CSCI 4430, 2014-2015 Term 2

Assignment 1: A Multi-Client File Transfer Application

Version 1.0, 2015 January 22

1 Introduction

In this assignment, we will implement a multi-client file transfer application called MYFTP. MYFTP is a simplified version of the File Transfer Protocol (FTP). It includes the basic functionalities of FTP, including authentication, file upload/download, and file listing. It also enables multiple clients to simultaneously connect to the server at the same time. Our objective is to develop hands-on skills of socket programming and multi-threading programming in C or C++.

2 Protocol Description

2.1 Overview

MYFTP is mainly composed of two entities: client and server. As their names suggest, the server hosts a file repository where the client can request to upload or download files.

Figure 1 is the state diagram that summarizes the client program flow of MYFTP. In a nutshell, the client first opens a connection to the server. It then authenticates itself to the server. If the authentication request is granted, then the client can perform three main functions that include (i)

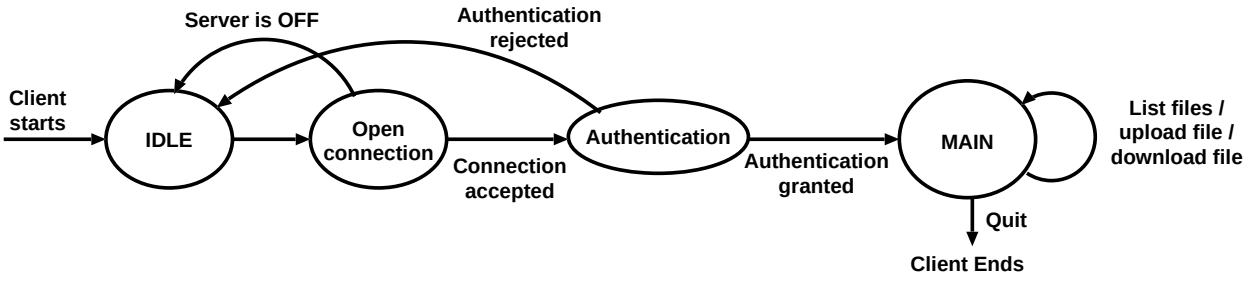

Figure 1: Client program: state diagram.

listing the files that are stored on the server, (ii) uploading files, or (iii) downloading files. When the client finishes, it quits and releases the connection with the server.

2.2 Protocol Messages

The client and the server will exchange protocol messages in order to perform the functions. All protocol messages are preceded with the protocol header shown in Figure 2.[1]

```
struct message_s {
    unsigned char protocol[6];   /* protocol magic number (6 bytes) */
    unsigned char type;          /* type (1 byte) */
    unsigned char status;        /* status (1 byte) */
    unsigned int  length;        /* length (header + payload) (4 bytes) */
} __attribute__ ((packed));
```

Figure 2: Protocol header.

- The `protocol` field is always set to be a series of six-byte data (remember, that is NOT a string) that starts with the byte '0xe3', followed by five characters "`myftp`".
- The `type` field has different possible values as specified in Section 2.3.
- The `status` field provides a YES/NO answer toward an incoming request.
- The `length` field is the length of the message (including the header and payload) to be sent over the network. The size of the payload is a variable depending on the message type. You can safely assume that the length is no larger than `INT_MAX`, described in "`limits.h`" in Linux.

We add "`__attribute__ ((packed))`" to avoid data structure alignment and pack all variables together without any gap[2]. Defining a packed structure makes our life easier when we calculate the size of a message to be sent over the network.

[1]For simplicity, we used "`unsigned char`" and "`unsigned int`" in Figure 2. However, it is more proper to use "`uint8_t`" and "`uint32_t`" defined in "`stdint.h`" for interoperability.

[2]See http://en.wikipedia.org/wiki/Data_structure_alignment for details

2.3 Protocol Details

We now explain the functions that MYFTP supports. We also introduce the protocol messages that accompany each function, as well as the formats of the protocol messages. For now, let us assume that there is only one single client.

