CS 118: Computer Network Fundamentals - Fall 2020

Project 1: File Transmission with CRC (Cyclic Redundancy Check)

❖ Overview

In this project, you will need to implement a simple client-server application that transfers a file over a TCP connection and uses CRC (Cyclic Redundancy Check) to detect any potential errors.

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 and multi-threading. We will also accept implementations written in C, however, the use of C++ is preferred.

The objective of this project is to understand how CRCs work, learn the basic functions of BSD sockets, understand the implications of using the API, and to discover common pitfalls when working with network operations.

You are required to use private `git` repositories 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.**

This is NOT a group project. Each student must work on the project individually.

You are PROHIBITED from making your code public during the class or any time after. You are encouraged to host your code in **PRIVATE** repositories on GitHub, GitLab, or other places.

❖ Task Description

The project contains three parts: a server, a client, and a CRC component.

- The server listens for TCP connections and saves all the received data from the client in a file. The server has to verify the received data by using the CRC code before saving it in the file.
- The client connects to the server and, as soon as the connection is established, sends the contents of a file to the server. The client adds a CRC code to each segment (chunk) of the content before sending it to the server.
- The CRC component is used by both the client and server to build a CRC-64 table and provide a function to obtain CRC codes.

1. Server Application Specification

The server application MUST be compiled into a binary called `server`, accepts the two command-line arguments:

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

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

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 numbers and exit with a non-zero error code (you can assume that the folder is always correct). In addition to exiting, the server must print out to standard error (`std::cerr`) an error message that starts with the `ERROR:` string.
- The server should exit with code zero when receiving `SIGQUIT`/`SIGTERM` signal.
- The server should be able to accept and process connections from multiple clients at the same time.

- The server must count all established connections (i.e., 1 for the first connection, 2 for the second connection, etc.). The file received over the connection must be saved to `<FILE-DIR>/<CONNECTION-ID>.file` (e.g., `/save/1.file`, `/save/2.file`, etc.).
- The server must assume there is an error if no data is received from a client for over `10 seconds`. In this case, it should abort the connection and write a single `ERROR` string (without end-of-line/caret-return symbol) into the corresponding file. Note that any partial input must be discarded.
- The server must use CRC to verify that the received data has no error.
- The server must abort the connection and write a single `ERROR` string (without end-of-line/caret-return symbol) into the corresponding file when there is a CRC verification error.
- The server should be able to accept and save files up to `100 MiB`.

2. Client Application Specification

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

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

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

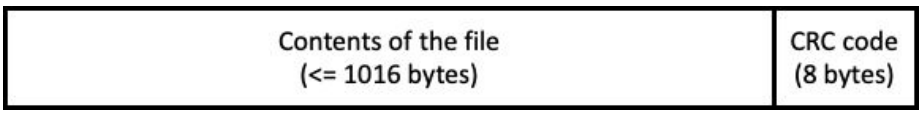
For example, the command below should result in a connection to a server on the same machine listening on port 5000 and transfer the content of `file.txt`:

```
$ ./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 hostnames and port numbers and exit with a non-zero exit code (you can assume that the specified file is always correct). In addition to exiting, the client must print out on standard error (`std::cerr`) an error message that starts with the string "`ERROR:`."

- The client application should exit with code zero after the file is successfully transferred to the server. It should support the transfer of files up to 100 MiB.
- The client should handle connection and transmission errors. The reaction time to network or server errors should be **no longer than 10 seconds**:
    - The timeout when connecting to a server should be no longer than `10 seconds`
    - The timeout when unable to send more data to the server (not being able to write to the send buffer) should be no longer than `10 seconds`.
- Whenever a timeout occurs, the client should abort the connection, print an error string starting with `ERROR:` to standard error (`std::cerr`), and exit with a non-zero code.
- The client must use a sending buffer with a size of 1024 `bytes`.
- The client must append an `8 bytes` CRC code to the contents of the file in each TCP segment. Therefore, the maximum number of bytes of the segment payload is `1016 bytes`.



3. CRC Component Specification

The CRC component provides functions to generate a CRC-64 table and to calculate the CRC of a given byte stream.

The client and the server must include the CRC64 header file to generate CRC codes. In this project, we use CRC-64-ECMA whose polynomial is as follows:

$$x^{64} + x^{62} + x^{57} + x^{55} + x^{54} + x^{53} + x^{52} + x^{47} + x^{46} + x^{45} + x^{40} + x^{39} + x^{38} + x^{37} + x^{35} + x^{33} +$$

$$x^{32} + x^{31} + x^{29} + x^{27} + x^{24} + x^{23} + x^{22} + x^{21} + x^{19} + x^{17} + x^{13} + x^{12} + x^{10} + x^{9} + x^{7} + x^{4} + x + 1$$

The coefficient of the polynomial can be represented as follows:

1 0100 0010 1111 0000 1110 0001 1110 1011 1010 1001 1110 1010 0011 0110 1001 0011

Note that we use MSB (Most Significant Byte) first to make data in network order.

Finally, we can write the polynomial in uint64_t in C/C++ as follows:

```
const uint64_t CRC64::m_poly =42F0E1EBA9EA3693
```

Using the CRC polynomial, we can build the CRC-table to store precomputed CRC codes of the 256 possible 8-bit bytes. The following example code is generating a CRC table by using CRC-16.

