Assignment 1

Computer Networks (CS 456/656)

Winter 2023

Introductory Socket Programming

Due Date: Monday, February 27, 2023, before midnight (11:59 PM)

Work on this assignment is to be completed individually

1 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 `Python or any other programming language` to design and implement a client program (`client`) and a server program (`server`) to communicate between themselves.

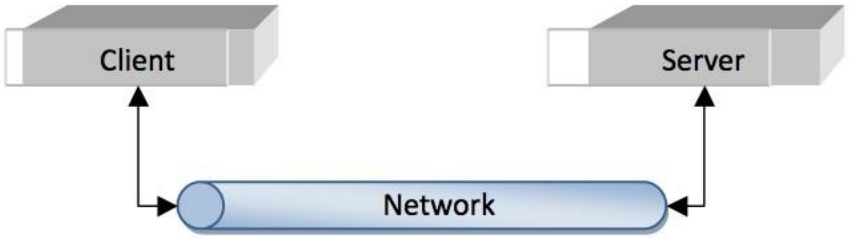

FIGURE 1

2 Assignment Specifications

2.1 Summary

In this assignment, the client will download a file from the server over the network using sockets.

This assignment uses a two-stage communication process. In the negotiation stage, the client and the server negotiate through a fixed negotiation port (`<n_port>`) of the server, a random port (`<r_port>`) for later use. Later in the transaction stage, the client and server connect through the selected random port (`<r_port>`) for actual file transfer.

Requests and responses in this assignment will mimic the File Transfer Protocol (FTP). FTP runs in one of two modes – Active mode or Passive mode, which dictate the negotiation and transaction stage procedures.

In Active mode, the client issues a `PORT` command to the server indicating a client-side port (`<r_port>`) that the server should connect to in the transaction stage. The server then connects to the client via this port and transfers the content of the file `<file_to_send>` to the client.

In Passive mode, the client issues a `PASV` command to the server indicating that the server should provide a server-side port (`<r_port>`) for the client to connect to in the transaction stage. The client then connects to the server via this port, and the server transfers the content of the file `<file_to_send>` to the client.

2.2 Protocol

The protocol is shown in Figure 2 (for Active mode) and Figure 3 (for Passive mode).

2.2.1 Active Mode

Stage 1. Negotiation using UDP sockets: In this stage, the client sends to the server (`<server_address>` and `<n_port>` as server address and port number respectively) a `PORT` request along with a port number `<r_port>`, and a request code `<req_code>`. `<r_port>` is the port number attached to a TCP socket running on the client, on which the client is expecting the file. `<req_code>` is a secret integer known to the client and the sever (e.g., `13`). If the client fails to send the expected `<req_code>`, the server responds with a `0` (i.e., a negative acknowledgement) and continues listening on `<n_port>` for subsequent client requests. Once the server verifies the `<req_code>`, it registers `<r_port>` internally and sends a `1` (i.e., a positive acknowledgement) back to the client. The two parties then transition to the Transaction stage.

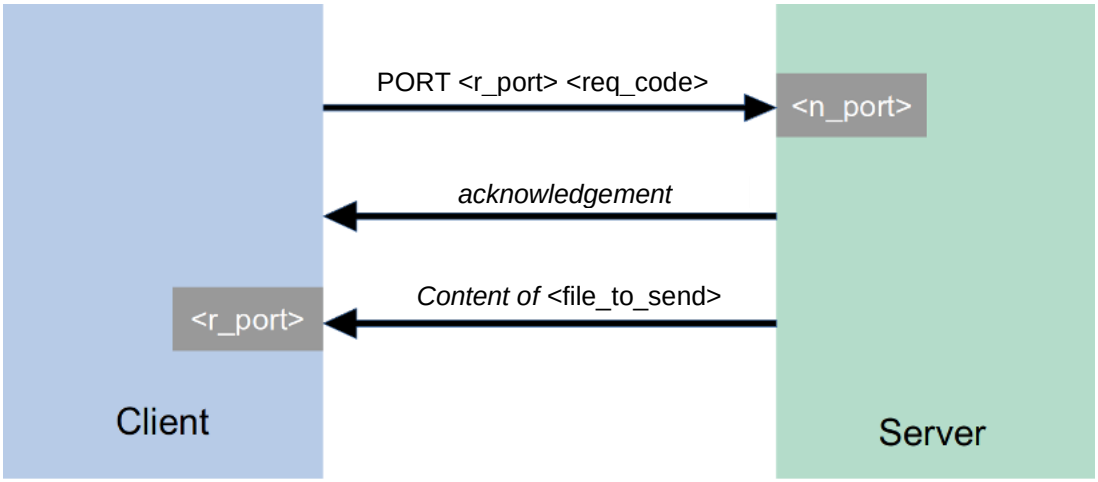

FIGURE 2

Stage 2. Transaction using TCP sockets: In this stage, the server initiates a TCP connection with the client on the client's port `<r_port>`, sends the content of `<file_to_send>`, and then closes the connection. On the other side, the client receives the data, saves the file as `<file_received>` closes its sockets and exits. Note that the server should continue listening on its port `<n_port>` for subsequent client requests. 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.

2.2.2 Passive Mode

Stage 1. Negotiation using UDP sockets: The client sends to the server (`<server_address>` and `<n_port>` as server address and port number respectively) a `PASV` request with the request code `<req_code>` (e.g., `13`). If the client fails to send the expected `<req_code>`, the server responds with a `0` (i.e., negative acknowledgment) and continues listening on its port `<n_port>` for subsequent client requests. Once the server verifies the `<req_code>`, it creates a server TCP

socket and replies with the socket's port number `<r_port>` where it will be waiting for the client to connect. The two parties then transition to the Transaction stage.

