**Attribution:** This series of programming assignments, including codebase and documentation, is adopted from 'Sponge' of Stanford CS144 Introduction to Computer Networking by Prof. Keith Winstein https://cs144.github.io/.

# Programming Assignment 0: Networking Warmup

Due: March 1, 2024. 23:59 pm

Late deadline: March 2, 2024. 23:59 pm (20% penalty)

In this warmup, you will set up an installation of Linux on your computer, learn how to perform some tasks over the Internet by hand, write a small program in C++ that fetches a Web page over the Internet, and implement (in memory) one of the key abstractions of networking: a reliable stream of bytes between a writer and a reader. We expect this warmup to take you between 2 and 6 hours to complete. Three quick points about the assignment:

- It's a good idea to read the whole document before diving in!
- Over the course of this 8-part assignments, you'll be building up your own implementation of a significant portion of the Internet—a router, a network interface, and the TCP protocol (which transforms unreliable datagrams into a reliable byte stream). *Most weeks will build on work you have done previously*, i.e., you are building up your own implementation gradually over the course, and you'll continue to use your work in future weeks. This makes it hard to "skip" a checkpoint.
- The documents aren't "specifications"—meaning they're not intended to be consumed in a one-way fashion. They're written closer to the level of detail that a software engineer will get from a boss or client.
- If you think something might not be fully specified, sometimes the truth is that it doesn't matter—you could try one approach or the other and see what happens.
- If you find something to be ambiguous and you think the answer matters, we encourage you to ask questions on the Q&A board at PLMS. Note that 1-on-1 emails are mainly for private inquiries (e.g., your own score issues).

## 1. Set up GNU/Linux on your computer

CSED353's assignments require the GNU/Linux operating system and a recent C++ compiler that supports the C++ 2017 standard.

**NOTE:** If you are outside POSTECH campus network, **you will need a VPN client** running on your local computer to access the URLs below. Find more information at https://vpn.postech.ac.kr/

The most convenient way to set up an environment is to install the official Stanford CS144 VirtualBox virtual-machine image (instructions at http://tomahawk.postech.ac.kr/csed353/assignments/vm/virtualbox) on your local computer.

If you have a MacBook with the ARM64 M1 chip, VirtualBox will not successfully run. Instead, please install the UTM virtual machine software and our ARM64 virtual machine image from http://tomahawk.postech.ac.kr/csed353/assignments/vm/.

## 2. Networking by hand

Let's get started with using the network. You are going to do two tasks by hand: retrieving a Web page (just like a Web browser). This task relies on a networking abstraction called a reliable bidirectional byte stream: you'll type a sequence of bytes into the terminal, and the same sequence of bytes will eventually be delivered, in the same order, to a program running on another computer (a server). The server responds with its own sequence of bytes, delivered back to your terminal.

### 2.1 Fetch a Web page

1. In a Web browser, visit http://tomahawk.postech.ac.kr/hello and observe the result.[1]
2. Now, you'll do the same thing the browser does, by hand.
    a. **On your VM**, run `telnet tomahawk.postech.ac.kr http`. This tells the `telnet` program to open a reliable byte stream between your computer and another computer (named `tomahawk.postech.ac.kr`), and with a particular *service* running on that computer: the "`http`" service, for the Hyper-Text Transfer Protocol, used by the World Wide Web.[2]
    If your computer has been set up properly and is on the Internet, you will see:

    ```
    user@computer:~$ telnet tomahawk.postech.ac.kr http
    Trying 141.223.65.44...
    Connected to tomahawk.postech.ac.kr.
    Escape character is '^]'.
    ```

    If you need to quit, press `ctrl+]`, and then type `close`↵.

    b. Type `GET /hello HTTP/1.1`↵. This tells the server the *path* part of the URL. (The part starting with the third slash.)
    c. Type `Host: tomahawk.postech.ac.kr`↵. This tells the server the *host* part of the URL. (The part between `http://` and the third slash.)
    d. Type `Connection: close`↵. This tells the server that you are finished making requests, and it should close the connection as soon as it finishes replying.
    e. Hit the Enter key one more time: ↵. This sends an empty line and tells the server that you are done with your HTTP request.

[1] If you are outside POSTECH campus network, you will need a VPN client running on your local computer to have access to the port 80 of an on-campus server. Please find more information at https://vpn.postech.ac.kr/

[2] The computer's name has a numerical equivalent (`141.223.65.44`, an *Internet Protocol v4 address*), and so does the service's name (`80`, a *TCP port number*). We'll talk more about these later.

f. If all went well, you will see the same response that your browser saw, preceded by HTTP *headers* that tell the browser how to interpret the response.

