Monday, 7 September 2020

EE 324, Assignment #1

Part 1: A simple client/server program

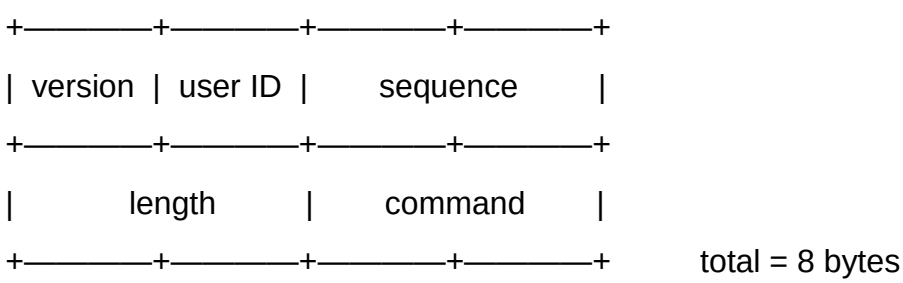
1. Overview

For your first assignment, you need to design and implement a simple client and a server application. They are communicating with a custom protocol based on TCP/IP. The server waits a connection trial from the client with port 12345, and the client will ask the server to conduct several jobs.

- NOTE: You don't need to employ a **non-blocking I/O**, which means that there is only one client for the server.

2. Protocol Specification

2.1 Header format



- version: static value, it should be 0x04
- user ID: static value, it should be 0x08
- sequence: initially **randomly** generated by a client, and the client should increment this value **by one** for the following packets in the same session
- length: total packet length (including the header)
- command: client command for the server job
  - 0x0001 - hello (client hello)
  - 0x0002 - hello (server hello)
  - 0x0003 - data delivery (server should receive the whole data from the client)
  - 0x0004 - data store (save the received data to a file, name should be explicitly specified)
  - 0x0005 - error (if there is any error)

0x0001, 0x0003, 0x0004 commands should be sent by the client

0x0002 command should be sent by the server

0x0005 command can be sent by any peer

- **NOTE: The field "sequence", "length", and "command" should be transferred as network byte order (i.e. big endian).**

3. Scenario

- Run your server (i.e. **"./server 12345"**).
- Run your client with the server's ip address, port, and file name (i.e. **"./client 127.0.0.1 12345 test.txt"**).
- After a TCP connection is set up between the client and the server, the client should send "client hello (0x0001)", and then the server should reply with "server hello (0x0002)".
- The client sends "data delivery (0x0003)" with the contents of the file.
- The server temporarily stores the contents into the memory (buffer).
- If the file transfer is complete, the client sends "data store (0x0004)" with the file name.
- Then, the server stores the contents of memory as a file.

4. More Requirements

- A client and a server should be different processes.
- You need to use "AF_INET" for connection (but you can test your program in a single machine).
- Even if the server's ip address comes in as hostname, it should be handled well.
- A client program should read a file and send to a server.
- A server should save a file with the received contents from the client.
- You don't need to consider a binary file transfer since TAs will test your program using **"*.txt"** files.
- If a data delivery fails, "error (0x0005)" should be delivered.
- Your program should be named as **"client"** for your client and **"server"** for your server.
- When a server replies with "server hello", the sequence of the packet should be the sequence of the "client hello" packet **plus one.**
- Again, the client should increment this value **by one** for the following packets in the same session.
- Since the **"length"** value of the header is 2 bytes, a large file should be split into multiple packets and sent.

5. Test Case

1) Does your program clearly send/receive packets with the specified packet header? - 25 points
   I. connection set up phase will be checked (i.e. client hello -> server hello) - 10 points
   II. all header values will be carefully checked - 15 points
      I. version & user ID – 2.5 points
      II. sequence – 2.5 points
      III. length – 2.5 points
      IV. command - 5 points
      V. network byte order – 2.5 points
2) Does your client properly transfer a file (up to **4MB**) to the server? – 12.5 points
3) Does your server properly store a file on the server side? (with the client command = 0x0004) – 12.5 points

Part 2: Process-based Concurrent client/server program

1. Overview

In the previous part, you developed the simple client/server program that conduct a simple file transfer. But, in practice, servers should process multiple requests sent from many different clients (e.g., Web server). For the part 2, you need to design and implement concurrent client/server programs that enable multiple simultaneous requests **by extending the previous work**. There are *two* required programs; *(1) a process-based server, (2) a client* that sends multiple requests to a server.