```
big_endian_table[0] := 0
crc := 0x8000 // Assuming a 16-bit polynomial
i := 1
do {
  if crc and 0x8000 {
    crc := (crc leftShift 1) xor 0x1021 // The CRC polynomial
  } else {
    crc := crc leftShift 1
  }
  // crc is the value of big_endian_table[i]; let j iterate over the already-initialized entries
  for j from 0 to i-1 {
    big_endian_table[i + j] := crc xor big_endian_table[j];
  }
  i := i leftshift 1
} while i < 256
```

Once we create the CRC table, we can calculate CRC codes by looking up the table. The following example code shows how to calculate CRCs with the CRC table.

```
rem = (rem leftShift 8) xor big_endian_table[string[i] xor ((leftmost 8 bits of rem) rightShift (n-8))]
```

Since the system automatically stores the generated CRC codes in little-endian byte order, your client needs to force it to keep it as big-endian byte order. After generating the CRC code by using the pseudo-code above, the client has to call the following function:

```
CRC = htobe64(get_crc_code(uint8_t *stream, int length))
```

Requirements:

- The constructor of the CRC component should create a CRC table which contains pre-calculated CRC codes for all possible combinations of 8-bit data.
- The CRC component must provide a function that returns a CRC code for a given byte stream.

❖ 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 a non-privileged user. This will allow you to notice reserved port restrictions from the kernel.

Here are some hints for using multi-threaded techniques to implement the server.

- For the server, you may have the main thread listening (and accepting) incoming **connection requests**.
    - Do you need a special socket API here?
    - How do you keep a listening socket to receive new requests?
- Once you accept a new connection, create a child thread for the new connection.
    - Should the new connection use the same socket as the the main thread?

Other resources:

- CRC (Cyclic Redundancy Check)
- 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 20.04-based virtual machine.

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

We provide a Vagrant file to help you set up the project environment, but the use of Vagrant is not mandatory. You can use any other virtualization software or directly use Linux machines.

Set Up Vagrant and Create VM Instance

Note that all example commands below are to be 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 gain access to the virtual machine and can compile your code there.

1. Download and install your favorite virtualization engine, e.g., VirtualBox
2. Download and install Vagrant tools for your platform
3. Set up the project and VM instance as follows:

i. **Clone project template**

```
git clone https://github.com/ksb2043/cs118_fall2020_project1 ~/cs118-proj1

cd ~/cs118-proj1
```

ii. **Initialize VM**

```
vagrant up
```

Do not start the VM instance manually from the VirtualBox GUI, otherwise you may encounter various problems (connection errors, connection timeouts, missing packages, etc.)

iii. **Establish an SSH session to the created VM**

```
vagrant ssh
```

- If you are using Putty on Windows, `vagrant ssh` will return information regarding the IP address and port to connect to your virtual machine.

iv. **Work on your project**

All files in the `~/cs118-proj1` folder on the host machine will be automatically synchronized with the `/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 you set up the environment.
- The codebase contains the basic files: `Makefile`, `server.cpp`, `client.cpp`, and `CRC.cpp`.
- You are now free to add more files and modify the Makefile to complete the `server` and `client` implementations.

❖ Submission Requirements

To submit your project, you need to prepare:

1. A `README.md` file placed in your code that includes:
    - Your name and UCLA ID
    - The high-level design of your server and client
    - The problems you ran into and how you solved them
    - List of any additional libraries used
    - Acknowledgment of any online tutorials or code example (except the class website) you made use of.
2. **If you need additional dependencies for your project, you must update the Vagrant file.**

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
```

The submission file (a `.tar.gz` archive) should contain all of your source code, along with the `Makefile`, `README.md`, `Vagrantfile`, and `.git` folder.

3. Then submit the resulting archive to the CCLE submission page.

Before submission, please make sure:

A. Your code compiles
B. Client and server conform to the specification
C. `.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 receive any credit.

❖ Grading

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

Your code will then be automatically tested against some test 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. (2.5 pts) Client connects and starts transmitting a file
6. (2.5 pts) Server accepts a connection and start saving a file
7. (10 pts) Client able to successfully transmit a small file (500 bytes) and Server able to receive a small file (500 bytes) and save it in `1.file`
8. (10 pts) Client able to successfully transmit a medium size file (1 MiB) and Server able to receive a medium file (1 MiB bytes) and save it in `1.file`
9. (5 pts) Client able to successfully transmit a large size file (100 MiB) and Server able to receive a large file (100 MiB bytes) and save it in `1.file`
10. (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)
11. (5 pts) Client handles abort connection attempt after 10 seconds.
12. (5 pts) Client aborts connection when server gets disconnected (server app or network connection is down)
13. (5 pts) Server aborts connection (a file should be created, containing only `ERROR` string) when doesn’t receive data from client for more than 10 seconds
14. (10 pts) CRC table contains correct CRC values
15. (10 pts) Client appends a CRC code to a payload of TCP segment
16. (10 pts) Server can verify CRC code by using the pre-populated CRC table
17. (5 pts) Server aborts connection (a file should be created, containing only `ERROR` string) when there is a CRC verification error