **IClientConnectionManagerListener** provides callbacks that are called when there is a connection failure.

The function **OnFailOverInProgress** 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 function **OnReconnected** informs the client that it is being reconnected, so it should start communicating again.

WinMain simply prints out messages in response to these events:

```
virtual void OnFailOverInProgress()
 {
        wprintf(L"OnFailOverInProgress\n");
 }

 virtual void OnReconnected()
 {
        wprintf(L"OnReconnected\n");
 }
```

Logging In

Once the connection is made, **ChatTest** asks for validation from the client via the **OnValidationRequest** callback. **WinMain** provides the name and password directly in the code and sends them to the server via the **SendValidationResponse** function:

```
virtual void OnValidationRequest(const std::vector<ICallback*>& callbacks)
 {
        for (size_t i = 0; i < callbacks.size(); ++i)
        {
                NameCallback* pNCB = dynamic_cast<NameCallback*>(callbacks[i]);
                if (pNCB)
                        pNCB->SetName(L"guest1");

                PasswordCallback* pPCB = dynamic_cast<PasswordCallback*>(callbacks[i]);
                if (pPCB)
                        pPCB->SetPassword(L"guest1");

                TextInputCallback* pTICB = dynamic_cast<TextInputCallback*>(callbacks[i]);
                if (pTICB)
                        {}
        }

        mpCCM->SendValidationResponse(callbacks);
 }
```

If the login is successful, **IClientConnectionManager** calls the callback **OnConnected** (part of the **IConnectionManagerListener** interface) in **WinMain**; this callback returns a userID for the user on this connection. **WinMain** prints the userID and opens a DCC (Direct Client to Client) channel for communication:

```
 virtual void OnConnected(const UserID& myID)
 {
        wprintf((std::wstring(L"Received User ID ") + myID.ToString() + L"\n").c_str());

        mpCCM->OpenChannel(L"__DCC_Chan");
 }
```

If the connection is refused (for example, due to an incorrect password), the **IClientConnectionManager** calls the **OnConnectionRefused** callback in **WinMain** and sends back a message that gives the reason for the failure, which **WinMain** prints:

```
 virtual void OnConnectionRefused(const std::wstring& message)
 {
        wprintf((std::wstring(L"Connection Refused ") + message + L"\n").c_str());
 }
```

Other Game Events

WinMain, as the **IClientConnectionManagerListener**, listens for **OnUserJoined** and **OnUserLeft** events, in response to which it prints out the user's userID:

```
	virtual void OnUserJoined(const UserID& user)
	 {
		wprintf((std::wstring(L"OnUserJoined ") + user.ToString() + L"\n").c_str());
	 }

	 virtual void OnUserLeft(const UserID& user)
	 {
		wprintf((std::wstring(L"OnUserLeft ") + user.ToString() + L"\n").c_str());
	 }
```

WinMain also prints out a message if it receives an **OnDisconnected** event, disconnecting it from the server:

```
	 virtual void OnDisconnected()
	 {
		wprintf(L"OnDisconnected\n");
```

Channel Events

When this user joins a channel via the **OpenChannel** function, **WinMain** receives an **OnJoinedChannel** event. In response it prints out the name of the channel and sets itself as the listener for events on this channel:

```
	virtual void OnJoinedChannel(IClientChannel* clientChannel)
	 {
		mpChannel = clientChannel;
		wprintf((std::wstring(L"OnJoinedChannel ") + clientChannel->GetName() +
L"\n").c_str());
		clientChannel->SetListener(this);
```

As the channel listener, **WinMain** listens for **OnPlayerJoined**, **OnPlayerLeft**, **OnDataArrived**, and **OnChannelClosed** events:

```
	// IClientChannelListener
	 virtual void OnPlayerJoined(IClientChannel* pChannel, const UserID& playerID)
	 {
		wprintf((std::wstring(L"OnPlayerJoined ") + playerID.ToString() + L"\n").c_str());
	 }

	 virtual void OnPlayerLeft(IClientChannel* pChannel, const UserID& playerID)
	 {
		wprintf((std::wstring(L"OnPlayerLeft ") + playerID.ToString() + L"\n").c_str());
	 }

	 virtual void OnDataArrived(IClientChannel* pChannel, const UserID& fromID, const byte* data,
size_t length, bool wasReliable)
	 {
		wprintf((std::wstring(L"OnDataArrived ") + fromID.ToString() + L"\n").c_str());

		mpChannel->SendUnicastData(fromID, data, length, wasReliable);
		mpCCM->SendToServer(data, length, wasReliable);
	 }

	 virtual void OnChannelClosed(IClientChannel* pChannel)
	 {
		wprintf((std::wstring(L"OnChannelClosed") + L"\n").c_str());
	 }
```

When a player leaves or joins a channel, it simply prints the message and the player's userID.

When data arrives, it sends the data back over the channel via the **SendUnicastData** function, and also sends it to the server via the **SendToServer** function.

