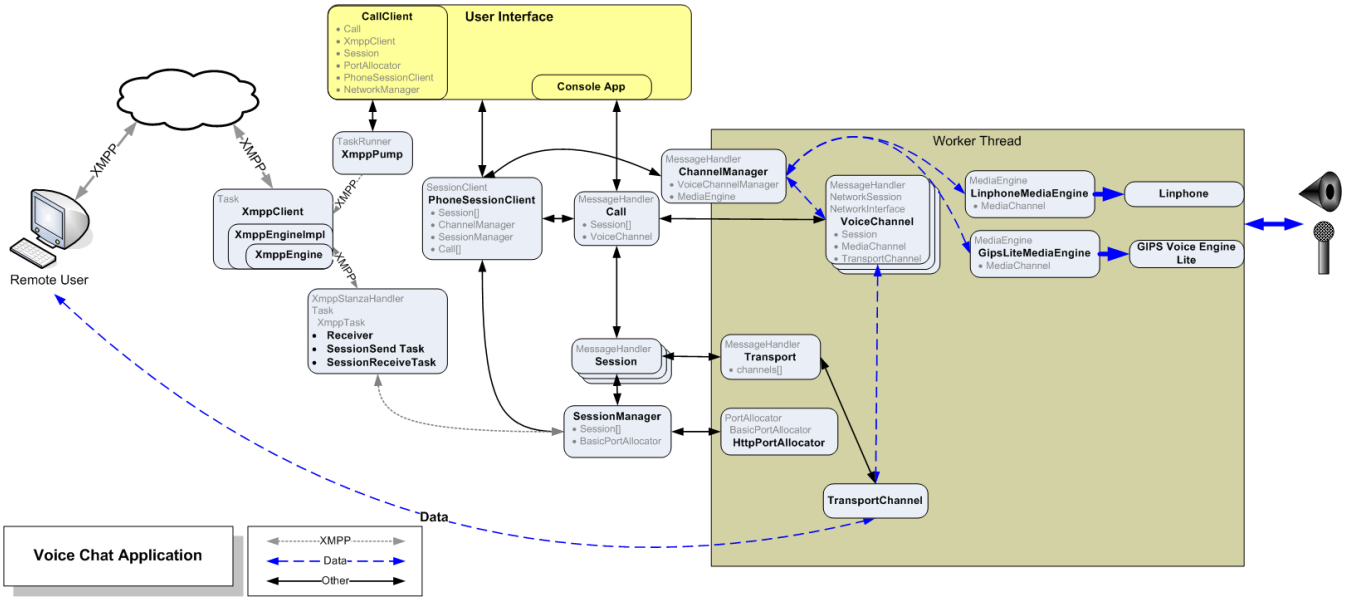
Voice Chat Applications

The call sample application is a voice chat application based on either GIPS VoiceEngine Lite or Linphone, depending on your operating system. It is a command-line application that provides basic alerts and presence notification. libjingle uses a variant of the Jingle extension for audio sessions (XEP-0167) to negotiate the details of the audio connection. These documents do not describe the actual protocol that manages an audio session. Instead, you should rely on the code to manage the protocol for you.

The following diagram shows the relationships between the main classes in the call application.



The previous diagram shows the main objects in the call code. The diagram shows the inheritance path of each object, shown in lighter text on the top right of the object, and important member variables, listed on the bottom of the object.

The user interface has two classes that are not discussed in detail here. The Console class manages asynchronous input and output to the monitor, and the CallClient class is the top level managing class that handles making and answering a call.

Some objects in this diagram are discussed in the overview of a general libjingle application. The following objects are specific to this application:

CallClient is the top level manager for all calls in a voice chat application.

PhoneSessionClient is the top level manager for Call objects, each of which represents one or more voice chat connections. It also is the top level manager for handles creating the offer description by collecting the list of codecs supported by the computer. In addition to the base **SessionClient** tasks, it handles voice chat specific tasks such as reading and generating codec lists, choosing a codec, and managing global audio settings. It creates **Call** objects and the **ChannelManager** object. The application instantiates this object.

