ECE361S Lab 2 Instructions:

File transfer/ FTP

Section 1: Simple file transfer

In this section, you will write a server-client pair (two applications; one running as server and one as client) that can communicate with each other to transfer a file. You should implement both server and client.

Scenario:

1. Similar to the chat client-server pair (Lab 1 instruction) you will need a `ServerSocket` in the server that waits until a client connects.

2. After having a socket connected, the client requests a file from server by sending the file name.

3. The server looks for the file in its system (only the current working directory). If the file exists, it will send a message to client notifying that the file transfer is about to begin. If the file does not exist, server sends an error message letting the client know that the file is not found on the server.

    - In order to open/create a file, you can use the `java.io.File` library (e.g. `File file=new File(filename);`).
    - In order to check for the existence of the file you can use the `file.exists()` method.
    - You can read and write from/to a file via the `read()` and `write()` methods of `java.io.FileInputStream fin = new FileInputStream(File file)` and `java.io.FileOutputStream fout = new FileOutputStream(File file)` respectively.
    - DON'T forget to close the input and output streams (in this case, `fin.close()` and `fout.close()`) after you are finished with the file. Otherwise there is a high chance that your file would not be stored properly.

4. Once the client receives the server's response to its request, if the response does not indicate an "error" it starts to receive the file content until the transfer is complete and the server closes the socket. Likewise, the server, after sending the notification to client, starts the file transfer. Once the transfer is complete, the server closes the connection and goes back to 1 (waiting for the next request to arrive).

5. In the previous lab instructions, you learned how to write and read lines of text using `java.io.DataOutputStream` and `java.io.BufferedReader` via the `writebytes(String str)` and `String str=readline()` methods. For transferring the file (bytes of data), you can use `DataInputStream din=new DataInputStream (socket.getInputStream())` and `DataOutputStream dout=new DataOutputStream (socket.getOutputStream())` via the `read(byte[] b, int off, int len)` and `write(byte[] b, int off, int len)` methods. To send the files, define a byte array with arbitrary size (e.g. `byte[]=new byte[1024]`) and

iteratively read chunks of data and send it to the client, until the end of file is reached. Note that the last chunk of data, is not necessarily equal to your defined array length. Therefore, you need to send only the amount of bytes that you have read from the file. To do that, you can look at the returned value of `fin.read()` method. The returned value is the number of read bytes. You can use it as the `len` input for the `write(byte[] b, int off, int len)` method when writing to socket.

6. Read the file name to be received in the client from the standard input (`System.in`)
7. Show proper output comments in the standard output (`System.out`) at each step.

8. Close the socket at the end of the program (use `Socket.close()`).

Section 1.2

1. Find the length of the file that was transferred and show it on both server and client's standard output. If you receive the file completely, the file lengths should be identical. You can use `file.length()` method to find the Length of a file.

2. Keep track of the transfer time using "`System.currentTimeMillis()`".

Question1: What was the data transfer rate? (You may need to transfer a big file to obtain a good approximation.)

Section 2: Use two connections for data transfer

In this section you should implement the data transfer part of the previous section, in a separate connection. The first connection in which the requests are send and the responses are received are called the "control connection" and the second connection is called "data connection". The resulting server-client pair works as follows:

1. The connection to the server, sending a file request, and receiving the response are the same as previous section.

2. This time, if the file is found, the server also sends a port number to the client. The client then makes a second connection to the server via the given port and starts receiving the data on the second connection (data connection).

3. Once the file transfer is over, the server will close the data connection and the client will be notified by checking if the socket is closed, as well as checking if the input stream still returns bytes. (If any of the `read()` methods of `DataInputStream` or `FileInputStream` reach the end of the stream, they will return 0 or -1 as the number of read bytes.)

Section 3 Use two threads for data transfer

Do the same as Section 2, but this time read the data on a separate thread. This can be useful when there is a need to communicate on the control connection, while a data transfer is in progress (e.g., abort transfer or parallel transfer). For example, if multiple downloads are needed to be supported, each file transfer should be on a separate thread. You will use multithreaded connections when you need to receive packet acks, in next labs.
You can look at the previous lab instruction (Section 5) for more information about multithreading.
If you want to run a new thread but do not need to define a new class, a very simple and integrated code that is able to run on a separate thread is given below:

```
new Thread(new Runnable() {
      @Override
      public void run() {
            //This section runs on a separate thread.
            //...
      }
}).start();
```

Section 4 (optional): An actual ftp client

The system you implemented in previous sections is the core part of an FTP application. You are very close to writing an actual FTP client!
In this section, you will implement a part of an actual FTP client. According to the file transfer protocol (FTP) (RFC959 and several RFCs afterwards), an FTP client is supposed to send its requests in a defined format:

- Each of the requests is a three or four letter text, and possibly one or two parameters after that.
- The parameters should be separated from the FTP commands by a white space.
- Each command should end with an end of line (\r\n) character.

