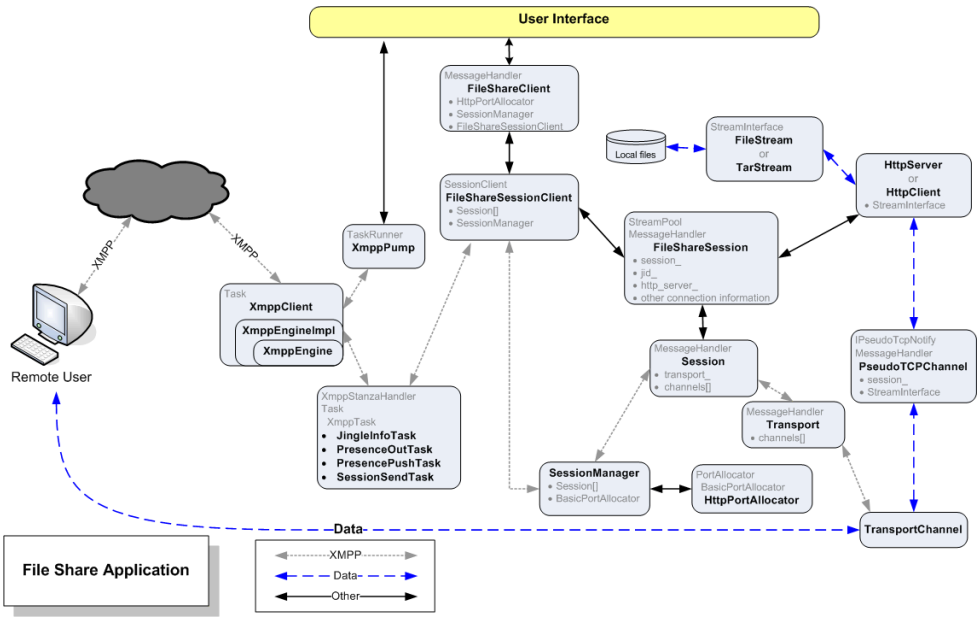
File Sharing Applications

A common use of peer-to-peer sessions is to exchange blocks of sequential data, such as files. The file share sample application shows how to use these classes in a peer-to-peer file copy application. See File Share sample application for information about how to run the sample.

Overview of the Sample File Sharing Application

The following diagram shows the important peer-to-peer file-sharing classes provided by libjingle. 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 preceding diagram shows the key elements of a file share application. The rest of this page discusses specifics of this application, and how they differ from a generic libjingle application.

Although you could modify the voice chat sample application to handle file data, the connection would only be as good as a UDP connection: that is, individual packets might be lost, re-ordered, or duplicated. While this would be acceptable for low-fidelity connections such as a voice chat, a file transfer program would require a more reliable streaming protocol, such as TCP. libjingle includes the **PseudoTcp** and **FileShareSession** classes to provide this capability. The file share application includes the following important classes:

FileShareClient is the top level manager for the file share application. After the application signs in to the server and creates a manifest containing all files to share, it instantiates the base **NetworkManager**, **HttpPortAllocator**, **SessionManager**, and **FileShareSessionClient** objects needed for all libjingle applications, sends and listens for presence notifications. When **FileShareClient** detects the presence of the requested share target, and when they are detected, calls **FileShareSession** with the manifest and JID of the target.

FileShareSessionClient Translates between **FileShareManifest**, used internally to represent the list of files and folders to share, and the XMPP stanza sent across the wire. It acts as a notifier to bubble up important notifications, such as session creation, to **FileShareClient**.

FileShareSession acts as the manager between the **Session** object and the data connection. It listens for incoming calls, and when it detects one, creates an **HttpServer** to serve out files, and handles file retrieval and streaming for requested files. For a client, it creates an **HttpClient** object, loops through the list of files passed in by **FileShareClient**, and requests each file in turn. More details are given in the next section.

PseudoTcpChannel enables sending TCP-like packets through a firewall. It is typically easier to make a UDP connection through a NAT than to make a TCP connection. Therefore, **PseudoTcpChannel** is provided to enable TCP-like functionality to UDP packets. Each **FileShareSession** object creates its own **PseudoTcpChannel** object when a connection is established. It creates a **TransportChannel**, which provides the external data connection, by calling **Session::CreateChannel** on the **Session** object passed into its constructor. It exposes a **StreamInterface** used by internal components to read/write data to the remote computer. In the file share application, **PseudoTcpChannel** acts as an intermediary between the channel and **HttpServer** or **HttpClient** to wrapping or unwrapping data with a pseudo-TCP layer. **PseudoTcpChannel** is created by **FileShareSession** when its associated Session object receives an informational XML stanza with a QN_SHARE_CHANNEL member.

HttpServer / HttpClient These two objects act as a requestor and server for content using HTTP GET/POST methods. The client is instantiated with a stream pool that provides **StreamInterface** handles to write to, and FileShareSession tells it to send a GET command to the URL of the server (which is based on the JID). When a response is received, it sends a notification, and the HTTP document object can be accessed by reading the **HttpClient::response()::document StreamInterface** passed into the completion notification (SignalHttpClientComplete). The file share sample passes its own **TarStream** interface (or **FileStream**, for a single file) into the object before the request, so that files will be copied to the temporary location of its choice.

