On Sending Files via OSCAR

Google Summer of Code 2005
Gaim Project

By Jonathan Clark

Table of Contents

Introduction........3
Contact Information........3
Acknowledgments........3
Who Should Read This........3
Background........3
Overview........4
Recommended Tools........4
Example Implementations........4
Conventions........4
Data Structures........5
Overview........5
Client to Messaging Server Communication........5
Rendezvous Data TLV Block........5
Server-Sent Acknowledge........8
Client to Client Communication via Proxy........8
Rendezvous Proxy: Header........9
Rendezvous Proxy: Error (0x0001)........9
Rendezvous Proxy: Initialize Send (0x0002)........10
Rendezvous Proxy: Acknowledge (0x0003)........10
Rendezvous Proxy: Initialize Receive (0x0004)........11
Rendezvous Proxy: Ready (0x0005)........12
Client to Client Direct Communication........12
OFT File Transfer Header........12
Sequence of Events........16
Overview of Scenarios........16
Direct Peer to Peer Connections........17
Redirected Connection........18
Stage 1 Proxy Server Connection........18
Stage 2 Proxy Server Connection........19
Stage 3 Proxy Server Connection........20
Exceptions to the Rules........21
Gaim's OSCAR Protocol Plug-In........22
Overview........22
Changes Made for Summer of Code 2005........22
Code Paths........22
Conclusion and Personal Notes........26
Errata........27
January 4, 2006........27

Introduction

Contact Information

If you would like more information, find any errors in this document, or would like to contribute additional information, you can reach me through email at ardentlygnarly@users.sourceforge.net or on AIM at ArdentlyGnarly. Also, I can occasionally be found on irc.freenode.net on channel #gaim.

Acknowledgments

The information here as well as the resulting code that is now a part of Gaim would not have been possible without the work of many other people. First, I thank my wonderful mentor Mark Doliner. Both his experience with OSCAR development and his reference material at http://www.kingant.net/oscar/ have been very valuable resources throughout development. Without his friendly advice, this project would have been very difficult. Also, Keith Lea of the Joust project has been big help both through his advice and his Java implementation of OSCAR at http://joust.kano.net.

Thanks to the developers, both present and past whose work I have built on this summer. The Gaim developers have all been very friendly and made this an enjoyable experience. Also, Sean Egan has put in a good deal of his time to make sure the Summer of Code was a success for Gaim. And of course, let's not forget Adam Fritzler who wrote the original libfaim code, on which the Gaim protocol plug-in is built.

My thanks also go out to the people that got me involved in the Summer of Code, too: First, to my uncle Robert Carruthers who found out about the program and gave me a copy of the information. Also, to Dr. Rinewalt of the TCU Computer Science department who helped me in in proofing my application form. And of course to my parents who have been very supportive and put up with my late nights of programming all summer.

Who Should Read This

The intended audience of this document is programmers who would like to implement file transfers via the OSCAR protocol and those who are interested in the very technical aspects of what actually happens during an AIM/ICQ file transfer. Before reading this, you should already have a working knowledge of the OSCAR protocol including TLV's, FLAP, SNAC, and ICBM's. Though some basic concepts are discussed, this document is not intended for absolute beginners.

Background

This documentation is being written for the Gaim project as a part of Google's Summer of Code 2005. In June, Google selected roughly 400 college students out of about 9000 who applied to contribute to one of over 40 open source projects. Once selected, each student was assigned a mentor from the project.

My application to the Gaim project to implement file transfers over the OSCAR protocol in Gaim was one of the accepted applications. In that application, I specified that the protocol messages involed in performing file transfers would be documented; this document is the result.

Also, as an aside, since the Google Summer of Code is all about open source software, this document and all accompanying diagrams were made using OpenOffice.Org 2.0 Beta.

Overview

OSCAR is the instant messaging protocol used in modern AOL, AOL Instant Messenger, and ICQ clients. Though ICQ originally used its own protocol, it now uses OSCAR file transfers just as AIM does. This change to ICQ is effective as of ICQ 5.02, released February 7, 2005; it allows ICQ users to route transfers through AOL proxy servers when a peer to peer connection fails. For direct connections and file transfers, a protocol called ODC/OFT (OSCAR Direct Connect/OSCAR File Transfer) is used. Sometimes this protocol is referred to as an OSCAR "rendezvous," but should not be confused with the Microsoft RVP protocol (which is also known as rendezvous) or the Apple's Bonjour (which was originally called rendezvous). The current version of the protocol is OFT2.

Recommended Tools

The typical method of writing code that implements the OSCAR protocol is to perform packet captures on the official Windows AIM client and then figure out what the data contained in each packet means. This type of reverse engineering has already been done for many types of OSCAR messages, but often a packet capture is the only way to find out what is really happening.

Before starting on any serious work with the OSCAR protocol, you will most likely want some sort of packet capture program so that you can see what is going on behind the scenes. For Linux, Ethereal is the standard; it can be downloaded at http://www.ethereal.com. For Windows, either Ethereal or Packetyzer (http://www.packetyzer.com) are good choices.

Example Implementations

Example implementations are perhaps one of the best ways to learn about the specifics of OSCAR file transfers. Some of the better examples available are Gaim and Joscar. The Gaim source code can be downloaded from Sourceforge at http://gaim.sourceforge.net and the Joscar source code is available at http://joust.kano.net/joscar/. These should be very valuable resources for anyone wishing to create an implementation of the modern OSCAR protocol.

Conventions

Throughout this document, several conventions are often used to improve readability. Shortened names of message components are given in parentheses and match up to a name in a table containing an example message, which should be somewhere above the text where the name was found. The length of message components is given in brackets to make it easier to find. Finally, example messages have been highlighted in color so that it is easier to match the hex components of the message to the shortened name of the component in question.

Data Structures

Overview

This section is dedicated to breaking down the data structures found in the packets exchanged during client-server and client-client file transfer communication. The structures outlined here are compatible with the OFT2 version of the protocol. It should be noted that in a packet capture, there will be much additional information contained in a packet that the instant messaging program does not handle. For example, each packet will likely contain Ethernet, Internet Protocol, and Transmission Control Protocol data; this type of information should be easily parsed out by modern packet capture programs and is not shown nor discussed in the description of these data types.

Client to Messaging Server Communication

Client to messaging server communication during file transfers is accomplished through ICBM's (Inter Client Basic Messages). ICBM's are somewhat complex and contains a considerable amount of information that does not pertain directly to file transfers. A good packet capture program or other documentation will give you insight into what some of the other components of this message mean. Here, ICBM's are only covered briefly and only the parts of the message that directly affect file transfers will be emphasized.

There are several layers that make up an ICBM; starting from the ground up: First, there is the FLAP layer, which begins with a command start of 0x2A, then a 1-byte channel number (file transfers are always on channel 0x02), followed by a 2-byte sequence number, a 2-byte length, 10 bytes of additional information, and, usually, a SNAC command. This additional information is broken down into a 2-byte family, a 2-byte subtype, 2-bytes of flags, and a 4 byte ID.

For a file transfer (as well as many other services, not discussed here), this remaining data is a SNAC command, which labels the data with a family. File transfer messages are classified as family 0x0004 (AIM Messaging).

