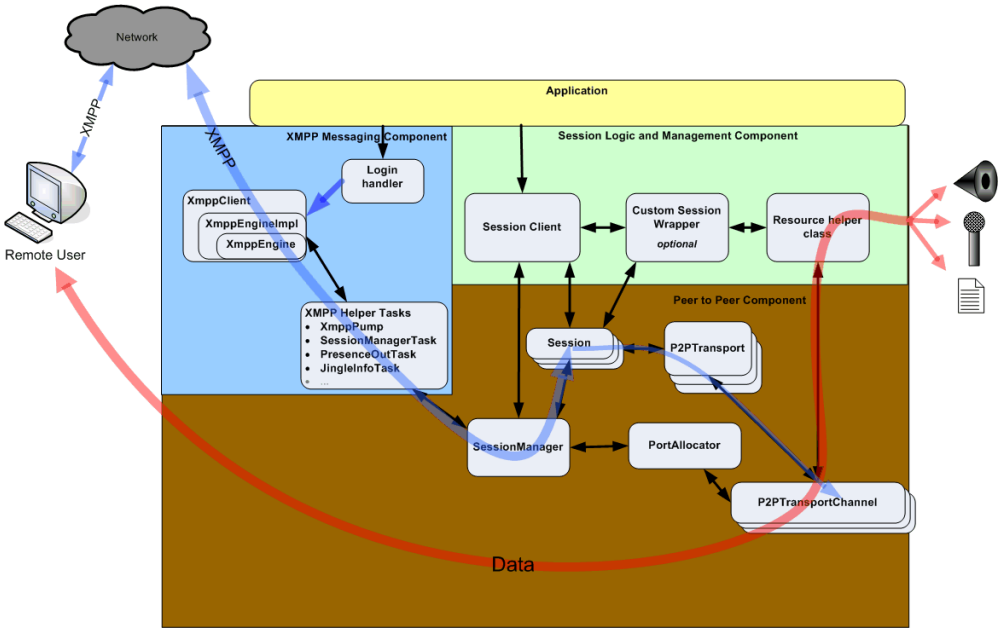
Generic libjingle Applications

The following diagram shows a simplified view of the three major components of a generic libjingle application. Some classes have been omitted for clarity. In this diagram, the data pathway is shown in red, and the XMPP pathway is shown in blue. Procedure calls between classes are shown as black arrows.



As the preceding diagram shows, a libjingle application typically consists of three main components, plus the user interface:

Application

On startup, the application (or a custom class in the session logic and management component) must explicitly create **SessionManager**, a **PortAllocator** subclass, a **SessionClient** subclass, a login handler, and any other custom objects from the Session Logic and Management Component that it needs. (More details on what an application needs to do are given in Creating a libjingle Application.) Communication between the application and these components is through direct method calling and callback notifications sent by the signaling mechanism (sigslot). libjingle supports multithreading so that the data channel and user interface thread do not block each other.

XMPP Messaging Component

This component is the gateway between the network and the program for XMPP stanzas. Incoming stanzas enter through this component and are typically routed to the Session Logic and Management Component. Outgoing stanzas are routed from the Session Logic and Management Component out to the network. This component is managed by a top level Task object, shown here as "Login handler." The sample applications use **XmppPump** as the handler. This handler manages **XmppClient**, which performs the actual sending and receiving of stanzas (with the help of **XmppEngine**). **XmppClient** also manages other helper task objects that handle specific jobs, such as listening for presence notifications (**PresencePushTask**) and sending presence notifications (**PresenceOutTask**). Several classes in other components host SignalRequestSignaling signals and OnRequestSignaling methods, which indicate that they are ready to send and receive stanzas.

Incoming session negotiation and information stanzas are typically routed to **SessionManager** via **SessionManagerTask**. **SessionManager** acts on them or forwards them to the appropriate **Session** object. XMPP stanzas from a connection request can filter down to the **P2PTransportChannel**, which reads the remote candidates when negotiating a connection. See Set Up the Session Management Pathway for more information about using the XMPP messaging component. Other stanzas, such as presence stanzas, are routed to any subscribed listeners; for example, **FileShareClient** creates a **PresencePushTask** to route presence stanzas to **FileShareClient**.

This component includes the following classes:

- **Login handler** is a helper object that handles the sign in process with the server. The application instantiates this class.
- **XmppClient** is the gateway for incoming and outgoing XMPP data and also manages the helper **Task** objects. The application instantiates this class, if the login handler does not.
- **SessionManagerTask /PresenceOutTask / JingleInfoTask** are helper **Task** objects managed by **XmppClient**. Each of these tasks is responsible for handling some job involving XMPP stanzas. **JingleInfoTask**, for instance, retrieves STUN and relay server information; **SessionManagerTask** is used by the **SessionManager** object to send redirection or offer messages from a **Session** object.

