Assignment 3

Implementing a simplified ftp client/server

Due: August 24, 2009, 1 pm

In this assignment, we will implement a simplified version of the file transfer protocol. The subset of the commands we will implement is small, ftp has a much richer functionality. Even within this subset, I have modified them to make things easily understandable and simpler. However, the focus in this assignment is to implement something exactly from the specifications given. Your client and server will NOT be tested against each other. Your client will be used to connect to others' servers and vice-versa.

In ftp, there are two separate connections made – a control connection used to send ftp commands and replies, and a data connection for sending and receiving data. The server waits on a predefined control port, and the control connection is made by the client. There are default data ports defined for both server and client; connections can be made to these ports to transfer data. The client side default data port can be changed to any port by the client and the server can be informed about this by the ftp PORT command. However, to keep things simple, we will not allow the default port to be changed in the middle of a ftp session. The control connection is kept open until the ftp session is over; however, the data connection can be created and destroyed by the server for each file transfer. More details of ftp are available in RFC 959.

You will write two C files – the ftp server ftpS.c, and the ftp client ftpC.c. The ftp server will be an iterative TCP server. The server will have two processes, S_C (the server's control process) and S_D (the server's data process). The process S_C will create a TCP socket S1, bind S1 to port X (to be specified later), and then wait on S1. The process S_D will wait to be woken up by the thread S_C when a data transfer has to be made. After waking up, it will create a TCP socket S2 and connect to the client's data port using it. S1 will be the control port for the server and S2 will be the data port for the server. After each transfer is over, it will close S2. S2 will be created again for the next transfer.

The client will also have two processes, C_C (the client's control process) and C_D (the client's data process). The process C_C will create a TCP socket C1, which will then connect to port X of the server (recall that the server's control process S_C is waiting on port X). The process C_D will create a TCP socket C2, bind C2 to port Y, and then wait on C2. C1 will be the control port for the client and C2 will be the data port for the client.

The basic flow of operation will be as follows. The client's control process C_C will send a command to the server's control thread S_C. Each command is a text word followed by 0 or more text word arguments. The command and the arguments are separated by one or more spaces, and the entire string is terminated by '\0'. The length of the string cannot be greater than 80 characters. In response, the server's control thread will send a 3-digit integer reply code (this is the only thing sent by the server in response, no string is sent.

Also, it is sent as an integer, and not as, for example, ascii code of the digits etc.). Based on the command, the server's control process S_C may wake up the server's data process S_D, which may then connect to port Y of the client (recall that the client's data process C_D is waiting on port Y), and some data transfer may take place. The server will close the data connection after every transfer is over. The details of the commands and replies are described next.

Your ftpC.c should connect to the server's control port, and just give a prompt ">" and wait for user commands. The user will type the commands (shown below), just like you do for a normal ftp client, and the commands below will be sent over the control connection. If the client receives an error code, it should print the code, along with some meaningful text message. If the client receives a non-error reply code, it should also print the code and a meaningful txt message.

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

1. **port** *Y* – this must be the first command sent to the server. Y is the port number of the data port at which the client waits. If any other command is sent as the first command, the server sends an error code 503, closes the connection, and returns to wait on socket S1. If the server receives it properly (checks that Y is between 1024 and 65535), it sends a reply code 200. Otherwise it sends an error code 550, closes the connection, and returns to wait on socket S1. If the client receives any error code from the server, it closes all connections and exits.
2. **cd** *dir_name* – this can be sent any time. It changes the server side directory to *dir_name*. If the directory change is successful, the server returns a reply code 200. Otherwise it sends an error code 501. If an error code is received, the client just prompts for the next command from the user (but does not exit). Note that the server also does not close the control connection, it just waits for the next command.
3. **get** *file_name* – This can be sent anytime after the *port* command. It gets the file named *file_name* from the server. *file_name* can be just a file name, in which case it must exist on the current directory in the server. Or *file_name* 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 ..). The file is stored with the same name in the local directory on the client side. If any such file already exists, it is overwritten.

