CTEC3604 – Multi-service Networks
Network File Transfer Application

Christian Manning

July 8, 2014

Contents

1 Introduction **2**

2 Similar Applications **2**

2.1 File Transfer Protocol (FTP) 2

2.2 Secure Copy (SCP) 2

2.3 rsync 3

3 Functional Requirements **3**

4 Testing **3**

5 Evaluation **4**

6 Conclusion **6**

1 Introduction

The aim of this project is to create an application which utilises TCP to transfer files over a network. The application is required to use Berkeley sockets and a client-server architecture allowing multiple client connections.

2 Similar Applications

This section contains the evaluations of similar applications or protocols in an attempt to gather the needed requirements for this project.

2.1 File Transfer Protocol (FTP)

FTP [2] is a protocol specifically designed for the transfer of files using TCP. It uses a client-server architecture, with many clients being allowed to connect to a server. For each client connected to the server, there is a control connection which is initiated by connecting to a pre-specified port (default 21). This control connection is the means by which commands are send from the client to the server, and replies are sent from the server to a client. This is achieved using the *Telnet* protocol [3].

When a command sent by the client requires data in reply, another connection is established specifically for data transfers, from the data port (default 20). Because of it using multiple ports, *FTP* is known as an out-of-band protocol. This data can be text or binary data and can be sent in blocks, as a stream or compressed.

FTP requires clients to login with a username and password, though the username may be 'anonymous' if the server is configured to allow anonymous access. The client then has the same access and permissions as that of the user they are logged in. The transferred data, commands and replies themselves are not encrypted in any way.

The client can provide a shell-like command line interface with which the user enters commands and text replies are shown, or a graphical user interface (GUI). Some commands are as follows: LIST (list directory entries), CWD (change working directory), DELE (delete file), RETR (transfer file), et al. These commands allow the user to browse the remote file system so that the available files can become known.

2.2 Secure Copy (SCP)

SCP is a secure network equivalent of the common *cp* (file copy) shell command. It uses the *SSH* [4] protocol, though provides no shell access, only the transferral of files. Because of it being based on *SSH* it is very secure as all network communications undertaken by *SCP* are encrypted. Its simple interface limits its usefulness to situations where only a simple copy is needed from one host to another. The remote file systems cannot be browsed as with *FTP*, requiring knowledge

from the user of the file system layout if the file is destined for anywhere other than the users remote home directory.

The security features of this application are one of its primary benefits, but they are beyond the scope of this project.

2.3 rsync

rsync is an application intended to replace *SCP* with a focus on synchronising files and directories across computers, which makes it a popular backup tool. It does this by comparing the checksums (or sometimes the modification date) of files on both hosts and if they differ, the transfer is initiated. Its interface is similar to that of *SCP* and is highly script-able. *rsync* can operate in daemon mode by listening on its default TCP port of 873, or it can operate using *SSH* requiring the *rsync* client to be on both hosts. Daemon mode is generally used for the purpose of mirroring servers, minimising data transfers for the server.

While *rsync* is more featureful than *SCP*, it still doesn't have the ability to browse the remote file system like *FTP*, requiring an alternate application (eg. *SSH*) for this functionality.

3 Functional Requirements

The following requirements were realised from determining which of the above existing applications' features were suitable for this project and combining some.

- Client-server architecture.
- Allow multiple clients to connect to a single server simultaneously.
- Shell-like user interface.
- Ability to browse the file system client and server-side, i.e. change working directory, list directory contents, get file size, etc.
- Transfer files from the server to the client, and vice versa.
- Separate control connection and data connection. Use the data connection for binary data only; the control connection can accept textual replies.

4 Testing

This section will show tests of one of the major features listed above: the file transfers themselves. The tests have been carried out over a Gigabit Ethernet LAN with both the client and server hosts running GNU/Linux. These tests will demonstrate the applications reliability in this situation and will also show its error handling capabilities. All files used are randomly generated.

C denotes the client, S denotes the server, $\rightarrow$ or $\leftarrow$ denotes direction of transfer. The files used in these tests are identified by their size.

#	Test	Expected Result	Actual Result
1	100MB file $S \rightarrow C$	MD5 checksums match	original: 8ddeb39c79c6429d684bd69a3c18a692 got: 8ddeb39c79c6429d684bd69a3c18a692
2	1000MB file $S \rightarrow C$	MD5 checksums match	original: 845087542715041a9241c07764c5bfb0 got: 845087542715041a9241c07764c5bfb0
3	4.5GB file $S \rightarrow C$	MD5 checksums match	original: 6faa2c90b333573cc040cede7203592e got: 6faa2c90b333573cc040cede7203592e
4	200MB file $C \rightarrow S$	MD5 checksums match	original: 08c2a37ef2bfa7c529a8d0071e64f9c0 got: 08c2a37ef2bfa7c529a8d0071e64f9c0
5	2000MB file $C \rightarrow S$	MD5 checksums match	original: 2963e7928261f9c7dd3386c86f73a330 got: 2963e7928261f9c7dd3386c86f73a330
6	3.5GB file $C \rightarrow S$	MD5 checksums match	original: d1acc5f957330dfdfedac7ab152d3937 got: d1acc5f957330dfdfedac7ab152d3937
7	User interrupt on client during transfer (ctrl-c)	Server safely stops operations on client	Server safely stops operations on client. Server messages detailed below.
8	Remove the Ethernet cable during transfer	TCP connection will eventually time out	TCP connection times out after 10+ minutes. If the cable is reconnected in this time, the transfer resumes successfully.

The following messages are produced by the server instance during test number 7, accompanied by some short explanations.

A network error occurred *sendFile* throws a NetworkErrorException with this message. This exception is caught in the *clientHandler* function which prints its message, then ends this specific server instance.

