A Simple File Transfer Protocol

CS4254: Project 1
Fall 2002
Deadline: 10/14/2002

Introduction

Most network applications rely on file transfer protocols in one form or the other. For instance, the HTTP protocol used in the Web is a generic file transfer protocol. A client (web browser) contacts the server and asks for a file (a web document). After the web server returns the file, the client may parse it – if it is a HTML document - and ask for inline images or other files included in the document. What makes the WWW tick is the underlying file transfer mechanism. Parsing, interpreting and displaying a web document a service added on to the underlying file transfer protocol. Most of the complexity of the web is at the client side and not on the server.

In this project, you will implement a client program and a server program for a simple file transfer protocol. The client program presents a command line interface that allows a user to

- Connect to a server
- List files located at the server.
- Get (retrieve) a file from the server.
- Send (put) a file from the client to the server.
- Terminate the connection to the server.

The server program binds to a port and listens for requests from a client. After a client connects to the server, the server waits for commands. When the client sends a terminate message, the server terminates the connection and waits for the next connection.

Implementation

You should implement your project in C as two independent programs, an ftp client called **`ftp_client`** and an ftp server called **`ftp_server`**. The ftp_client program presents a command line interface, which supports the following commands. The format of a command is

`command_name <command arguments> \n`

- `connect <server name> <server port>`
- `list`
- `get <filename 1> <filename 2>`
- `put <filename 1> <filename 2>`
- `quit`

All commands entered at the client will be as ASCII strings. The client should also send the commands to the server as ASCII strings. On receiving a command, the server should parse the command and perform the appropriate action.

1. `connect <server name> <server port>`: This command allows a client to connect to a server. The arguments are the name of the server – for instance vtopus.cs.vt.edu – and the port number on which the server is listening for connections. On receiving this command at the command line, the client should initiate a connection to the server. **You should not send this command to the server.**

2. `list`: When this command is sent to the server, the server returns the list of files in the current directory on which it is executing, as a sequence of newline terminated strings. The client should get the list and display it on screen. To get the list of files in the current directory at the server, take a look at the man pages for
    - getcwd
    - opendir
    - readdir (man 3 readdir)
    - closedir

3. `get <filename 1> <filename 2>`: This command allows a client to get the file specified by filename 1 from the server. The file should always be treated as a binary file, i.e. use read()/write() or fread()/fwrite() calls to get the data from the file. Don't use fprintf() or fscanf(). On receiving this command, the server should open the file specified by filename 2 and transfer the contents to the client. The client should create a new file – filename 2 – and save the contents of the received file into filename 2.

4. `put <filename 1> <filename 2>`: This command allows a client to send the file specified by filename 1 to the server. Similar to the get command, you should treat the contents of the file as binary data. On receiving this command the server should create a new file specified by filename 2 and save the contents of the message.

5. `quit`: This command allows a client to terminate the connection. On receiving this command, the client should send it to the server and terminate the connection. When the ftp_server receives the quit command it should close its end of the connection.

Client/Server Interaction

To implement the communication between the client and the server you need to use 2 TCP sockets – a control socket and a data socket. The control socket between the ftp_client and the ftp_server is used for sending control messages. Control messages are the commands sent by the client. The data socket should be used for sending/receiving data between the ftp_client and the ftp_server programs. This interaction is illustrated with 2 examples:

- `get <filename 1> <filename 2>`: The client program sends the get command over the control socket. On receiving this command, the server sends the data back on the data socket. The client reads data from the data socket and saves it to the file specified by filename 2.

- `put <filename 1> <filename 2>`: This command from the client is sent to the server over the control socket. After the ftp_server connects to the data port, the client sends the data corresponding to filename 1 over the data socket to the server. The ftp_server reads the data from the data socket and saves it to filename 2.

The client server interaction also has another interesting aspect. The ftp_server program acts as the server on the control port, and the ftp_client acts as the server for the data port. By "acting as a server" I mean creating a socket, binding to the socket and listening for and accepting connections on the socket. This interaction happens in two steps. First, the ftp_client connects to the ftp_server over the control port. After the client-server connection is established, you need to establish a data connection. **The data connection is established and torn down after each command.** After the ftp_client sends a command to the server where it expects a response – list, get, put – it opens the data port as the server. The ftp_server program connects to the data port established by ftp_client and sends the response. In the case of the put command, you need to wait for a connection to be established by the ftp_server before you send any data. When you receive and EOF (end-of-file) on the data connection, you know that the server has no more data to send and the data connection can be terminated.

Why all this complexity? Why can't we just use one socket and use TCP's ability to do bi-directional communication on a socket to send data both ways. A little thought will show that you now have to worry about both ASCII control messages as well as binary data flowing over the socket. Think about the sequence of actions that happen on the put command. The client sends the put message and then starts transferring the file. If you happen to transfer a file that has your grocery list in it and the first line of the file says list, the server will get confused. Also, if you use a single connection you would need to worry about the length of each command and response. If you use separate control and data connections, where the data connection is torn down after each command, you can use the EOF indication on the data connection to infer that there is no more data to send.

The next question to ask is what port numbers should you use for the control port and the data port. For the control port (server port) the port number should be the last 4 digits of your SSN. If the last 4 digits of your SSN are less than 1024, add 10000 to the last 4 digits. For example, if the last 4 digits of your SSN are 0123, your control port number should be 10123. For the data port number add 1 to the control port number using the same calculation as above. For example if your control port is 3478, your data port should be 3479. You may hard code the control port number in your server program.

Submission information

NOTE: For this project up to 2 students may pair up to form a group.
Your code should be implemented as C programs that run under Linux. You need to submit an electronic version of your project to chekhov.cs.vt.edu/cs4254.

The submission should contain

- Source code

- All binaries

Your source code should also contain a comment on the first few lines with the authors of the code and the VT ID numbers of the authors.

The submission format is tar. Use the tar program to archive the contents of your project directory as follows:
If your project directory is called `project1` and it is under your home directory, then go to your home directory and issue the following command:

```
tar cvf filename.tar project1
```

`filename.tar` now contains the archived version of your project directory.

Each group is allowed 3 submissions with try numbers numbered 1, 2 and 3. Any submissions past 3 will be ignored. The last submission will be graded.

Happy debugging!