UDP File Transporter

Trevor Spiteri

12 January, 2006

Contents

1 Overview

A client (*uft*) and a server (*uftd*) application were designed that can transfer files reliably in both directions using the UDP transport mechanism.

The project was implemented using the C++ programming language.

2 Invoking

2.1 The server application

The server application accepts the following command line options:

--port PORT — sets the server port number (default 2986).

--verbose VERBOSITY — sets the verbosity level (default 1).

--help — prints help on the command line options.

--version — prints the version of the file transporter.

2.2 The client application

The client application accepts the following command line options:

--server HOST — sets the server host (default 127.0.0.1). The host can be either in dot notation or a name.

--port PORT — sets the server port number (default 2986).

--clientport PORT — sets the client port number.

--verbose VERBOSITY — sets the verbosity level (default 1).

--help — prints help on the command line options.

--version — prints the version of the file transporter.

3 Command line interface

The server application does not read standard input and outputs messages according to the preset verbosity level.

The client application reads commands from standard input and outputs results and messages according to the preset verbosity level. It accepts the following commands:

dir — lists the directory contents of the server.

ldir — lists the local directory contents.

del *filename* — deletes a file found on the server.

ldel *filename* — deletes a local file.

ren *filename filename2* — renames a file on the server.

lren *filename filename2* — renames a local file.

get *filename* — downloads a file from the server. This overwrites any local file with the same name.

reget *filename* — resumes downloading a file from the server.

put *filename* — uploads a file onto the server. This overwrites any file with the same name.

reput *filename* — resumes uploading of a file onto the server.

exit — exits the client application.

4 Protocol

Communication between the client and the server is performed using UDP datagrams. The datagram size was selected to be efficient over Ethernet, while allowing for frame formats which are slightly smaller than Ethernet. For example, PPPoE has a frame size of $1500 - 8 = 1492$. So the maximum datagram size, including IP and UDP headers, was selected to be 1484.

The datagram format is shown in Table 1. The numbers in parenthesis are the number of octets of each field.

Table 1: The datagram format.

0	1	2	3
CRC (4)			
Type (1)	Sequence ID (3)		
Payload (0–1448)			

The CRC covers the frame including the header (type and sequence id) and the payload. It does not cover the IP and UDP headers.

The type has 4 bits for the operation and 4 bits for flags. The operation code (higher 4 bits) can be any of the operations listed in Table 2. The flags which can be set are shown in Table 3.

Table 2: The operations permitted.

Code	Operation
1	List directory
2	Delete file
3	Rename file
4	Download file
5	Upload file

Table 3: The available flags.

Code	Flag
0001	Error (sent from server)
0010	End-Of-File
0100	Resume (sent from client)

The sequence id is a unique number for each datagram from a particular address (host and port number).

Each time the server replies a request, the reply datagram has the same operation id (the higher 4 bits of the type field) and the same sequence id. When an error has occurred, the server sets the Error flag.

The only operations that require a connection are directory listing (§ 4.1) and uploading of files (§ 4.5). For these operations, the server provides a 32-bit id (list id and upload id). These have to be included in the client requests after the connection is established.

Offsets in the protocol are always 64-bit.

In datagrams, filenames are sent as a single octet containing the filename length followed by the filename itself. The filename length can be in the range 1–255 inclusive. Filenames can not contain slashes or backslashes.

All numbers are sent in network byte order, that is, most significant byte first.

4.1 List directory

To list the directory contents of the server, the first datagram sent by the client has form (a) in Table 4. The server creates a 32-bit list id and sends a response of form (a) in Table 5. If all the listing is contained in the first response, the End-Of-File flag is set.

After the connection is established, the client sends datagrams of form (b) in Table 4. The server continues to respond using form (a) in Table 5, setting the End-Of-File flag when the whole listing is transferred.

Table 4: Client request for directory listing

	Type	Payload	Flags
(a)	1	empty	none
(b)	1	list id, offset	none

Table 5: Server reply for directory listing

	Type	Payload	Flags
(a)	1	list id, listing	End-Of-File (optional)
(b)	1	empty	Error

4.2 Delete file

To delete a file found on the server, the client sends a datagram of form (a) in Table 6, containing the filename of the file to delete. On success, the server sends a response of form (a) in Table 7, otherwise, the server sends a response of form (b).

Table 6: Client request for deleting a file

	Type	Payload	Flags
(a)	2	filename	none

Table 7: Server reply for deleting a file

	Type	Payload	Flags
(a)	2	empty	none
(b)	2	empty	Error

4.3 Rename file

To rename a file on the server, the client sends a datagram of form (a) in Table 8, containing the filename of the file to rename and the new filename. On success, the server sends a response of form (a) in Table 9, otherwise, the server sends a response of form (b).

Table 8: Client request for renaming a file

	Type	Payload	Flags
(a)	3	old filename, new filename	none

Table 9: Server reply for renaming a file

	Type	Payload	Flags
(a)	3	empty	none
(b)	3	empty	Error

4.4 Download file

To download a file segment, the client sends a datagram of form (a) in Table 10 containing the offset from which to start and the filename. The server responds with a datagram of form (a) in Table 11. If the end of the file is reached, the server sets the End-Of-File flag. If an error occurs, the server responds with a datagram of form (b) in Table 11.

