CS 456/656 Fall 2017 - Assignment 3

Teaching Assistants:
Milad Ghaznavi (eghaznav@uwaterloo.ca**)**
Dhinakaran Vinayagamurthy (dhinakaran2705@gmail.com**)**
Phil Pawlega (filip.pawlega@uwaterloo.ca**)**
Office Hours: Monday 12:00 to 01:00pm, DC 3549

Due date: **Dec 1, 2017 11.59pm**

1 Summary

The goal of this assignment is a server that facilitates a file exchange between clients. You need to write two programs: a server and a client with the specifications given below. Communication between client and server uses TCP. You can implement your programs in any of the following programming languages: `C`, `C++`, `Java`, and `Python`. You can, and are very much encouraged to, use standard library container types for `string`, `list`, `set`, etc. However, do not use any libraries that substantially alter and/or enhance the basic socket interface. Further, if you choose a multi-threaded design, you must not use any automatic concurrent data structures from a library, such as the container data structures from `java.util.concurrent`, or libraries which implement higher level concurrency constructs such as "Monitors". If in doubt, consult with the instructor.

2 Details

2.1 Server

You must implement a server program that can handle an arbitrary number of concurrent connections and file exchanges, only limited by system configuration or memory. The server is started without any parameters and creates a TCP socket at an OS-assigned port. It must print out the assigned port number and store it in a local file `port`, which is used when starting clients. The server listens on its main socket and accepts client connections as they arrive. Clients perform an upload or download operation, or instruct the server to terminate.

Both upload and download operations specify a key that is used to match clients with each other, i.e., a client asking for downloading with a specific key receives the file that another client uploads using that key. Files are NOT stored at the server, but instead

clients wait for a match and then data is forwarded directly from the uploader to the downloader. The server must always match a pair of uploader and downloader with the same key. For simplicity, we assume that only a single downloader matches an uploader, and an uploader never starts before its matched downloader.

The server must support concurrent file exchanges; however, you can make the following simplifying assumptions when using an event-based, single-threaded design:

Assume file read/write system calls never block. Assume network write/send system calls never block. Assume that when the first byte from a new client arrives, the complete initial command (9 bytes, see below) can be received without blocking. In other words, the server only needs to multiplex potentially blocking system calls for accepting new connections and receiving upload data. When the server receives the termination command from a client, it must close all waiting connections from unmatched clients and not accept any further connections. It must, however, complete ongoing file exchanges and terminate only after all file exchanges are finished.

2.2 Communication

The data stream sent from the client to the server must adhere to the following format:

- `command`: 1 ASCII character: `G` (get = download), `P` (put = upload), or `F` (finish = termination)
- `key`: 8 ASCII characters (padded at the end by '\0'-characters, if necessary)

In case of an upload, the above 9-byte control information is immediately followed by the binary data stream of the file. In case of download, the server responds with the binary data stream of the file. When a client has completed a file upload, it closes the connection. Then the server closes the download connection to the other client.

2.3 Client Program

The client takes up to 6 parameters and can be invoked in 3 different ways:

1. terminate server: `client <host> <port> F`
2. download: `client <host> <port> G<key> <file name> <recv size>`
3. upload: `client <host> <port> P<key> <file name> <send size> <wait time>`

The client creates a TCP socket and connects to the server at `<host>` and `<port>`. It then transmits the command string given in the 3rd shell parameter to the server as described above, i.e., with padding. When transmitting an 'F' command, the client must still send an empty key, i.e., 8 '\0'-characters.

When requesting an upload or download, the client reads data from or stores data to, respectively, the file specified in the 4th parameter.

The 5th parameter specifies the size of the buffer that is transmitted during each individual `write/send` or `read/recv` system call during the file transfer - except for the final data chunk that might be smaller.

When uploading a file, the 6th parameter specifies a wait time in milliseconds between subsequent `write/send` system calls. Assume the wait time is always below 1 second, so that you can use `usleep` to implement the wait. This parameters allows for a simple form of rate control that is important to test the concurrency of the server.

2.4 Additional Comments/Hints

1. There are different ways to concurrently handle multiple connections. These include using I/O multiplexing system calls, such as `select()`, `poll()`, `epoll()`, or multi-threading. You can choose either of these methods.

2. **IMPORTANT**: Do not resort to busy-looping, i.e., repeatedly checking for new arrivals without blocking. Use any of the above methods to block while there is nothing to do.

3. **IMPORTANT**: Follow the hand-in instructions below carefully - or risk loosing a substantial number of marks!

3 Procedures

3.1 What to hand in

Hand in source code files, including appropriate comments. All files must be stored in the same directory. There should be no directory hierarchy or package definitions. Your assignment must come with a Makefile. The targets in the Makefile must include:

- `'clean'` to remove all object files, addressing files, as well as all log and temporary files; and
- `'all'` to build all object and executable files

After executing `'make all'`, the following executables (or start scripts) must exist in the current directory. The server program must be started without any arguments:

- `server`

The client program must accept up to 6 parameters:

- `client <host> <port> <op-string> <file name> <send|recv size> <wait time>`

'`README`' file (in plain ASCII text): Report which machines your program was built and tested on. Also, document which parts of the assignment have or have not been completed. Describe the basic design ideas/justifications for your program(s), especially the server. This file does not need to be long, but should succinctly provide all the requested information.

3.2 Evaluation

The assignment must be completed individually. Your program must work in the `linux.student.cs` environment. Your program should not silently crash under any circumstances. At the very least, all fatal errors should lead to an error message indicating the location in the source code before termination. Marks will be assigned as follows:

- Functionality and Correctness: 80%
- Code Quality and Design Documentation: 20%

3.3 Submission Instructions

After you have completed testing your code, hand it in using the dropbox feature of the Learn environment. Combine all files into a `zip/tar/gzip/bzip2` archive with any of the following corresponding names: `a3.{zip,tgz,tbz}`. Make sure to execute '`make clean`' before submitting, and do not include temporary or object files!