Project 1: Send Files over TCP Using C++ and Android based Client Applications

Overview

In this project, you will need to implement a simple client-server application that transfers a file over a TCP connection. Part 1 consists of writing a client and server applications in C++. Part 2 consists of writing a mobile client application (eg android, IOS, PI) that connects to the same server application in Part 1.

Part 1 : Recommeded that all implementations are written in C++ using BSD sockets (http://en.wikipedia.org/wiki/Berkeley_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. We will also accept implementations written in C, but it will be the responsibility of each student to ensure that their C code works with the provided test-cases. We will also accept python implementation as long as no high level abstractions and modules used to handle the requirements stated in the project. The use of C++ is preferred and encouranged.

Part 2 : Write another client application on a mobile platform (android or iOS or Raspberry PI).

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. There are several resources that provide further information about git (https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners) and GitHub (https://docs.github.com/en/desktop/installing-and-

configuring-github-desktop/getting-started-with-github-desktop).

> You are encouraged to host your code in private repositories on GitHub (https://github.com/), GitLab (https://gitlab.com/), or other places. 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 honestly 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 the `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 must count all established connections (1 for the first connection, 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.
- You can assume that the server will not need to handle and process data from multiple clients at the same time. However, a client can establish sequential connections to the server (e.g., connect to the server, send a file, disconnect, connect again, send another file, disconnect, etc.)
- The server must assume error if no data received from the client for over `10 seconds` . It should abort the connection and write a single `ERROR` string (without end-of-line/carret-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

Part 1: C++ App

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 10 seconds**:
    - Timeout to connect to server should be no longer than `10 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 `10 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.

Part 2: Android App (you can choose another mobile platform)

The client android application must have all functionalities listed above. Student need to design a user friendly interface that connects to the server (same server application developped earlier) and exchanges a stored file. The application must provide the user with at least the following functionalities: connection, sending, disconnection, measuring the time of the connection and exchange as well as the bandwidth between the client and the server.

Refer to the following tutorial (https://www.tutorialspoint.com/sending-and-receiving-data-with-sockets-in-android) for starter code and ideas on developping this application. Note that the tutorial helps implement a server app while you are required to use the previously described app.

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.

Optional: 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 is some sample code:

- A simple server that echoes back anything sent by client: server.cpp (./Project-1/server.cpp), client.cpp (Project-1/client.cpp)

Other resources

- Guide to Network Programming Using Sockets (http://beej.us/guide/bgnet/)

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 favorite virtualization engine (VirtualBox, VMware) using the Vagrant platform and steps outlined below.

Set Up Vagrant and Create VM Instance

You are NOT required to use vagrant for your project. You are welcome to develop your project in any way you see fit. However, this step is needed if you want your project to work universally across different operating system platforms and hardware specifications (remember each student may have a different laptop model and/or operating system). 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 (https://www.virtualbox.org/wiki/Downloads)
- Download and install Vagrant tools (https://www.vagrantup.com/downloads.html) for your platform
- Set up project and VM instance
  - Clone project template

    ```
    git clone https://github.com/amtibaa-cmu/CMPSCI5792_P1.git
    cd Project-1
    ```

  - 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 your project 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, please read this article (https://www.osradar.com/how-to-install-and-configure-vagrant-in-windows-10/) 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
```

- 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, studentID
   - 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 `.zip` archive.

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

   ```
   make tarball
   ```

   Then submit the resulting archive to canvas.

Before submission, please make sure:

1. Your code compiles
2. Client and server conforms to the specification
3. `.zip` 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 detecting 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. You will also be requested to demo your code and the execution (via zoom). Projects receive full credit if only all these checks are passed.

Grading Criteria

1. (5 pts) At least 3 git commits
2. (10 pts) readme file describing the functionalities implemented/missing
3. (5 pts) Client/Server handles incorrect hostname/port
4. (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. (10 pts) Client able to successfully transmit a small file (1 MB)
8. (20 pts) Client able to successfully transmit a large size file (100 MB)
    - sending in large chunks without delays
    - receiving file sent in large chunks without delays
    - receiving file sent in small chunks with delays
9. (5 pts) Client handles abort connection attempt after 10 seconds.
10. (5 pts) Client aborts connection when server gets disconnected (server app or network connection is down)
11. (25 pts) Android client app implements all above functionalities.

Grading Hint

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