Session Logic and Management Component

This component handles the specific logic required for each session type. This component is the one that is most specialized for different applications, and most of your custom classes will probably be in this component. This component forwards incoming stanzas down to the Peer to Peer Component, which handles most of the connection details, and then reads and writes data from that component to a file, speaker, microphone, or other capture/render object. Specific tasks handled by this component include: translating custom session descriptions from XMPP to internal structures (and back again); accepting or rejecting incoming connection requests; initiating connection requests; and creating and managing capturers/renderers and other hardware and software resources. One of the specialized classes in this component manages the individual **Session** objects that receive XMPP stanzas to negotiate a connection. Another communicates with **P2PTransport** to send and receive the actual peer to peer data for that session. The session negotiation channel is managed by the **Session** object (which is *not* overridden for different session types).

This component includes the following classes:

- **SessionClient** understands the high level tasks common to all sessions. You must extend this base class to convert session-specific offer descriptions (such as a manifest, for file transfers, or a list of codecs used for voice calls) between the XMPP and any data structures you use internally . The application creates this object. See Extending SessionClient for more information. When **SessionManager** calls **SessionClient::OnSessionCreate** for either incoming or outgoing calls, your implementation must create any classes required for a specific type of peer to peer session (such as audio renderers).
- **Resource Allocator/Resource Classes** are additional helper classes specific to your session type. These include media rendering or capture classes, management or monitoring classes, as required. One of these classes is responsible for requesting a **TransportChannel** from **Session**, and using it to send or receive data.
- **Session** is the high-level wrapper for a session negotiation channel. Each **Session** object hosts one or more **Transport** objects, which create the data channel. **Session** communicates with **Transport** to create and send a list of local connection candidates, and monitor connection writability. You should not need to extend this class, but you will probably want to create a wrapper object (discussed below). **Session** objects are created using the **SessionManager**: for incoming connection requests, they are created automatically when **SessionClient** forwards an incoming connection request to **SessionManager**; for outgoing requests, the session wrapper explicitly requests creation of the object.
- The **SessionManager** object creates and destroys **Session** objects and forwards incoming stanzas to the correct **Session** object. All incoming stanzas are routed through the **SessionManager**, which tries to find the right **Session** object for the stanza. If this is a request for a new session from another user, it will create a new **Session** object, and pass the stanza to it, which will eventually bubble up an alert to the SessionClient, which should notify the user of the request. It also retains a map of SessionClient instances for specific Session types (identified by a unique ID in the request stanza as well as a global variable in the code). The application creates this object.

- **Custom Session Wrappers** are sometimes used to handle helper tasks associated with individual sessions. The file sharing example, for instance, uses **FileShareSession** to manage the conversion of a manifest to a **FileShareDescription** (a task associated with **SessionClient**). The voice chat example uses **Call** as a sort of session wrapper for multiple sessions, as a way to manage call initiation and termination, audio settings for multiple sessions, and also kicks off channel connection.

Peer-to-Peer Component

This component manages the connection between the local and remote computer. It sends and receives peer to peer data across the network, generates local candidates, allocates sockets, and manages and monitors connection quality. The data channel is created and managed by the top level **P2PTransport** object, which hosts a **P2PTransportChannel** that is the actual endpoint for data reading and writing. libjingle provides several port types, including UDP, TCP, and a mock SSL port. **P2PTransportChannel** abstracts data sending and receiving to a simple read or write command. The Session Logic and Management Component uses this component as a black box to generate candidates, maintain the connection, and send/receive data as requested. In general, you should not need to modify items in this component for different session types, although there are a very few scenarios in which you might.