Following the SNAC is the ICBM. This contains a cookie (generated by one of the clients upon sending the first message in a sequence), the channel (still 0x0002), and ICBM data. This data contains the buddy name of the sender and, usually, a TLV of type 0x0005 (Rendezvous Data). If this Rendezvous TLV block exists, it contains the value section of the rendezvous data is the message type, an ICBM cookie, a 16-byte capability block for a Send File (a constant), and then another TLV chain. Each of these message types has different TLV chains associated with it; the differences in message types are discussed in the subheadings below.

Rendezvous Data TLV Block

During an OSCAR file transfer, ICBM's are used to help in negotiating the transfer. These messages are sent to the AOL messaging server and not directly to the other client involved in the transfer. Thus, these messages are more a part of the OSCAR protocol than the OFT protocol.

First, let us examine the FLAP layer: The messages in which we are interested are of the family 0x0004 (AIM Instant Messaging) and have a subtype of either 0x0006 (Outgoing) or 0x0007 (Incoming). The rest of the FLAP and SNAC layers of this message are already well-documented[1] except for the type 0x0005 (Rendezvous) TLV block. The rest of this section will be dedicated to the information contained in the value portion of the rendezvous TLV block. The components of the value section of this TLV are as follows (in order):

- Message Type - [2 bytes] This describes the purpose of this message. Possible values are:
    - → 0x0000 – Request. This is the bulkiest type. There will be a TLV chain following the ICBM cookie and the send file capability block for this type. (See below)
    - → 0x0001 – Cancel. One of the clients involved in this file transfer is no longer interested in transferring the file.
    - → 0x0002 – Accept. One of the clients is ready to begin the transfer. It is worth noting that another message indicating that the transfer is ready to begin is sent across the OFT connection. That is to say, even though this accept message must be sent to successfully emulate the official Windows AIM client, there are other messages sent during negotiation that are better suited to use as triggers to begin transmitting raw data. (See the section on "OFT Header")
- ICBM Cookie - [8 bytes] This information has already been sent in this message in the SNAC layer. However, we must duplicate it here for unknown reasons. This cookie will be the same for all messages related to this transfer.
- Send File Capability Block – This is a constant value that indicates that that this client is capable of sending files and that this message is in regard to a file transfer. The constant value is (all values in hex, separated by spaces for readability): 0x0946 1343 4C7F 11D1 8222 4445 5354 0000 [16 bytes]
- A TLV chain of variable length follows if this is a rendezvous request (Message Type is 0x0000). The TLV's that have been observed to be a part of this chain are as follows (in the order in which the official Windows client writes them to the chain):
    - → 0x000A – Request Number. This is a 1-based sequential identifier for which request this is. It can take up to 3 requests for a file transfer to be negotiated. This value will start at 0x0001 for the first request sent for a given file transfer; each additional request sent during a file transfer will have a request number incremented by 1. The value portion of this TLV has a length of [2 bytes].
    - → 0x000F – Mystery Block. As of this writing, the significance of this TLV block is unknown. The value portion of this TLV has a length of [0 bytes].
    - → 0x000E – Language. This is likely in ISO639 format and is expressed in lower-case ASCII. For example, "en" is a possible value. For more information on ISO639

1 For more information about the structure of incoming (subtype 0x0007) & outgoing (subtype 0x0006) instant messages (family 0x0004), see http://joust.kano.net/wiki/oscar/moin.cgi/InstantMessages.

codes, see http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt. The value portion of this TLV has a length of [2 bytes].

→ 0x000D – Character Set. This indicates the default encoding for the sender of the file. The value portion of this TLV has been observed to have a length of [8 bytes] when the value is “us-ascii”. This is likely not a constant length. For more example character sets, see http://a4esl.org/c/charset.html.

→ 0x000C – User Message. This is a message to be displayed to the user when prompting the receiver as to whether or not the transfer should be allowed. The official Windows AIM client sends the client's message in HTML format or just “<HTML>” for a blank message and the official Windows ICQ client sends a message surrounded by opening and closing <ICQ_COOL_FT> tags. The message sent by the ICQ client includes various pieces of information about the transfer including the name of the sender and the name of the file to be transfer; this information is formatted by the client and displayed in the conversation window. This TLV has variable length.

→ 0x0002 – Proxy IP. This is the IP address of the proxy server that the receiver should attempt to connect to if a proxy server is to be used. If a proxy is to be used, the 0x0010 TLV must be included later in this chain. Otherwise, the value of this block must match the client IP address (type 0x0003). The value portion of this TLV has a length of [4 bytes].

→ 0x0016 – Proxy IP Check. The value of this block is the bitwise complement of the value of the 0x0002 TLV block. This is most likely either an attempt at a checksum or an attempt to deter outsiders from reverse engineer the protocol. The value portion of this TLV has a length of [4 bytes].

→ 0x0003 – Client IP Address. This is the IP address that the client thinks it has. This might be different than the value the outside world sees if the client is behind a NAT device. The value portion of this TLV has a length of [4 bytes].

→ 0x0005 – External Port. For direct connections, this is the port that the other client should use when attempting to connect. When using a proxy server, this “port” is actually just a check value and the client should actually connect to port 5190 of the proxy server.

→ 0x0017 – Port Check. The value of this block is the bitwise complement of the value of the 0x0005 TLV block. This is most likely either an attempt at a checksum or an attempt to deter outsiders from reverse engineer the protocol. The value portion of this TLV has a length of [2 bytes].

→ 0x0010 – Proxy Flag. If present, this value indicates that the IP contained in the 0x0002 block is to be used as the proxy IP address and that this transfer should be routed through that proxy server. The value portion of this TLV has a length of [0 bytes].

→ 0x02711 – Capability Data. The value portion of this block can be broken up into 4 separate pieces of data:

- The first 2 bytes are the Multiple Files Flag. A value of 0x0001 indicates that only one file is being transferred while a value of 0x0002 indicates that more than one file is being transferred.
- The next 2 bytes is the File Count, the total number of files that will be transmitted during this file transfer.
- The next 4 bytes is the Total Bytes, the sum of the size in bytes of all files to be transferred.
- The remaining bytes is a null-terminated string that is the name of the file.

→ 0x02712 – File Name Encoding. This is a lower-case ASCII string indicating how the file name was encoded. An example value is "us-ascii". The value portion of this TLV block has a length of [8 bytes].

→ 0x0004 – Verified IP Address. This block is the IP address that the messaging server thinks the client has. This might be different than the IP address the client thinks it has if the client is behind a NAT device. This TLV block is added by the messaging server and not the sending client for obvious reasons. The value portion of this TLV block has a length of [4 bytes].

Server-Sent Acknowledge

This message is a bit different from those discussed previously in that it does not have any rendezvous or SNAC data associated with it. The basics of the FLAP are not discussed here are they are already well-documented.[2] The most important parts of this message are the family and subtype within the FLAP layer of the message. The family of 0x0004 is AIM messaging, and the subtype of 0x000C indicates that this is an server-sent acknowledge. It is sent in response to a client adding a zero length TLV block of type 0x0003 to the main TLV chain of an outgoing ICBM.

