Developer Scenarios

You can modify libjingle code to support a number of different scenarios:

Scenario	Description
Video chatting	To create a video chat application, start with the call sample code and add a media engine to render and capture video. If you are using Jingle, use the Jingle video media description format (XEP-0180).
Music streaming	Streaming live music could be done either by modifying the call sample code to enable one-way communication only, or by using the File Share sample application and replacing **FileStream** with **MemoryStream**. You would have to modify the code to set one computer as the server, and handle the session negotiation to determine which computer is serving, similar to how the file share sample application specifies the serving computer. (Part of this would include modifying the **SessionDescription** to include custom data to indicate the server). You could replace the voice engines with the basic audio renderers on the operating system.
Tuning the port allocation	You might want to extend the **HttpPortAllocator** class to specify how relay resources are allocated. You could also modify the **Transport** class to change the candidate prioritization algorithm.
Supporting a different signaling protocol	Although libjingle uses XMPP as its session negotiation protocol, you could modify the code to support a different protocol. If you use an XML-based protocol you could reuse most of the existing code but change the **SessionDescription** object and modify the **PhoneSessionClient::TranslateSessionDescription** methods, and provide your own QName values to describe the elements and attributes of your protocol. For non-XML based protocols you would have to provide a more extensive reworking of the XMPP messaging component, **Session**, **SessionClient**, and **SessionManager** classes.

Most of these scenarios involve the following two tasks:

- Supporting additional session types
- Supporting additional media engines

Supporting Additional Session Types

libjingle provides code to support voice chat applications and file transfers. If you want to support additional types of data transfer, you will have to extend a number of base classes, and possibly provide new rendering, wrapping, or helper classes. The following list describes the steps to support an additional session type.

- Extend **SessionClient** to manage the new session type. Your extension should handle any tasks specific to your session type, including generating or reading stanzas that pertain to those tasks. For example, the call sample application requires both sides to generate a list of codecs that they support; **PhoneSessionClient** parses and generates the XMPP stanzas containing those codecs, and also initiates the request to its media engines to list the codecs that they can support.
- Extend **SessionDescription** to describe your own session-specific information. **SessionDescription** provides extended information needed to describe a particular type of session (codecs for voice sessions, an arbitrary string for tunnel sessions). SessionDescription is created by the whatever object needs to send a session request (typically, the object that extends **SessionClient**).

- Create a session wrapper to wrap individual **Session** objects in a way appropriate to your new session type. The call example uses the **VoiceChannel** class to instantiate and manage a rendering engine and a **P2PTransportChannel**, and the **Call** class to store and associate **Session** objects with **VoiceChannel** objects; the File Share sample application uses **FileShareSession** to handle both jobs (manage a **Session** object and associate it with the proper **P2PTransportChannel**.
- If your session includes a different media type from the provided examples, you will have to create an additional media engine wrapper.

Extending SessionClient

The **SessionClient** class manages all session objects of a specific type for the application. It handles tasks that are common across all sessions, such as specifying which connection should be active, accepting or rejecting a session request, and initiating connection requests. Extensions of the **SessionClient** class typically create any additional helper classes you might need, such as media engines or stream wrappers. You must extend **SessionClient** and **SessionDescription** in order to handle new connection types.

When extending **SessionClient**, you should override the following virtual methods as described.

SessionClient Method	Description
OnSessionCreate	Called by the **SessionManager** object when it creates a new **Session** object for either an incoming our outgoing connection request. In the implementation, connect to the **Session** object's **Session::SignalState** and **Session::SignalError** signals. Your handler method should parse the session-specific description passed in for information, such as codecs, and take the appropriate action.
OnSessionDestroy	Called by the **SessionManager** when it destroys a **Session** object. Your implementation should clean up all resources allocated for the **Session** object and links to the **Session** object created by your **SessionClient** implementation.
CreateSessionDescription	Translates an incoming XMPP stanza describing the specifics of the session into a **SessionDescription** subclass. This is described in the following section.
TranslateSessonDescription	Converts a **SessionDescription** object sent by **Session** into an XMPP stanza ready for transmission across the network.

Extend SessionDescription

When negotiating a connection between two clients, you may need to include additional information in the stanzas sent between the two computers. For instance, when setting up an audio session, both computers must exchange a list of audio codecs supported by each application. Some protocols have requirements about what must be included (for example, audio sessions exchange audio codec lists), but you can include any additional information that is understood by both parties in the connection. This additional information is wrapped by the **SessionDescription** class when translated from the XMPP stanza.