- **NOTE: You don't need to make separated source file for process-based server. You can just write the process-based server by extending the previous server code (part 1).**

2. Process-based Server

- It automatically spawns a child process when a new TCP connection is requested.
- Therefore, you should use the **"fork()"** function to generate child process in your code.
- The spawned child process should follow *the scenario of the part 1* (i.e., handshake and file transfer with the defined protocol).
- But, the parent process should be still listening for other new connections after spawning the child process like typical Linux daemons.
- **NOTE: You should reap all zombie processes by using "waitpid()"**

3. Client

- In this part, you need to slightly modify the previous client program.
- To generate simultaneous requests, we will use **"fork()"** function as we used to write the process-based server program.
- The binary must be named as "`client_multi`" to make a difference from the single client program.
- And, the command line is changed as **"`./client_multi [server_ip] [server_port] [number_of_requests]`. (e.g., `./client 127.0.0.1 12345 20`).** Note that client_multi does not accept any input file as in client (part 1).
- After execution, the main process generates processes as the number of request from the argument (put the sleep function for 100 milliseconds for each cycle).
- Then, each process creates a new socket, connects the server.
- Next, using our defined protocol, it handshakes with the target server and sends 0x0003 with the string in a following form: "`I am a process [process id]`".
  - e.g. if the process ID is 325, the string should be "I am a process 325"
  - "getpid()" returns the process ID (PID) of the calling process.
  - **Note that client_multi sends the string above to the server, not an input file.**
- The client_multi should send 0x0004 with the file name " [process_id].txt" to the server.
  - e.g. if the process ID is 325, the filename should be "325.txt"
- You can check if the [process_id] is well delivered using "`printf()`" function in the client program.

4. Test Case

1) Process-based Server - 45 points
   I. Does the server spawn a child process using fork() whenever it receives a new connection? - 5 points
   II. Does the server properly store files on the server side? (without losing any files) – 10 points
   III. Can the server process 20 (approximate) simultaneous connections from the client? - 20 points
   IV. Can the server process more than 20 simultaneous connections from the client? – 10 points
2) Client - 5 points
   I. Does the client spawn a child process using fork() for a new connection? - 5 points

6. Instruction for Submission

- **You should upload your code on GitLab.** (You don't need to submit any files on KLMS.)
- TAs will download your project when the due is over.
- **Fork the GitLab repository to your group and clone the forked repository to your local machine before you start.**
  - You need to make a group named after your student id (e.g.20201234) before forking. Refer to the submission guideline slide.
  - Your forked repository should be in form of (if your ID is 20201234): `https://gitlab.ee324.kaist.ac.kr/20201234/assignment-1`
  - Make sure you are not directly cloning from the assignment repository.
- In the GitLab, there are five files. You may add more files as you wish. However, the total size of the submission must not exceed 10MB. Please do not add unnecessary files to your commit (e.g. log files).
  1. `README.md`
  2. `Makefile`
  3. `src/client.c`
  4. `src/server.c`
  5. `src/client_multi.c`
- Make sure that your `Makefile` correctly compiles your source code. The `Makefile` must reside in the root of the repository.
- In the README.md:
  - You should put your name, student ID.
  - Write your name and date on the ethics oath.
  - In addition, briefly describe how you designed and implemented your programs.
  - Describe whatever help (if any) you received from others while doing the assignment, and write the names of any individuals with whom you collaborated, as prescribed by the course Policy web page.
- Write the concise comments in the source code to understand your program.

! IMPORTANT 1: Please strictly follow the above structure and name policy; if not, TAs will not evaluate your program.

! IMPORTANT 2: Please make sure that there is no compile and execution error. TAs will not give you any score in case of the problems.

7. Test Environment

- Language: C or C++
- Test O/S: **Ubuntu 16.04 LTS (Xenial Xerus), Ubuntu 18.04 LTS (Bionic Beaver),**

20.04 LTS (focal fossa) 64bit

- NOTE: We will not consider your compilation and execution problems due to the different OS versions.

8. Due Date

- 23:59, Sep. 27, 2020 (Sunday)

- Please check if the code is properly uploaded on the GitLab. (The website often becomes unavailable, so please make sure that you submit your assignment early)

- 10% late penalty per day

- Hard deadline: **23:59, Oct. 01. 2020 (Thursday)**
  - We will not receive additional submissions after the hard deadline.
  - Exceptions: documented medical/personal emergency
- Please DO NOT e-mail Prof. Han or TAs to submit your assignments.

9. Plagiarism

- You can discuss with your colleagues, but you should turn in your own programs
  - ✓ Copy and Paste
    - Will run plagiarism detection on source code
    - “Copy and paste” codes will get severely penalized
    - If detected, 0 point for all assignments (both providers and consumers)
    - But you will have a chance to defend yourself

10. Questions?

- Please use Piazza for asking any questions.