CSCI4430/ESTR4120 (Spring 2019)

Assignment 1: A Multi-Client File Transfer Application

Due on February 14, 2019, 23:59:59

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, such as 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-threaded programming in C/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.

MYFTP supports three main functions: (i) list the files that are stored on the server, (ii) upload files from the client to the server, or (iii) download files from the server to the client.

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 below:

```
struct message_s {
    unsigned char protocol[5];   /* protocol string (5 bytes) *
    unsigned char type;          /* type (1 byte) */
    unsigned int length;         /* length (header + payload) (4 bytes) */
} __attribute__ ((packed));
```

The `protocol` field is always set to be the string that starts with the string "myftp". The `type` field has different possible values as specified in Section 2.3. 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 $2^{32} - 1$.

We add `__attribute__ ((packed))` to avoid *data structure alignment* and pack all variables together without any gap[1]. Defining a packed structure makes our life easier when we calculate the size of a message to be sent over the network.

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. Table 1 lists all protocol messages that are used.

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

LIST_REQUEST	`protocol`	myftp
	`type`	0xA1
	`length`	header length
LIST_REPLY	`protocol`	myftp
	`type`	0xA2
	`length`	header length + payload length
	`payload`	file names, in null-terminated string
GET_REQUEST	`protocol`	myftp
	`type`	0xB1
	`length`	header length + payload length
	`payload`	file name, in null-terminated string
GET_REPLY	`protocol`	myftp
	`type`	0xB2 or 0xB3
	`length`	header length
PUT_REQUEST	`protocol`	myftp
	`type`	0xC1
	`length`	header length + payload length
	`payload`	file name, in null-terminated string
PUT_REPLY	`protocol`	myftp
	`type`	0xC2
	`length`	header length
FILE_DATA	`protocol`	myftp
	`type`	0xFF
	`length`	header length + file size
	`payload`	file payload

Table 1: Protocol messages

2.3.1 List Files

Suppose that the client wants to list all files that are stored in the server. The client will issue a command `list`. Then it 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 `data/`, which is located under the working directory of the server program. We assume that the directory has been created before the server starts. To list files in a directory, you may use the function `readdir()` (use `man readdir` to see its usage). All file names are delimited by the newline character '\n'.

2.3.2 File Download

Suppose that the client wants to download a file from the server. The client will issue a command `get FILE`, where `FILE` is the name of the file to be downloaded. It first sends a protocol message **GET_REQUEST** to the server. The server will reply a protocol message **GET_REPLY** with the type field set to 0xB2 if the file exists or 0xB3 otherwise. If the file exists, the server will send a **FILE_DATA** message that contains the file content following the **GET_REPLY** message.

Note that a file can be in either ASCII or binary mode. You should make sure that both ASCII and binary files are supported. Here, we assume that the file size is at most 1 GiB (which is equal to 2^{30} bytes).

2.3.3 File Upload

Suppose that the client wants to upload a file to the server. The client will issue a command `put FILE`, where `FILE` is the name of the file to be uploaded. First, the client will check whether the file exists locally. If not, the client will display an error message saying the file doesn't exist. If the file exists, 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 (we use the same **FILE_DATA** message as in file download).

Here, we assume that the file to be uploaded resides in the same working directory as the client program. If the server has already stored the file before, the file will be overwritten. *Note again that all files are in ASCII or binary mode*. We again assume that the file size is at most 1 GiB.

2.4 Interfaces

The server uses the following command-line interface:

```
sparc1> ./myftpserver PORT_NUMBER
```

You may fill in the port number as desired. The client uses the following command-line interface:

```
client> ./myftpclient <server ip addr> <server port> <list|get|put> <file>
```

Note that the `file` argument only exists for the get and put commands. You may define your own output format for any error message.

2.5 Implementation Issues

The client and the server can reside in different machines with different operating systems (e.g., Linux vs. Unix), and all functions should remain correct. *Make sure all byte ordering issues are properly addressed (see Tutorial notes).*

The server may support multiple clients that may upload or download files simultaneously. We assume that no two clients will upload or download a file with the same file name simultaneously (to make your life easier), but it is possible that a client downloads a file that is previously uploaded by another client.

3 Milestones

Your implementation should have the following functions:

- 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 locally, the client displays an error message.
- The client and the server can reside in different machines with different operating systems.
- Now the server needs to support a general number N of active connections (assume that $N \leq 10$). You will have to implement a multi-client version using multi-threading based on the pthread library.

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

4 Submission Guidelines

You *must* at least submit the following files, though you may submit additional files that are needed:

- *myftp.h*: the header file that includes all definitions shared by the client and the server (e.g., the message header)
- *myftp.c*: the implementation of common functions shared by both the client and the server
- *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 Unix/Linux machines in the department. Please refer to the course website for the submission instructions. We will arrange demo sessions to grade your assignments. Have fun! :)