Creating a libjingle application

Creating a libjingle Application

libjingle includes C++ code that can be compiled and run on two main target operating systems: Linux and Windows.

The libjingle build computer has the following requirements:

- UNIX/Linux or Windows operating system
- Standard Template Library
- Linphone (UNIX/Linux) or GIPS VoiceEngine Lite (Windows) to use the voice chat application as written.

Systems running a libjingle application have the following requirements:

- UNIX/Linux or Windows operating system
- For UNIX/Linux systems, you will need expat, OpenSSL, oRTP, and GLib.
- 512K RAM (if not using the external media engine classes) .
- Approximately 900 KB disc space, though you could optimize the code to require significantly less.

Logging

libjingle includes code to enable message logging to an output stream. The code that specifies whether or not logging is enabled is in logging.h. If `LOGGING` is defined in the code, libjingle will enable the logging macros defined in that file, including LOG, PLOG, and LOG_ERR. (The Visual Studio solution defines `_DEBUG`, which causes the code to define `LOGGING`; in other environments, the md_common file defines `LOGGING`). libjingle code logs various messages and errors as it works.

Logging is enabled by default in the debug build. The default logging level in debug builds is LS_INFO.

In addition, **XmppClient** exposes two signals, **SignalLogInput** and **SignalLogOutput**, which can be used to log incoming and outgoing XMPP stanzas. See pcp_main.cc for an example of how to listen for those two signals and use the output to display stanzas as they arrive or are sent.

The sample code sends messages to `stdout`. You can change the standard output stream by calling **LogMessage::LogToStream**. You can set the minimum severity message to log by calling **LogMessage::LogToDebug** (the default value when logging is enabled is all messages).

Key Tasks

This section and the following sections describe all the high level actions that a program must take in order to use libjingle functionality. If you want to customize the libjingle code to handle different stream types, or to perform different actions, you should learn more about the underlying system by reading Generic libjingle Applications, Voice Chat Applications, and File Sharing Applications, and also read Scenarios to learn about different ways that you can modify the code.

The sample applications have a main() function that starts the sign in process, and a top-level managing class that handles all the other steps. In the file share example, the handler is **FileShareClient**, which is defined in pcp_main.cc; in the voice chat example, the handler is **CallClient**, which is defined in callclient.h/.cc. The steps detailed below are the high level steps; you might only see or need to modify some of these steps, depending on how much of the code you want to reuse.

For an application to use libjingle, it must perform following steps:

1. **Create the signaling thread for your application.** This thread is used by many components, and is key to the internal messaging system in libjingle. This thread must be created and started prior to instantiating **SessionManager**, or running any **Task** objects (such as **XmppPump**). You must create a **PhysicalSocketServer** object and pass it to a new **Thread** instance (which uses the socket server), then pass the new thread to the global static **ThreadManager** class. The simplest way to create the signaling thread is to use **AutoThread**. An example of this is given in the linked document for the next step.
2. **Sign in to the server.** The first step is to sign in to the XMPP server that acts as a central contact point to find other computers. You must instantiate the XMPP task manager and **XmppClient** objects in order to sign in. The sample applications use **XmppPump** to manage sign in and instantiate **XmppClient**.
3. **Send and request presence.** You will need to find out who is online and request their JID in order to request connections with another computer.
4. **Set up the session management pathway**. This is the set of Session Logic and Management components that listen for and respond to connection requests, or which are used to initiate your own requests. These are all custom classes, that vary with the application type.
5. **Send connection requests, or accept incoming connection requests**. In order to listen for incoming connections, you must sign up to receive notifications of session connections from **SessionManager**, sign up for session state changes, create any custom session descriptions when sending a connection request, and perform other management tasks. Many of these jobs are handled by wrapper classes in the Session Logic and Management Component.

Last updated March 23, 2012.