Extend **SessionDescription** to include any kinds of methods and variables you need to describe your session. Both the **Session** and **SessionClient** objects must be customized to be able to parse and use the additional information. The **SessionClient** subclass is responsible for translating between the stanza and the **SessionDescription** class in its implementation of **CreateSessionDescription** and **TranslateSessionDescription**.

Translating a SessionDescription object to an XmlElement object

libjingle uses the **XmlElement** object to wrap an XMPP stanza. The following methods and describe how to build an **XmlElement** from a **SessionDescription** object.

To create a stanza encapsulating a session description, you must create an **XmlElement** object with the name "description" and a namespace unique to your session type. The **XmlElement** object constructor takes both of these parameters wrapped in a **QName** object as shown here (taken from **PhoneSessionClient**):

```
const std::string NS_PHONE("http://www.google.com/session/phone");
const buzz::QName QN_PHONE_DESCRIPTION(true, NS_PHONE, "description");
buzz::XmlElement* description = new buzz::XmlElement(QN_PHONE_DESCRIPTION, true);
```

The **XmlElement** object defines the following constructor:

XmlElement(QName &*name*, bool *useDefaultNs*)

name

A **QName** (qualified name) object combines the element name and a namespace into a single object. **QName** objects in libjingle are typically global scope. The **PhoneSessionClient** class, for example, defines several **QName** objects required by Jingle Audio.

useDefaultNs

A boolean value specifying whether to qualify with the namespace when printing XML. You will always need to create an XML element with the name "description" and a namespace unique to your session type.

You may then need to add child elements to this root description element with the **AddElement** method. A file sharing session description might create one element for each file; **PhoneSessionClient** creates one for each supported codec. The following code from **PhoneSessionClient** demonstrates creating a **SessionDescription** with a sub-element for each codec:

```
buzz::XmlElement *PhoneSessionClient::TranslateSessionDescription(const SessionDescription *_session_desc)
 {
  // Cast the generic SessionDescription to the specific type needed here.
  const PhoneSessionDescription* session_desc = static_cast<const PhoneSessionDescription*>(_session_desc)
;

  // Create the wrapper element.
  buzz::XmlElement* description = new buzz::XmlElement(QN_PHONE_DESCRIPTION, true);

  // Loop through the codecs in the SessionDescription and add one child element to
  // the stanza for each codec.
  for (size_t i = 0; i < session_desc->codecs().size(); ++i) {
    buzz::XmlElement* payload_type = new buzz::XmlElement(QN_PHONE_PAYLOADTYPE, true);

    char buf[32];
    sprintf(buf, "%d", session_desc->codecs()[i].id);
    payload_type->AddAttr(QN_PHONE_PAYLOADTYPE_ID, buf);

    payload_type->AddAttr(QN_PHONE_PAYLOADTYPE_NAME,
                          session_desc->codecs()[i].name.c_str());

    description->AddElement(payload_type);
  }

  return description;
}
```

The code shown above creates the following audio description stanza:

```
<description xmlns='http://www.google.com/session/phone'>
    <payload-type id='18' name='G729'/>
    <payload-type id="97" name="IPCMWB"/>
    <payload-type id='98' name='L16'/>
    <payload-type id="103" name="ISAC"/>
```

```
    <payload-type id="102" name="iLBC"/>
    <payload-type id="4" name="G723"/>
    <payload-type id="100" name="EG711U"/>
    <payload-type id="101" name="EG711A"/>
    <payload-type id="0" name="PCMU"/>
    <payload-type id="8" name="PCMA"/>
    <payload-type id="13" name="CN"/>
</description>
```

Translating an XmlElement object to a SessionDescription object

Reading data from an **XmlElement** is similarly straightforward. Your **CreateSessionDescription** method will be passed the received `<description/>` element, from which you should follow whichever protocol being used to extract data out of it. There are a few libjingle functions that should be particularly helpful.

