Important Concepts

You should understand the following important concepts about libjingle:

Signals

libjingle uses the **sigslot** library to facilitate communication between objects. sigslot is a generic framework that enables you to connect a calling member to a receiving function in any class (including the same class) very simply. The way it works is this:

1. The sending class declares a member variable, called a *signal*, using a special template-like syntax. This signal defines the parameters of the listening function.
2. The listening class implements a function with the same number, type, and sequence of parameters as the signal. This is sometimes called the receiver or the *slot*. (Note: this can even be the same class as the one that declared the signal.) This function cannot return a value (e.g., returns void). The receiver must inherit `sigslot::has_slots<>`.
3. The listener connects to the signal by calling the signal's **connect** method, passing in a pointer to the instance of the listening object, and the address of the implementing class function.
4. The sender calls its signal member as if it were a function, passing in the appropriate parameter types as declared. It can pass parameters by either value or reference.

You can connect as many signals as you like to a common slot. libjingle sometimes assigns multiple signals to a single slot in order to consolidate its message handling. Conversely, several objects declare a signal object in order to broadcast commonly needed messages from a single point (for example, alerts sent when a connection state changes). `sigslot` takes care of disconnecting callbacks and dereferencing when objects are destroyed.

The following code demonstrates using sigslot:

```
// Class that sends the notification.
class Sender  {

  // The signal declaration.
  // The '2' in the name indicates the number of parameters. Parameter types
  // are declared in the template parameter list.
  sigslot::signal2<string message, std::time_t time> SignalDanger;

  // When anyone calls Panic(), we will send the SignalDanger signal.
  void Panic(){
    SignalDanger("Help!", std::time(0));
  }

 // Listening class. It must inherit sigslot.
class Receiver : public sigslot::has_slots<>{

  // Receiver registers to get SignalDanger signals.
  // When SignalDanger is sent, it is caught by OnDanger().
  // Second parameter gives address of the listener function class definition.
  // First parameter points to instance of this class to receive notifications.
  Receiver(Sender sender){
        sender->SignalDanger.connect(this, &Receiver.OnDanger);
  }

  // When anyone calls Panic(), Receiver::OnDanger gets the message.
  // Notice that the number and type of parameters match
  // those in Sender::SignalDanger, and that it doesn't return a value.
  void OnDanger(string message, std::time_t time){
    if(message == "Help!")
    {
      // Call the police
      ...
```

```
    }
  }
...
}
```

Many classes in the code send signals to notify listeners of important events. For example, **Call::SignalSessionState** sends notifications when you send or receive a connection attempt. Your application must connect to these signals and act appropriately.

The general convention in libjingle code is to prefix the name of a signal with *Signal:* e.g., SignalStateChange, SignalSessionState, SignalSessionCreate. Listener methods intended to connect to signals are typically prefixed with *On*, e.g., OnPortDestroyed(), OnOutgoingMessage(), OnSendPacket().

See the sigslot documentation for more details.

Threads

libjingle supports multithreading in order to improve the performance of your application. libjingle components use one or two globally available threads:

- The **signaling thread** is the thread used to create all the basic components, such as the Session Management and Control and XMPP Messaging components.
- The **worker thread** (sometimes called the *channel* thread in the code) is used by the Peer to Peer component objects to handle more resource intensive processes, such as data streaming. Putting these on a separate thread prevents data flow from blocking or being blocked by XMPP or user interface components. Classes using the worker thread include **ChannelManager**, **SocketMonitor**, **P2PTransportChannel**, and the **Port** objects. To enable a second thread, you must create and pass a new **Thread** object to the **SessionManager** constructor. (If no thread is passed in, the thread in which **SessionManager** was created will be used as the worker thread). **CallClient::InitPhone** demonstrates creating a worker thread for the low-level components.

Additionally, libjingle now provides a base class called **SignalThread**. Extend this class to enable an object that exists on its own thread, and which can be instantiated, started, and left alone to complete and delete itself when done. See signalthread.h/.cc for more information.