Call handles one or more **Session** objects, each representing a connection between two users. A multi-person chat would consist of a single **Call** object managing several **Session** objects, one per connection. The **Call** object handles the top-level jobs for that call, for example ending sessions, making a call to another user, accepting an incoming call, handling audio tasks such as mute and monitoring, and so on. It sends signals when sessions are created, ended, or change state (connecting, running, and so on). For outgoing calls, the application instantiates this object; for incoming calls, it is instantiated for you by **PhoneSessionClient**.

ChannelManager is responsible for setting up and destroying the voice channels used to conduct voice chats, as well as specifying some audio settings on those channels. It creates a helper object, **MediaEngine** (not shown), that is specific to the type of media engine used (Linphone or GIPS VoiceEngine Lite), and also creates the individual **VoiceChannel** objects, one for each **Session** object. **ChannelManager** is instantiated by **PhoneSessionClient** when it is instantiated.

VoiceChannel connects the capturing/rendering object (**MediaEngine**) and the socket that sends and receives audio data from the network via a **TransportChannel**. Each voice channel is associated with one **Session**, one **TransportChannel**, and one **MediaChannel**. It monitors and sends notifications about the quality of the media channel with the help of the **AudioMonitor** helper class (not shown). **VoiceChannel** is instantiated by the **ChannelManager** at the request of the **Call** object.

MediaChannel is a base class that is extended to control specialized third-party audio packages (such as GIPS VoiceEngine Lite or Linphone). It is used by the specialized **MediaEngine** instance for that audio package to handle lower level tasks such as sending and receiving audio packets. Each voice channel requires a **MediaChannel** handle. When **ChannelManager** creates a new voice channel it also instantiates the proper derived version of this class.

LinphoneMediaEngine / **GipsLiteMediaEngine** are specialized extensions of **MediaEngine**. They control the higher-level aspects of audio, such as setting the hardware and audio options. It creates and returns a specialized **MediaChannel** implementation to a caller, which uses that to control the settings in the **MediaEngine**. **ChannelManager** instantiates this class when it is created. There can be only one media engine per **PhoneSessionClient**.

Linphone / **GIPS VoiceEngine Light** are third-party audio packages that work with the hardware on your computer to render or capture audio. GIPS VoiceEngine Lite is a limited version of GIPS for Windows computers; Linphone is used on Linux/UNIX computers. The appropriate **MediaEngine-**derived class instantiates these objects.

Tasks, **Session**, **SessionManager**, **P2PTransport**, **ProxyConnection/TCPConnection**, **XmppPump**, and the **Port** objects are all covered in the discussion about generic libjingle applications.

Call Application Specifics

The call sample application follows the general outline given in Generic libjingle Applications but has a few details worth noting:

- **Session** objects are bundled into groups managed by a **Call** object. The reason for this is to be able to handle related connections as a single unit (e.g., a multi-party chat). The **Call** object can be used to mute or activate a bundle of chats with a single method call. By default, each incoming or outgoing **Session** object is handled by a new **Call** object, but you can combine **Session** objects into a single **Call** object with a single method call (**Call::Join**) if you determine that they should be part of the same chat.
- **VoiceChannel** signs up for **Session::SignalState** signals. When a connection request acceptance stanza is received, **VoiceChannel** gets the Session::STATE_RECEIVEDINITIATE state signal. It then reconciles the list of sent codecs and received codecs to find the list of acceptable codecs, chooses the most preferred one, and tells the **MediaEngine** to use that codec.
- The **VoiceChannel** object tells the **Session** object to create a **TransportChannel** object, and wraps the created channel.

The voice chat creates RTP channels, which enables faster data streaming at the expense of potentially lost packets. The channel type is specified in the **Session::CreateChannel** method by specifying "rtp" as the channel name.

