Lab 0: networking warmup

Due: September 30, 5 p.m. (hard deadline: October 2, 5 p.m. with late penalty)

Welcome to CS144: Introduction to Computer Networking. In this warmup lab, 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 (future labs will take much more of your time). It's a good idea to read the whole lab before diving in.

0 Collaboration Policy

The programming assignments must be your own work: You must write all the code you hand in for the programming assignments, except for the code that we give you as part of the assignment. Please do not copy-and-paste code from Stack Overflow, GitHub, or other sources. If you base your own code on examples you find on the Web or elsewhere, cite the URL in a comment in your submitted source code.

Working with others: You may not show your code to anyone else, look at anyone else's code, or look at solutions from previous years. You may discuss the assignments with other students, but do not copy anybody's code. If you discuss an assignment with another student, please name them in a comment in your submitted source code. Please refer to the course administrative handout for more details, and ask on Piazza if anything is unclear.

Piazza: Please feel free to ask questions on Piazza, but please don't post any source code.

1 Set up GNU/Linux on your computer

CS144's assignments require the GNU/Linux operating system and a recent C++ compiler that supports the C++ 2017 standard. Please choose one of these three options:

1. **Recommended:** Install the CS144 VirtualBox virtual-machine image (instructions at https://stanford.edu/class/cs144/vm_howto/vm-howto-image.html).
2. Run Ubuntu version 18.04 LTS, then run our setup script. You can do this on your actual computer, inside VirtualBox, or on EC2 or another virtual machine (a step-by-step guide is at https://stanford.edu/class/cs144/vm_howto/vm-howto-iso.html).
3. Use another GNU/Linux distribution, but be aware that you may hit roadblocks along the way and will need to be comfortable debugging them. Your code will be tested on Ubuntu 18.04 LTS with `g++` 8.2.0 and must compile and run properly under those conditions. Tips at https://stanford.edu/class/cs144/vm_howto/vm-howto-byo.html.

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) and sending an email message (like an email client). Both of these tasks rely on a networking abstraction called a *reliable bidirectional in-order 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://cs144.keithw.org/hello and observe the result.
2. Now, you'll do the same thing the browser does, by hand.
   (a) **On your VM**, run `telnet cs144.keithw.org http`. This tells the `telnet` program to open a reliable byte stream between your computer and another computer (named `cs144.keithw.org`), and with a particular *service* running on that computer: the "`http`" service, for the Hyper-Text Transfer Protocol, used by the World Wide Web.[1]
   If your computer has been set up properly and is on the Internet, you will see:

   ```
   user@computer:~$ telnet cs144.keithw.org http
   Trying 104.196.238.229...
   Connected to cs144.keithw.org.
   Escape character is '^]'.
   ```

   If you need to quit, hold down ctrl and press], 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: cs144.keithw.org` ↵. This tells the server the *host* part of the URL. (The part between http:// and the third slash.)
   (d) Hit the Enter key one more time: ↵. This tells the server that you are done with your HTTP request.
   (e) 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://cs144.keithw.org/lab0/*sunetid*,

[1] The computer's name has a numerical equivalent (`104.196.238.229`, an *Internet Protocol v4 address*), and so does the service's name (`80`, a *TCP port number*). We'll talk more about these later.

replacing *sunetid* with your own primary SUNet ID. You will receive a secret code in the `X-Your-Code-Is:` header. Save your SUNet ID and the code for inclusion in your writeup.

2.2 Send yourself an email

Now that you know how to fetch a Web page, it's time to send an email message, again using a reliable byte stream to a service running on another computer.

1. **From within Stanford's network**, run `telnet smtp-unencrypted.stanford.edu smtp`. (If you are not on Stanford's network, log in to `cardinal.stanford.edu` first and then run these commands.) The "`smtp`" service refers to the Simple Mail Transfer Protocol, used to send email messages. If all goes well, you will see:

   ```
   user@computer:~$ telnet smtp-unencrypted.stanford.edu smtp
   Trying 171.64.13.18...
   Connected to smtp-unencrypted.stanford.edu.
   Escape character is '^]'.
   220 smtp-unencrypted.stanford.edu ESMTP Postfix
   ```

2. First step: identify your computer to the email server. Type `HELO mycomputer.stanford.edu` ↵. Wait to see "`250 smtp-unencrypted.stanford.edu`".

