Project 2: TritonHTTP

Due: Monday, May 6th at 11:59pm

TA in charge: Manas Sharma

Project Overview

In this project, you are going to build a simple web server that implements a subset of the HTTP/1.1 protocol specification called TritonHTTP.

- Starter code: https://classroom.github.com/a/y-k-1497

Note that the starter code README.md file has a succinct summary of what you should implement, and that is probably your best starting point on beginning the project. The information below goes into more details on the points raised in the starter code.

Basic Web Server Functionality

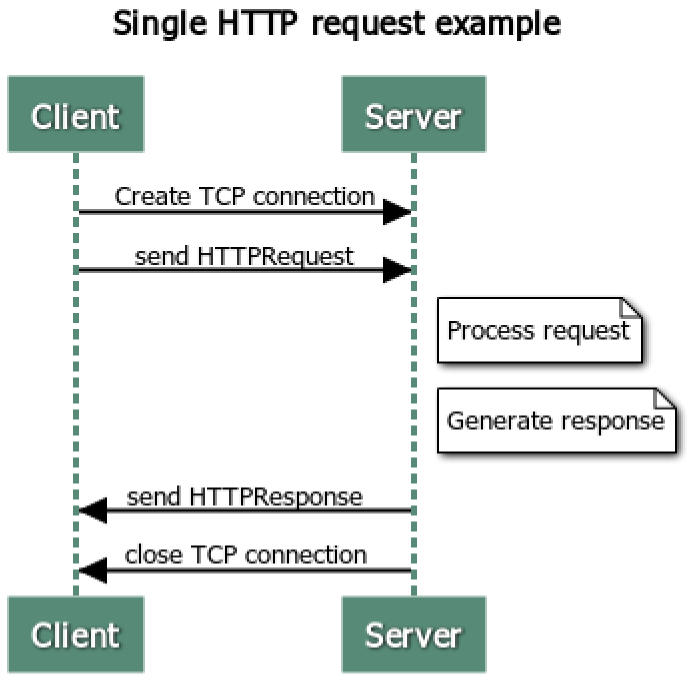
At a high level, a web server listens for connections on a socket (bound to a specific port on a host machine). Clients connect to this socket and use the HTTP protocol to retrieve files from the server. Your server will read data from the client, using the framing and parsing techniques discussed in class to interpret one or more requests (if the client is using pipelined requests). Every time your server reads in a full request, you will service that request and send a response back to the client. After sending back one (or more) responses, your server will either close the connection (if instructed to do so by the client via the “Connection: close” header, described below), or after an appropriate timeout occurs (also described below). Your web server will then continue waiting for future client connections. Your server should be implemented in a concurrent manner, so that it can process multiple client requests overlapping in time.