Appendix: Source Code for the WinMain Sample Client Program

```
#define UNICODE
#define _UNICODE
#define STRICT
#define NOMINMAX
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#pragma warning(disable: 4100)          // warning C4100: 'XXX' : unreferenced formal parameter

// SGS Core and Interfaces
#include "SGSClient.h"

// SGS Implementation Classes
#include "Discovery\URLDiscoverer.h"
#include "Client\ClientConnectionManager.h"
#include "Client\DefaultUserManagerPolicy.h"

using namespace SGS;

class ConsoleListener :
        public IClientConnectionManagerListener,
        public IClientChannelListener
{
public:
        ConsoleListener(IClientConnectionManager* pCCM) : mpCCM(pCCM) { }

private:
        IClientConnectionManager* mpCCM;
        IClientChannel* mpChannel;

        // IClientConnectionManagerListener
        virtual void OnValidationRequest(const std::vector<ICallback*>& callbacks)
        {
                for (size_t i = 0; i < callbacks.size(); ++i)
                {
                        NameCallback* pNCB = dynamic_cast<NameCallback*>(callbacks[i]);
                        if (pNCB)
                                pNCB->SetName(L"guest1");

                        PasswordCallback* pPCB = dynamic_cast<PasswordCallback*>(callbacks[i]);
                        if (pPCB)
                                pPCB->SetPassword(L"guest1");

                        TextInputCallback* pTICB = dynamic_cast<TextInputCallback*>(callbacks[i]);
                        if (pTICB)
                                {}
                }

                mpCCM->SendValidationResponse(callbacks);
        }

        virtual void OnConnected(const UserID& myID)
        {
                wprintf((std::wstring(L"Received User ID ") + myID.ToString() + L"\n").c_str());

                mpCCM->OpenChannel(L"__DCC_Chan");
        }

        virtual void OnConnectionRefused(const std::wstring& message)
        {
                wprintf((std::wstring(L"Connection Refused ") + message + L"\n").c_str());
        }

        virtual void OnFailOverInProgress()
        {
                wprintf(L"OnFailOverInProgress\n");
        }

        virtual void OnReconnected()
        {
                wprintf(L"OnReconnected\n");
        }
```

```
        virtual void OnDisconnected()
        {
                wprintf(L"OnDisconnected\n");
        }

        virtual void OnUserJoined(const UserID& user)
        {
                wprintf((std::wstring(L"OnUserJoined ") + user.ToString() + L"\n").c_str());
        }

        virtual void OnUserLeft(const UserID& user)
        {
                wprintf((std::wstring(L"OnUserLeft ") + user.ToString() + L"\n").c_str());
        }

        virtual void OnJoinedChannel(IClientChannel* clientChannel)
        {
                mpChannel = clientChannel;
                wprintf((std::wstring(L"OnJoinedChannel ") + clientChannel->GetName() + L"\n").c_str());
                clientChannel->SetListener(this);
        }

        virtual void OnChannelLocked(const std::wstring& name, const SGS::UserID& userID)
        {
                wprintf((std::wstring(L"OnChannelLocked ") + name + L"\n").c_str());
        }

        // IClientChannelListener
        virtual void OnPlayerJoined(IClientChannel* pChannel, const UserID& playerID)
        {
                wprintf((std::wstring(L"OnPlayerJoined ") + playerID.ToString() + L"\n").c_str());
        }

        virtual void OnPlayerLeft(IClientChannel* pChannel, const UserID& playerID)
        {
                wprintf((std::wstring(L"OnPlayerLeft ") + playerID.ToString() + L"\n").c_str());
        }

        virtual void OnDataArrived(IClientChannel* pChannel, const UserID& fromID, const byte* data,
size_t length, bool wasReliable)
        {
                wprintf((std::wstring(L"OnDataArrived ") + fromID.ToString() + L"\n").c_str());

                mpChannel->SendUnicastData(fromID, data, length, wasReliable);
                mpCCM->SendToServer(data, length, wasReliable);
        }

        virtual void OnChannelClosed(IClientChannel* pChannel)
        {
                wprintf((std::wstring(L"OnChannelClosed") + L"\n").c_str());
        }

};

int wmain()
{
#ifdef _DEBUG
        _CrtSetDbgFlag(_CRTDBG_LEAK_CHECK_DF | _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG));
#endif

        std::auto_ptr<IUserManagerPolicy> pPolicy(new DefaultUserManagerPolicy());
        std::auto_ptr<IDiscoverer> pDiscoverer(new URLDiscoverer(L"http://FakeDiscovery.xml"));
        std::auto_ptr<IClientConnectionManager> pCCM(new ClientConnectionManager(L"ChatTest",
pDiscoverer.get(), pPolicy.get()));

        std::auto_ptr<IClientConnectionManagerListener> pCCMListener(new ConsoleListener(pCCM.get()));
        pCCM->SetListener(pCCMListener.get());

        std::vector<std::wstring> classNames = pCCM->GetUserManagerClassNames();
        for (size_t i = 0; i < classNames.size(); ++i)
        {
                if (pCCM->Connect(classNames[i]))
                {
                        // Keep the connection up for 60s, allowing the network to update every 10ms
                        for (int i = 0; i < 6000; ++i)
                        {
                                pCCM->Update();
                                Sleep(10);
                        }
                        pCCM->Disconnect();
                }
        }
        pCCM->SetListener(NULL);
```

```
        return 0;
}
```