Project 1: Client Server File Transfer

Overview

In this project, you will need to implement a simple client-server application that transfers a file over a TCP connection.

All implementations should be written in C++ using BSD sockets. **No high-level network-layer abstractions (like Boost.Asio or similar) are allowed in this project.** You are allowed to use some high-level abstractions, including C++11 extensions, for parts that are not directly related to networking, such as string parsing, multi-threading.

The objective of this project is to learn basic operations of BSD sockets, understand implications of using the API, as well as to discover common pitfalls when working with network operations.

You are required to use `git` to track the progress of your work. **The project can receive a full grade only if the submission includes git history no shorter than 3 commits.**

You are encouraged to host your code in private repositories on on GitHub, GitLab, Bitbucket, etc. At the same time, you are PROHIBITED to make your code for the class project public during the class or any time after the class. If you do so, you will be violating academic honesty policy that you have signed, as well as the student code of conduct and be subject to serious sanctions.

Task Description

The project contains two parts: a server and a client.

- The server listens for TCP connections and saves all the received data from the client in a file.

- The client connects to the server and as soon as connection established, sends the content of a file to the server.

Server Application Specification

The server application MUST be compiled into `server` binary, accepting two command-line arguments:

```
$ server <PORT> <FILE-DIR>
```

- `<PORT>`: port number on which server will listen on connections. The server must accept connections coming from any interface.
- `<FILE-DIR>`: directory name where to save the received files.

For example, the command below should start the server listening on port `5000` and saving received files in the directory `/save`.

```
$ ./server 5000 /save
```

Requirements:

- The server must open a listening socket on the specified port number
- The server should gracefully process incorrect port number and exit with a non-zero error code (you can assume that the folder is always correct). In addition to exit, the server must print out on standard error (`std::cerr`) an error message that starts with `ERROR:` string.
- The server should exit with code zero when receiving `SIGQUIT`/`SIGTERM` signal
- The server should be able to accept and process multiple connection from clients at the same time
- The server must count all established connections (1 for the first connect, 2 for the second, etc.). The received file over the connection must be saved to `<FILE-DIR>/<CONNECTION-ID>.file` file (e.g., `/save/1.file`, `/save/2.file`, etc.). If the client doesn't send any data during gracefully terminated TCP connection, the server should create an empty file with the name that corresponds to the connection number.
- The server must assume error if no data received from the client for over `15 seconds`. It should abort the connection and write a single `ERROR` string (without end-of-line/carriage-return symbol) into the corresponding file. Note that any partial input must be discarded.

- The server should be able to accept and save files up to `100 MiB`

Client Application Specification

The client application MUST be compiled into `client` binary, accepting three command-line arguments:

```
$ ./client <HOSTNAME-OR-IP> <PORT> <FILENAME>
```

- `<HOSTNAME-OR-IP>`: hostname or IP address of the server to connect
- `<PORT>`: port number of the server to connect
- `<FILENAME>`: name of the file to transfer to the server after the connection is established.

For example, the command below should result in connection to a server on the same machine listening on port 5000 and transfer content of `file.txt`:

```
$ ./client localhost 5000 file.txt
```

Requirements:

- The client must be able to connect to the specified server and port, transfer the specified file, and gracefully terminate the connection.
- The client should gracefully process incorrect hostname and port number and exist with a non-zero exit code (you can assume that the specified file is always correct). In addition to exit, the client must print out on standard error (`std::cerr`) an error message that starts with `ERROR:` string.
- Client application should exit with code zero after successful transfer of the file to server. It should support transfer of files that are up to 100 MiB file.
- Client should handle connection and transmission errors. The reaction to network or server errors should be **no longer that 15 seconds**:
    - Timeout to connect to server should be no longer than `15 seconds`
    - Timeout for not being able to send more data to server (not being able to write to send buffer) should be no longer than `15 seconds`.
- Whenever timeout occurs, the client should abort the connection, print an error string starting with `ERROR:` to standard error (`std::cerr`), and exit with non-zero code.

A Few Hints

General hints:

- If you are running the client and the server on the same machine, you can use "localhost" (without quotes) or "127.0.0.1" (without quotes) as the name of the server.
- You should NOT use port numbers in the range of 0-1023 (these are reserved ports). Test your client/server code by running as non-privileged user. This will allow you to capture reserved port restrictions from the kernel.

Here are some hints of using multi-thread techniques to implement the server.

- For the server, you may have the main thread listening (and accepting) incoming **connection requests**.
    - Any special socket API you need here?
    - How to keep the listening socket receiving new requests?
- Once you accept a new connection, create a child thread for the new connection.
    - Is the new connection using the same socket as the one used by the main thread?

Here are some sample codes:

- A simple server that echoes back anything sent by the client: server.cpp, client.cpp
- A simple multi-thread countdown: multi-thread.cpp

Other resources

- Guide to Network Programming Using Sockets

Environment Setup

The best way to guarantee full credit for the project is to do project development using a Ubuntu 16.04-based virtual machine.

You can easily create an image in your favourite virtualization engine (VirtualBox, VMware) using the Vagrant platform and steps outlined below.

Set Up Vagrant and Create VM Instance

Note that all example commands are executed on the host machine (your laptop), e.g., in `Terminal.app` (or `iTerm2.app`) on macOS, `cmd` in Windows, and `console` or `xterm` on Linux. After the last step (`vagrant ssh`) you will get inside the virtual machine and can compile your code there.