Note: Although libjingle supports multiple threads, only certain methods support thread safety by verifying the calling thread, and very few methods do any locking. The following snippet demonstrates how a method verifies which thread it is being called on:

```
// Check that we're being called from the channel (e.g., worker) thread.
ASSERT(talk_base::Thread::Current() == channel_thread_);
channel_thread_->Clear(this);
```

libjingle wraps all threads, the signaling thread, the worker thread, and any other threads, with the **talk_base::Thread** object (or a derived object). All **Thread** objects are managed by **ThreadManager**, which retrieves them on request. **SessionManager** calls **ThreadManager::CurrentThread** to provide it with a signaling thread (and a worker thread, if none is provided) when it is instantiated; **XmppPump** uses the current thread for its signaling thread. Therefore, you must create a **Thread** object (or derived object) for the signaling thread and push it into **ThreadManager**'s thread pool before creating a **SessionManager** object, or before expecting the **XmppPump** to start working. (See Signing In to a Server for example code.) There are two ways to create a **Thread**:

- **AutoThread** This wraps the existing operating system thread with a libjingle **Thread** object and makes it the current thread in the **ThreadManager** object's thread pool (that is, will return the thread if **Thread::CurrentThread** is called).
- **Thread** This creates and wraps a new thread to use, typically for a worker thread. In order to use this thread, you have to create a new **Thread** object, call **ThreadManager::Add** or **ThreadManager::SetCurrent** to add it to the pool, and call **Run** to start it in a blocking loop, or **Start** to start the thread listening.

Threads provide a conduit for messages between (or within) objects. For instance, **SocketManager** sends a message to itself on another thread to destroy a socket, or to **SessionManager** when connection candidates have been generated. The **Thread** object inherits **MessageQueue**, and together they expose **Send**, **Post**, and other methods for sending messages synchronously and asynchronously. An object that will receive messages sent using **MessageQueue** must inherit and implement **MessageHandler**. **MessageHandler** defines the **OnMessage** method, which is called with the **MessageQueue** messages.

You can send messages to any object that inherits `talk_base::MessageHandler` over any thread. However, if sending a message to perform a resource-intensive thread, you should send the message over the worker thread. You can get a handle to the worker thread by calling **SessionManager::worker_thread()**. You can get a handle to the signaling thread by calling **SessionManager::signaling_thread()**.

An object has several ways to access a specific thread: it can request and store a thread pointer as an input parameter; it can assume that the current thread when it is created (accessed by **ThreadManager::CurrentThread** in its constructor) is a particular thread and cache a member pointer to it; it can call **SessionManger::signal_thread()** or **SessionManager::worker_thread()** to retrieve threads. All three techniques are used in libjingle.

Because an object can be called on any thread, an object may need to verify which thread a method is being called from. To do this, call **Thread::Current** (which retrieves the current thread) and compare that value against a known thread--this can be one of the threads exposed by **SessionManager**, or the object can store a pointer to its initial thread in the constructor. Here is a more extended example of calling a method in the same object on another thread.

```
// Note that worker_thread_ is not initialized until someone
// calls PseudoTcpChannel::Connect
// Also note that this method *is* thread-safe.
bool PseudoTcpChannel::Connect(const std::string& channel_name) {
  ASSERT(signal_thread_->IsCurrent());
  CritScope lock(&cs_);
    if (channel_)
      return false;
    ASSERT(session_ != NULL);
    worker_thread_ = session_->session_manager()->worker_thread();
...
}

void PseudoTcpChannel::SomeFunction(){
  ...
  // Post a message to yourself over the worker thread.
  worker_thread_->Post(this, MSG_PING); // <- Goes in here....
  ...
}

// Handle queued requests.
void PseudoTcpChannel::OnMessage(Message *pmsg) {
  if (pmsg->message_id == MSG_SORT)
    OnSort();
  else if (pmsg->message_id == MSG_PING) // -> And comes out here!
    // Check that we're in the worker thread before proceding.
    ASSERT(worker_thread_->IsCurrent());
    OnPing();
  else if (pmsg->message_id == MSG_ALLOCATE)
    OnAllocate();
  else
    assert(false);
}
```

Naming Conventions

libjingle has some naming conventions that it is useful to be aware of:

- On*SomeMethod* Methods beginning with "On" are often connected to a signal, either from this or another object. If called from the same object, it is probably called on a different thread.
- *SomeMethod*_w Methods ending with "_w" exist in the "worker thread" and are called from another thread
- Signal*SomeName* These are the signals that send messages to callback methods.

