Setting Up the Session Management Pathway

Once you've signed in to the server, and sent and received presence information about roster members, you'll want to set up the session management pipeline. This is the set of Session Management and Logic components that monitor incoming connection requests and make outgoing connection requests. This is the most customized part of a libjingle application, because different applications have different requirements for sending or receiving connections: for example, do you need to create or read a list of files to share? Do you need to instantiate additional media engines? Do you need to generate a list of codecs as part of the session offer?

What tasks you perform to send a connection request, or to respond to a connection request, depends on your specific application. The basic underlying steps in session connection are described in How libjingle applications work. However, all applications will need to instantiate the following core libjingle objects:

1. **Instantiate your NetworkManager, PortAllocator subclass, and SessionManager objects.** This must be done before you try to initiate or receive session requests. The example code waits until it has successfully signed in to the server before creating these objects in order to avoid unnecessary object creation if a sign in fails; however, you can create these objects earlier if you want.
2. **Create a new Thread object to use as the worker thread [*optional*].** If your application hosts a separate worker thread, you should create a new **talk_base::Thread** object and pass it in to the **SessionManager** constructor. Otherwise, the thread in which you create the **SessionManager** will be used as the worker thread. The file share example uses only a single thread (created to host the **FileShareClient** manager object, which creates **SessionManager**), while the voice chat example creates a dedicated worker thread inside **CallClient::InitPhone**. Be sure to call **Start**, not **Run**, on the worker thread, since **Run** is a blocking call.
3. **Register your SessionClient subclass with SessionManager. SessionManager** calls **SessionClient::OnSessionCreate** when it creates a new **Session**. (This notification is sent for both outgoing and incoming requests; a parameter in the signal specifies the direction of the request.) **SessionManager** holds a map of *ID*/**SessionClient** instance values, where *ID* is a unique ID string value present in both the sending and receiving code. This ID is included in the XMPP stanzas passed between the computers, and parsed out by **SessionManager** on incoming stanzas.
4. **Enable SessionManager to listen for and send XMPP messages. SessionManagerTask** enables **SessionManager** to send and receive XMPP messages**. SessionManagerTask** is the middleman between the XMPP messaging component and the Peer to Peer component. These messages include session requests, replies to your session requests, and candidate lists.
5. **Reset the STUN and relay port information.** libjingle uses Google STUN and relay servers by default, but you should request information dynamically for your actual servers and reset these values in **HttpPortAllocator**. You can request this information dynamically with the **JingleInfoTask** object.

The following code demonstrates the most high level of these steps. It is taken from pcp_main.cc, and modified for brevity and concept.

```
// Create the HttpPortAllocator and HttpPortAllocator, and SessionManager objects.
// We use the current thread (signaling thread) as the worker thread. To create
// a new worker thread, pass a new Thread object into the SessionManager constructor.
// See CallClient::InitPhone for an example of creating a worker thread.
talk_base::NetworkManager network_manager_;
talk_base::HttpPortAllocator *port_allocator_ = new talk_base::HttpPortAllocator(&network_manager_, "pcp")
;
cricket::SessionManager *session_manager_ = new cricket::SessionManager(&port_allocator, NULL);

// Create the object that will be used to send/receive XMPP session requests
// and start it up.
cricket::SessionManagerTask *session_manager_task = new cricket::SessionManagerTask(xmpp_client_, session_
manager_);
session_manager_task->EnableOutgoingMessages();
session_manager_task->Start();

// Query for the STUN and relay ports being used. This is an asynchronous call,
// so we need to register for SignalJingleInfo, which sends the response.
buzz::JingleInfoTask *jingle_info_task = new buzz::JingleInfoTask(xmpp_client_);
jingle_info_task->RefreshJingleInfoNow();
jingle_info_task->SignalJingleInfo.connect(this, &FileShareClient::OnJingleInfo);
jingle_info_task->Start();

// Hook up a custom logic class that is used for file sharing in the File Share sample app.
file_share_session_client_.reset(new cricket::FileShareSessionClient(session_manager_.get(),
xmpp_client_->jid(), "pcp"));
file_share_session_client_->SignalFileShareSessionCreate.connect(this, &FileShareClient::OnFileShareSessio
nCreate);

// Associate the FileShareSessionClient with its global ID in SessionManager.
session_manager_->AddClient(NS_GOOGLE_SHARE, file_share_session_client_.get());

...

// Called by JingleInfoTask::SignalJingleInfo with the ports being used.
void OnJingleInfo(const std::string &relay_token,
                  const std::vector<std::string> &relay_addresses,
                  const std::vector<talk_base::SocketAddress> &stun_addresses) {
  port_allocator_->SetStunHosts(stun_addresses);
  port_allocator_->SetRelayHosts(relay_addresses);
  port_allocator_->SetRelayToken(relay_token);
}
```

Last updated March 23, 2012.