2.3.1 Open a Connection

The following outlines the steps taken by the client while opening a connection with the server:

1. When a user starts a client and inputs the following command: "`open SERVER_IP SERVER_PORT`", where `SERVER_IP` and `SERVER_PORT` are the IP address and port numbers of the server, respectively, then a TCP connection should be initiated to the server's IP address and port.
2. Next, the client will send a protocol message **OPEN_CONN_REQUEST** to the server to request to open a connection.
3. If the server is running (i.e., it is listening to the server port number), it replies a protocol message **OPEN_CONN_REPLY**.
4. Last, the client receives the server's reply.

Figure 3 illustrates the message flow of how a connection is opened, and the protocol messages are shown in Figure 4. Note that:

- We assume that when a server is up, it always replies the **OPEN_CONN_REPLY** with the `status = 1`.
- When a field is stated as *unused*, neither the client nor the server would read the field content.

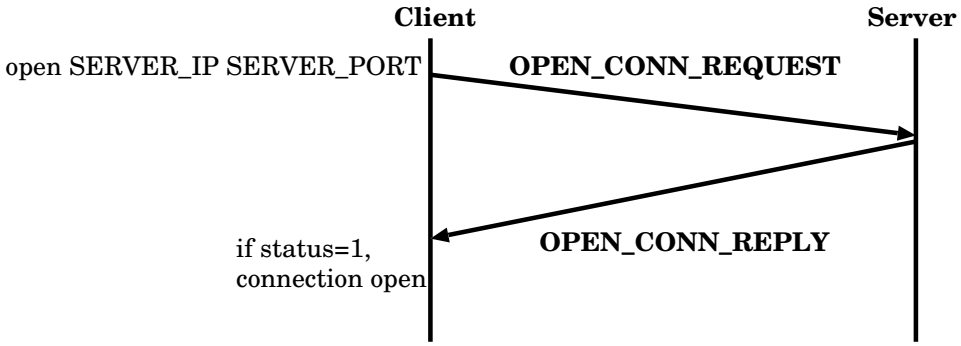

Figure 3: Flow diagram: open a connection.

OPEN_CONN_REQUEST	`protocol`	0xe3myftp
	`type`	0xA1
	`status`	unused
	`length`	12

OPEN_CONN_REPLY	`protocol`	0xe3myftp
	`type`	0xA2
	`status`	1
	`length`	12

Figure 4: Protocol messages: open a connection.

2.3.2 Authentication

After the connection is successfully open, the user needs to authenticate himself by issuing the command "`auth USER PASS`", where `USER` and `PASS` are the username and password, respectively. Then, the client will send a protocol message **AUTH_REQUEST**, which contains both the username and password, to the server. The server will then check against an ASCII text file named `access.txt` which is in the following format:

```
alice pass1
bob pass2
......
```

Each line of `access.txt` corresponds to a pair of username and password for some particular user. We assume that the username and the password are alphanumeric strings (i.e., strings with only A-Z, a-z, 0-9) separated by **one space character**. The following is a list of assumptions over `access.txt`.

- Every user ID in `access.txt` must be unique, and the maximum length is 32 bytes.
- The password of every user must be non-empty, and the maximum length is 32 bytes.
- There is always a newline character at the end of each `(User ID, Password)` duple.
- `access.txt` is stored in the same directory as the server program. Such a file always exists on the server.

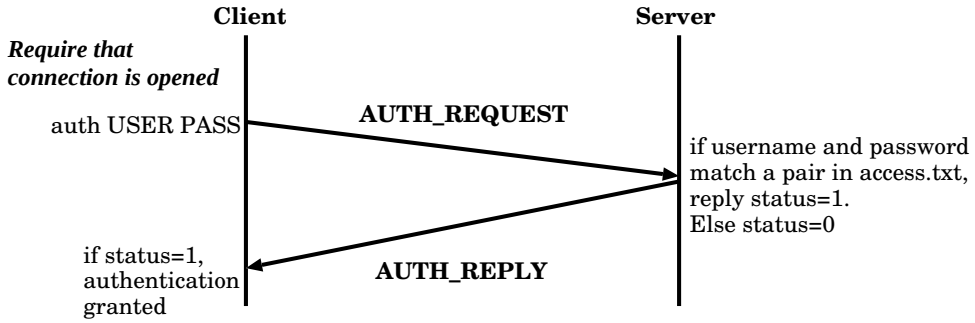

Figure 5: Flow diagram: authentication.