Client to Client Communication via Proxy

"Client to client" communication when using a rendezvous proxy server is a bit different, especially since the clients actually have no direct communication this way. Before any OFT communication is done, both clients must log into the proxy server. This is done via a protocol that is not quite like any of the protocols discussed thus far. There are 5 different types of commands that can be sent via this protocol, which are outlined in the following subheadings. Upon successful login to a proxy server, the connection then uses the OFT protocol as if the clients were directly connected.

2 For more information about the structure of basic OSCAR instant messages, see http://joust.kano.net/wiki/oscar/moin.cgi/InstantMessages.

Rendezvous Proxy: Header

The rendezvous proxy header consists of the first 12 bytes of a rendezvous proxy packet. A typical rendezvous proxy header might look like this:

0010 044A 0001 0000 0000 0220...	Len PackVer CmdTypeUnknown Flags...

They can be broken down as follows:

- 2 bytes: the length of the data (Len) in bytes that should be expected to follow. The length field itself does not count toward the length of the data. This means that we expect to see the length field having a value 2 less than the overall length of the packet. For instance, there should be 16 bytes following "Len" above.
- 2 bytes: the packet version (PackVer). In modern clients, this value is always 0x044A.
- 2 bytes: are the command type (CmdType). This specifies what type of data follows. Valid values are 0x0001 – 0x0005; these values are listed in parentheses next to each of the rendezvous proxy packets below. It is worth noting that all commands types having an odd value are sent from server to client and those having even value are sent from client to server.
- 4 bytes: Unknown. These could be from a legacy version of the protocol or to allow for future expansion. So far, they have always been observed to be null.
- 2 bytes: Flags. These have been observed to be 0x0220 when the message originates from the server and 0x0000 when the message originates from the client.
- Remaining bytes: command-specific data

Rendezvous Proxy: Error (0x0001)

The rendezvous proxy error command consists of the rendezvous proxy header (see above) plus an additional byte for a total of 14 bytes. A typical rendezvous proxy error command looks like this:

000C 044A 0001 0000 0000 0220 000D	Len CmdType Flags ErrCode

Notice that:

- The Length (Len) will always be 0x000C (Decimal: 12 bytes)
- This message has a command type (CmdType) of 0x0001
- It is sent from the server to the client, so the flags are 0x0220

The following values of the 2 byte error code have been observed:

- 0x000D – Bad Request
- 0x0010 – Initial Request Timed Out. This happens after 10 seconds if the required login information is not sent.
- 0x001A – Accept Period Timed Out. This happens when the other client does not accept the file transfer in a timely manner and the proxy server gets tired of waiting.

Rendezvous Proxy: Initialize Send (0x0002)

The rendezvous proxy initialize send command consists of the rendezvous proxy header (see above) plus additional values, including the variable length screenname data. A typical rendezvous proxy initialize send command looks like this:

0036 044A 0002 0000 0000 0000	Len CmdType Flags
0F46 6965 7263 656C 7947 6E61	SnLen Sn
726C 7979 3A83 1900 7945 0000	Cookie
0001 0010 0946 1343 4C7F 11D1	Capability
8222 4445 5354 0000	

Notice that:

- The Length (Len) is variable, due to the screenname field.
- This message has a command type (CmdType) of 0x0002
- It is sent from the client to the server so the flags are 0x0000

The data after the header (after the flags) is structured as follows:

- 1 byte: length of screen name (SnLen)
- Variable length: the screen name in ASCII (Sn)
- 8 bytes: ICBM Cookie (Cookie). This must match the cookie that is sent with the rendezvous ICBM request.
- 20 Bytes: Send File Capability TLV (Capability). Here, the type is 0x0001 since this is the first and only TLV block in this chain. The length is 0x0010 (Decimal: 16 bytes). The final 16 bytes are the value of the send file capability; this is a constant.

Rendezvous Proxy: Acknowledge (0x0003)

The rendezvous proxy acknowledge command consists of the rendezvous proxy header (see above) plus an additional 6 bytes for a total of 18 bytes. A typical rendezvous proxy acknowledge command looks like this:

0010 044A 0003 0000 0000 0220	Len CmdType Flags
2337 400C C986	Port IP

Notice that:

- The Length (Len) will always be 0x0010 (Decimal: 16 bytes)
- This message has a command type (CmdType) of 0x0003
- It is sent from the server to the client, so the flags are 0x0220

The data after the header (after the flags) is structured as follows:

- 2 bytes: "Port." This is not the port that should be used when connecting (5190 is always used for the actual connection); rather, this "port" is sent as a check value.
- 4 bytes: Proxy IP Address (IP). This is the IP address of the proxy server that the other client should use to connect; it is not the same as the IP address of the proxy from which this proxy acknowledge came.

Rendezvous Proxy: Initialize Receive (0x0004)

The rendezvous proxy initialize receive command consists of the rendezvous proxy header (see above) plus additional values, including the variable length screenname data. This command is nearly identical to a initialize send command except that this contains a "port number." A typical rendezvous proxy initialize receive packet looks like this:

0038 044A 0004 0000 0000 0000	Len CmdType Flags
0F46 6965 7263 656C 7947 6E61	SnLen Sn
726C 2337 7979 3A83 1900 7945	Port Cookie
0000 0001 0010 0946 1343 4C7F	Capability
11D1 8222 4445 5354 0000	

Notice that:

- The Length (Len) is variable, due to the screenname field.
- This message has a command type (CmdType) of 0x0004
- It is sent from the client to the server so the flags are 0x0000

The data after the header (after the flags) is structured as follows:

- 1 byte: length of screen name (SnLen)
- Variable length: the screen name in ASCII (Sn)
- 2 bytes: "Port." This is not the port that we use in connecting with the proxy server, but rather just serves as a check value when logging in. It must match the port number that the other client was given in the rendezvous proxy acknowledge message.
- 8 bytes: ICBM Cookie (Cookie). This must match the cookie that is sent with the rendezvous ICBM request.
- 20 Bytes: Send File Capability TLV (Capability). Here, the type is 0x0001 since this is the first and only TLV block in this chain. The length is 0x0010 (Decimal: 16 bytes). The final 16 bytes are the value of the send file capability; this is a constant.

Rendezvous Proxy: Ready (0x0005)

The rendezvous proxy ready command is nothing more than a rendezvous proxy header; no additional information is included. Thus the total length is only 12 bytes. A typical rendezvous proxy ready command looks like this:

000A 044A 0005 0000 0000 0220	Len CmdType Flags

Notice that:

- The Length (Len) will always be 0x000A (Decimal: 10 bytes)
- This message has a command type (CmdType) of 0x0005
- It is sent from the server to the client, so the flags are 0x0220

Client to Client Direct Communication

Normal client to client communication during a file transfer is done via the OFT protocol. Really, this is fairly simple protocol in that there's only one major message type, plus the raw data that is sent in between these packets.