SSL Support

libjingle supports two types of SSL:

- OpenSSL (for UNIX)
- SChannel (for Windows)

To use SSL, you must perform the following steps:

1. **#define FEATURE_ENABLE_SSL** (in the Visual Studio project, this value is defined in the project settings, not in the code).
2. **Ensure that *either* SSL_USE_OPENSSL or SSL_USE_SCHANNEL are #defined in ssladapter.cc**. One of these should be defined by default, depending on the build settings for your operating system.
3. **Call InitializeSSL to initialize required components**. This function is defined in ssladapter.cc. When the application closes down, call CleanupSSL. You do not need to call InitializeSSLThread (it is used internally by InitializeSSL).

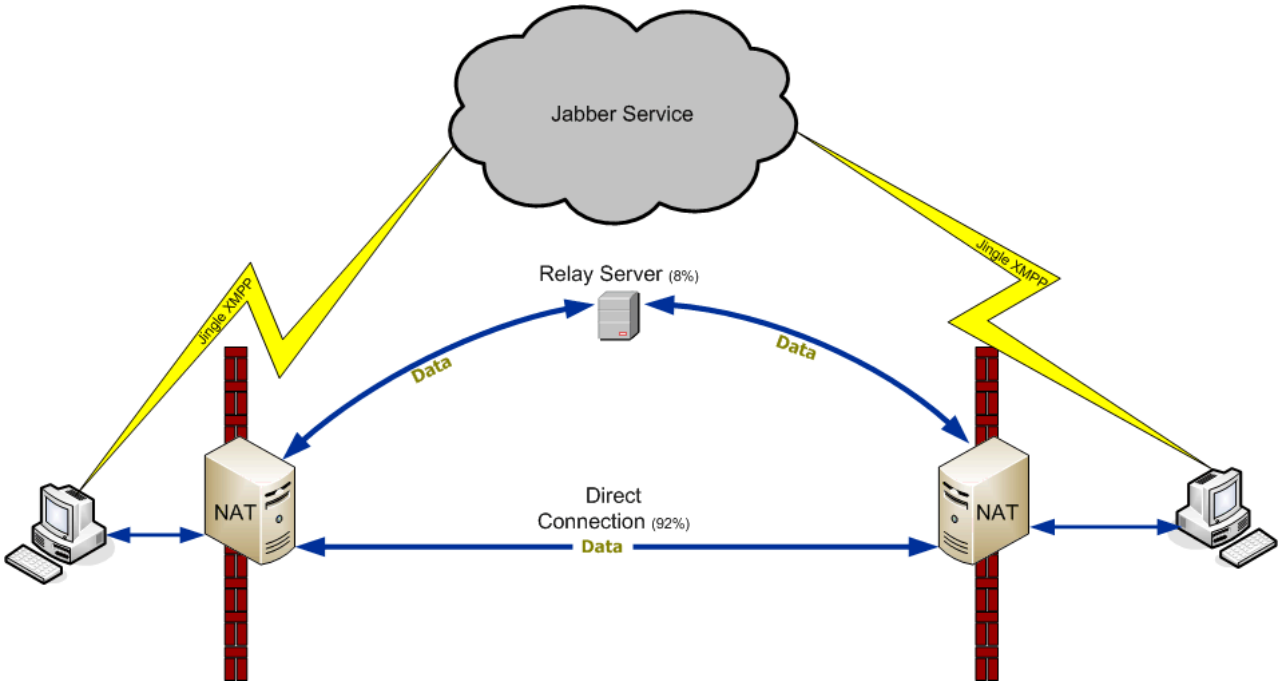
Connections

A libjingle peer-to-peer connection actually consists of two channels:

- The **session negotiation channel** (also called the signaling channel) is the communication link used to negotiate the data connection. This channel is used to request a connection, exchange candidates, and negotiate the details of the session (such as socket addresses, codecs needed, files to be exchanged, connection change requests, and termination requests). This is the first connection made between computers, and only after this connection is made can the data channel be established. libjingle uses a precursor to Jingle to specify the stanzas and responses needed to establish the data connection (see Jingle and libjingle) This channel sends stanzas through an intermediary XMPP server; the example code uses the Google Talk server as the intermediary.
- The **data channel** carries the actual data (audio, video, files, etc) exchanged in the peer-to-peer session. Data channel data is wrapped in TCP or UDP packets, depending on the transport negotiated, and does not go through the XMPP server.