TritonHTTP Specification

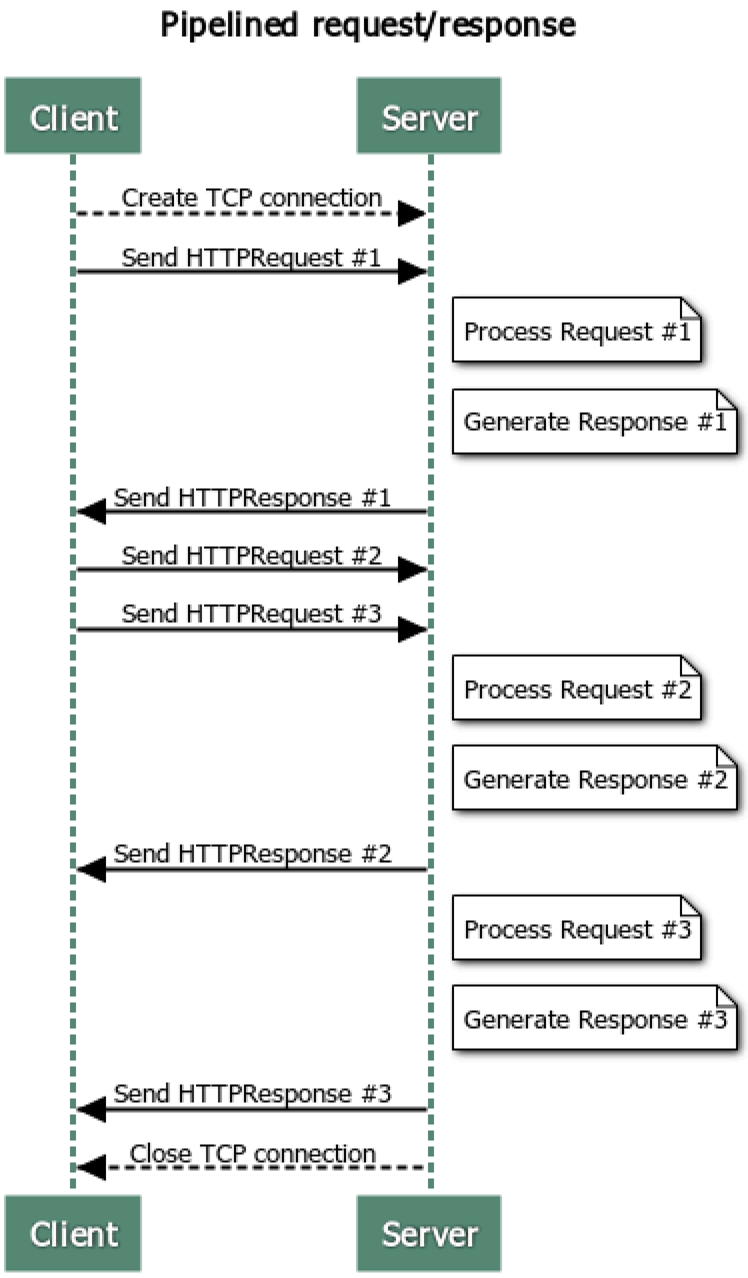
This section describes a minimal subset (which also differs in details) of the HTTP/1.1 protocol specification. For this class, please consider this spec as the definitive guide for implementing HTTP – as such, we are going to call it **TritonHTTP**. Portions of this specification are courtesy of James Marshall, used with permission from the author.

Client/Server Protocol

TritonHTTP is a client/server protocol that is layered on top of the reliable stream-oriented transport protocol TCP. Clients send request messages to the server, and servers reply with response messages. In its most basic form, a single TritonHTTP-level request/response exchange happens over a single, dedicated TCP connection. The client first connects to the server, and then sends the TritonHTTP request message. The server replies with a TritonHTTP response, and then closes the connection:



Repeatedly setting up and tearing down TCP connections reduces overall network throughput and efficiency, and so TritonHTTP has a mechanism whereby a client can reuse a TCP connection to a given server (HTTP **persistent connection**). The idea is that the client opens a TCP connection to the server, issues a TritonHTTP request, gets a TritonHTTP response, and then issues another TritonHTTP request on the already open outbound part of the connection. The server replies with the response, and this can continue through multiple request/response interactions. The client signals the last request by setting a “Connection: close” header, described below. The server indicates that it will not handle additional requests by setting the “Connection: close” header in the response. Note that the client can issue more than one TritonHTTP request without necessarily waiting for full HTTP replies to be returned (HTTP **pipelining**).



To support clients that do not properly set the "Connection: close" header, the server must implement a **timeout** mechanism to know when it should close the connection (otherwise it might just wait forever). For this project, you should establish a server timeout of 5 seconds. If this timeout occurs and the client has sent part of a request, but not a full request, then the server should reply back with a 400 client error (described below). If this timeout occurs and the client has not started sending any part of a new request, the server should simply close the connection.

HTTP Messages

TritonHTTP request and response messages are plain-text ASCII (the body of a response can also contain binary data). Both requests and responses start with a header section. Responses optionally contain a body section which is separated from the header section by a blank line. The header consists of an initial line (which is different between requests and responses), followed by zero or more key-value pairs. Every line is terminated by a CRLF (carriage-return followed by a line feed, or “\r\n”).