OFT File Transfer Header[3]

The OFT header contains at least 256 bytes worth of data, though most of it is not used in an OFT2 transfer. A typical header might look something like this:

4F46 5432 0100 0101 F8FA 0200 686B 0000	ProtVer Len Type Cookie
0000 0000 0001 0001 0001 0001 0000 0039	Encrypt Comp TotFil FilLft TotPrts PrtsLft TotSz
0000 0039 42D9 37C4 EEAF 0000 FFFF 0000	Size ModTime Checksum RfrcSum
0000 0000 0000 0000 FFFF 0000 0000 0000	RfSize CreTime RfcSum nRecvd
FFFF 0000 436F 6F6C 2046 696C 6558 6665	RecvCsum IDString
7200 0000 0000 0000 0000 0000 0000 0000	IDString(c'td)
0000 0000 201C 1100 0000 0000 0000 0000	IDString(c'td) Flags NameOff SizeOff Dummy
0000 0000 0000 0000 0000 0000 0000 0000	Dummy(c'td)
0000 0000 0000 0000 0000 0000 0000 0000	Dummy(c'td)
0000 0000 0000 0000 0000 0000 0000 0000	Dummy(c'td)
0000 0000 0000 0000 0000 0000 0000 0000	Dummy(c'td) MacFileInfo
0000 0000 0000 0000 0000 0000 0000 0000	MacFileInfo(c'td) Encoding Subcode
6379 6777 696E 2E62 6174 0000 0000 0000	FileName
0000 0000 0000 0000 0000 0000 0000 0000	FileName(c'td)
0000 0000 0000 0000 0000 0000 0000 0000	FileName(c'td)
0000 0000 0000 0000 0000 0000 0000 0000	FileName(c'td)

3 Much of the information here was taken from net/kano/joscar/rvproto/ft/FileTransferHeader.java by Keith Lea and gaim/src/protocols/oscar/ft.c. See the section on "Example Implementations" to obtain these sources.

In order, the components of the above message can be broken down as follows:

- The Protocol Version (ProtVer) is “OFT2” in ASCII for modern clients. [4 bytes]
- The length defaults to 0x0100 (Decimal: 256 bytes). This length will expand to accommodate additional data; specifically, the length will be greater than 256 bytes if the filename is more than 63 characters. This length counts all of the data in the message including the protocol version and the length field itself. [2 bytes]
- The “Type” value specifies the purpose of this header. [2 bytes] The 0x0100 family comes from the sender and the 0x0200 family comes from the receiver. (Note: If the type is for a resumed transfer (see below), then the checksums and bytes received must all reflect the data that is currently already on the receiver's computer.) The possibilities are:
    - → 0x0101 – Prompt. This is sent by the file sender to indicate that the client is ready to begin sending data.
    - → 0x0202 – Acknowledge. This is sent by the file receiver in response to a prompt to indicate that the client is ready to begin receiving.
    - → 0x0204 – Done. This is sent by the file receiver after all raw file data has been sent to indicate that the client is done receiving.
    - → 0x0205 – Receiver Resume. The receiver wishes to begin the transfer starting at a point up to which the receiver has already received data.
    - → 0x0106 – Sender Resume. The sender has agreed to begin the transfer at the point the receiver specified.
    - → 0x0207 – Resume Acknowledge. The receiver is ready to begin receiving data from the specified point.
- The cookie must match the ICBM cookie sent with the original file request. [8 bytes]
- Encryption (Encrypt) is set to 0x0000 to indicate that encryption will not be used [2 bytes]
- Compression (Comp) is set to 0x0000 to indicate that compression will not be used [2 bytes]
- Total Files (TotFil) is the total number of files being sent during this file transfer [2 bytes]
- Files Left (FilLft) is the number of files still waiting to be sent during this file transfer [2 bytes]
- Total Parts (TotPrts) is the total number of “parts” being send during this file transfer. This is 0x0001 for most files and 0x0002 for files with a Macintosh resource fork. [2 bytes]
- Parts Left (PrtsLeft) is the number of “parts” (see directly above) still remaining to be sent during this file transfer. [2 bytes]
- Total Size (TotSz) is the total size in all bytes of all parts to be sent during this transfer. [4 bytes]

- The size (Size) is the number of bytes that have been sent so far during this file transfer. [4 bytes]
- Modification Time (ModTime) is the last time (if any) that the file was modified in seconds since the UNIX epoch. [4 bytes]
- Checksum (Checksum) is a checksum for the current file transferred. This checksum is not updated during the transfer as the received checksum is (see below). This is not an intuitive checksum; for the details of its implementation, try the source files of net/kano/joscar/rvproto/ft/FileTransferChecksum.java in Joscar or gaim/src/protocols/oscar/ft.c in Gaim. See the section on “Example Implementations” for information on obtaining the source code for both of these. The starting value for this checksum is 0xFFFF0000. [4 bytes]
- The Received Resource Fork Checksum (RfrcvCsum) is a checksum for the resource fork (if any). This checksum changes every time any raw resource fork data is transmitted across the file transfer connection. See above for where to find more information on this checksum. [4 bytes]
- The Resource Fork Size (RfSize) is the size in bytes of the resource fork (if any). [4 bytes]
- The Creation Time (CreTime) is the number of seconds since the UNIX epoch. The official Windows AIM client always sends 0x0000 for this value and the creation date for the file is set to the time the transfer was begun. [4 bytes]
- The Resource Fork Checksum (RfCsum) is a checksum value for the resource Macintosh resource fork (if any). This checksum is not updated during the transfer as the received resource fork checksum is (see above). See above under “Checksum” for where to find more information on this checksum. [4 bytes]
- The Bytes Received (nRecvd) value is the number of bytes transmitted so far during this file transfer. [4 bytes]
- The Received Checksum (RecvCsum) is a checksum for the portion of the file transmitted up to this point. This checksum changes every time any raw resource fork data is transmitted across the file transfer connection. See above under “Checksum” for where to find more information on this checksum. [4 bytes]
- The Identification String (IDString) has unknown significance as of this writing. The values transmitted by the official Windows AIM client prior to version 5.2 was "OFT_Windows ICBMFT V1.1 32" and from version 5.2 onward has been “CoolFileXfer” [32 bytes]
- The Flags (Flags) help specify the header type. [1 byte]
  They have been observed to have the following values:
  - 0x20 – Negotiation. This is sent to indicate that the file transfer is not yet complete.
  - 0x01 – Done. This is sent to indicate that the file transfer is over
- The List Name Offset (NameOff) has unknown significance as of this writing. However, this value defaults to 0x1C (Decimal: 28). [1 byte]

- The List Size Offset (SizeOff) also has unknown significance. Its default value is 0x11 (Decimal: 17). [1 byte]
- The "Dummy" block (Dummy) is a large NULL block sent by the official Windows AIM client. This is likely for supporting legacy transfers or to allow for future expansion of the OFT protocol. [69 bytes]
- The Macintosh File Information (MacFileInfo) is null when coming from a non-Macintosh user. Otherwise it contains the information described at http://developer.apple.com/documentation/Carbon/Reference/Finder_Interface/index.html?http://developer.apple.com/documentation/Carbon/Reference/Finder_Interface/finder_interface/chapter_1.1_section_1.html. [16 bytes]
- The Encoding (Encoding) specifies the character set that was used to encode the file name string. [2 bytes] Observed values include:
  - → 0x0000 for ASCII
  - → 0x0002 for UTF-16BE or UCS-2BE
  - → 0x0003 for ISO-8859-1
- The Encoding Subcode (Subcode) gives additional information about how the string was encoded. So far, this value has been observed to be only 0x0000. [2 bytes]
- The Filename (Filename) is a string indicating the name of the file on the sender's computer. It is always null terminated, though additional null bytes may occur within a Unicode string. The length of this field is equal to the length field (see above) minus 192 bytes (the length of all data up to this point). If a filename is less than 63 characters long, then it is padded with additional null bytes until it reaches a 64 byte minimum. [Variable length, 64 bytes minimum]

Sequence of Events

Overview of Scenarios

Though sending files was a relatively simple operation several years ago, the world of firewalls, routers, and NAT's (Network Address Translation devices) has complicated things. Normally, during a file transfer, the receiver connects to the sender. However, if the sender is behind a NAT, this is not possible. One early solution to this was to "redirect" the connection if the receiver could not connect to the sender; that is, the sender connects to the receiver. That way, if the receiver is not behind a NAT, the file transfer is still successful.