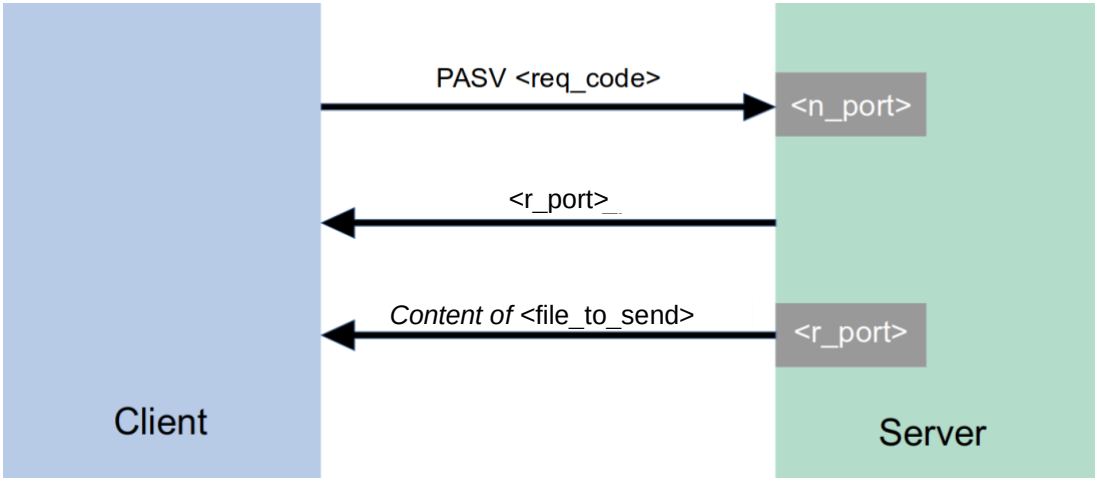

FIGURE 3

Stage 2. Transaction using TCP sockets: In this stage, the client initiates a TCP connection to the server on the server's port `<r_port>`. The server accepts the connection, transfers the file `<file_to_send>` to the client and closes the TCP connection. Once received, the client saves the file as `<file_received>`, closes all the sockets and exits. Note that the server should continue listening on its port `<n_port>` for subsequent client requests. 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.

2.3 Server Program (server)

You will implement a server program, named `server`. The server will take `<req_code>` and `<file_to_send>` as command line parameters. The file `<file_to_send>` will be in the same directory as the server program. When you run the server and the server creates a UDP socket for the negotiation stage, it must print out the port number `<n_port>` of the socket, in the following format as the first line in the stdout:

`SERVER_PORT=<n_port>`

For example, if the negotiation port of the server is `52500`, then the server should print:

`SERVER_PORT=52500`

2.4 Client Program (client)

You should implement a client program, named `client`. It will take five command line inputs: `<server_address>`, `<n_port>`, `<mode>`, `<req_code>`, and `<file_received>` in the given order.

`<mode>` can either be `'P'` or `'A'` indicating whether the client should run in Passive or Active mode.

<file_received> is the name under which the client will save the file downloaded from the server. This file should be saved in the same directory as the client program.

2.5 Example Execution

Two shell scripts named `server.sh` and `client.sh` are provided. Modify them according to your choice of programming language. You should execute these shell scripts which will then call your client and server programs.

- Run server: `./server.sh <req_code> 'sent.txt'`
- Run client: `./client.sh <server address> <n_port> <mode> <req_code> 'received.txt'`
- After execution, client saves the recieved content into `'received.txt'` locally

2.6 File Size

For simplicity, <file_to_send> will be at most 1024 bytes long, and so the server should only require one transaction to transfer the file to the client.

3 Hints

Below are some points to remember while coding/debugging to avoid trivial problems.

- You can use and adapt the sample codes of TCP/UDP socket programming in Python slides (last few slides Chapter 2).
- 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 <n_port> where the server will be listening and make sure that the client is trying to communicate with the server on that same port for negotiation.
- If both the server and the client are running in the same system, 127.0.0.1 or localhost can be used as the destination host address.
- You can use help on 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.

4 Procedures

4.1 Due Date

The assignment is due on Monday, February 27, 2023, before midnight (11:59 PM).

4.2 Hand in Instructions

Submit all your files in a single compressed file (.zip, .tar etc.) using LEARN in dedicated Dropbox. The filename should include your username and/or student ID.
You must hand in the following files / documents:

- Source code files.
- Makefile: your code must compile and link cleanly by typing "make" or "gmake" (when applicable).
- 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.
- Modified `server.sh` and `client.sh` scripts.

Your implementation will be tested on the machines available in the undergrad environment `linux.student.cs` which includes the following hosts:

Hostname	***CPU Type***	***# of CPUs***	***Cores /CPU***	***Threads /Core***	***RAM (GB)***	***Make***	***Model***
ubuntu2004-002.student.cs.uwaterloo.ca	AMD EPYC 7532	2	32	2	256	Dell Inc.	PowerEdge R7525
ubuntu2004-004.student.cs.uwaterloo.ca	AMD EPYC 7532	2	32	2	256	Dell Inc.	PowerEdge R7525
ubuntu2004-008.student.cs.uwaterloo.ca	Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz	2	16	2	256	Dell Inc.	PowerEdge R730
ubuntu2004-010.student.cs.uwaterloo.ca	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz	2	20	2	384	Supermicro	SYS-6029P-WTRT
ubuntu2004-014.student.cs.uwaterloo.ca	Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz	2	14	2	256	Dell Inc.	PowerEdge R730

4.3 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 or sloppy.

4.4 Evaluation

Work on this assignment is to be completed individually.

5 Additional Notes:

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