Deliverables

You should deliver a **compressed file** containing all the code that you produced according to the specifications below. You have to include a `README.txt` file indicating how to test your program.

Pair Programming

You are required to work with your assigned partner on this assignment. You must observe the pair programming guidelines outlined in the course syllabus — failure to do so will be considered a violation of the Davidson Honor Code. *Collaboration across teams is prohibited and at no point should you be in possession of any work that was completed by a person other than you or your partner.*

1 Introduction

In this assignment, you will develop the network module and the HTTP engine of an Web Server, using any API similar to the UNIX *sockets*, which we studied in class.

As we also discussed in class, your browser makes **HTTP requests** to a web server in the following minimal format:

```
GET /monsters_inc.jpg HTTP/1.1
Host: localhost:4000
```

The server waits for those requests, parses them, extracts the filename (in this case, "monsters_inc.jpg"), and sends a response in this (minimal) form:

```
HTTP/1.1 200 OK
Filename: monsters_inc.jpg
Content-Length: 414372

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (file contents)
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

The part before the empty line is called **HTTP response header**, the part after the empty line is called **HTTP response body**, and the response altogether is called simply **HTTP response**.

1.1 Material

You can use any API similar to the UNIX **sockets** interface. Here are references for Python and Java. Using another language and library is fine, but first consult me to check if the library you want to use is similar enough to the UNIX **sockets** API in order to be allowed.

1. Python – recommended language/library
   - Guide in `https://docs.python.org/3/howto/sockets.html`
   - Reference in `https://docs.python.org/3/library/socket.html`
2. Java
   - Guide in `https://docs.oracle.com/javase/tutorial/networking/sockets/index.html`
   - Reference: use the general Javadoc class references.
3. C/C++
   - Guide in `https://www.beej.us/guide/bgnet/`
   - Reference: use the manual pages on Linux

I believe that Python has the best support for what you are trying to do in this homework. I suggest using Python unless both group members only know Java.

2 (40 pts) Network Module

In this part, you will implement some of the networking functions of the web server.

Step 1 (E, 25 pts) : Create a basic **server** using the API of your choice. The server should create a socket, bind to a user-provided port, listen to connections, and accept new connections just as in the C `sockets` library. A good test for this step is (i) have your server echo to the clients whatever is sent to it; (ii) connect to your server via `ncat`; and (iii) write messages to your server via `ncat`, expecting a response that matches.

Step 2 (E, 15 pts) : When your server produces a client-connected socket via `accept()`, immediately pass the client-connected socket to a function running in a separate thread. That way, your server can continue to accept connections meanwhile previous clients are handled. We discussed multi-threading in class. Example code on how to create a thread in Python and Java have been provided with this homework.

3 (50 pts) Implementing a one-shot GET handler

In this part, use `ncat` to send a valid HTTP GET request to your server. A valid HTTP GET request is in the form:

```
HTTP GET /path HTTP/1.1\r\nHeader1: data1\r\nHeader2: data2\r\nHeaderN: dataN\r\n\r\n
```

The `\r\n` characters are special non-printable characters, so we name/type them using a backslash. The character "\r" is just one character; the character "\n" is also just one character. The two-character sequence "\r\n" in HTTP signifies line-skipping. In other words, most systems would print the string above as:

```
HTTP GET /path/to/file HTTP/1.1
Header1: data1
Header2: data2
HeaderN: dataN

--------- (end) ----------
```

Note the empty line in the end, defined because we had two "\r\n" groups in the end of our HTTP GET request. Your server should receive the string above and:

Step 3 (M, 25 pts) Accumulate a request in a buffer until "\r\n\r\n" arrives (we will call this the "double-double"). By accumulate I mean that your server must be prepared to receive, say, half of the header without the double-double, waiting and accumulate more data until the double-double "\r\n\r\n" indeed arrives, or until your buffer that holds the header gets full. If you receive the header, call a function that will parse the header and produce an HTTP GET **response**, a further step. If your buffer where you accumulate the header fills up before you detect the double-double "\r\n\r\n", simply close the connection because you may have a malicious client connecting to your server. A good buffer size would be about 4K – that is, if your accumulated header ever gets to 4K without the double-double, close the connection.

Step 4 (M, 25 pts) Parse the string received in your HTTP GET request, and figure out the filename the client requested. If the client requested the "/path/to/file", open the local file "path/to/file" (see how "/" is relative to the server's current working directory?) and generate an HTTP reply back to the user, containing the file.

The HTTP reply should be in the following format, in case "path/to/file" can be opened:

```
HTTP/1.1 200 OK
```

```
Filename: /path/to/file (with leading slash)
Content-Length: 424242

ABCDE....
```

In the reply above, the number following "Content-length" should be replaced with the `size` of the file requested by the client; "ABCDE..." should be replaced with the `contents` of the file. Note that whenever we skip lines, we use the sequence "\r\n". So, between "424242" and "ABC..." above, we have two line skips – the double-double: "\r\n\r\n".

In case "path/to/file" cannot be opened, reply as:

```
HTTP/1.X 404 Not Found
Filename: <whatever_the_client_requested>

<HTML>
<!-- Put some nice HTML code here -->
</HTML>
```

You *can*[1] even add a 403 Forbidden reply in case the file is found, but cannot be opened due to permissions:

```
HTTP/1.X 403 Forbidden
Filename: <whatever_the_client_requested>

<HTML>
<!-- Put some nice HTML code here -->
</HTML>
```

4 (10 pts) Multi-shot GET handler

Step 5 (E, 10 pts) Make sure that after your *thread* that handles an HTTP request sends a reply, you get back to the mode where you wait for an HTTP Header. In practice, this means that whoever is connecting to your server (likely a browser) will be able to make multiple requests using a single TCP connection. This step may or may not have been already implemented by you. The step here is just making sure that your client-handling thread, as soon as replies have been sent, returns back to the mode

[1] That is, you do not need to.

where it expects the header.

5 Testing

Test your program. Launch your server at port 4000 (say), and connect with your **browser**, asking for a **picture file** that you store in the **same directory** as your server. Your browser request needs to include the port in the address, and the picture filename should follow.

```
http://localhost:4000/monsters_inc.jpg
```

In my case here, after implementing the server, I can see the picture of the movie "Monsters Inc" being loaded in each of the tabs.

6 Style Deductions

I may deduct up to 15% of every grade item to account for bad style, which includes:

1. Poor indentation;
2. Cryptic variable names;
3. Poor error treatment;
4. Naming style inconsistencies (adopt one style with your partner and stick to it);
5. Overly-complicated code.

Good luck,
- Hammurabi