Another solution was to try connecting using different IP addresses. The client may think it has an address of 192.168.1.100 (let's call this the client IP), but the server may think the same client has an IP of 233.1.1.1 (let's call this the verified IP). If two clients are behind the same NAT (as is the case for many corporate and campus networks), then connecting via the client IP will be successful. However, if clients are behind NAT's then we are stuck trying to negotiate the connection using the verified IP.

The next big step in making file transfers successful was the use of proxy servers. This allowed both clients to connect to a remote server that served only to relay data between the two clients. Though slower, this method almost always results in a successful connection. As an added benefit, it is slightly more secure since neither client has to reveal its IP address. This is provided that a valid proxy server IP is supplied to both clients (as might not be the case if one user is sporting a malicious client).

All of these new methods of making a connection have left OSCAR file transfers with a large number of possible connection scenarios. First, a peer to peer connection with the receiver attempting to connect to the sender is usually attempted since it is faster than having to route the transfer through a proxy server. This connection is attempting using both the client IP address and the verified IP address (see above for an explanation of these terms). If this fails, then the connection is "redirected" and the sender then attempts to connect to the receiver. If the connection is still unsuccessful, the receiver will then request that a "stage 3" proxy be used. In a stage 3 proxy connection, 3 rendezvous request messages are sent between the clients before a successful connection is made.

However, if one of the clients has specified that a proxy server will be necessary to make a connection in the client settings, then the use of a proxy server is forced and any attempt at making a peer to peer connection is skipped. If it is the sender of the file that requests that the proxy be used, then this is called a "Stage 1" proxied transfer since it happens at the earliest possible time. Likewise, if the receiver requests that a proxy be used, this it is called a "Stage 2" proxied transfer since it happens at the second possible juncture. Also, note that it takes 1 rendezvous request message to negotiate a stage 1 proxied transfer and 2 rendezvous request messages to negotiate a stage 2 proxied transfer.

The following sections are dedicated to discussing the sequence of messages that are sent during each of these scenarios. The structure of each of these messages is outlined in the "Data Structures" section of this document.

Direct Peer to Peer Connections

The following sequence of events takes place during a direct peer to peer connection:

1. A rendezvous request is sent from the sender to the receiver via the messaging server. A server acknowledgment is requested in this message's main TLV chain. The message contains the following TLV's in its rendezvous data (in order of appearance):
   a) Request Number (0x000A)
   b) Mystery Block (0x000F) (optional)
   c) Locale (0x000E) (optional)
   d) Encoding (0x000D) (optional)
   e) Transfer Message (0x000C) (optional)
   f) Proxy IP Address (0x0002)
   g) Proxy IP Check (0x0016)
   h) Client IP Address (0x0003)
   i) External Port (0x0005)
   j) Port Check (0x0017)
   k) Capability Data (0x2711)
   l) File Name Encoding (0x2712)
   m) Verified IP Address (0x0004) (Added by the server)
2. The messaging server transmits a server acknowledge to the file sender.
3. The receiver establishes a connection to the sender.
4. The receiver transmits a rendezvous accept to the sender via the messaging server.
5. The sender transmits an OFT header with the “Prompt” type via the direct connection.
6. The receiver transmits an OFT header with the “Acknowledge” type via the direct connection.
7. The sender transmits the raw data of the file over the direct connection.
8. The receiver transmits an OFT header with the “Done” type via the direct connection.

Redirected Connection

A redirected connection is just a variation on a direct peer to peer connection. After step 2 above:

1. A reply rendezvous request is sent from the receiver to the sender via the messaging server. The message contains the following TLV's in its rendezvous data (in order of appearance):
    a) Request Number (0x000A) (now has value 0x0002 instead of 0x0001)
    b) Proxy IP Address (0x0002)
    c) Proxy IP Check (0x0016)
    d) Client IP Address (0x0003)
    e) External Port (0x0005)
    f) Port Check (0x0017)
    g) Verified IP Address (0x0004) (Added by server)
2. Now, instead of the receiver establishing a connection to the sender, the sender establishes a connection to the receiver.
3. Things continue as normal on the direct peer to peer connection step #4.

Stage 1 Proxy Server Connection

For a stage 1 proxy server connection, there is a bit more preparation that needs to be done. Variations from a peer to peer connection are italicized. The sequence of events for this scenario is:

1. *The sender establishes a connection with ars.oscar.aol.com*
2. *The sender transmits a rendezvous proxy Initialize Send message to the proxy server.*
3. *The sender receives a rendezvous proxy Acknowledge from the proxy server.*
4. A rendezvous request is sent from the sender to the receiver via the messaging server. The proxy IP, external port, and cookie for this message have already been established via the previous messages exchanged with the proxy server. A server acknowledgment is requested in this message's main TLV chain. The message contains the following TLV's in its rendezvous data (in order of appearance): **Note that these are the same TLV's sent for a direct peer to peer connection with the addition of the proxy flag**
    a) Request Number (0x000A)
    b) Mystery Block (0x000F) (optional)
    c) Locale (0x000E) (optional)
    d) Encoding (0x000D) (optional)
    e) Transfer Message (0x000C) (optional)

f) Proxy IP Address (0x0002)

g) Proxy IP Check (0x0016)

h) Client IP Address (0x0003)

i) External Port (0x0005)

j) Port Check (0x0017)

k) Proxy Flag (0x0010)

l) Capability Data (0x2711)

m) File Name Encoding (0x2712)

n) Verified IP Address (0x0004) (Added by the server)

5. The messaging server transmits a server acknowledge to the file sender.

6. The receiver establishes a connection to the IP of the proxy server specified in the request.

7. The receiver transmits a rendezvous proxy initialize receive message to the proxy server.

8. The proxy server transmits a rendezvous proxy Ready message to both the sender and receiver.

9. The receiver transmits a rendezvous accept to the sender via the messaging server.

10. The sender transmits an OFT header with the “Prompt” type via the direct connection.

11. The receiver transmits an OFT header with the “Acknowledge” type via the direct connection.

12. The sender transmits the raw data of the file over the direct connection.

13. The receiver transmits an OFT header with the “Done” type via the direct connection.

Stage 2 Proxy Server Connection

For a stage 2 proxy server connection, things go just as a peer to peer connection until the receiver intervenes, so the first 2 steps are the same as a direct peer to peer connection. Variations from a peer to peer connection are italicized. The sequence of events AFTER step #2 of what started as a direct peer to peer connection are:

1. The receiver establishes a connection with ars.oscar.aol.com

2. The receiver transmits a rendezvous proxy Initialize Send message to the proxy server.

3. The receiver receives a rendezvous proxy Acknowledge from the proxy server.

4. A reply rendezvous request is sent from the receiver to the sender via the messaging server. The proxy IP, external port, and cookie for this message have already been established via the previous messages exchanged with the proxy server. The message contains the following TLV's in its rendezvous data (in order of appearance):

a) Request Number (0x000A) (now has value 0x0002 instead of 0x0001)

b) Proxy IP Address (0x0002)

c) Proxy IP Check (0x0016)

d) External Port (0x0005)

e) Port Check (0x0017)

f) Proxy Flag (0x0010)

5. *The sender transmits a rendezvous accept to the receiver via the messaging server.*

6. The sender transmits an OFT header with the “Prompt” type via the direct connection.

7. The receiver transmits an OFT header with the “Acknowledge” type via the direct connection.

8. The sender transmits the raw data of the file over the direct connection.

9. The receiver transmits an OFT header with the “Done” type via the direct connection.

Stage 3 Proxy Server Connection

A stage 3 proxy server connection is the result of all other forms of connection failing. When that happens, the receiver transmits a message to the sender asking for a proxy server to be used. The sequence of events AFTER step #2 of what started as a direct peer to peer connection are:

1. A reply rendezvous request is sent from the receiver to the sender via the messaging server. A null client IP address is sent with this message. The message contains the following TLV's in its rendezvous data (in order of appearance):

a) Request Number (0x000A) (now has value 0x0002 instead of 0x0001)

b) Proxy IP Address (0x0002) (has value 0x0000 0000)

c) Proxy IP Check (0x0016) (has value (0xFFFF FFFF)

d) Client IP Address (0x0003) (has value 0x0000 0000)

2. The sender establishes a connection with ars.oscar.aol.com

3. The sender transmits a rendezvous proxy Initialize Send message to the proxy server.

4. The sender receives a rendezvous proxy Acknowledge from the proxy server.

5. A 3rd rendezvous request is sent from the sender to the receiver via the messaging server. The proxy IP, external port, and cookie for this message have already been established via the previous messages exchanged with the proxy server. The message contains the following TLV's in its rendezvous data (in order of appearance):

a) Request Number (0x000A) (now has value 0x0003)

b) Proxy IP Address (0x0002)

c) Proxy IP Check (0x0016)

d) External Port (0x0005)

e) Port Check (0x0017)

f) Proxy Flag (0x0010)

6. From this point forward, the transfer behaves just as a stage 1 proxy connection transfer beginning at that transfer's step #6.

Exceptions to the Rules

If a file already exists in its entirety on the destination computer, then the client will reply to the OFT header of type “Prompt” with a header of type “Done.”

Gaim's OSCAR Protocol Plug-In

Overview

Gaim is a very layered application written in C. Within the main program, the core and the User Interface (UI) are kept separate from each other in what is known as the core-UI split. This leaves the possibility of writing a new UI on top of the existing Gaim code.

Also, Gaim is based on a plug-in design. In its most basic form, Gaim does not have any knowledge of how to interact with any protocols. The ability to communicate with each protocol is added at runtime by loading protocol plug-ins (aka PRPL's). This allows for new protocols to be easily added to the program. For the programmer, it also means there are many function pointers and callbacks floating around.

Within the OSCAR PRPL itself, there are still more layers. Gaim's OSCAR code is based on libfaim, a library started by Adam Fritzler in 1998 for connecting to the AIM network. On top of the libfaim code are handlers for the protocol messages and an the interface between the PRPL and Gaim.

Changes Made for Summer of Code 2005

The first change made to Gaim's OSCAR Protocol Plug-in for the Summer of Code was to enable sending files to ICQ users. As with all changes for this project, this will affect modern ICQ clients (version 5.02 or better). This change was released in Gaim 1.4.0.

Next, support was added for attempting a peer to peer connection via both the client IP and verified IP addresses (see “Overview of Scenarios” for an explanation of this terminology). This allowed users behind the same NAT to a make a successful connection. Also, a mechanism for timing out each connection attempt (and eventually the transfer as a whole) was added. Gaim 1.5.0 included these new features.

Finally, the bulk of the work that was done for this project was adding support for using proxy servers for AIM and ICQ file transfers. During this period of development, more thorough in-code documentation was added. All of the many scenarios encompassed by proxied file transfers are outlined in the “Overview of Scenarios” section. These changes will be released in Gaim 2.0.0.