The voice chat example attempts to start sending and receiving data without waiting for the call recipient to accept.

Running a Voice Chat Application

The following sections cover the basic steps handled by the voice chat example:

- Making a Call
- Receiving a Call

In the call sample program, the custom CallClient object handles most of these steps. To see how to set up and use CallClient, see the **main** function in call_main.cc, or examine callclient.cc to learn how CallClient handles the main tasks.

Making a Call

The high-level object that manages the important actions in a voice call is called (appropriately) **Call**. A **Call** object manages any number of peer-to-peer **Session** objects, each representing one peer-to-peer connection. The **Call** object is the top level object to make calls, accept or reject incoming calls, monitor the status of the call, and performs other high level actions on call connections. CallClient wraps all required steps for making a call in its MakeCallTo method. Here are the basic steps taken by CallClient:

1. Create the **Call** object by calling **PhoneSessionClient::CreateCall**
2. Connect to the **PhoneSessionClient::SignalCallDestroy** to monitor when the all sessions in the call have ended and the call object is being destroyed. A call is destroyed when all the **Session** objects are destroyed, which can happen by request of the current user, by request of the other user, or by a connection failure.
3. Connect to the new **Call** object's **SignalSessionState** signal to monitor progress of the connection and send notifications to the user ("sent initiate ," "received initiate ," "in progress," and so on).
4. Send the connection request to the other user. Call **Call::InitiateSession** with the JID of the person to connect to.
5. Listen for the STATE_INPROGRESS message associated with that session, which will indicate that the connection request has been accepted and begun. The **Call** object will handle all the details of connecting and managing the connection for you.
6. When you are connected, you can mute or unmute a call with the **Call::Mute** method or terminate a session with the **Call::Terminate** method. (Some methods exposed by **Call** apply to specific sessions, while others apply to all sessions. See the reference documentation for details.)

The following code from CallClient::MakeCallTo starts a call to another user, specified by JID.

```
// Let us know when the Call object is destroyed, so we can close the UI
// or alert the user.
// This only needs doing once per PhoneSessionClient.
phone_client()->SignalCallDestroy.connect(this, &CallClient::OnCallDestroy);

...

// Create the call object.
call_ = phone_client()->CreateCall();

// Connect to receive session notifications.
call_->SignalSessionState.connect(this, &CallClient::OnSessionState);

// Make the connection request to the other user
session_ = call_->InitiateSession(buddy_jid);

// libjingle audio engine only handles one active voice channel at a time.
// Set the focus on the newly created conversation.
phone_client()->SetFocus(call_);

...
// Listen to the progress of the call and alert the user.
void CallClient::OnSessionState(cricket::Call* call,
                                cricket::Session* session,
                                cricket::Session::State state) {
    if (state == cricket::Session::STATE_INPROGRESS) {
        console_->Print("Call connected.");
    } else if (state == cricket::Session::STATE_RECEIVEDREJECT) {
        console_->Print("Other side rejected the request.");
    }
    ...other conditions...
 }
```