A request message has this form:

```
<initial request line>[CRLF]
Key1: value1[CRLF]
Key2: value2[CRLF]
…
KeyN: valueN[CRLF]
[CRLF]
```

A response message has this form:

```
<initial response line>[CRLF]
Key1: value1[CRLF]
Key2: value2[CRLF]
…
KeyN: valueN[CRLF]
[CRLF]
<optional body>
```

Note that the optional body section is not terminated by a CRLF delimiter. Instead, the end of that body will be indicated via the Content-Length header, described below. There is no specific limit to the size (in bytes) of a request or response message, and no specific limit to the number of key-value pair headers each could contain.

Initial Request Line

Line of a TritonHTTP request header has three components:

`GET <URL> HTTP/1.1`

The **method** field GET indicates that the client wants to download the content located at the provided URL. Real web servers support other methods such as PUT and POST, which are used to upload data to websites. We will only implement the GET method.

The **URL** specifies the location of the resource the client is interested in. Examples include /images/myimg.jpg and /classes/fall/cs101/index.html. A well-formed URL always starts with a / character. If the slash is missing, send back a 400 error. Note that if the URL ends with / like

/path/to/dir/, then you should interpret that as if the client requested the URL /path/to/dir/index.html.

The protocol **version** field takes the form HTTP/x.y, where x.y is the highest version that the client supports. For this course we'll always use 1.1, so this value should be HTTP/1.1.

The fully formed initial request line would thus look something like:

```
GET /images/myimg.jpg HTTP/1.1
```

Any request that doesn't align with the specifications outlined in this document should be responded to with a 400 status code.

Initial Response Line

The initial line of a TritonHTTP response also has three components, which are slightly different than those in the request line:

```
HTTP/1.1 <status code> <status description>
```

The first term is the highest HTTP **version** that the server supports, in our case HTTP/1.1.

The next term is a three-digit numeric code indicating the **status code** of the request (e.g., whether it succeeded or failed, including more fine-grained information about how to interpret this response).

The third term is a human-friendly **status description**, which can contain spaces.

In this project we will support the following three response statuses:
- **200 OK**: The request was successful
- **400 Bad Request**: The client sent a malformed or invalid request that the server doesn't understand
- **404 Not Found**: The requested content wasn't there

The fully formed initial response line would thus look something like:

```
HTTP/1.1 200 OK
```

Header Lines

After the initial request/response line, the TritonHTTP message can optionally contain zero or more key-value pairs that add additional information about the request or response (called "HTTP Headers"). Some of the keys are specific to the request message, some are specific to response messages, and some can be used with both requests and responses. The exact format of a key-value header is:

`<key><colon>(<space>*)<value><CRLF>`

The key starts the line, followed by a colon and zero or more spaces, and then the value (each key-value pair is terminated by a CRLF delimiter). <key> is composed of one or more alphanumeric or the hyphen "-" character (i.e. <key> cannot be empty). It is **case-insensitive**. <value> can be any string not starting with space, and not containing CRLF. It is **case-sensitive**. As a special case <value> can be an empty string.

Note that when storing the <key> internally, we require it to be converted into the canonical format, with the first letter and any letter following a hyphen to be upper case. For example, the canonical format for "content-type" is "Content-Type".

This spec is intentionally different from the actual HTTP/1.1 spec for the purpose of simplification. A few examples:

- `“Content-length: 324\r\n”`
  - Key: `“Content-Length”`, value: `“324”`
- `“Content-length:324\r\n”`
  - Key: `“Content-Length”`, value: `“324”`
- `“Content-length:        324\r\n”`
  - Key: `“Content-Length”`, value: `“324”`

For this assignment, you must implement or support the following HTTP headers:

- Request headers:
  - Host (required, 400 client error if not present)
  - Connection (optional, if set to “close” then server should close connection with the client after sending response for this request)
  - You should gracefully handle any other valid request headers that the client sends. Any request headers not in the proper form (e.g., missing a colon), should signal a 400 error.
- Response headers:
  - Date
  - Last-Modified (required only if return type is 200)
  - Content-Type (required only if return type is 200)
  - Content-Length (required only if return type is 200)
  - Connection: close (returned in response to a client “Connection: close” header, or for a 400 response)

You do not need to support duplicate keys in a single request. So, for example, requests will never have two Host headers.

