Assignment 1

Computer Networks (CS 456/656)

Winter 2024

Introductory Socket Programming

Due Date: Monday, February 14, 2024, 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.

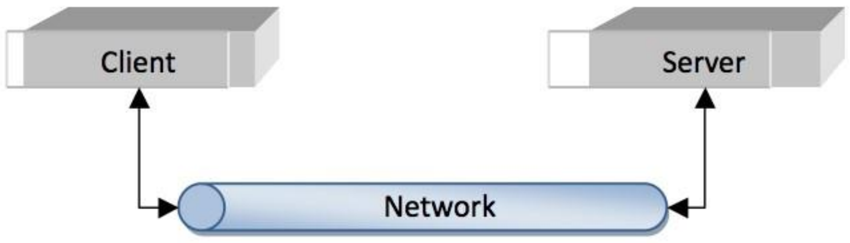

FIGURE 1

2 Assignment Specifications

2.1 Summary

In this assignment, the client will download a file from/upload a file to 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 the server connect through the selected random port (`<r_port>`) for actual file transfer.

Requests and responses in this assignment will mimic the negotiation and transaction stage procedures of File Transfer Protocol (FTP). This protocol will have a GET command for downloading files and a PUT command for uploading files.

In GET command, the client issues a `GET` command to the server indicating a file download and provides 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 `<filename>` to the client.

In PUT command, the client issues a `PUT` command to the server indicating a file upload. Then the server should provide a server-side port (`<r_port>`), which client then connects to and transfers the content of the file `<filename>` to the server.

2.2 Protocol

The protocol is shown in Figure 2 (for GET command) and Figure 3 (for PUT command).

2.2.1 GET Command

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 `GET` request along with a port number `<r_port>`, and the name of the file `<filename>` (e.g., `hello.txt`). `<r_port>` is the port number attached to a TCP socket running on the client, on which the client is expecting the file. `<filename>` is the name of the file the client is requesting from the server. If the server fails to find the file of `<filename>`, the server responds with a `”404 Not Found”` (i.e., a negative acknowledgement) and continues listening on `<n_port>` for subsequent client requests. The client terminates upon receiving the `”404 Not Found”` response. Otherwise, it sends a `“200 OK”` (i.e., a positive acknowledgement) back to the client, and registers `<r_port>` internally. The two parties then transition to the Transaction stage.

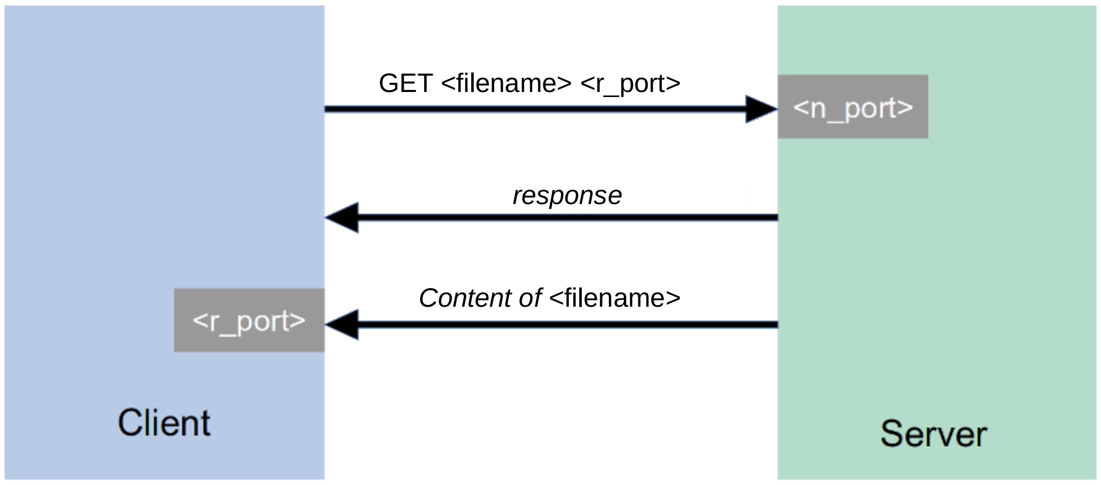

FIGURE 2

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 `<filename>`, and then closes the connection. On the other side, the client receives the data, saves the file as `<filename>` 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 PUT Command

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 `PUT` request with the request code

`<filename>` (e.g., `hello.txt`). The server 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.