   Before the client sends the **get** command to the server, it should notify the process C_D about the file name and the command name (**get**). You need to design some appropriate way of implementing this.

   On receiving the **get** command, the server first checks if the file exists and whether it can be opened for read. If the file does not exist, or if it cannot be opened for read, an error code 550 is sent to the client. If the file exists, the server first notifies the process S_D of the command name (**get**), the filename and the port

Y received from the client earlier. The process S_D then creates a socket S2 and opens a data connection to port Y of client using socket S2, transfers the file, and closes the connection. It then notifies the server process S_C that the transfer is over. It should also notify the process S_C if some error occurs during transmission. You need to design some appropriate way of implementing this communication between the two processes. If the transfer is successful, the server sends the reply code 250 to the client over the control connection; else it sends an error code 550. The server waits for the next command after sending the reply/error code. The client's C_D process, on finding that the server's S_D process has closed the connection, closes its end of the connection and comes back to wait on C2 (you will need to design some appropriate way of implementing this communication between the two processes so that the client process closes the connection only after the server thread has done so).

If an error code is received, the client just prompts for the next command from the user (and does not exit).

4. **put** *file_name* – This can be sent anytime after the *port* command. It puts the file named *file_name* from the local directory of the client to the local directory of the server. The *file_name* can be just a file name, no path is allowed; so the file must exist on the local directory of the client. The file is stored with the same name in the local directory on the server side. If any such file already exists, it is overwritten.

   Before the client sends the **put** command to the server, it should notify the process C_D about the file name and the command name (**put**). You need to design some appropriate way of implementing this.

   On receiving the **put** command, the server first notifies the process S_D of the command name (**put**), the filename and the port Y received from the client earlier. The process S_D then creates a socket S2 and then opens a data connection to port Y of client using socket S2, and waits to read the file. On the client side, the process C_D, which was waiting on the socket C2, now comes out of the accept call (as the server has connected to it). It then sees the command is **put**, reads the file and sends it to the server over the data connection. The server's S_D process, on receiving the file, stores it with the same name in the local directory and closes the data connection. It then notifies the server process S_C that the transfer is over. It should also notify the process S_C if some error occurs during transmission. You need to design some appropriate way of implementing this communication between the two processes. If the transfer is successful, the server sends the reply code 250 to the client over the control connection; else it sends an error code 550. The server waits for the next command after sending the reply/error code. The client's C_D process, on finding that the server's S_D process has closed the connection, closes its end of the connection and comes back to wait on C2. (you will need to design some appropriate way of implementing this communication between the two processes so that the client process closes the connection only after the server thread has done so).

If an error code is received, the client just prompts for the next command from the user (and does not exit).

5. **quit –** The server, on receiving this command, sends a reply code 421 and closes the connections with the client, and goes back to wait on the socket S1 for the next client. The client, on receiving this reply code, closes connections and exits.

In addition to the above, the following two checks are always done by the server. If a command is received that is not one of the above commands, the server sends an error code 502. It will then wait for the next command from the client. Also, if a server receives a valid command, but an invalid argument (for ex., the **get** command is sent with two filenames etc.), it sends the error code 501 and then waits for the next command from the client.

Also remember that if the client receives an error code, it should print the code, along with some meaningful text message, before it does anything else (for ex., closing a connection, or sending the next command etc.). Similarly, if the client receives a non-error reply code, it should also print the code and a meaningful txt message before it does anything else.

You can also assume that the **port** command is not sent more than once by the client. So you do not need to check for this.

The file transfer does not happen in one step, as the file may be 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, any other character indicates it is not the last block of the file), and a 16-bit integer (short) indicating the length in bytes of the data in this block sent (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 (what will be the block size?), prefix it with the header, and send it.

Use port X = 50000, and Y = 55000. Submit only the two C files.