The format for the Date and Last-Modified header is: <day-name>, <day> <month> <year> <hour>:<minute>:<second> GMT. Date refers to the current time on the server. Last-Modified refers to the time the file being accessed was last modified. For example, we could have:

```
Last-Modified: Tue, 19 Oct 2021 18:12:55 GMT
```

We have provided a utility function in starter code to help you format time with this spec.

The value of the Content-Type header could be determined using this standard library function. We provided another utility function in starter code demonstrating how to do this. The charset need not be specified in the header. For e.g Content-Type: text/html is a valid header and value.

For the ease of testing, please write response headers in sorted order / alphabetical order. (Note for real HTTP, there is no need to write headers in any particular order)

Message Body

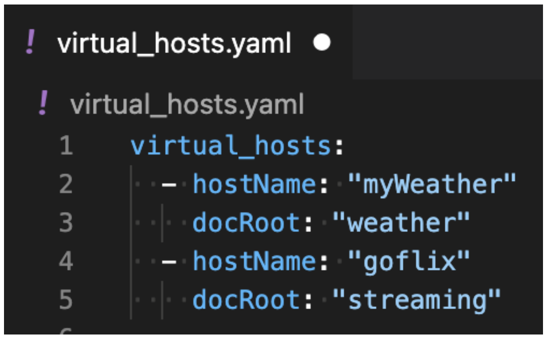
In TritonHTTP, request messages would never have a body, because we only support the GET method. In real-world HTTP, request body might be needed for some other methods.

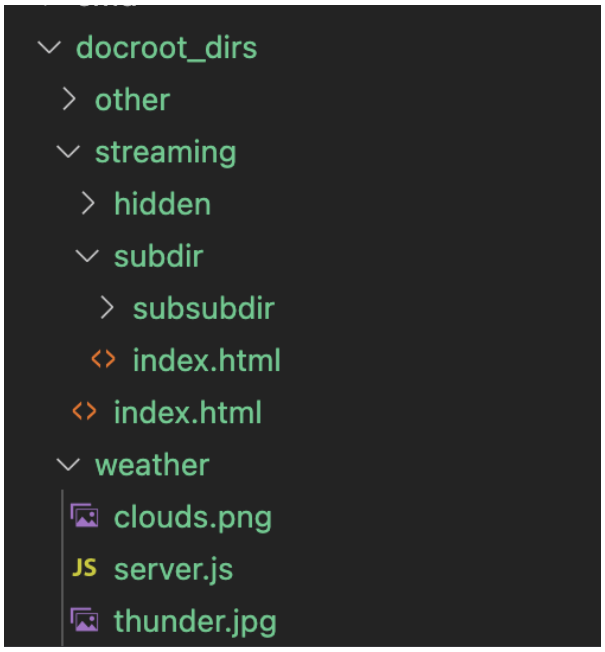
Our response messages might have a body if it’s a 200 response. In this case, the message body is basically the bytes of the requested file to serve to the client. 400 and 404 messages don’t have a body.

Virtual Hosting

In some cases, it is desirable to host multiple web servers on a single physical machine. This allows all the hosted web servers to share the physical server’s resources such as memory and processing, and in particular, to share a single IP address. This project implements virtual hosting by allowing TritonHTTP to host multiple servers. Each of these servers has a unique host name and maps to a unique docroot directory on the physical server. Every request sent to TritonHTTP includes the “Host” header, which is used to determine the web server that each request is destined for.