XmlElement::FirstNamed returns the first child element with a given qualified name. **PhoneSessionClient::CreateSessionDescription** calls **FirstNamed** to get the first `<payload-type/>` child:

```
  const buzz::XmlElement* payload_type = element->FirstNamed(QN_PHONE_PAYLOADTYPE);
```

You can then loop through each `<payload-type/>` element by using the **XmlElement::NextNamed** method:

```
    payload_type = payload_type->NextNamed(QN_PHONE_PAYLOADTYPE);
```

The specifics of how to interpret the XMPP stanza will vary depending on the session type. For example, because Jingle Audio calls for a number of `<payload-type>` elements, it uses a loop to add each one to the **PhoneSessionDescription** object:

```
int id = atoi(payload_type->Attr(QN_PHONE_PAYLOADTYPE_ID).c_str());
  int pref = 0;
  std::string name = payload_type->Attr(QN_PHONE_PAYLOADTYPE_NAME);
  desc->AddCodec(MediaEngine::Codec(id, name, 0));
```

Create a Session Wrapper

In libjingle, the **Session** object handles the common mechanics of the session negotiation (sending and receiving invitations, creating a **TransportChannel** to manage the data channel, and ending the session), but you must create a higher level wrapper object to manage both of these objects appropriately for a session type. For example, in the file share code, **FileShareSession** creates and hosts the objects that make and serve the HTTP request, and sends the requests for each file in turn.

The following table describes the most important signals and methods that your controlling classes will need to know about the **Session** object.

Session Signal or Method	Description
Initiate	Sends a connection request to another computer.
Accept	Accepts an incoming request. The incoming session will be sent to **SessionClient::OnSessionCreate**.
Reject	Rejects an incoming request. The incoming session will be sent to **SessionClient::OnSessionCreate**.
Terminate	Ends a current session.
CreateChannel	Creates a **TransportChannel** object, used to send and receive data from the other party. See

	"Using the P2PSocket Object" below.
SignalState	Sent when the session state changes, as described in the voice chat description.
SignalError	Sent when an error occurs.
SignalInfoMessage	Sent when a **Session** object receives an information message specific to the session type (for example, "share complete" when all files have been requested in the file sharing example).
SignalChannelGone	Called when the connection has been broken for some reason.
SendInfoMessage	Sends an XMPP message to the other party, containing information specific to the session type (for example, file not found in the file sharing application).

Using the TransportChannel Object

Once you have a **Session** object managing the connection, you can stream data between computers. The top level object managing the data channel is **TransportChannel** (or a derived class). All data is sent or received across the network using this socket, which manages connectivity and performance for you. libjingle supports two kinds of **TransportChannel** objects: TCP-type sockets, which provide a lossless data connection with TCP-like functionality that supports redundancy and robustness for data-critical applications like file transfer; and RTP-type sockets, which provide a higher transmission speed for real-time applications like voice chat. How you read and write to the socket depends on what kind of socket you create.

RTP-type Channels

1. Create the **TransportChannel** using the **Session** object. If this is an incoming connection, it is the **Session** passed to your **SessionClient::OnSessionCreate** implementation. If this is an outgoing connection, it is the **Session** object you explicitly created. Create a **TransportChannel** by calling **Session::CreateChannel** with "rtp" as the name.
2. Connect to receive messages from the new socket. You should connect to **Socket::SignalWritableState** and **Socket::SignalReadPacket** to receive state messages and received data. See the reference documentation for more information about those methods.
3. Send data to the other computer by calling **TransportChannel::SendPacket**; read data that is sent through the **TransportChannel::SignalReadPacket** signal. See voicechannel.cc for example code.

TCP-type Channels

TCP sockets require several more helper classes and layers of indirection in order to support data redundancy and a stream-like interface. The file share example uses **PseudoTcpChannel** to expose stream read/write functionality. See File Share Application and StreamInterface for more information.

Supporting Additional Media Engines

libjingle ships with two media engines, Linphone and GIPS VoiceEngine Lite, that enable live audio to be captured and rendered on your computer. If you want to add a new media engine to support audio capturing or rendering, or if you want to support a new media type such as video, you must override the **MediaEngine** class and the **MediaChannel** class as described next. (Supporting new media types requires a much more extensive redesign of the supporting classes, as described in Supporting Additional Session Types above.)

MediaEngine

The **MediaEngine** base class represents the components that manage rendering and capturing hardware. Specialized classes extend **MediaEngine** to support different media hardware. The examples included in libjingle use two **MediaEngine** specializations called **LinphoneMediaEngine**, which uses Linphone code, and **GipsLiteMediaEngine** which uses GIPS VoiceEngine Lite. The **MediaEngine** class is responsible for handling global audio settings and creating **MediaChannel** objects.