The received response from the server is also expected to be in a defined format:

- The server always responds with a three digit return code which depending on the command may be followed by some extra parameters, followed by an arbitrary human readable text comment.
- The return code, the parameters and the comment are separated by white space and the command ends with an end of line (\r\n) character. However, for some commands (none of which we implement here), there might exist some end of line characters in the middle of the parameters.

A complete list of possible FTP commands and server return codes are available online and can be found using your favorite search engine.
According to the FTP protocol, two connections must be used for control purposes and data transfer.

The default port of control connection is 21 and the default port of data transfer is 20. However, there should be no problem with running FTP on arbitrary ports.

Sample FTP commands:

USER	Authentication username.
SYST	Return system type.
EPSV	Enter extended passive mode.
LIST	Returns information of a file or directory if specified, else information of the current working directory is returned.
RETR	Retrieve a copy of the file
STOR	Accept the data and to store the data as a file at the server site
QUIT	Disconnect.

Sample server return codes:

229	Entering Extended Passive Mode (|||port|).
150	File status okay; about to open data connection.
226	Closing data connection. Requested file action successful (for example, file transfer or file abort).
215	NAME system type.
230	User logged in, proceed. Logged out if appropriate.
220	Service ready for new user.
221	Service closing control connection.
450	File not found

Emulated Server:

You are given an emulated FTP server with limited implemented commands (in the terminal run "`java MainClass`"). Write a client application that connects to the emulated FTP server. The connection that you make will count as the control connection and will be used for sending FTP commands and reading reply codes.
Implement the following commands for an FTP client:

Section 4.1 Implement the USER and SYST commands

The USER command is used for logging a user via its username. If the server requires password-protected authentication, after entering the username, it will ask for password (code 331). The emulated server that you are given does not require authentication, so you can enter any arbitrary username for logging in.

By convention, once the client application runs, it will ask for the username and sends the USER request to the server. However, during the FTP session, the user can always change his/her username. The format of the command is "USER *username*". Once this request is sent to the server, the server will check if the username is acceptable (in our server all usernames are acceptable!). If the username is accepted the server responds with a reply code "230" followed by an arbitrary comment.
- You can also try to send the server the SYST command to retrieve the type of the machine running the server.

Section 4.2 Implement the GET command

Implement the GET command. The GET command is used to retrieve a file. The user of the ftp client will enter the command "GET *filename*". The *filename* is the file that the server should send to the client. In order to receive the file, you can use a code similar to the code written in Section 2. First you have to ask the server to open the data connection. In FTP, this is called the "passive mode":

Entering the passive mode:

- The request command that you should send to the server is "EPSV" for entering the passive mode.
- If everything is OK, the server should respond with the ok code "229". Wait to receive the "229" response code as well as the port for data connection. The response from server looks something like this:
  `229 Entering Extended Passive Mode (|||16703|),`

  where the number 16703 is the given data connection port. You can parse the port number in the given text by looking at the "pipe" indicators (i.e., the "|" character). An easy way to do that is to look for the last pipe character and read all the numbers before it. You can also use `StringTokenizer st=new StringTokenizer(String str,"\r\n\t |.()"/*these are the delimiters*/);` along with the method `st.nextToken();`.

Retrieving the file:

- Once the data connection is established, send the request code "RETR *filename*" on the control connection. This command is used for retrieving files.
- Wait to receive the "150" ready code on the control connection. Once the code is received, you should start receiving the data on the data connection.
- Once the transfer is complete, close the data connection and wait on the control connection to receive the "226" response code.

Section 4.3 Implement the "put" command

Implement the FTP "put" command. The "put *filename*" command is similar to the "get *filename*" command, except that this time you will be the one who sends a file and the server will be the one to receive it.

- The request command that you should send to the server is first EPSV (similar to get) for entering the passive mode.
- Wait to receive the 229 code as well as the port for data connection.
- Once the data connection is established, send the request code "STOR *filename*" on the control connection.
- Wait to receive the "150" ready code. Once the code is received, you should start to transfer the data on the data connection.
- Once the transfer is complete, close the data connection and wait on the control connection to receive the "226" response code.

Section 4.4 Implement the LIST command

The LIST command lists the directories and files that exist in the current working directory of the server (similar to what "ls" does in the terminal on the local computer). You can also add an extra parameter to it (which is the directory you want to be listed, e.g. *ls /usr/bin*). The logic of the command is exactly the same as the GET command. This time after entering the passive mode, you should send the request "LIST [*directory*]". For receiving the directory, you can use `BufferedReader`, instead of `DataInputStream`, since the server will send you the list of directories line by line.

Section 4.5

Use your client to connect to an actual ftp server. There are plenty of free ftp servers that do not need registration or username/passwords. A sample one: `loki.cciw.ca` (requires you to implement the PASS command for password)

NOTE: Don't forget to close the sockets at the end of the application.

Goodluck. ☺