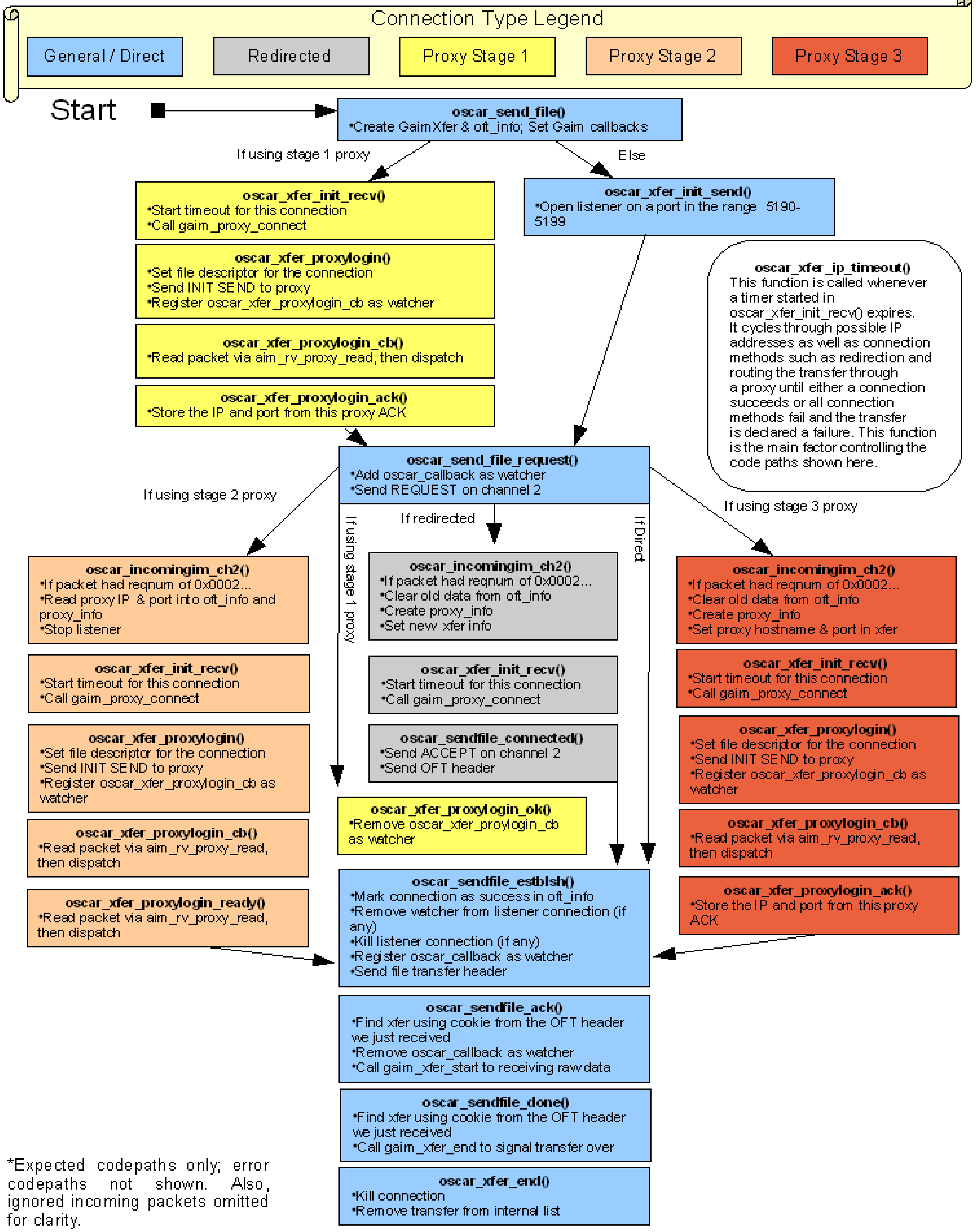
Code Paths

The following diagrams are intended to make further development on the file transfer code easier. However, the diagrams follow only the “meat” of the code paths found in oscar.c and show the expected code path for the most typical file transfers; code paths for special cases may be found easily by reading the section on “Sequence of Events” and then applying that knowledge to these diagrams. Also, each of the functions listed in these diagrams may make calls to the Gaim core or libfaim to accomplish its task. As a guide for finding these functions here is a brief list of the files involved in Gaim's OSCAR file transfer code (paths are listed with respect to the base Gaim directory):

- src/ft.c – File transfer functions from the Gaim core.

- src/proxy.c – Functions for making a connection with or without a proxy from the Gaim core.
- src/protocols/oscar/oscar.c – Main interface between Gaim and the OSCAR PRPL.
- src/protocols/oscar/aim.h – Structures and #defines for libfaim.
- src/protocols/oscar/bstream.c – libfaim functions for reading and writing bitstreams, the format used for handling incoming and outgoing messages.
- src/protocols/oscar/ft.c – libfaim file transfer functions.
- src/protocols/oscar/im.c – libfaim IM sending & receiving functions; this includes the formation and parsing of such messages as rendezvous request, accept, and cancel messages.
- src/protocols/oscar/rxqueue.c – libfaim receiving queue functions; these are the first place that incoming messages appear.
- src/protocols/oscar/rxhandlers.c – libfaim received message handlers; these functions are used for routing incoming messages to functions that deal with them.
- src/protocols/oscar/txqueue.c – libfaim transmit queue functions; these are where most outgoing messages end up ultimately.

Gaim OSCAR PRPL File Send Code Paths*



oscar_send_file()
•Create GaimXfer & oft_info; Set Gaim callbacks

oscar_xfer_init_recv()
•Start timeout for this connection
•Call gaim_proxy_connect

oscar_xfer_proxylogin()
•Set file descriptor for the connection
•Send INIT SEND to proxy
•Register oscar_xfer_proxylogin_cb as watcher

oscar_xfer_proxylogin_cb()
•Read packet via aim_rv_proxy_read, then dispatch

oscar_xfer_proxylogin_ack()
•Store the IP and port from this proxy ACK

oscar_xfer_init_send()
•Open listener on a port in the range 5190-5199

oscar_xfer_ip_timeout()
This function is called whenever a timer started in oscar_xfer_init_recv() expires. It cycles through possible IP addresses as well as connection methods such as redirection and routing the transfer through a proxy until either a connection succeeds or all connection methods fail and the transfer is declared a failure. This function is the main factor controlling the code paths shown here.

oscar_send_file_request()
•Add oscar_callback as watcher
•Send REQUEST on channel 2

oscar_incomingim_ch2()
•If packet had reqnum of 0x0002...
•Read proxy IP & port into oft_info and proxy_info
•Stop listener

oscar_xfer_init_recv()
•Start timeout for this connection
•Call gaim_proxy_connect

oscar_xfer_proxylogin()
•Set file descriptor for the connection
•Send INIT SEND to proxy
•Register oscar_xfer_proxylogin_cb as watcher

oscar_xfer_proxylogin_cb()
•Read packet via aim_rv_proxy_read, then dispatch

oscar_xfer_proxylogin_ready()
•Read packet via aim_rv_proxy_read, then dispatch

oscar_incomingim_ch2()
•If packet had reqnum of 0x0002...
•Clear old data from oft_info
•Create proxy_info
•Set new xfer info

oscar_xfer_init_recv()
•Start timeout for this connection
•Call gaim_proxy_connect

oscar_sendfile_connected()
•Send ACCEPT on channel 2
•Send OFT header

oscar_xfer_proxylogin_ok()
•Remove oscar_xfer_proylogin_cb as watcher

oscar_incomingim_ch2()
•If packet had reqnum of 0x0002...
•Clear old data from oft_info
•Create proxy_info
•Set proxy hostname & port in xfer

oscar_xfer_init_recv()
•Start timeout for this connection
•Call gaim_proxy_connect

oscar_xfer_proxylogin()
•Set file descriptor for the connection
•Send INIT SEND to proxy
•Register oscar_xfer_proxylogin_cb as watcher

oscar_xfer_proxylogin_cb()
•Read packet via aim_rv_proxy_read, then dispatch

oscar_xfer_proxylogin_ack()
•Store the IP and port from this proxy ACK

oscar_sendfile_estblsh()
•Mark connection as success in oft_info
•Remove watcher from listener connection (if any)
•Kill listener connection (if any)
•Register oscar_callback as watcher
•Send file transfer header