The session negotiation channel is established first, as the computers negotiate the details of the data channel; after the data connection is made, most activity occurs on the data channel, except for the occasional requests for a codec change, a new file request, a redirect request, or a termination request.

The following diagram shows these two pathways. In the diagram, two alternate data paths are shown, although only one data pathway will be active in a connection. This is because the data pathway can be either a direct connection (92% of connection attempts can take place directly) or through a relay server (8% of connection attempts require an intermediary relay server). A third data pathway, not shown, is a direct connection from computer to computer when there is no intermediary firewall.



Notes:

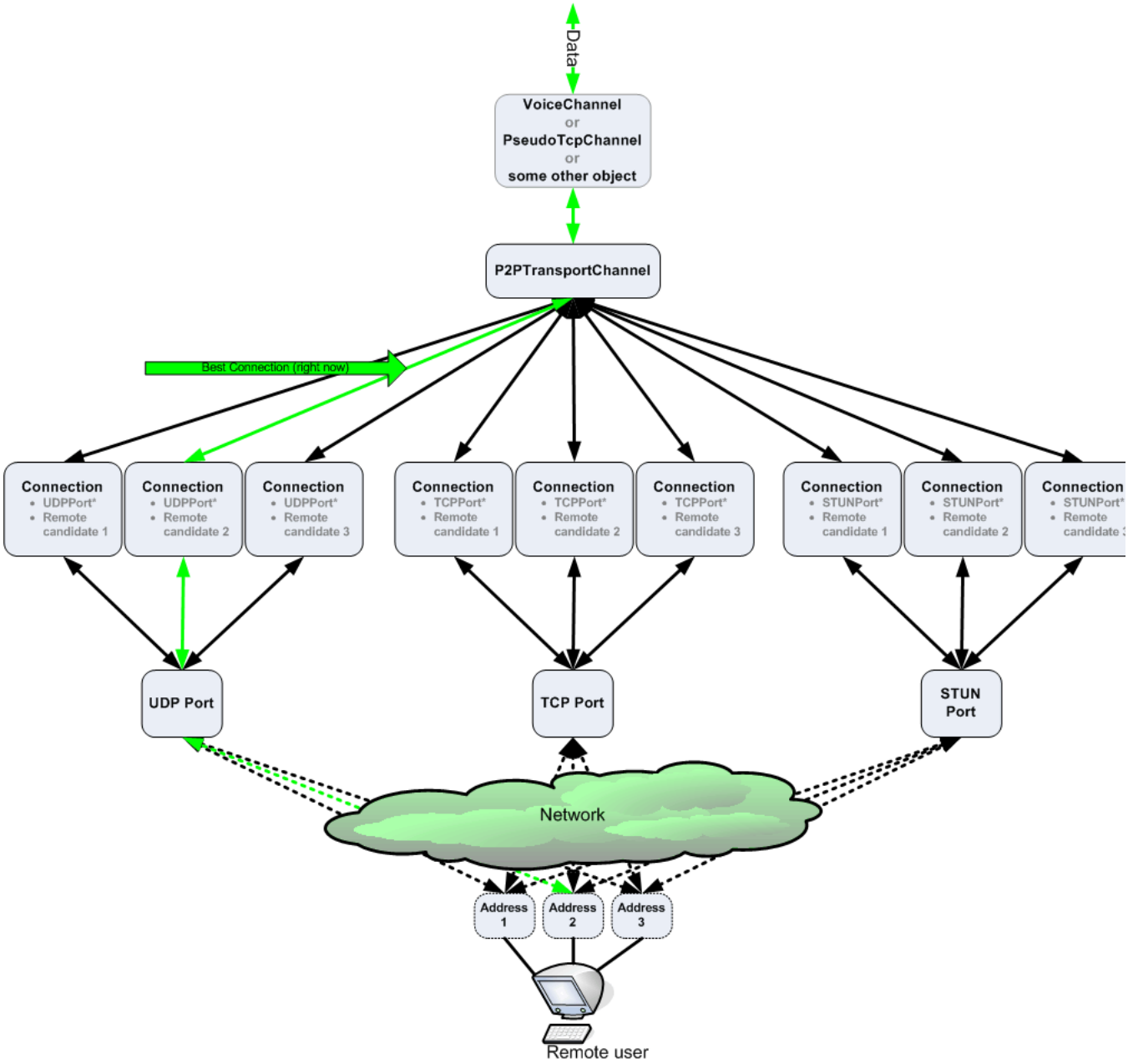
- libjingle sends out occasional STUN packets to maintain writability, keep firewall and NAT address bindings active, and check connection latency.
- libjingle assigns a user name and password to connection ports. This ensures that the computer connecting on the data channel is the same one that negotiated the connection over the signaling channel. ***Because these username and password values are sent over XMPP and may or may not be encrypted with TLS, these values in a STUN packet are used only for identification, not cryptographic authentication.***