In the code provided, the file **virtual_hosts.yaml** provides the mapping for every hostname to its base directory. Suppose you have two different websites running on your physical machine - myWeather and goflix. Your directory structure and virtual_hosts.yaml might look something like this.


```
virtual_hosts.yaml
virtual_hosts:
  - hostName: "myWeather"
    docRoot: "weather"
  - hostName: "goflix"
    docRoot: "streaming"
```


```
docroot_dirs
  other
  streaming
    hidden
    subdir
      subsubdir
      index.html
    index.html
  weather
    clouds.png
    server.js
    thunder.jpg
```

Any request for the myWeather application should not be able to access data outside the scope of the weather directory; similarly, any request for goflix should not be able to access resources that are not inside the streaming directory. None of these services should be able to access data present outside of "docroot_dirs" or inside the "other" directory.

Implementation Guide

Mapping relative URLs to absolute file paths

Clients make requests to files using a Uniform Resource Locator, such as `/images/cyrpto/enigma.jpg`. One of the key things to keep in mind in building your web server is that the server must translate that relative URL into an absolute filename on the local filesystem. For example, you might decide to keep all the files for your server in `~aturing/cse101/server/www-files/`, which we call the document root. When your server gets a request for the above-mentioned enigma.jpg file, it will prepend the document root to the specified file to get an absolute file path of `~aturing/cse101/server/www-files/images/crypto/enigma.jpg`. You need to ensure that malformed or malicious URLs cannot "escape" your document root to access other files. For example, if a client submits the URL `/images/../../../.ssh/id_dsa`, they should not be able to download the ~aturing/.ssh/id_dsa file. If a client uses one or more .. directories in such a way that the server would "escape" the document root, you should return a 404 Not Found error back to the client. However, it is valid if the client requests a URL like `/../../../cse101/server/www-files/images/crypto/enigma.jpg`, which translates to `~aturing/cse101/server/www-files/images/crypto/enigma.jpg`, where the URL does not escape from the document root.

Golang has a function called `filepath.Clean()` that will take a file path and clean up duplicate slashes, '..'s, etc. You will likely find it useful for this aspect of the project. For example:

```
fmt.Println(filepath.Clean("/subdir1/../subdir2/index.html"))
```

Results in:

```
/subdir2/index.html
```

And in general, you shall find the path/filepath package useful.

Note that we are just bringing your attention to this function. It's ultimately up to you whether you use this function or some other functions, and how you use them.

Tips for getting started

You are free to implement this project any way you want, as long as it meets the specification. However, I wanted to pass along one potential strategy that you may find helpful.

One approach you can take to the project is to develop it in pieces, so that you can debug and test each piece as you go. That way, you don't have to try to debug a huge system, which can be very difficult!

The goal here is to make a web server that works "end to end" very quickly, even though its features are very very limited. Then you add one feature at a time, so you can test as you go. As an example, consider the following set of milestones for your server:

1. Your server accepts incoming connections and just prints out to the screen whatever the client sends. This way, you should be able to see the HTTP request and headers from the client on your terminal.
2. Next, have your server simply send to the client a "static" HTTP response no matter what request it sends. For example, an HTTP 200 OK response with a Content-Type of "text/html" and an appropriate Content-Length, where the body of the response is a very simple HTML page (I had one in my slides during class, though examples of HTML documents are easily found on the web)
3. Now, instead of sending a static HTML page back to the client, generate an HTTP 200 OK response that sends the headers back as the body of a "text/html" document. In a sense, at this point you've implemented something very similar to server3.go from chapter 1 of the Go Programming Language book

4. Now that you have a server that simply replies to the client with the request that the client sent, you can start to work on your parsing and framing code. Have your server modify the HTML you send back to, for example, highlight the URL, or some of the headers you're supposed to handle.
5. Once you've identified the URL, you can return (as an HTML document) to the client information about the file, such as whether it exists or not, or what its size is
6. At this point, you can fully implement code for 200 and 404 return types, and complete the project.

Again, this isn't the only way to implement the project but one that some might find helpful.