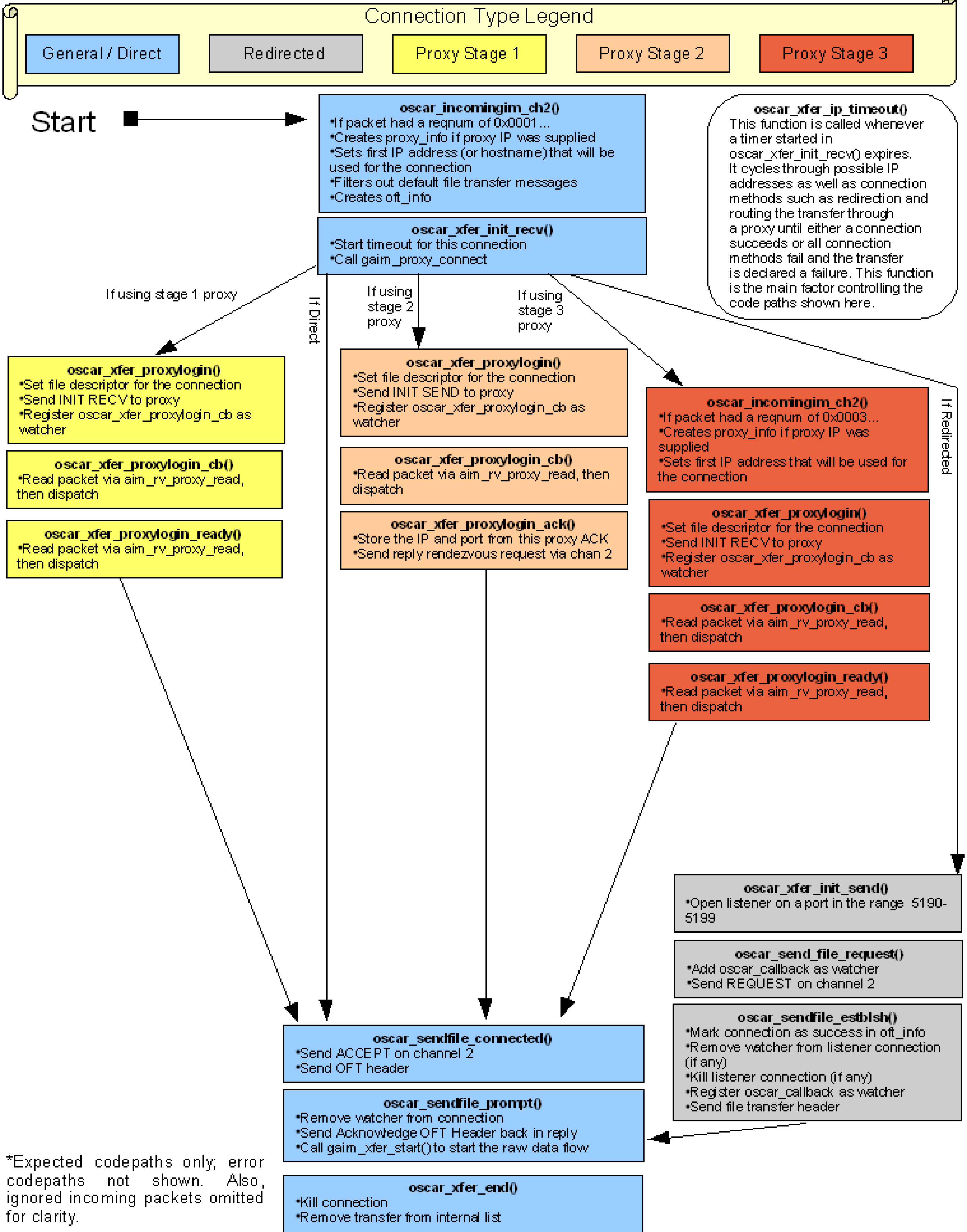
oscar_sendfile_ack()
•Find xfer using cookie from the OFT header we just received
•Remove oscar_callback as watcher
•Call gaim_xfer_start to receiving raw data

oscar_sendfile_done()
•Find xfer using cookie from the OFT header we just received
•Call gaim_xfer_end to signal transfer over

oscar_xfer_end()
•Kill connection
•Remove transfer from internal list

*Expected codepaths only; error codepaths not shown. Also, ignored incoming packets omitted for clarity.

Gaim OSCAR PRPL File Receive Code Paths*



oscar_incomingim_ch2()
•If packet had a reqnum of 0x0001...
•Creates proxy_info if proxy IP was supplied
•Sets first IP address (or hostname) that will be used for the connection
•Filters out default file transfer messages
•Creates oft_info

oscar_xfer_init_recv()
•Start timeout for this connection
•Call gaim_proxy_connect

oscar_xfer_ip_timeout()
This function is called whenever a timer started in oscar_xfer_init_recv() expires. It cycles through possible IP addresses as well as connection methods such as redirection and routing the transfer through a proxy until either a connection succeeds or all connection methods fail and the transfer is declared a failure. This function is the main factor controlling the code paths shown here.

oscar_xfer_proxylogin()
•Set file descriptor for the connection
•Send INIT RECV to proxy
•Register oscar_xfer_proxylogin_cb as watcher

oscar_xfer_proxylogin_cb()
•Read packet via aim_rv_proxy_read, then dispatch

oscar_xfer_proxylogin_ready()
•Read packet via aim_rv_proxy_read, then dispatch

oscar_xfer_proxylogin()
•Set file descriptor for the connection
•Send INIT SEND to proxy
•Register oscar_xfer_proxylogin_cb as watcher

oscar_xfer_proxylogin_cb()
•Read packet via aim_rv_proxy_read, then dispatch

oscar_xfer_proxylogin_ack()
•Store the IP and port from this proxy ACK
•Send reply rendezvous request via chan 2

oscar_incomingim_ch2()
•If packet had a reqnum of 0x0003...
•Creates proxy_info if proxy IP was supplied
•Sets first IP address that will be used for the connection

oscar_xfer_proxylogin()
•Set file descriptor for the connection
•Send INIT RECV to proxy
•Register oscar_xfer_proxylogin_cb as watcher

oscar_xfer_proxylogin_cb()
•Read packet via aim_rv_proxy_read, then dispatch

oscar_xfer_proxylogin_ready()
•Read packet via aim_rv_proxy_read, then dispatch

oscar_xfer_init_send()
•Open listener on a port in the range 5190-5199

oscar_send_file_request()
•Add oscar_callback as watcher
•Send REQUEST on channel 2

oscar_sendfile_estblsh()
•Mark connection as success in oft_info
•Remove watcher from listener connection (if any)
•Kill listener connection (if any)
•Register oscar_callback as watcher
•Send file transfer header

oscar_sendfile_connected()
•Send ACCEPT on channel 2
•Send OFT header

oscar_sendfile_prompt()
•Remove watcher from connection
•Send Acknowledge OFT Header back in reply
•Call gaim_xfer_start() to start the raw data flow

oscar_xfer_end()
•Kill connection
•Remove transfer from internal list

*Expected codepaths only; error codepaths not shown. Also, ignored incoming packets omitted for clarity.

Conclusion and Personal Notes

Google's Summer of Code has been a wonderful thing for both the open source community as a whole and for me personally. Hundreds of projects just made large amounts of progress in just two months and just as many students gained valuable experience. For the Gaim project, users of AOL and ICQ have file transfer capability comparable to the modern official clients.

No longer, while using Gaim, when one of my friends attempts to transfer a file to me and must I explain to them that I must switch to another AIM client before the file transfer will work. Previously, my friends would focus on this one deficiency rather than seeing the other great features of the program. Not any more. Now I can say “Yeah, Gaim can do that.” Adding this functionality to Gaim has strengthened its position as a first-class instant messaging program and will hopefully attract new users.

I have learned much from coding on this project. This has been my first time to work on a program with thousands upon thousands of lines of code. Luckily, top developers there to share their experience and help me along every step of the way. Now, I feel comfortable building on top of existing code and would like to continue this type of work in my free time. As I have learned: **Hacking open source is fun.** Thanks again to the friendly developers of Gaim and the very generous people at Google for making this experience possible.

Errata

Just like code, bugs pop up in documentation, too. The following corrections have been made to this document since its original release:

January 4, 2006

- Incorrect hex length values on example packets
- Initialize send and initialize receive are variable length packets
- Part of screename was incorrectly highlighted on initialize send and initialize receive packets.
- The length of the error code in the rendezvous proxy error packet is 2 bytes.
- Gaim version 1.5.1 was never to be; Documentation now references Gaim 2.0.0