To see the actual stanzas being sent, run the file share sample application.

Transports, Channels, and Connections

Each **P2PTransportChannel** represents a data channel between the local and remote computers. This channel actually obscures a complex system designed for robustness and performance. **P2PTransportChannel** manages a number of different **Connection** objects, each of which is specialized for a different connection type (UDP, TCP, etc). A **Connection** object actually wraps a pair of objects: a **Port** subclass, representing the local connection; and an address representing the remote connection. If a particular connection fails, **P2PTransportChannel** will seamlessly switch to the next best connection.

The following diagram shows a high level view of the data pathway inside the Peer to Peer Component.



When a libjingle application negotiates a connection with a remote computer, it creates a list of potential connection points, called *candidates*, on the local computer. Local candidates wrapped by **Port** objects, which are allocated by the **PortAllocator** subclass. Local **Port** objects are created by the offering computer before it sends out the offer, or by the receiving computer when it gets the request (if it hasn't already generated a list of local ports for any other reason). When **P2PTransportChannel** receives a connection offer from another computer (which includes the remote candidates), it creates one **Connection** object to wrap each remote candidate/local **Port** pair.

libjingle also defines a class named **RawTransport** that supports direct connections between two UDP connections, without using ICE. This transport would be used when you can create a direct UDP connection, or if one of the parties doesn't understand the ICE mechanism.

P2PTransportChannel creates and manages multiple **connection** objects. It evaluates them by writability and preference (UDP has a higher preference than a relay port connection, for example), and selects the best one to use. This may change as connections are broken or performance changes, but **P2PTransportChannel** switches connections seamlessly and invisibly to any classes higher up the chain.

P2PTransport (not shown) is the top-level creation and management object for the P2P data system. It creates and destroys the **P2PTransportChannel**, and monitors its general performance, but does not actually handle data; the true entry point for the data pipeline is **P2PTransportChannel**. **VoiceChannel** and **PseudoTcpChannel** connect to **P2PTransportChannel** to read and write their data.

The **Session** object hosts the **P2PTransport** object and requests creation of data channels. Although the **Session** object can potentially host multiple instances and subclasses of **Transport** objects, the current version of the code only defines and uses one instance of the **P2PTransport** subclass.