Using this mechanism, a whole file can be downloaded without the need of a connection. Since the client always sends the offset and the filename, the server need not record which clients are downloading which files, and the transfer is completely controlled by the client. It also means that if the server crashes and is restarted before the client times out the operation, the file transfer will resume automatically.

The server must always send equally-sized datagrams to make the client implementation easier in the case where the client sends requests for several parts of a file.

If a client requests an offset past the end of the file, the server responds with a datagram containing no contents and with the End-Of-File flag set.

Table 10: Client request for downloading a file

	Type	Payload	Flags
(a)	4	offset, filename	none

Table 11: Server reply for downloading a file

	Type	Payload	Flags
(a)	4	contents	End-Of-File (optional)
(b)	4	empty	Error

4.5 Upload file

To upload a file, a connection is required. The first datagram sent by the client is of form (a) in Table 12. This contains the filename of the file to upload and the first part of the contents. If the whole file is contained in the

first datagram, the End-Of-File flag is set. This deletes any file on the server having the same name. To resume uploading of a file, the Resume flag is set. When the Resume flag is set in the first datagram, the server does not delete the existing file, but appends new data accordingly. The server notifies the client where to continue using the complete offset described below.

The server creates a 32-bit upload id and sends a response of form (a) in Table 13. The complete offset is the number of contiguous bytes written in the file. That means that if, for example, the first 8000 bytes have been received and stored, the complete offset is 8000. Note that the server keeps a copy of bytes received from the client even if they can not be stored immediately because of a missing preceding datagram. The client can know which datagrams are stored in such a manner by examining the sequence id of the response in addition to the complete offset.

The capacity offset is a sort of advertisement to notify the client how much bytes it can send. No bytes with offset higher than the capacity offset can be sent by the client. As the transfer progresses, the capacity offset returned will be progressively larger.

After the connection is established, the client sends datagrams of form (b) in Table 12, containing the offset and contents. When the end of the file is reached, the End-Of-File flag is set. The server continues to respond using form (a) in Table 13.

If the server crashes and is restarted before the client times out, the server receives a datagram of form (b) in Table 12. But the server has no upload ids stored, so it sends back an error response of form (b) in Table 13. When the client receives such a response, it attempts to resume the upload. This means that if the server is restarted before the client times out, the file upload will resume automatically without the need of an explicit command.

Table 12: Client request for uploading a file

	Type	**Payload**	**Flags**
(a)	5	0 (32-bit), filename, contents	End-Of-File (optional), Resume (optional)
(b)	5	upload id, offset, contents	End-Of-File (optional)

5 Timeout calculation

The datagram timeout is calculated using the estimates for the round trip time (RTT) and the RTT deviation. When a packet is transmitted only

Table 13: Server reply for uploading a file

	Type	Payload	Flags
(a)	5	upload id, complete offset, capacity offset	none
(b)	5	empty	Error

once and an acknowledgment is received, the estimates for RTT and RTT deviation are updated as follows:

$$\begin{aligned} \textit{RTT difference} &= \textit{actual RTT} - \textit{RTT estimate} \\ \textit{new RTT estimate} &= \textit{RTT estimate} + \tfrac{1}{8}\textit{RTT difference} \\ \textit{new RTT deviation} &= \tfrac{3}{4}\textit{RTT deviation} + \tfrac{1}{4}|\textit{RTT difference}| \end{aligned}$$

The timeout is then calculated to be:

$$\textit{Timeout} = \textit{RTT estimate} + 3\textit{RTT deviation}$$

The datagram timeout has a minimum value of 2 ms and a maximum value of 15 s.

6 Congestion avoidance

When sending multiple datagrams for the same operation, a window size is used to decide how much datagrams to send before acknowledgments arrive. The window size is calculated in datagrams.

When starting transmission, the window size is set to one datagram. Initially, each acknowledgement received for a datagram which was transmitted only once will increase the window size by one. Once a timeout occurs, it is assumed that the datagram was lost because of a congested network, so the window size is halved. This is done for every retransmission required. When the congestion clears and acknowledgement are received for datagrams that were sent only once, the window starts to grow again. The window size is increased by one for every acknowledgement received until a certain threshold is reached, after which it is increased by one for every whole window transmitted and acknowledged without retransmissions. The threshold is half the window size when the congestion first occurred.

7 File locking

Since multiple clients can connect to the server simultaneously, a file locking mechanism was required to avoid situations where multiple clients are up-

loading the same file. A file can be read by multiple clients, but if one client is uploading a file, no other client can read from, write to, delete or rename the file.

8 Comments

The application was not optimized for efficient use of the CPU. The objective was to have a reliable file transport mechanism which works well on an unreliable connection.

Since the server has to handle multiple clients concurrently, it was designed to have less work to do than the client. The server implementation was kept as simple as possible. The client is solely responsible for flow control and congestion avoidance. The server always processes and answers a datagram it receives immediately.

The server keeps some state information to avoid performing the same operation twice. As an example, consider the case when the client asks for a file to be deleted. The server keeps a list of recently-performed deletions together with the corresponding address and sequence id. If the server's acknowledgment is lost, the client resends the delete request. The server matches the retransmitted request and resends its acknowledgment without performing any operation.

The server also keeps some state information for listing directories (§ 4.1) and for uploading files (§ 4.5).

9 Code organization

The program source code consists of four files.

transporter.h — the header file containing common declarations, constants and inline functions.

utils.cpp — functions and class definitions used by both the client and the server applications.

uft.cpp — the client application.

uftd.cpp — the server application.