CS 39006: Assignment 3
File Transfer Using Sockets
Date: 24-Jan-2022
Deadline: 07-Feb-2022, 2 pm

In this assignment, we will implement a simplified file transfer protocol. The actual working of ftp is more complex, this is a simplified version of it.

You will write two C files – the ftp server ftpS.c, and the ftp client ftpC.c. The ftp server will be a concurrent TCP server.

You will need to setup a few things first. In the directory from which you will run the server program, create a file called *user.txt*. Each line of the file contains a one word user login name and a one word password, separated by one or more spaces, in that order. There should be nothing else in the file. Create the file to contain 2 users, U1 and U2. Give arbitrary passwords.

Your ftpC.c should give a prompt "myFTP>" and wait for user commands. The user will type the commands (shown below), and the commands below will be sent to the server (except for the command **open** to open a connection (for which no command is sent to the server as there is no connection opened before that), **lcd** and **quit**).

At different times (see the commands below), the server may send a code back to the client. If a command is executed successfully, the server sends back the code 200; on an error, the server sends back 500 by default, unless otherwise mentioned in the specific command below. If the client receives an error code, it should print the code, along with a text message "Error executing command", otherwise the client should print "Command executed successfully" after the command finishes successfully. In both the cases, the client gives the prompt again and waits for the next command. The client terminates only when the **quit** command is given (see below).

The following commands are to be implemented for the system. The command is shown in bold, and the arguments in italics.

1. **open** *IP_address port* – This causes the client to open a TCP connection to the server with the designated IP address and port. This should be the first command entered on the client side (the client should check for it and not accept any other command before this). The IP address is to be given in dotted decimal form (like 10.15.60.3). Port should be any number between 20000 and 65535 for this assignment. The client prints a text message if the connection is opened successfully, else prints an error message.
2. **user** *username* – This must be the first command sent by the client to the server, the server should check for this. If any other command is sent as the first command, the server should send an error code of 600. If the username exists in the user.txt file of the server, a reply code of 200 is sent. If no such user exists, reply code of 500 is sent.
3. **pass** *password* – This must be the second command sent to the server. If any other command is sent as the second command, the server sends an error code 600. If the

password matches the user's password specified in the user.txt file, the server sends a reply code 200. If the password does not match, the server sends an error code 500.

The commands from 4 to 10 below should only be executed if the username and password have already matched. Both the client and the server should check that before executing any of these commands. If the username matches but the password does not, the user must enter both in that order again (not just the password).

4. **cd** *dir_name* – It changes the server side directory to *dir_name*. If the directory change is successful, the server sends a reply code 200; otherwise the server sends an error code 500.
5. **lcd** *dir_name* – It changes the local (client side) directory to *dir_name*. Nothing is sent to the server.
6. **dir** – It shows the contents of the current directory of the server. The client sends the command to the server and the server sends the directory content to the client.
7. **get** *remote_file local_file* – It gets the file named *remote_file* from the server and stores it under the name *local_file* on the client. File names can be just a file name, in which case it must exist on the current directory in the server. Or there can be an absolute path name to a file (ex. /usr/home/agupta/myfile.txt). The file name cannot be relative to the current directory or parent directory (i.e., cannot start with . or ..). If any file with the name *local_file* already exists in the client, it is overwritten. The client first checks if it can open the local file for writing. If not, a message is printed and no command is sent to the server. Otherwise, the command is sent. If the remote file does not exist in the server, it sends an error code 500 to the client, and transfers nothing. Otherwise the server first sends the code 200, followed by the contents of the file in the format specified later.
8. **put** *local_file remote_file* – It puts the file named *local_file* from the local directory of the client to the file *remote_file* in the server. The file names are similar to that in the *get* command. If any file with the name *remote_file* already exists in the server, it is overwritten. The client first checks if it can open the local file for reading. If not, a message is printed and no command is sent to the server. Otherwise, the command is sent. If the server is able to open the remote file, it sends a code 200. The client then transfers the file in the format specified later. If the server is unable to open the remote file for some reason, it sends the error code 500 and the client sends nothing.
9. **mget** *file1, file2,…* - gets a set of files given by the filenames file1, file2, … from the server, and copies them to the current directory on the client under the same name. The filename convention is the same as in **get** and **put**. The command fails if any of the file transfers fails (some of the others may have already been completed by then, that's fine). This command is to be executed by sending the **get** command multiple times with the proper arguments.
10. **mput** *file1, file2,…* - same behavior as mget, just puts files from the client on to the current directory of the server. This command is to be executed by sending the **put** command multiple times with the proper arguments.
11. **quit** – closes the connection (if open) and exits.

The file transfer does not happen in one step when the file is arbitrarily long. The file is sent in binary mode as a sequence of bytes, block by block. Each block is prefixed with a header containing two fields, a character ('L' indicates it is the last block of the file, and a character 'M" indicates it is not the last block of the file and more blocks will follow), and a 16-bit integer (short) indicating the length in bytes of the data sent in this block (not including the header bytes). So a file transfer is over after a block with 'L' in the character field is received. Note that this check is sufficient for detecting the end of the file, as TCP ensures that data bytes are received in order. Thus, the sender of the file will read the file block by block (with red call), prefix it with the header, and send it (with send call). Some points to remember:

- This being a TCP connection, there is no guarantee that the entire block will reach together to the other side, you should take care of that.
- You should also take care of the fact that the architectures on both sides may be different
- The files can contain anything, do not assume txt files

The commands are to be sent exactly as shown as null-terminated strings from the client to the server, except for **open** (connection open, no command to send), **lcd** (local operation at client), **quit** (connection close only), and **mget/mput** (sends get and put commands).

Note that any ftp client should work with any ftp server, not necessarily written by the same group. So you should carefully read the specifications and follow them strictly, not just somehow make your own client and server work together. Protocol specification is important for interoperability; in the real world, different entities write the client and server respectively (thin a web server and a firefox/chrome client). If you see any ambiguity or incomplete specification anywhere that may make a client not work with an arbitrary server, please let us know

Submit the files ftpS.c and ftpC.c in moodle in the submission link in a single zip file.