If `access.txt` contains a pair of username and password that matches the pair provided by the client, then the server will reply **AUTH_REPLY** with a status field set to 1. Otherwise, the server will reply **AUTH_REPLY** with the status field set to 0.

If the client receives the status equal to 0, then it **closes the connection** immediately (i.e., returning to the IDLE state as in Figure 1). Figure 5 illustrates the message flow of the authentication steps and the protocol messages are shown in Figure 6.

Note that according to Figure 1, a client program has to check whether the server program is running or not. If not, an appropriate error message should be printed.

AUTH_REQUEST	`protocol`	0xe3myftp
	`type`	0xA3
	`status`	unused
	`length`	12 + strlen(payload) + 1
	`payload`	`USER␣PASS` (as one null-terminated string)
		`USER` and `PASS` are separated by one space.

AUTH_REPLY	`protocol`	0xe3myftp
	`type`	0xA4
	`status`	0 or 1
	`length`	12

Figure 6: Protocol message: authentication.

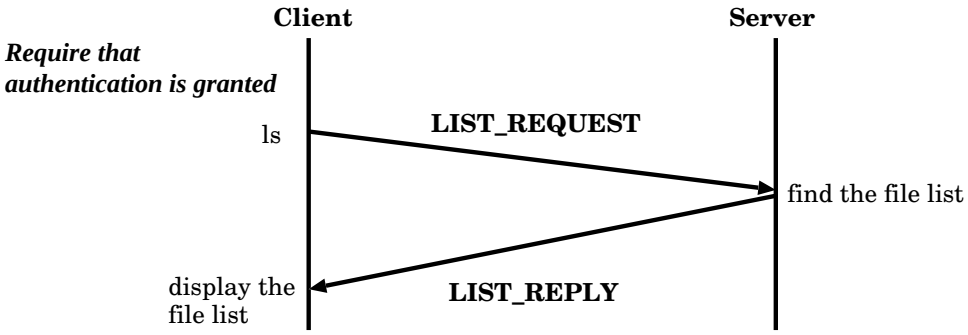

Figure 7: Flow diagram: list files.

LIST_REQUEST	`protocol`	0xe3myftp
	`type`	0xA5
	`status`	unused
	`length`	12

LIST_REPLY	`protocol`	0xe3myftp
	`type`	0xA6
	`status`	unused
	`length`	12 + strlen(payload) + 1
	`payload`	file names, as one null-terminated string

Figure 8: Protocol message: list files.

2.3.3 List Files

After the user is authenticated, he can perform the main functions. Suppose that the user wants to list all files that are stored in the server. The user will issue a command "`ls`". Then, the client will send a protocol message **LIST_REQUEST** to the server. The server will reply a protocol message **LIST_REPLY**, together with the list of available files.

All files are stored in a repository directory called "`filedir`", which is located under the working directory of the server program. The following lists the assumptions over the repository directory:

- The directory has been created before the server starts.
- The directory may contain files when the server starts.
- The directory contains regular files only (i.e., no sub-directories, no link files, etc).

- The server process has the necessary privilege to access the directory as well as the files inside the directory.
- The file names contain the following set of characters only: alphabets (a-z and A-Z) and digits (0-9).

To read the directory on the server side, you should use the function `readdir_r()` (run "`man readdir`" to see its usage). All file names sent in the **LIST_REPLY** message are followed by a newline character '\n'. Figure 7 shows the message flow of how to list files, and the protocol messages are shown in Figure 8.

2.3.4 Download file

Suppose that the user wants to download a file from the server. Then, he issues a command "`get FILE`", where `FILE` is the name of the file to be downloaded. The client will then send a protocol message **GET_REQUEST** to the server. On the server side, the server should perform the following actions:

- The string "FILE" supplied by the client is a path pointing to a name in the server repository.
- For examples, "`./hello.txt`" and "`hello.txt`" are names referring to the server repository.
- Some bad examples are: "`/etc/passwd`" and "`../hello.txt`".
- However, how to resolve the paths? Use the library call "`realpath()`". Please read the man page of the call by yourself.
- Then, the server will check whether the file is available in its repository directory or not.
- If not, the server will reply a protocol message **GET_REPLY** with the status field set to 0.
- If the file exists, the server will reply a protocol message **GET_REPLY** with the status field set to 1, followed by a **FILE_DATA** message that contains the file content.