3. **Assignment**: Now that you know how to fetch a Web page by hand, show us you can! Use the above technique to fetch the URL http://tomahawk.postech.ac.kr/csed353/assn0/*povisid*, replacing povisid with your own **POVIS ID (not STUDENT NUMBER).** You will receive a secret code in the `X-Your-Code-Is:` header. Save your **POVIS ID** and the code for inclusion in your writeup.

## 2.2 Listening and connecting

You've seen what you can do with telnet: a **client** program that makes outgoing connections to programs running on other computers. Now it's time to experiment with being a simple **server**: the kind of program that waits around for clients to connect to it.

1. In one terminal window, run `netcat -v -l -p 9090` on your VM. You should see:

   ```
   user@computer:~$ netcat -v -l -p 9090
   Listening on [0.0.0.0] (family 0, port 9090)
   ```

   **If you are a UTM user** and have an error `getnameinfo: Temporary failure in name resolution`, you should add `-n` option in your netcat command.

2. Leave `netcat` running. In another terminal window, run `telnet localhost 9090` (also on your VM).
3. If all goes well, the `netcat` will have printed something like "`Connection from localhost 53500 received!`".
4. Now try typing in either terminal window—the `netcat` (server) or the `telnet` (client). Notice that anything you type in one window appears in the other, and vice versa. You'll have to hit ↵ for bytes to be transferred.
5. In the netcat window, quit the program by typing `ctrl-C`. Notice that the `telnet` program immediately quits as well.

# 3. Writing a network program using an OS stream socket

In the next part of this warmup assignment, you will write a short program that fetches a Web page over the Internet. You will make use of a feature provided by the Linux kernel, and by most other operating systems: the ability to create a *reliable bidirectional byte stream* between two programs, one running on your computer, and the other on a different computer across the Internet (e.g., a Web server such as Apache or nginx, or the `netcat` program).

This feature is known as a *stream socket*. To your program and to the Web server, the socket looks like an ordinary file descriptor (similar to a file on disk, or to the `stdin` or `stdout` I/O streams). When two stream sockets are *connected*, any bytes written to one socket will eventually come out in the same order from the other socket on the other computer.

In reality, however, the Internet doesn't provide a service of reliable byte-streams. Instead, the only thing the Internet really does is to give its "best effort" to deliver short pieces of data, called

*Internet datagrams*, to their destination. Each datagram contains some metadata (headers) that specifies things like the source and destination addresses—what computer it came from, and what computer it's headed towards—as well as some *payload* data (up to about 1,500 bytes) to be delivered to the destination computer.

Although the network tries to deliver every datagram, in practice datagrams can be (1) lost, (2) delivered out of order, (3) delivered with the contents altered, or even (4) duplicated and delivered more than once. It's normally the job of the operating systems on either end of the connection to turn "best-effort datagrams" (the abstraction the Internet provides) into "reliable byte streams" (the abstraction that applications usually want).

The two computers have to cooperate to make sure that each byte in the stream eventually gets delivered, in its proper place in line, to the stream socket on the other side. They also have to tell each other how much data they are prepared to accept from the other computer, and make sure not to send more than the other side is willing to accept. All this is done using an agreed-upon scheme that was set down in 1981, called the Transmission Control Protocol, or TCP.

In this assignment, you will simply use the operating system's pre-existing support for the Transmission Control Protocol. You'll write a program called "`webget`" that creates a TCP stream socket, connects to a Web server, and fetches a page—much as you did earlier in this assignment. In future assignments, you'll implement the other side of this abstraction, by implementing the Transmission Control Protocol yourself to create a reliable byte-stream out of not-so-reliable datagrams.

## 3.1 Let's get started—fetching and building the starter code

1. The assignments will use a starter codebase called "Sponge." **On your VM**, run `git clone https://github.com/POSTECH-HIS/sponge` to fetch the source code for the assignment.
2. Optional: Feel free to backup your repository to a **private** GitHub/GitLab/Bitbucket repository (e.g., using the instructions at https://stackoverflow.com/questions/10065526/github-how-to-make-a-fork-of-public-repository-private), **but please make absolutely sure that your work remains private**.
3. Enter the Assignment 0 directory: `cd sponge`
4. Create a directory to compile the assignment software: `mkdir build`
5. Enter the build directory: `cd build`
6. Set up the build system: `cmake ..`
7. Compile the source code: `make` (you can run `make -j4` to use four processors).
8. Outside the `build` directory, open and start editing the `writeups/assn0.md` file. This is the template for your assignment writeup and will be included in your submission.

## 3.2 Modern C++: mostly safe but still fast and low-level

The assignments will be done in a contemporary C++ style that uses recent features to program as safely as possible. This might be different from how you have been asked to write C++ in the past. For references to this style, please see the C++ Core Guidelines (http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines).