- **PortAllocator**, together with its helper class **PortAllocatorSession** (not shown), allocates local ports in response to requests from **P2PTransport**. The application creates this object. It is extended to enable new kinds of ports and protocols—libjingle provides a subclass, **HttpPortAllocator**, which supports relay ports more naturally. A custom application Session Logic and Management Component class instantiates this object.
- **P2PTransport** is the top level creation object for the peer to peer data channel. Its purpose is to create and monitor the **P2PTransportChannel** objects that represent data connections, but it does not see any of the actual data itself. After creating a data connection, it hands this connection (a **P2PTransportChannel**) off to the requestor (the Session object). Only one channel, judged the best connection by write latency, is active at any given time. Incoming latency is not checked. In turn, each **Channel** has one or more **Connection** objects, which represent a local **Port**/remote address pair representing the actual data stream. **P2PTransport** switches connections automatically when a connection becomes unwritable. **P2PTransport** uses the helper class **PortAllocator** to find and allocate physical ports/sockets on the computer. **P2PTransport** is instantiated by **Session** when it is initiated by either party. Although **Session** can hold multiple **Transport** objects of different types, in the current code only one **P2PTransport** instance is used. (See Transports, Channels, and Connections for more information)
- **P2PTransportChannel** wraps a number of connections and continually monitors and selects the best one. This is the end of the data pipeline. Applications read data by listening for **P2PTransportChannel::SignalReadPacket** and calling **P2PTransportChannel::SendPacket** (or by calling similar methods on a wrapper class, such as **PseudoTcpChannel**).
- **Ports** and **Sockets** (not shown) are wrappers that handle the actual requests to send and receive data to and from the remote computer. A **Port** object wraps a physical port on the local computer by means of the **Socket** object that it hosts. A **Socket** object represents the physical socket on the local computer. The base **Socket** class is overridden for specific socket types. **Socket** objects are additionally wrapped by the **AsyncPacketSocket** class, which enables asynchronous packet reception. Sockets are created by **Session** or other objects and passed to a **Port** when it is created. The basic **Port** class is extended for specific types of ports (UDP, TCP, and so on). A **Port** object generates candidate lists (actually each port only generates one candidate) at a request from the **PortAllocator** object, which asks each type of port (TCP, UDP, Relay, etc) to generate a candidate list. Ports also send and approve bind requests from remote computers, read and write data to the socket asynchronously. **Port** objects are responsible for generating local candidates to send to the remote computer.

Details of a Session Connection

A peer to peer session is actually made up of two streams of information:

1. The high-level session management connection, which negotiates the who, what, and how of making a connection. This is XMPP information, and is sometimes called the signaling channel.
2. The data channel, which is the actual bytes of file or voice data being exchanged.

The diagram above shows the pathways for each of these two types of data.

Here is a general overview of how the different pieces of a libjingle application work together to handle a peer to peer session.

Startup

In order to start running a libjingle application, you must first instantiate and set up a number of required objects. This includes a few objects from each component: the XMPP task manager, **SessionManagerTask** to listen for incoming messages, **NetworkManager**, **PortAllocator**, **SessionManager**, a **SessionClient** subclass and perhaps another Session Logic and Management Component, plus any required **Thread** objects.

Signing in

Once your application is running, you must sign in to the XMPP server in order to connect to other computers. Send the user's XMPP server name and password to **XmppClient::Connect** directly, or else use the **XmppPump::DoLogin** helper task (recommended). **XmppClient** sends status signals indicating the status of the sign in request. See Signing In to a Server for more details.

If sign in has succeeded, the **XmppClient** sends the SignalStateChange(STATE_OPEN) signal. The application then sends user status information to the server and requests status information for the user's roster members. **PresenceOutTask** creates and sends the user status out to the server. **PresencePushTask** listens for roster member presence notifications from the server, and whenever it receives a presence stanza, it sends a SignalStatusUpdate signal with information, and the application can alert the user.

The following sections describe a connection attempt between two users, Romeo and Juliet.

Connecting

After the application receives roster member status, it can display the list of available members to the user, who can then decide to send a connection request as described here.

In order to connect the data stream, the two computers must negotiate the data connection details over XMPP. This negotiation involves the exchange of two important pieces of information:

- **Session request specifics** [from the initiator to the target]. This describes who you are (by JID), what you want (to connect), and what you want to exchange (files, voice data). It includes a description element specific to the session type: for a voice chat, it would include a codec list; for a file exchange, it would include a file name, data direction, and other information. Both applications must be configured to understand a custom session description stanza. In reply, the target sends an acceptance or rejection stanza. An acceptance stanza might include a description element of its own.
- **A transport list** [from each computer to the other computer]. This is a list of connection information such as local connection candidates (generated by the **Port** objects under the direction of the **P2PTransport** object), as well as other connection information such as priority and password information. For a list of what types of what types of candidates are generated, see Candidates and Transports .

In this latest version of libjingle, both of these are in the initial XMPP connection request stanza; in the previous version, the transport list was a candidate list, and it was contained in a separate stanza. The person receiving the connection request must reply with a selected transport in order for the connection to begin.

Romeo sends a connection request to Juliet

Romeo's application instantiates a new **Session** object. It does this by creating the top level Session Logic and Management wrappers, and having one of them call **SessionManager::CreateSession**, which creates the new **Session** object and returns it to **SessionClient::OnSessionCreate**. After creating **Session**, the application connects to all its signals to be alerted when the session changes status (accepted or rejected), as well as to receive notifications of incoming XMPP messages.

Next, the application creates a description of the session and wraps it in an appropriate **SessionDescription** subclass. Voice call descriptions include codecs; file share descriptions include file names.