Note that a file can be in either ASCII or binary mode. You should make sure that both ASCII and binary files are supported. Also, the program will overwrite existing, local files. On the other hand, we assume that for each client, only one file is downloaded at a time before the user issues the next command. Figure 9 shows the message flow of how to download a file and the protocol messages are shown in Figure 10.

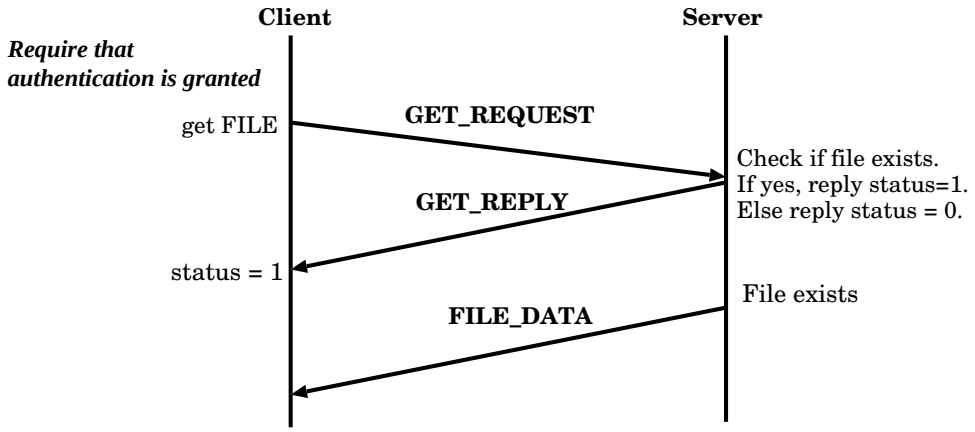

Figure 9: Flow diagram: download a file.

GET_REQUEST	`protocol`	0xe3myftp
	`type`	0xA7
	`status`	unused
	`length`	12 + strlen(payload) + 1
	`payload`	one file name, in one null-terminated string

GET_REPLY	`protocol`	0xe3myftp
	`type`	0xA8
	`status`	0 or 1
	`length`	12

FILE_DATA	`protocol`	0xe3myftp
	`type`	0xFF
	`status`	unused
	`length`	12 + file size
	`payload`	file payload

Figure 10: Protocol message: download a file.

2.3.5 Upload file

Suppose that the user wants to upload a file to the server. Then, he issues a command "`put FILE`", where `FILE` is the name of the file to be uploaded. First, the client uses "`realpath`" to check if

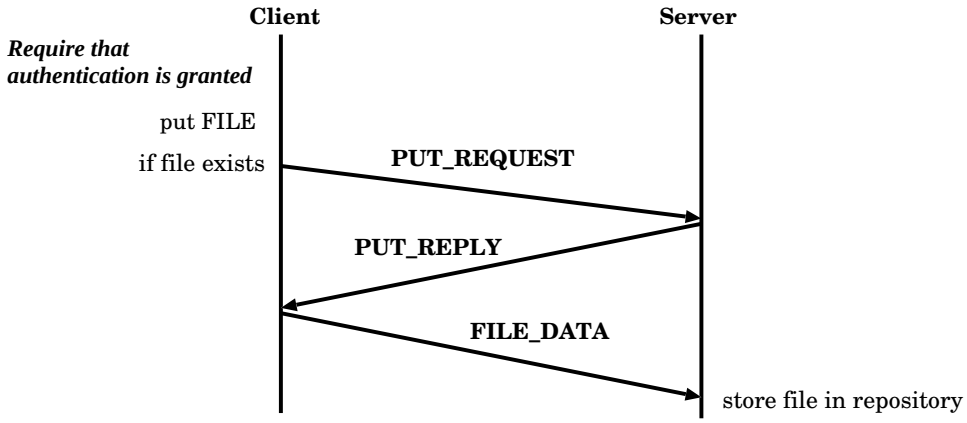

Figure 11: Flow diagram: upload a file.