Experimentation

We have provided you with starter code that relies on Go's own internal http web server (Look in the Makefile for the command to start Go's internal server). You can use this to experiment and explore what a "real" webserver does under a variety of conditions. Note that the full HTTP specification is several thousand pages long, and we certainly do not expect you to implement all of that! We're only focusing on a relatively small subset of the overall protocol. As a result, you may see Go's implementation of the web server doing things we didn't ask you to (for example, Go's web server returns a header called "X-Content-Type-Options" which we don't cover in our class). You can just ignore those differences and focus on the subset of HTTP described in the TritonHTTP specification described above.

If an aspect of what we're expecting from you is unclear, please ask. We encourage you to experiment with Go's in-built web server to have an idea of a reasonable web server's behavior. But do note that TritonHTTP can behave differently from this default server. **So don't use it as THE standard.**

Testing Strategies

As you work to develop your web server, you are going to need to send it various types of input (good, malformed, etc) to ensure that it is working correctly. The purpose of this section is to give you some practical strategies for doing so.

Using Pure Command Line Tools

There are several commands that might be useful for you to use

netcat

With the -l flag, you are creating a server socket, listening on the "localhost" address. You bind to a particular port via the -p flag.

```
$ nc -l -p 8885
```

This will listen for connections on port 8885.

Try it:
Start a server session using “nc -l -p xxxx”, and then point a web browser to that host/port and see what it prints out. What do you see?

In this version, you create connection to a server (and port) as indicated

```
$ nc google.com 80
```

After you create the connection, you would be able to type in the message/request you want to send to the server. Note that if you use your keyboard as input, when you hit the “enter” or “return” key, it isn’t clear what character(s) will be sent to the server. Some platforms send a carriage return (\r), some send a new line (\n), and some send both a carriage return and a newline (\r\n). It depends on whether you’re on Linux, Mac, or Windows (or perhaps something else?)

We’re going to want more control over **exactly** what gets sent to the server. Furthermore, we’re going to want to build up a set of *test vectors*, or testing inputs, that we can send to the server. Rather than typing them out each time, we’d like to be able to read and write them to a file.

However, writing the request message each time would be tedious. So we can use:

printf & pipe

printf works just like the printf you might have used in your C/C++ program.

```
$ printf 'GET /index.html HTTP/1.1\r\nHost: MyHost\r\n\r\n'
```

Try it:
Type the above command into Linux and see what it produces. Now you have control over exactly whether you send a \r or a \n or a \r\n.

With the string literal get printed, you can pipe it into a file with ‘>’
printf 'GET /index.html HTTP/1.1\r\nHost: MyHost\r\n\r\n' > request.input

After that, you would have a file that contains the request content you want to send to the server, and you don’t need to type the content again and again.

Putting it together:
We can compose the above two steps so that we can generate a possible input

```
$ printf 'GET /index.html HTTP/1.1\r\nHost: MyHost\r\n\r\n' > request.input
```

Then we can send that input to your webserver via:

```
$ cat request.input | nc localhost 8885
```

You can even store the output of your server into a file as well!

```
$ cat request.input | nc localhost 8885 > response.output
```

hexdump

You can dig into the content of the file with hexdump. This part is optional but make sure the content you want to send is correct!

This gives you the hexadecimal form of the content in your file

```
$ hexdump [filename]
```

This gives you character form of the content in your file

```
$ hexdump -c [filename]
```

Try it!

Combining the tools for testing

1. Create a test input file that contains the request message

```
$ printf 'GET /index.html HTTP/1.1\r\nHost: MyHost\r\n\r\n' > request.input
```

2. Verify the content is what you want to send

```
$ hexdump -c request.input
```

3. Use nc to send the request and see what the response from the server is

```
$ cat request.input | nc -l 8885
```

Using Python

```
from socket import socket

# Create connection to the server
s = socket()
s.connect(("localhost", 8885));

# Compose the message/HTTP request we want to send to the server
msgPart1 = b"GET /index.html HTTP/1.1\r\nHost: Ha\r\n\r\n"

# Send out the request
s.sendall(msgPart1)

# Listen for response and print it out
print (s.recv(4096))

s.close()
```