Assignment 1

Computer Networks (CS 456)

Spring/Summer 2016

Introductory Socket Programming

Due Date: on Friday, June 24th, 2016, at midnight (11:59 PM).

Work on this assignment is to be completed individually

Assignment Objective

The goal of this assignment is to gain experience with both TCP and UDP socket programming in a client-server environment (Figure 1). You will use Java/Python/C/C++ to write a set of sockets-based file copying programs. You'll write get and put clients and a server. Your program has to follow the protocol described in this document.

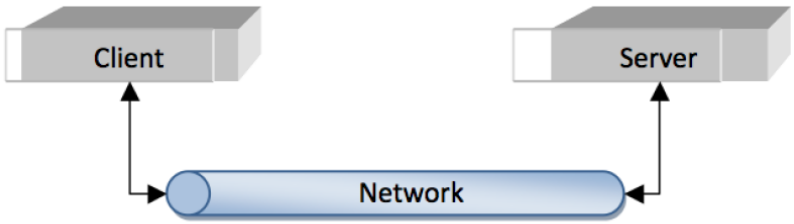

Figure 1

Assignment Specifications

1. Summary

In this assignment, the client will send requests to the server to upload (put) and download (get) files over the network using sockets. This assignment uses a two stage communication process. In the negotiation stage, the client asks the server to send it a list of files that it has. Later in the transaction stage, the client connects to the server through a selected port for actual data transfer. Note: The assignment description in the following assumes an implementation in C.

2. Languages

You may do this assignment in either C, C++, Java or Python.

3. Signaling

The signaling in this project is done in two stages as described below.

Stage 1. Negotiation using UDP sockets: In this stage, the client creates a UDP socket to the server using (serverAddr) as the server address and (uPort) as the negotiation port on the server (where the server is

listening). The client sends a request (list) to get the list of files that the server already has. Once the server receives this request it answers back with the list of files that reside in the server folder (home_folder).

Stage 2. Transaction using TCP sockets: In this stage, the client creates a TCP connection with the server using (serverAddr) as the server address and (tPort) as the negotiation port.

[optional: extra credit]: Instead of having uPort and tPort hard coded in your server and client programs add port negotiation to the negotiation step (i.e. the client can first ask the server for a random port number (rPort) then the client can create the TCP connection using (rPort) as the port number).

4. SERVER PROGRAM

After accepting a connection, the server will receive a line of text that contains a request to either get a file or put a file. This line of text that is received by the server is one of two possible messages:

put *filename*

get *filename*

The put and get commands are followed by whitespace — one or more space (' ') or tab ('\t') characters — and then followed by a filename. The filename is ASCII text that identifies the file that will be either sent or requested.

Handling put requests

If the first line is a put request, all bytes after the newline comprise the data of the file that will be stored on the server.

The request will either be accepted or rejected. An acceptance message is a newline-terminated string containing the string "ok". A rejection message is a line of ASCII text that contains the string error followed by a space and then a string containing the message.

ok

error *error_message_text*

Now the server tries to create the local file with the given name. If that fails, it sends back a rejection message to the client. If the local file was successfully created, the server is ready to get the contents. It sends back a success message to the client. The server now loops, reading chunks of data from the client and writing it to the local file until there's no more data to receive.

Handling get requests

The server now attempts to open the local file that is being requested. If the requested file is not in the list the server sent to the client in the negotiation step the server should send a rejection message to the client. If the file exists but open fails, it sends another rejection message to the client. If the local file was successfully opened for reading, the server is ready to send the file. It sends back a success message to the client. The server now loops, reading chunks of data from the file and sending it to the client until

there's no more data to read. The server closes the file and the socket and returns to accept the next connection. For simplicity, we assume, there will be only one client in the system at a time. So, the server does not need to handle simultaneous client connections

[Optional: extra credit] The server will handle more than one client at a time Note: See threading and select command instead of accept

5. CLIENT PROGRAMS

You should implement 2 client programs, named put and get.

The put client

The put client is named put and accepts the remote and local file names as command-line parameters. It reads the contents of the file and sends them to the server where the server will create a file with the same name and contents. If the names are missing, it should print an error and exit.

put local_filename remote_filename

The client opens the specified file (exiting with an error message if the open fails).

The client should follow the protocol dictated by the server. It sends the request: a line containing put followed by the file name.

The program now reads a response. As described earlier, the response is a single newline-terminated string containing either ok or an error message. If the response is an error message, print the message and exit.

Now, the client reads data from the opened file and sends it to the server block by block (block size can be 256B) until it reaches the end of the file. When it reaches the end of file, it closes the connection on the socket.

Close the local file, close the socket, and exit the program.

The get client

The get client is named get and accepts local and remote file names as command-line parameters. It sends the remote file name to the server, which will then send the contents back to the client. The client writes those contents to a local file. If the names are missing, it should print an error and exit.

get remote_filename local_filename

The client should follow the protocol dictated by the server. It sends a line containing get request followed by the file name.

The program now reads a response. As described earlier, the response is a single newline-terminated string containing either ok or an error message. If the response is an error message, print the message and exit.

Now, the client reads data from the opened file and sends it to the server block by block (block size can be 256B) until it reaches the end of the file, closing the connection when the end of file is reached. If there's a mismatch between the number of bytes actually received versus the number of bytes expected, print an error message to that effect.

Close the local file, close the socket, and exit the program.

HINTS

- You can use the sample codes of TCP/UDP socket programming in C from lecture 6 slides.
- Below are some points to remember while coding/debugging to avoid trivial problems.
    - Use port id greater than 1024, since ports 0-1023 are already reserved for different purposes (e.g., HTTP @ 80, SMTP @ 25)
    - If there are problems establishing connections, check whether any of the computers running the server and the client is behind a firewall or not. If yes, allow your programs to communicate by configuring your firewall software.
    - Make sure that the server is running before you run the client.
    - Also remember to print the where the server will be listening and make sure that the client is trying to connect to that same port for negotiation.
    - If both the server and the client are running in the same system, 127.0.0.1 (i.e., localhost) can be used as the destination host address.
    - You can use help on Java or C/C++ or Python network programming from any book or from the Internet, **if you properly refer to the source in your programs**. **But remember, you cannot share your program or work with any other student**.

PROCEDURES

1. DUE DATE

The assignment is due on Friday, June 24th, 2016, at midnight (11:59 PM).

2. HAND IN INSTRUCTIONS

Submit all your files in a single compressed file (.zip, .tar etc.) using LEARN in dedicated Dropbox. You must hand in the following files / documents:

- Source code files.
- Makefile: your code must compile and link cleanly by typing “make” or “gmake”.
- README file: this file must contain instructions on how to run your program, which undergrad machines your program was built and tested on, and what version of make and compilers you are using.
- server.sh and client.sh scripts that calls your client and server programs. Your implementation will be tested on the machines available in the undergrad environment.

3. DOCUMENTATION

Since there is no external documentation required for this assignment, you are expected to have a reasonable amount of internal code documentation (to help the markers read your code). You will lose marks if your code is unreadable, sloppy, or not efficient.

4. EVALUATION

Work on this assignment is to be completed individually.

ADDITIONAL NOTES

1. You have to ensure that both (uPort) and (tPort) are available. Just selecting a random port does not ensure that the port is not being used by another program.
2. All codes must be tested in the linux.student.cs environment prior to submission.
   2.1. Run client and server in two different student.cs machines
   2.2. Run both client and server in a single student.cs machine
3. Make sure that no additional (manual) input is required to run any of the server or client.