PUT_REQUEST	`protocol`	0xe3myftp
	`type`	0xA9
	`status`	unused
	`length`	12 + strlen(payload) + 1
	`payload`	file name, in null-terminated string

PUT_REPLY	`protocol`	0xe3myftp
	`type`	0xAA
	`status`	unused
	`length`	12

Figure 12: Protocol message: upload a file.

the user input a filename referring to the current working directory. If not, an appropriate error message should be printed. Second, the client will check whether the file exists locally (i.e., on the client side). If not, the client will display an error message saying the file does not exist. So far, no protocol messages were involved, and the client-side checking is done.

If the file exists locally, then the client will send a protocol message **PUT_REQUEST** to the server. The server will reply a protocol message **PUT_REPLY** and awaits the file. Then the client will send a **FILE_DATA** message that contains the file content.

Here, we assume that the uploading file resides in the same directory as the client program. Also, only one file is uploaded at a time before the user issues the next command. Last, the server stores the uploaded file in the repository directory "`filedir`", using the name provided by the user.

Note that the server may overwrite existing files. Again, all files can be either ASCII or binary. You should make sure that binary files are supported.

Figure 11 shows the message flow of how to upload a file. The protocol messages are defined in Figure 12 (note that **FILE_DATA** is defined in Figure 10).

2.3.6 Close the Connection and Quit

To close the connection, the user issues a command "`quit`". The client will send a protocol message **QUIT_REQUEST** to the server. The server will reply a protocol message **QUIT_REPLY** to the client, and the client should release the connection (i.e., closing the TCP connection). Last, the client program will quit. Figure 13 shows the message flow of how to close the connection and the protocol messages are defined in Figure 14.

On the server side, since the connection is terminated, the corresponding thread should be terminated as well.

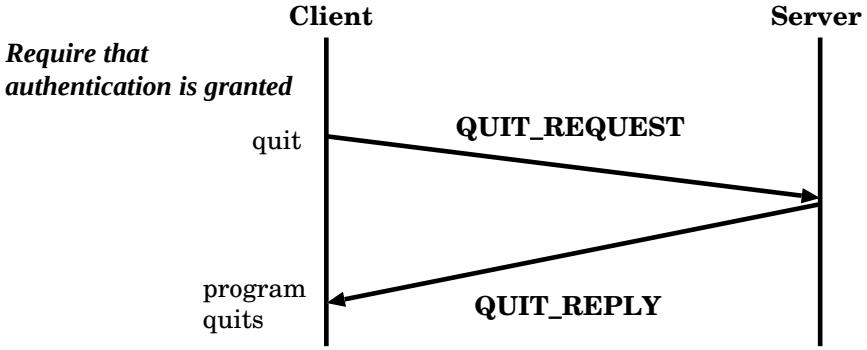

Figure 13: Flow diagram: close the connection.

2.4 User Interface

You may first invoke the server as follows:

```
linux1> ./myftpserver PORT_NUMBER
```

You may fill in the port number as desired. Although there is no restrictions over the interface of the server program, the server program should at least print the following information to its standard output stream:

- When a client establishes a TCP connection with it successfully, print out the IP address and the port number used by the client.

QUIT_REQUEST	`protocol`	0xe3myftp
	`type`	0xAB
	`status`	unused
	`length`	12

QUIT_REPLY	`protocol`	0xe3myftp
	`type`	0xAC
	`status`	unused
	`length`	12

Figure 14: Protocol message: quit.

- When a client tears down a TCP connection, print out the IP address and the port number used by the client.

For the client program, while you can design your own user interface, a fancy interface is NOT required (and NOT recommended). Nevertheless, in order to faciliate our grading process, your client program should provide the following commands:

- `open [IP address] [port number]`: send the `OPEN_CONN_REQUEST` message.
- `auth [username] [password]`: send the `AUTH_REQUEST` message.
- `ls`: send the `LIST_REQUEST` message.
- `get [filename]`: start the download procedures.
- `put [filename]`: start the upload procedures.
- `quit`: send the `QUIT_REQUEST` message.