Candidates

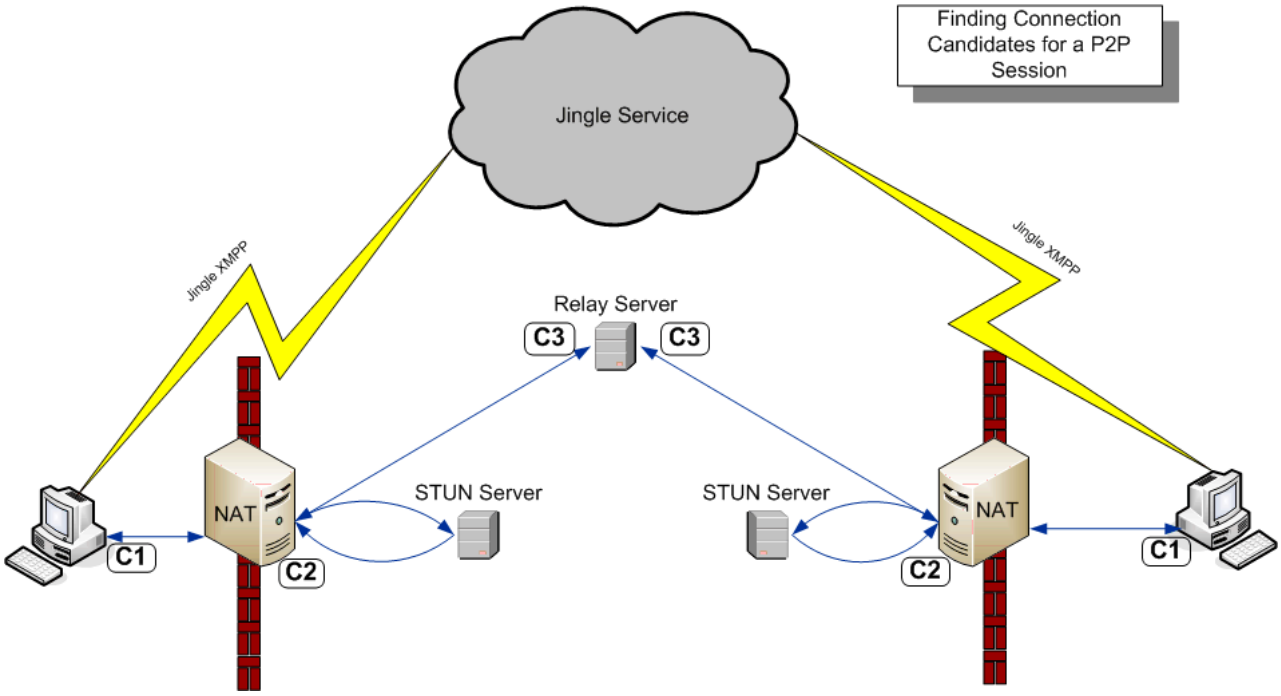
One of libjingle's key benefits is its ability to negotiate connections across firewalls or Network Address Translation (NAT)-enabled devices. libjingle uses the Interactive Connectivity Establishment (ICE) procedure to connect through a firewall. The first step a libjingle application does when trying to negotiate a connection is to generate a list of potential local port addresses for the other computer to connect to. Each of these potential

addresses is called a *candidate*. Candidates are IP:port pairs to which both the application and the other computer can connect (technically, it only listens on the local connection). libjingle provides a robust mechanism for discovering valid local connection candidates that other computers can access, even through NAT devices or firewalls.

In order to provide as many potential connection addresses as possible to the other computer, libjingle generates three kinds of local candidates:

- **Local IP addresses** One candidate is a local IP addresses on the computer. Other computers sharing the same local network should be able to access this address.
- **Global addresses** A second candidate is an external address on a NAT or firewall device between the two computers. If this is outside a NAT device, libjingle uses STUN to cause the NAT to bind to your computer and expose a global address. This address is used as a candidate to connect from outside the NAT device.
- **Relay server addresses** Approximately 8% of clients attempting to connect across a firewall cannot make contact through either of the previous methods. A third method of connecting is through a relay server, a free-standing server that lives in the network space between the two firewalls. Although libjingle is capable of using a relay server, no relay server URI is provided. However, libjingle includes code for a relay server (relayserver.h/cc). You can build and run this server yourself, and use its IP address as the third parameter in the **BasicPortAllocator** constructor.

The following diagram shows two computers generating local address candidates (C1), external NAT candidates (C2), and Relay server candidates (C3).

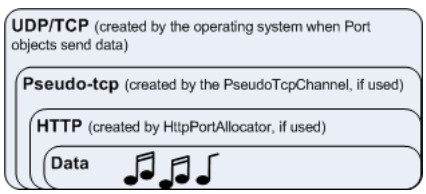


libjingle stores the full list of potential candidates so that even after a connection is made, it can quickly switch to a new connection if the current connection slows down or breaks.

libjingle now includes support for multiple transports, in the spirit of the Jingle `<transport>` element. A transport can contain much more information than a simple candidate address: for example, the ICE transport tag supports ICE-specific information such as priority, password, and user fragments. Although this is the preferred way to negotiate connections, for backward compatibility purposes libjingle still supports clients that still use the older bare `<candidate>` stanza. See the Jingle ICE Transport Specification for an example of a transport specification.

Data Packets

Peer to peer data sent between computers can be wrapped in multiple layers of protocols, as shown here, depending on the application:



Not all applications use all layers: for instance, the file share application uses pseudo-tcp, but the voice chat application does not.

Last updated March 23, 2012.