Next, the Session Logic and Management wrapper calls **Session::Initiate** on the new session, passing in the description. This causes it to take the following steps:

- **Session** creates a set of potential transports it can use. Each **P2PTransport** creates a **Port** object, each of which generates a candidate, and sends the compiled list back to **Session**. This list includes a default transport as its highest priority transport.
- **Session** sorts the transport offer list with highest preference first, and generates an XMPP offer stanza with the description and transport list and sends it to Juliet.
- **Session** tries immediately to connect with its default channel.

The **Session** object sends signals whenever it changes state as a result of incoming or outgoing stanzas (sent session invitation, received session invitation, sent accept, running, and so on). **Session** also sends a signal (SignalInfoMessage) whenever it receives an informational message. These messages are customized to the session type--so, for example, the file sharing application sends a "no more files" informational message when all files have been requested so that the receiver can shut down.

Juliet receives the connection request

The stanza is caught by **SessionManagerTask**, which forwards it to **SessionManager**, which recognizes a connection request. **SessionManager** creates a new **Session** object and copies the description and candidate list to the new **Session** object. When **SessionManager** creates the new session, it sends a SignalSessionCreate notification, which includes the direction of the request (here: *incoming* to indicate that it is an incoming request). Juliet's **SessionClient** subscribed to that notification when it was created, and it pops up a dialog box telling Juliet that her roster buddy Romeo is requesting a connection.

Without waiting for Juliet to accept, **Session** tells **P2PTransport** to start generating its own local list of transports. When these are ready, **Session** searches for the first matching transport from Romeo's list that is also in its own list.

If, after being notified about the incoming request, Juliet decides to accept Romeo's request, she calls **Session::Accept** on the **Session** object that her application created for the new connection request. Session then creates its own connection description, which includes her chosen transport as well as its own session-specific description object (if necessary) and sends this back to Romeo. It immediately tries connecting over the accepted transport.

If Juliet decides not to accept, she calls **Session::Reject**, which sends a rejection message and destroys the associated **Session** object. If Juliet cannot find a transport on her list that she can accept, she will send a rejection message, and **SessionManager** will destroy the session.

(The order of stanza exchanges differs for old clients that do not support the `<transport>` tag but libjingle is backward compatible with these clients.)

Romeo receives Juliet's acceptance stanza

The received stanza is sent down to the **Session** object, which calls OnTransportAcceptMessage to verify that one of the transports it sent out was included in the reply. It then sets the returned transport as the active **P2PTransport** for the **Session**, and tells the transport to create and connect channels with the candidate accepted. As soon as the socket can be written to, Romeo can begin sending data.

Session changes its state to STATE_RECEIVEDACCEPT, which will later be changed to STATE_INPROGRESS. **Session** sends a signal with the new state, so the application can start sending data when appropriate.

See Transports, Channels, and Connections for more information about connections.

The Peer to Peer session begins

The data session continues until one of the issuers sends a redirect or terminate request, or the data channel is broken.

Cleaning Up

Exact details about cleanup vary depending on the client type, but the cleanup of the common objects (**Session** and **P2PTransport**) are the same. When a client calls **Session::Reject** (for an incoming request) or **Session::Terminate** (for an established connection), the client will send a Terminate message to the other client, and tell **SessionManager** to delete the session, and call the derived version of **SessionClient::OnSessionDestroy**. This derived function must free up any allocated resources for that session, and also delete the associated **P2PTransport** object (by calling **Session::DestroyChannel** on the session sent to **OnSessionDestroy**).

To sign out of a session with the XMPP server, call **XmppClient::Disconnect**.

When a client receives a Terminate message from the other party, it will send the message to the proper **Session**, which handles it the same way as when a client calls **Session::Terminate** itself.

When a connection is broken during an exchange of data, **P2PTransport** tries to reconnect over all the connections for a specific period of time. If the connection cannot be reestablished within a timeout period, it terminates the session as just described. The timeout period is specified by the timeout_ member of **SessionManager**.

When a connection is broken during an exchange of XMPP information, the sending client will receive an error message, which will cause the session to terminate as described previously.

If one user signs out without explicitly terminating the session, no Terminate message is sent. If the user that signs out actually closes the network connection, the session will terminate because of connectivity as described earlier whether or not data is being sent at the time (libjingle sends periodic pings over all the ports to determine connectivity, and if they all fail and it cannot reestablish a connection, it will terminate). However, if the network connection is not broken, these pings will succeed, and the session will not end until one or the other client either closes or the network connection is broken. In order to avoid this scenario, your application should monitor changes to the roster (by using **PresencePushTask**).

More Details

For more specifics about how to run a libjingle application at a high level, see Creating a libjingle Application.

For more specifics about how the file share or voice chat applications work, see File Share Application or Voice Chat Application.

Last updated March 23, 2012.