Of course, the above set of commands does not just send a message, but also applies the application logic behind the commands / messages.

```
linux1> ./myftpclient
Client> open 137.189.99.9 12345
Server connection accepted.
Client> auth alice passpass
ERROR: Authentication rejected. Connection closed.
Client> open 137.189.99.9 12345
Server connection accepted.
Client> auth alice pass1
Authentication granted.
Client> ls
----- file list start -----
file1.txt
file2.txt
----- file list end -----
Client> get file1.txt
File downloaded.
Client> put file1.txt
File uploaded.
Client> quit
Thank you.
linux1>
```

The above example shows that if the server rejects the authentication request, then the client program should terminate the connection. It needs to re-initiate the connection again.

2.5 Unexpected Message Handling

There are occasions that the server or the client programs receive unexpected messages:

- A message does not start with our protocol magic number;
- A message contains an unknown protocol message type; or
- A message has a wrong length field.

In any of the above scenarios, the party which encounters such an unexpected message must print a message, stating that an unexpected message is received. Then, it must terminate the connection immediately. For the client, you should terminate process as well. For the server, you only need to terminate the corresponding thread after disconnecting.

For a well-implemented client and server programs, it would not send any unexpected messages. Therefore, our tutor will implement some malicious clients and servers for the sake of testing the robustness of your client and your server programs.

Note that you can assume that valid messages always come in a correct order, i.e., no unexpected, well-formatted message would arrive.

3 Milestones

3.1 Milestone 1 (60%)

In this milestone, we assume that only one client connects to the server at any given time. Your implementation should have the following functions:

- The client can open a connection to the server.
- The client can authenticate:
  - Authentication can be done only if a connection is opened
  - The server can either accept or reject the authentication
- The client can list files in the repository directory of the server.
- The client can download a file (either an ASCII or binary file) from the server. If the file doesn't exist, the server replies with an error message.
- The client can upload a file (either an ASCII or binary file) to the server. If the file doesn't exist, the client displays an error message.
- The client can list files, download/upload files only if authentication is granted.
- The client and the server can reside in different machines with different operating systems.

Aside the above functions, you may make assumptions to address the features that are not mentioned in this specification.

3.2 Milestone 2 - Multi-Client Support (30%)

Now the server needs to support N active, concurrent connections. You will have to implement a multi-client version using POSIX threads. The assumptions and the requirements are as follows.

- During the grading process, we will set $N < 5$. Of course, you are not required to keep the number of concurrent connection always below 5.

- You must use POSIX threads to realize the implementation of the server. The server should concurrently serve requests from N clients at the same time. A suggested implementation is to create a thread per connected client.
- Multiple clients must be able to concurrently upload and download files at the same time. The client program used in this milestone should be the same as the one used in the previous milestone.
- Multiple users will be using the same repository directory and you are NOT required to implement any locking mechanism over the directory nor the files inside the directory.

On top of that, your multi-client version of the server program needs to support all functions described in Milestone 1.

3.3 Milestone 3 - Handling of Unexpected Messages (10%)

Both the server and the client programs should be able to handle invalid messages.

4 Submission Guidelines

You must submit the following files (although you may submit additional files that are needed):

- myftpclient.c: the client program
- myftpserver.c: the server program
- Makefile: a makefile to compile both client and server programs

Your programs must be implemented in C/C++. They must be compilable on 32-bit, Intel-based, Linux machines (and you are welcome to support 64-bit machines) in the department. Please refer to the course website for the submission instructions.

We will arrange demonstration sessions to grade your assignments. Our tutors will user "*human eye*" to grade your work, instead of an automatic judge program. Nevertheless, our tutors will implement their version of client and server programs in order to test your implementation. You are encouraged to test your implementation against your classmates' work. Of course, you should exchange the compiled program, NOT SOURCE CODES. Have fun! :)

Deadline: 23:59, 2015 Feburary 15 (Sunday)

Change Log

Time	Affected Version(s)	Details
2015 Jan 22	NIL	Release version 1.0.