Receiving a Call

1. An incoming call triggers **PhoneSessionClient** to send its **SignalCallCreate** signal. You connected to this signal earlier as part of your initial setup. When an incoming call request is received, the **PhoneSessionClient** creates a new **Call** object and sends this signal, along with the **Call** object. Because **SignalCallCreate** is sent whether you or someone else created the **Call** object, the only way to find out what caused this call is to connect to the **Call** object's **SignalSessionState** signal.
2. The new **Call** object sends a **SignalSessionState** signal describing the new connection request. **SignalSessionState** notifications include the managing **Call** object, the **Session** object (which contains information about the caller), and an enumeration indicating what is happening (outgoing call, incoming request, etc). See **Session** for a description of the important enumeration values. Incoming call requests are sent STATE_RECEIVEDINITIATE.
3. Alert the user that a new call request has been made, and allow them to accept or reject the request. libjingle will respond to the call request automatically with session negotiation stanzas, but will not begin exchanging data until the user has explicitly accepted a connection request. Accept a call using **Call::AcceptSession**; reject a call using **Call::RejectSession**. You must pass in the session holding this request. Either use the **Session** object you received from **SignalSessionState**, or use the first **Session** object in the **Call** object's **Session** collection (accessed using **Call::Sessions**). Each new connection request generates a new **Call** object with one **Session** object, so there shouldn't be a problem with finding the right session on an incoming call.
4. If the user accepted the call, activate the voice channel of the new call. Before you can begin talking over a session, you must first call **PhoneSessionClient::SetFocus**(**Call*** *call*) to activate the channel. This is because the audio engines in the example implementation only allow one voice channel to be active at a time. Once a session has focus, you can start talking over it.

The following example code demonstrates receiving a call. This code is taken from callclient.cc.

```
// Handler connected to PhoneSessionClient::SignalCallCreate on initialization.
void CallClient::OnCallCreate(cricket::Call* call) {
  call->SignalSessionState.connect(this, &CallClient::OnSessionState);
}

// Handler called when a Session object changes state or is first created.
void CallClient::OnSessionState(cricket::Call* call,
                                cricket::Session* session,
                                cricket::Session::State state) {
  if (state == cricket::Session::STATE_RECEIVEDINITIATE){
      // This is an incoming call. Alert the user of the caller's JID.
      buzz::Jid jid(session->remote_address());
      console_->printf("Incoming call from '%s'", jid.Str().c_str());

      // In this thread or in a new thread, request user input to accept or reject the call.
             // Input not shown.
      ... bool acceptCall = user input value;

             // Accept or reject the call.
      // The call will be the first item in the Call::Sessions() collection.
             if(acceptCall){
                 call_->AcceptSession(call_->sessions()[0]);
          phone_client()->SetFocus(call_);
      }
             else{
                 call_->RejectSession(call_->sessions()[0]);
      }
    }
  ... other session states ...
}
```

Cleaning Up

There are no additional steps to take for cleaning up beyond the standard cleanup tasks, (The **Call** object handles deletion of the **VoiceChannel** objects by calling **ChannelManager::DestroyVoiceChannel**.)

Threads

The **call** sample application exposes one additional thread: this is the input/output thread used by the Console class to write and read user input. The other two threads are described in the threading topic. **ChannelManager** and the **MediaEngine**/**MediaChannel** classes also live in the worker thread. In the call example, **CallClient::InitPhone()** creates a worker thread for you.

Signals to Listen For

The following table lists the most important signals sent by the voice chat application. How to listen for signals is described in Signals.

Signal	Description
SessionManager::SignalRequestSignaling	Sent by the **SessionManager** to indicate that it is ready to start sending messages over the messaging thread. Signals include candidate generation and socket availability. When this signal is sent, simply respond by calling **SessionManager::OnSignalingReady**.
PhoneSessionClient::SignalCallCreate	Sent whenever a **Call** object is created. An incoming call request will trigger libjingle to create a new **Call** object that hosts a new **Session** object, and you must be prepared to catch that event. The call sample application also creates **Call** objects for outgoing calls in a different class in response to an XMPP request, so catching this signal will also alert you as to when your outgoing call is connecting. In the listening function, connect to the new **Call** object's **SignalSessionState**.
PresencePushTask::SignalStatusUpdate	Sent by the presence task whenever someone on your roster joins or leaves the XMPP server. You can catch this message to add or remove entries in the user interface. See Sending and Querying Presence for more information on using this object.
Call::SignalSessionState	Sent when a session's state changes. You will need to receive these notifications to recognize whether a new session is an incoming or outgoing call request, whether an outgoing request has been accepted or rejected, if the connection has been made, and other notifications.

Last updated March 23, 2012.