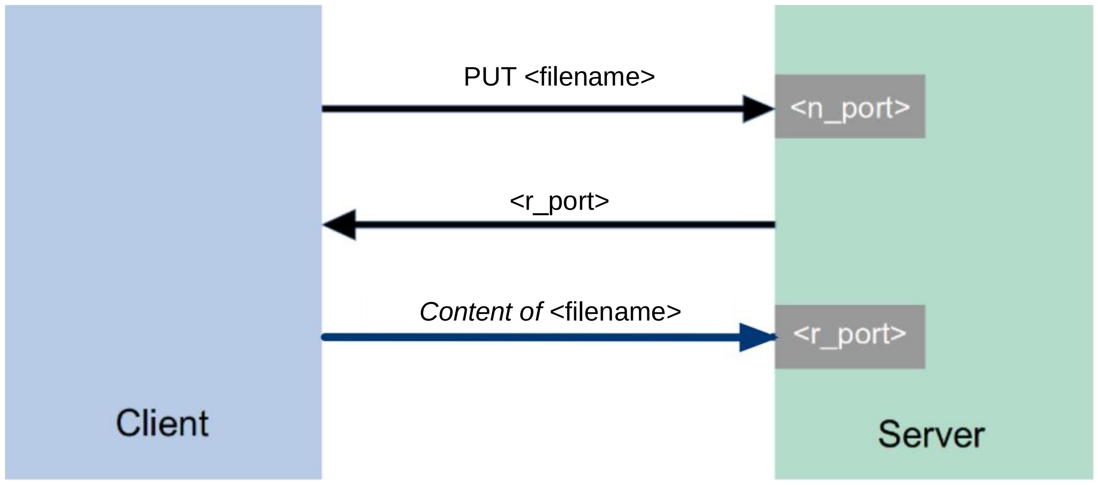

FIGURE 3

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, receives the file `<filename>` from the client and closes the TCP connection. Once sent, the client closes the connection and exits. Afterwards, the server saves the file as `<filename>` (overwrite if exists) and closes the `<r_port>` socket. 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 folder `<storage_directory>` as command line parameters. The files the server sends and receives are and will be stored in the `<storage_directory>`. 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 four command line inputs: `<server_address>`, `<n_port>`, `<command>`, and `<filename>` in the given order.

`<command>` can either be `'GET'` or `'PUT'` indicating whether the client should send GET or PUT command.

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 './storage'`
- Run client: `./client.sh <server address> <n_port> <command> 'hello.txt'`

2.6 File Size

There is no assumption on the size of the file being sent/received. You can assume the file will not be larger than the storage size or other hardware constraint.

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.
- The file being sent/received can be transferred in chunks of 1024 bytes.

4 Procedures

4.1 Due Date

The assignment is due on Monday, February 14, 2024, 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***
ubuntu2204-002.student.cs.uwaterloo.ca	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz	2	20	2	384	Supermicro	SYS-1029U-E1CR25M
ubuntu2204-004.student.cs.uwaterloo.ca	AMD EPYC 7532 @ 2.4GHz	2	32	2	512	Dell Inc.	PowerEdge R7525
ubuntu2204-006.student.cs.uwaterloo.ca	AMD EPYC 7532 @ 2.4GHz	2	32	2	1024	Dell Inc.	PowerEdge R7525
ubuntu2204-010.student.cs.uwaterloo.ca	AMD EPYC 9654	2	96	2	1536	Supermicro	AS-2125HS-TNR
ubuntu2204-012.student.cs.uwaterloo.ca	AMD EPYC 9654	2	96	2	1536	Supermicro	AS-2125HS-TNR
ubuntu2204-014.student.cs.uwaterloo.ca	AMD EPYC 9654	2	96	2	1536	Supermicro	AS-2125HS-TNR

You can see the status of each of these machines here.

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.