The server is much simpler, listening for requests, and sending a SignalHttpRequest notification when it is received. It can stream content back to the requester by specifying a **StreamInterface** object in the **HttpTransaction::Response::success** member returned in the HttpServer::Respond method.

FileStream / **MemoryStream** / **TarStream** / **StreamAdapterInterface** These are stream classes used to manage reading and writing data, either locally or between computers. **FileStream** is used to read to/from a local file; **MemoryStream** is used to read/write to RAM; **TarStream** is to read a compressed TAR file; **StreamAdapterInterface** is used as a base class for classes that want access to the data as it is streaming through (for example, **StreamCounter**, which signals a count of bytes read or written). There are other stream types not mentioned here.

XmppClient, **SessionManager**, **HttpPortAllocator**, **Transport, TransportChannel, FileStream, TarStream** These objects are covered in the description of a generic libjingle application.

How the File Share Application Works

The file share example application uses HTTP to request and send files across the network. The client runs an instance of **HttpClient**, which sends a basic GET request to the connection negotiated by the **P2PTransport** objects. The server runs an instance of **HttpServer**, which receives connection requests, creates a **PseudoTcpChannel** to handle the connection, then creates and sends a pointer to a StreamInterface object that provides access to the files. The file is read using this stream, sent out through the **PseudoTcpChannel** object, which wraps the data, out through its **TransportChannel**, across the network to the receiver, which receives data through a **TransportChannel**, which converts it back to a **StreamInterface** stream, fed back through the **PseudoTcpChannel** to the **HttpClient** object, which stores the data on disk.

Here are the basic steps taken by **FileShareSession** to manage file/folder transfer. The file share example application uses a managing object, **FileShareClient**, defined in pcp_main.cc to manage the high level aspects of file transfers.

1. The sender creates a **FileShareSession** object and an associated **Session** object with the JID of the person to share with. **Session** handles session negotiation and connection is handled as described in Generic libjingle Applications.
2. The sender calls **FileShareSession::Share**. The object creates an **HttpServer** object that will manage sending the data, creates a **FileSessionDescription** object from the manifest passed in, generates unique identifiers (directory paths) to indicate preview and non-preview and preview paths in the description, and then calls **Session::Initiate** to

begin session negotiation with the receiver. Note that **FileSessionClient::CreateSessionDescription** automatically generates a preview image request for every image request; this preview image is prefixed with the preview identifier path in the manifest.

3. The receiver gets the request, which causes the **SessionManager** to create a **Session** object and send the STATE_RECEIVEDINITIATE message. **FileShareSession** captures this and calls **OnInitiate**, which extracts the source and preview identifiers from the description, and caches the direction of the session (incoming our outgoing). Asynchronously, the application sends a notification of the incoming request.
4. If the user accepts the connection request, the application calls **FileShareSession::Accept**, which creates an **HttpClient** object that will send file requests, and an **HttpServer** object that will serve previews of already downloaded content. The application calls **Session::Accept** to send the reply, and calls **FileShareSession::NextDownload** to start requesting the items in the manifest.
5. **NextDownload** first determines whether all manifest items have been downloaded: if so, it replies with an XMPP informational message that requests are complete; if not, it gets the name of the next item in the manifest, by means of a global counter, which is then incremented. The function then creates a new local file/folder object with the same name as the name of the item being downloaded (a file or a folder), creates a stream object for writing into this object (a **TarStream** for files, or a **FileStream** for files), creates an HTTP GET query for the item with the **HttpClient** object, associates the stream with this query (by setting **HttpClient::response()::document**), and finally sends the HTTP request. This function also sends an XMPP info stanza with QN_SHARE_CHANNEL.
6. The sender receives the QN_SHARE_CHANNEL message and creates a **PseudoTcpChannel** connection to the requesting computer. It receives the GET request in its **HttpServer** object, which calls **FileShareSession::OnHttpRequest**. This method checks parses the query, determines whether it is for an image preview (indicated by the presence of the preview ID path), and if so sends the SignalResampleImage signal, which indicates that the application should resample the image down to the specified preview size (in **FileStreamClient** this is handled by a dummy method that does not actually resample the image; you should implement this image to enable preview thumbnails). Next, the method creates a **StreamInterface** subclass stream reader for the specified file and wraps it in the current **Transaction** object, and calls **HttpServer::Respond** with the **Transaction** to start sending data back to the receiver.
7. The receiver continues streaming data through its **StreamInterface**, until all data is done, and the **HttpClient** object sends **HttpClient::SignalHttpClientComplete**, which is handled by **FileShareSession::OnHttpClientComplete**. This method moves the download from the cache directory to another download directory, then calls **NextDownload**, which will send the next GET request.

There are a few complications not covered in this description, such as the counter object that tracks the bytes downloaded, but they are not essential to understanding the process.

Threads

The file share example uses only a single main thread: the worker and signaling thread are the same thread, created by the main() function. However, HTTP requests are handled on their own threads, enabled by the **AsyncHttpRequest** object, which always has its own thread (it extends **SignalThread**).

Last updated March 23, 2012.