The basic idea is to make sure that every object is designed to have the smallest possible public interface, has a lot of internal safety checks and is hard to use improperly, and knows how to clean up after itself. We want to avoid "paired" operations (e.g. malloc/free, or new/delete), where it might be possible for the second half of the pair not to happen (e.g., if a function returns early or throws an exception). Instead, operations happen in the constructor to an object, and the opposite operation happens in the destructor. This style is called "Resource acquisition is initialization," or RAII.

In particular, we would like you to:

- Use the language documentation at https://en.cppreference.com as a resource.
- Never use `malloc()` or `free()`.
- Never use **new** or **delete**.
- Essentially never use raw pointers (*), and use "smart" pointers (`unique_ptr` or `shared_ptr`) only when necessary. (You will not need to use these in CSED353.)
- Avoid templates, threads, locks, and virtual functions. (You will not need to use these in CSED353.)
- Avoid C-style strings (`char *str`) or string functions (`strlen()`, `strcpy()`). These are pretty error-prone. Use a `std::string` instead.
- Never use C-style casts (e.g., `(FILE *)x`). Use a C++ `static_cast` if you have to (you generally will not need this in CSED353).
- Prefer passing function arguments by `const` reference (e.g.: `const Address & address`).
- Make every variable `const` unless it needs to be mutated.
- Make every method `const` unless it needs to mutate the object.
- Avoid global variables, and give every variable the smallest scope possible.
- Before handing in an assignment, please run `make format` to normalize the coding style.

**On using Git:** The assignments are distributed as Git (version control) repositories—a way of documenting changes, checkpointing versions to help with debugging, and tracking the provenance of source code. **Please make frequent small commits as you work, and use commit messages that identify what changed and why.** The Platonic ideal is that each commit should compile and should move steadily towards more and more tests passing. Making small "semantic" commits helps with debugging (it's much easier to debug if each commit compiles and the message describes one clear thing that the commit does) and protects you against claims of cheating by documenting your steady progress over time—and it's a useful skill that will help in any career that includes software development. The graders will be reading your commit messages to understand how you developed your solutions to the assignments. If you haven't learned how to use Git, please consult a tutorial (e.g., https://guides.github.com/introduction/git-handbook). Finally, you are welcome to store your code in a **private** repository on GitHub, GitLab, Bitbucket, etc., but please **make sure your code is not publicly accessible**.

## 3.3 Reading the Sponge documentation

To support this style of programming, Sponge's classes wrap operating-system functions (which can be called from C) in "modern" C++.

1. Using a Web browser, read over the documentation to the starter code at http://tomahawk.postech.ac.kr/csed353/assignments/assn0/.
2. Pay particular attention to the documentation for the `FileDescriptor`, `Socket`, `TCPSocket`, and `Address` classes. (Note that a `Socket` is a type of `FileDescriptor`, and a `TCPSocket` is a type of `Socket`.)
3. Now, find and read over the header files that describe the interface to these classes in the `libsponge/util` directory: `file_descriptor.hh`, `socket.hh`, and `address.hh`.

## 3.4 Writing webget

It's time to implement webget, a program to fetch Web pages over the Internet using the operating system's TCP support and stream-socket abstraction—just like you did by hand earlier in this assignment.

1. From the `build` directory, open the file `../apps/webget.cc` in a text editor or IDE.
2. In the **get_URL** function, find the comment starting "`// Your code here.`"
3. Implement the simple Web client as described in this file, using the format of an HTTP (Web) request that you used earlier. Use the `TCPSocket` and `Address` classes.
4. Hints:
   - Please note that in HTTP, each line must be ended with "\r\n" (it's not sufficient to use just "\n" or `endl`).
   - Don't forget to include the "Connection: close" line in your client's request. This tells the server that it shouldn't wait around for your client to send any more requests after this one. Instead, the server will send one reply and then will immediately end its outgoing bytestream (the one *from* the server's socket *to* your socket). You'll discover that your incoming byte stream has ended because your socket will reach "EOF" (end of file) when you have read the entire byte stream coming from the server. That's how your client will know that the server has finished its reply.
   - Make sure to read and print all the output from the server until the socket reaches "EOF" (end of file)—**a single call to `read` is not enough**.
   - We expect you'll need to write about ten lines of code.

5. Compile your program by running `make`. If you see an error message, you will need to fix it before continuing.
6. Test your program by running `./apps/webget tomahawk.postech.ac.kr /hello`. How does this compare to what you see when visiting http://tomahawk.postech.ac.kr/hello in a Web browser? How does it compare to the results from Section 2.1?
7. When it seems to be working properly, run `make check_webget` to run the automated test. Before implementing the get URL function, you should expect to see the following:

   ```
   1/1 Test #31: t_webget .........................***Failed  0.00 sec
   Function called: get_URL(tomahawk.postech.ac.kr, /hasher/xyzzy).
   Warning: get_URL() has not been implemented yet.
   ERROR: webget returned output that did not match the test's expectations
   ```

   After completing the assignment, you will see:

   ```
   1/1 Test #31: t_webget .........................  Passed  0.04 sec
   100% tests passed, 0 tests failed out of 1
   ```
8. The graders will run your webget program with a different hostname and path than make check runs—so make sure it doesn't only work with the hostname and path used by make check.