A customized **MediaEngine** implementation must include the following:

- The ability to capture or render media data as required from the computer hardware.
- The ability to encode and decode media using suitable codecs. Currently, to work with Google Talk, you must support at least one of the following codecs: PCMA, PCMU, G.723, iLBC, ISAC, IPCMWB, EG711U, or EG711A.
- The ability to understand Real-Time Protocol (RTP). **MediaEngine** must work with the **P2PTransportChannel** class, meaning it should be capable of receiving incoming packets by a function call and sending outgoing packets by a callback.
- The ability to create and manage **MediaChannel** objects (described below). A **MediaChannel** object represents an individual media stream. As with **MediaEngine**, subclass the **MediaChannel** class to support your specific media components.

The **MediaEngine** class exposes eight methods that you should override. If your implementation handles a non-audio media type, it should add corresponding methods to handle that medium.

MediaEngine Method	Description
CreateChannel	Creates a new **MediaChannel** object, representing a media stream associated with a session. There is one **MediaChannel** associated with each **Session**.
GetInputLevel	Returns the current input level, used to show your outgoing volume in the UI. This isn't used by libjingle, but can be used by your application.
Init	Initializes your media library and populates the vector of codecs that your engine supports (see below).
SetAudioOptions	Sets audio options found in the **MediaEngineOptions** enum. Currently, only one option exists: `AUTO_GAIN_CONTROL`. This isn't used by libjingle, but can be used by your application.
SetSoundDevices	Sets the sound hardware to use. Your application and your **MediaEngine** extension should both understand what the integer arguments mean. This isn't used by libjingle, but can be used by your application.
Terminate	Cleans up the resources used by the **MediaEngine**.
codecs	Returns a list of codecs supported by the engine.
FindCodec	Indicates whether a specific codec is supported by the engine.

Overriding MediaEngine::Init

In your **MediaEngine::Init** implementation, you should populate the internal codecs_ member vector with **Codec** structs representing the codecs that your **MediaEngine** class supports. **PhoneSessionClient** sorts these codecs in order of preference before sending the session description. The less-than operator used to sort codecs is also encapsulated within the struct; the higher the number, the greater the preference. Create **Codec** objects with their constructor and push them onto the codecs_ vector as shown here:

```
codecs_.push_back(Codec(110, "speex", 8));
codecs_.push_back(Codec(102, "iLBC", 4));
codecs_.push_back(Codec(0, "PCMU", 2));
```

When the **PhoneSessionClient** class specifies which codec to use, it does so only by the codec's *name* property . The codec used varies from voice session to voice session.

Customizing the MediaChannel Class

The **MediaChannel** class represents the data flow between the **P2PTransportChannel** and the **MediaEngine** in one-to-one audio conversation, with its own codec and RTP streams. It contains a number of overridable methods for negotiating which codec to use, whether to play or mute audio, and the current audio quality and volume of the voice session.

The **MediaChannel** class is responsible for sending and receiving data from the media renderers to the peer-to-peer session. When the `session` component of libjingle receives a packet from the **P2PTransportChannel** object, it calls **MediaChannel::OnPacketReceived**. When the **MediaChannel** object needs to send a packet, it sends it via the **NetworkInterface** class.

The **NetworkInterface** class has a single method, **SendPacket**, which **MediaChannel** uses to send packets to the **P2PTransportChannel** object. Other classes can subclass from **NetworkInterface** and then set themselves as the network interface with **MediaChannel::SetInterface**.

The overridable functions of the **MediaChannel** class are:

MediaChannel Method	Description
GetCurrentQuality	Returns the audio quality. This isn't used by libjingle, but may be used by your application.
GetOutputLevel	Returns the volume of the incoming audio on this stream. This isn't used by libjingle, but by the application.
OnPacketReceived	Called when a packet has been read from the network. The **MediaChannel** should handle the incoming RTP packet and play the audio.
SetCodec	Called when a codec for this session has been negotiated. It is passed the codec as a string.
SetPlayout	Sets whether or not to play audio from this session.
SetSend	Sets whether to send audio over the network from this session. This is used by the Mute feature.

Last updated March 23, 2012.