Client thread ending Client thread safely ending.

Removing client. No. clients: 0 Client thread removed and the client count is updated. The server is still operating as normal and can accept new client connections.

5 Evaluation

Both the server and client were implemented using the D programming language [1] with the standard library and no external libraries. Version 2.058 of the compiler, runtime and standard library was used throughout the implementation of this project.

The code for this project has been split into three parts, or modules: *client*, *server* and *util*. The *util* module contains all functionality common to both the client and the server, and also

some usage specific code. This was done to reduce code duplication and also so that the main logic of *client* and *server* was kept to a minimum in an attempt to increase their readability and ease understanding.

The server has successfully enabled the simultaneous connection of multiple clients by utilising a multi-threaded architecture thanks to the technique of message passing employed by the *std.concurrency* module (*std.* prefix refers to the D standard library). This is made very easy and safe by D as it uses thread-local storage by default, limiting data sharing to explicit and controlled operations. To take advantage of this, *server* was created in a modular fashion: there is the main thread, the listener thread, and a thread for each connected client. These threads communicate using the following functions: *send*, *receive*, *receiveOnly* and *receiveTimeout*. These are not to be confused with socket operations, which are methods of the *Socket* class.

Socket operations are, by default on POSIX compliant systems, blocking, meaning the operation stalls its thread until it is completed or it failed. This is problematic, as if the sockets block then there is no way to receive messages from other threads, meaning all network operations will need to finish before all threads can end. A solution to this problem is to use I/O multiplexing with the *select* system function with a short ($10ms$) time out value, in combination with a call to *receiveTimeout* with a $1\mu s$ time out. The time out value for *select* was chosen due to being an insignificant period of time, though long enough that not too many CPU cycles are used. *receiveTimeout*'s time out value was chosen to be so small so that it wouldn't block. This is a better solution than just adding a time out value to a blocking socket as *select* polls multiple sockets until one or more is ready. This has allowed for receiving errors during a file transfer, for example, an operation that couldn't be accomplished previously without blocking one or the other operation.

FTP's method of using separate sockets for control and data has worked well for this project, though it is perhaps an unnecessary overhead. Protocols such as *HTTP* do not use this method and cope well with many situations. A server without this structure would mean that any transfers, whether file data, text, errors, etc. would all use the same connection. For implementing this, all data would have to be contained in a common structure for transportation so that the message type could be determined and processed appropriately. This would be a great improvement for a future revision of this project.

The shell-like interface provides a highly usable user experience, though there are some limitations. The implementation simply consists of a loop, with a call to *readln* at its beginning. While this method has merit in its simplicity, it does not include any key handling functionality. This means that the user cannot use the arrow keys for doing things such as moving the cursor to edit a command, or selecting a previously entered command. It should be noted however, that Windows integrates these features in its *cmd* and *PowerShell* applications. On a POSIX compliant system, the interface is able to produce colours using ANSI colour codes. This is used for differentiating between files and directories when listing the contents of a remote or local directory. It is also possible for this to be implemented on Windows based systems using

the *SetConsoleTextAttribute* function; a potential future improvement. Another feature of the interface is the presence of progress bars, indicating progress of a file transfer to the client user. This is implemented using ANSI escape codes on POSIX systems and carriage return on Windows. While this is a nice feature, it currently doesn't show the current speed of transfer, or an estimated finishing time, only the overall average speed and time taken values are printed on completion. This is a desirable feature and could improve usability in a future version.

The commands implemented in this project work correctly for both remote and local usage, where appropriate. The local commands are the same as the remote, but prefixed with *loc*. The commands currently included are: *pwd*, *cd*, *ls*, *du*, *mkdir*, *rm*, *cptr* and *cpfr*. These commands provide basic functionality for browsing and manipulating the local and remote file systems, though others could be included, such as a local copy and move.

There is also a major security risk in this projects implementation, due to no awareness of users. Currently the connected client has all the permissions of the user the server is being ran as, which means there is the potential for the client to accomplish things that they shouldn't, especially if the server is ran as *root*. This issue coupled with the unencrypted data transmission could allow remote intrusions to the server. This could be improved by forcing clients to login as a user on the server or, like *FTP*, anonymously, and the server should operate as a non-interactive user. Encryption of traffic could also be utilised through libraries such as OpenSSL or OpenSSH. Both of these features were considered beyond the scope of this project.

6 Conclusion

This project has succeeded in implementing the required functionality, though it has several flaws in security and usability, as noted above, along with potential improvements to help combat these flaws.

One thing that hasn't been mentioned is that the server does not function properly on Windows systems. The exact cause of this is unknown, though the problem manifests as an indefinite stall when opening files for reading or writing. It is likely that the problem is caused by the Windows implementation of *std.concurrency* as this functionality works as intended in the single-threaded client, but that is yet to be determined.

References

[1] Walter Bright and Andrei Alexandrescu et al. *D Programming Language*. Digital Mars. 2012. URL: `http://dlang.org` (visited on 03/19/2012).

[2] J. Postel and J. Reynolds. *File Transfer Protocol*. RFC 959 (Standard). Updated by RFCs 2228, 2640, 2773, 3659, 5797. Internet Engineering Task Force, Oct. 1985. URL: `http://www.ietf.org/rfc/rfc959.txt`.

[3] J. Postel and J.K. Reynolds. *Telnet Protocol Specification*. RFC 854 (Standard). Updated by RFC 5198. Internet Engineering Task Force, May 1983. URL: `http://www.ietf.org/rfc/rfc854.txt`.

[4] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253 (Proposed Standard). Internet Engineering Task Force, Jan. 2006. URL: `http://www.ietf.org/rfc/rfc4253.txt`.