3. Next step: who is sending the email? Type `MAIL FROM: sunetid@stanford.edu` ↵. Replace *sunetid* with your SUNet ID.[2] If all goes well, you will see "`250 2.1.0 Ok`".

4. Next: who is the recipient? Type `RCPT TO: sunetid@stanford.edu` ↵. Replace *sunetid* with your SUNet ID. If all goes well, you will see "`250 2.1.5 Ok`."

5. It's time to upload the email message itself. Type `DATA` ↵ to tell the server you're ready to start. If all goes well, you will see "`354 End data with <CR><LF>.<CR><LF>`".

6. Now you are typing an email message to yourself. First, start by typing the *headers* that you will see in your email client. Leave a blank line at the end of the headers.

   ```
   354 End data with <CR><LF>.<CR><LF>
   From: sunetid@stanford.edu ↵
   To: sunetid@stanford.edu ↵
   Subject: Hello from CS144 Lab 0! ↵
   ↵
   ```

7. Type the *body* of the email message—anything you like. When finished, end with a dot on a line by itself: `.` ↵. Expect to see something like: "`250 2.0.0 Ok: queued`".

[2] Yes, it is possible to give a fake "from" address. Electronic mail is a bit like real mail from the postal service, in that the accuracy of the return address is (mostly) on the honor system. Please do not abuse this!

8. Type `QUIT` ⏎ to end the conversation with the email server. Check your inbox and spam folder to make sure you got the email.

9. **Assignment:** Now that you know how to fetch a Web page by hand, show us you can. Use the above technique to send an email, from yourself, to `cs144grader@gmail.com`.

2.3 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)
   ```

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 transfered.

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 lab, 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 in-order 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 lab, 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 lab. In future labs, 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 lab assignments will use a starter codebase called "Sponge." **On your VM**, run `git clone https://github.com/cs144/sponge` to fetch the source code for the lab.
2. Enter the Lab 0 directory: `cd sponge`
3. Create a directory to compile the lab software: `mkdir build`
4. Enter the build directory: `cd build`
5. Set up the build system: `cmake ..`
6. Compile the source code: `make` (you can run, e.g., `make -j4` to use four processors when compiling).
7. Outside the `build` directory, open and start editing the `writeups/lab0.md` file. This is the template for your lab writeup and will be included in your submission.

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

The lab assignments will be done in a contemporary C++ style that uses recent (2011) 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 CS144.)
- Avoid templates, threads, locks, and virtual functions. (You will not need to use these in CS144.)
- 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 CS144).
- 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 labs 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 labs. If you haven't learned how to use Git, please do ask for help at the CS144 office hours or 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 https://cs144.github.io/doc/lab0.
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 lab.

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`).
   - When you have finished writing your request to the socket, tell the server that you're done making requests by ending your outgoing bytestream (the bytestream *from* your socket *to* the server's socket). You can do this by calling the `shutdown`

method of a `TCPSocket` with an argument of `SHUT_WR`. In response, the server will send you one reply and then will end its own 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. (If you don't shut down your outgoing byte stream, the server will wait around for a while for you to send additional requests and won't end its outgoing byte stream either.)

- 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 cs144.keithw.org /hello`. How does this compare to what you see when visiting http://cs144.keithw.org/hello in a Web browser? How does it compare to the results from Section 2.1? Feel free to experiment—test it with any `http` URL you like!

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 #25: lab0_webget .....................***Failed    0.00 sec
Function called: get_URL(cs144.keithw.org, /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:

```
4/4 Test #4: lab0_webget ......................   Passed    0.14 sec

100% tests passed, 0 tests failed out of 4
```

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 in-order 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 week's lab, you will implement, in memory on a single computer, an object that provides this abstraction. (You may have done something similar in CS 110.) 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*: it's initialized with a particular capacity: the maximum number of bytes it's willing to store in its own memory. The byte stream will limit the writer in how long a string it can write, 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.

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. 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 interface, no longer in memory, but instead over an unreliable network. This is the Transmission Control Protocol.

5 Submit

1. 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)
2. Finish editing `writeups/lab0.md`, filling in the number of hours this assignment took you and any other comments.
3. When ready to submit, please follow the instructions at https://cs144.github.io/submit.
4. Please let the course staff know ASAP of any problems by posting a question on Piazza. Good luck and welcome to CS144!