- Download and install your favourite virtualization engine, e.g., VirtualBox
- Download and install Vagrant tools for your platform (these two steps should have been finished as part of project 0)
- Set up project and VM instance
    - Project 1 skeleton is hosted in public repo `https://github.com/CS118W19/winter19-project1.git`. Similar to project 0, you need to duplicate it to your personal private repo (You should NOT fork it since github only support public fork now). Note github provides unlimited private repo for student accounts.
    - Clone project template

      ```
      git clone [private git repo path] ~/cs118-proj0
      ```

    - For example (replace the public repo path below with your private one):

```
git clone https://github.com/CS118W19/winter19-project1.git ~/cs118-proj0
```

    - Initialize VM

      ```
      vagrant up
      ```

      Do not start VM instance manually from VirtualBox GUI, otherwise you may have various problems (connection error, connection timeout, missing packages, etc.)

    - To establish an SSH session to the created VM, run

      ```
      vagrant ssh
      ```

- If you are using Putty on Windows platform, `vagrant ssh` will return information regarding the IP address and the port to connect to your virtual machine.
- Work on your project
- All files in `~/cs118-proj1` folder on the host machine will be automatically synchronized with `/vagrant` folder on the virtual machine. For example, to compile your code, you can run the following commands:

```
vagrant ssh
cd /vagrant
make
```

Notes

- If you want to open another SSH session, just open another terminal and run `vagrant ssh` (or create a new Putty session).
- If you are using Windows, read this article to help yourself set up the environment.
- The code base contains the basic `Makefile` and two empty files `server.cpp` and `client.cpp`.

```
$ vagrant ssh
vagrant@vagrant:~$ cd /vagrant
vagrant@vagrant:/vagrant$ ls -a
. .. client client.cpp .git .gitignore Makefile README.md server
server.cpp .vagrant Vagrantfile
```

- You are now free to add more files and modify the Makefile to make the `server` and `client` full-fledged implementation.

Submission Requirements

To submit your project, you need to prepare:

1. A `README.md` file placed in your code that includes:
    - Your name, UCLA ID

    - The high level design of your server and client
    - The problems you ran into and how you solved the problems
    - List of any additional libraries used
    - Acknowledgement of any online tutorials or code example (except class website) you have been using.

      If you need additional dependencies for your project, you must update Vagrant file.

2. All your source code, `Makefile`, `README.md`, `Vagrantfile`, and `.git` folder with your git repository history as a `.tar.gz` archive (and any files from extra credit part).

   To create the submission, **use the provided Makefile** in the skeleton project. Just update `Makefile` to include your UCLA ID and then just type

   ```
   make tarball
   ```

   Then submit the resulting archive to CCLE.

Before submission, please make sure:

1. Your code compiles
2. Client and server conforms to the specification
3. `.tar.gz` archive does not contain temporary or other unnecessary files. We will automatically deduct points otherwise.

> Submissions that do not follow these requirements will not get any credit.

Grading

Your code will be first checked by a software plagiarism detection tool. If we find any plagiarism, you will not get any credit.

Your code will then be automatically tested in some testing scenarios.

We may test your server against a “standard” implementation of the client, your client against a “standard” server, as well as your client against your server. Projects receive full credit if only all these checks are passed.

Grading Criteria

1. (2.5 pts) At least 3 git commits
2. (2.5 pts) Client handles incorrect hostname/port
3. (2.5 pts) Server handles incorrect port
4. (2.5 pts) Server handles `SIGTERM` / `SIGQUIT` signals
5. (5 pts) Client connects and starts transmitting a file
6. (5 pts) Server accepts a connection and start saving a file
7. (5 pts) Client able to successfully transmit a small file (500 bytes)
8. (5 pts) Client able to successfully transmit a medium size file (1 MiB)
    - sending in large chunks without delays
9. (5 pts) Client able to successfully transmit a large size file (100 MiB)
    - sending in large chunks without delays
10. (5 pts) Server able to receive a small file (500 bytes) and save it in `1.file`
11. (15 pts) Server able to receive a medium file (1 MiB bytes) and save it in `1.file`
    - (5 pts) receiving file sent in large chunks without delays
    - (10 pts) receiving file sent in small chunks with delays
12. (15 pts) Server able to receive a large file (100 MiB bytes) and save it in `1.file`
    - (5 pts) receiving file sent in large chunks without delays
    - (10 pts) receiving file sent in small chunks with delays
13. (10 pts) Server can properly receive 10 small files (sent without delays) in `1.file`, `2.file`, ... `10.file`
    - a single client connects sequentially
    - 10 clients connect simultaneously (our test will ensure proper ordering of connections)
14. (5 pts) Client handles abort connection attempt after 15 seconds.
15. (5 pts) Client aborts connection when server gets disconnected (server app or network connection is down)
16. (5 pts) Server aborts connection (a file should be created, containing only `ERROR` string) when doesn’t receive data from client for more than 15 seconds
17. (5 pts) Client able to successfully send and server properly receive and save large file over lossy and large delay network (we will use `tc` based emulation).

Grading Hint

- If you see 100% load on your CPU when you running your client/server, it is an indication that something is wrong and test script will legitimately (but randomly) fail.

Deductions

1. (-5 pts) The submission archive contains temporary or other non-source code file, except `README.md`, `Vagrantfile`, files under `.git` subfolder.