## 4. An in-memory reliable byte stream

By now, you've seen how the abstraction of a *reliable byte stream* can be useful in communicating across the Internet, even though the Internet itself only provides the service of "best-effort" (unreliable) datagrams.

To finish off this assignment, you will implement, in memory on a single computer, an object that provides this abstraction. Bytes are written on the "input" side and can be read, in the same sequence, from the "output" side. The byte stream is finite: the writer can end the input, and then no more bytes can be written. When the reader has read to the end of the stream, it will reach "EOF" (end of file) and no more bytes can be read.

Your byte stream will also be *flow-controlled* to limit its memory consumption at any given time. The object is initialized with a particular "capacity": the maximum number of bytes it's willing to store in its own memory at any given point. The byte stream will limit the writer in how much it can write at any given moment, to make sure that the stream doesn't exceed its storage capacity. As the reader reads bytes and drains them from the stream, the writer is allowed to write more. Your byte stream is for use in a *single* thread—you don't have to worry about concurrent writers/readers, locking, or race conditions.

To be clear: the byte stream is finite, but it can be *almost arbitrarily long*[3] before the writer ends the input and finishes the stream. Your implementation must be able to handle streams that are much longer than the capacity. The capacity limits the number of bytes that are held in memory (written but not yet read) at a given point, but does not limit the length of the stream. An object with a capacity of only one byte could still carry a stream that is terabytes and terabytes long, as long as the writer keeps writing one byte at a time and the reader reads each byte before the writer is allowed to write the next byte.

[3] At least up to $2^{64}$ bytes, which in this class we will regard as essentially arbitrarily long

Here's what the interface looks like for the writer:

```
// Write a string of bytes into the stream. Write as many
// as will fit, and return the number of bytes written.
size_t write(const std::string &data);

// Returns the number of additional bytes that the stream has space for
size_t remaining_capacity() const;

// Signal that the byte stream has reached its ending
void end_input();

// Indicate that the stream suffered an error
void set_error();
```

And here is the interface for the reader:

```
// Peek at next "len" bytes of the stream
std::string peek_output(const size_t len) const;

// Remove ``len'' bytes from the buffer
 void pop_output(const size_t len);

// Read (i.e., copy and then pop) the next "len" bytes of the stream
std::string read(const size_t len);

bool input_ended() const;        // `true` if the stream input has ended
bool eof() const;                // `true` if the output has reached the ending
bool error() const;              // `true` if the stream has suffered an error
size_t buffer_size() const;      // the maximum amount that can currently be
peeked/read
bool buffer_empty() const;       // `true` if the buffer is empty

size_t bytes_written() const;        // Total number of bytes written
size_t bytes_read() const;           // Total number of bytes popped
```

Please open the libsponge/byte stream.hh and libsponge/byte stream.cc files, and implement an object that provides this interface. **You may add any private members and member functions, but you must not change its public interface.** As you develop your byte stream implementation, you can run the automated tests with `make check_lab0`.

What's next? Over the next four weeks, you'll implement a system to provide the same inter- face, no longer in memory, but instead over an unreliable network. This is the Transmission Control Protocol.

## 5. Submit

1. In your submission, please only make changes to webget.cc and the source code in the top level of `libsponge` (`byte_stream.hh` and `byte_stream.cc`). Please don't modify any of the tests or the helpers in `libsponge/util`.
2. Before handing in any assignment, please run these in order:
   (a) `make format` (to normalize the coding style)
   (b) `make` (to make sure the code compiles)
   (c) `make check_lab0` (to make sure the automated tests pass)
3. Finish editing `writeups/assn0.md`, filling in the number of hours this assignment took you and any other comments.
4. When you have finished your implementation, do not forget to **commit all changes you made!** This **should be done before creating an archive** of your git repository. If you create an archive before committing changes, the archive will include only part or none of the changes you made. This may result in a lower score than you anticipated, or worse zero.

   If you are unsure whether your archive includes all changes you made, please refer to our Q&A item: "Before submitting my bundle, I want to make sure the bundle includes all my commits".
5. To archive your git repository, use the following command from the git working directory on your VM:

   ```
   git bundle create /tmp/<your_student_id>.git --all
   ```

   **Important**: This command will create an archive of your git history, which means that your solution must be committed to your git repository! In addition, we will be grading only the master branch of your repository, so please be sure that branch corresponds to the solution you want to turn in.
6. Please upload your archive (bundle) file to the proper assignment entry on PLMS.