# Background

TABLE OF CONTENTS

Here is some basic information on the HTTP server that you will be implementing.

## Structure of an HTTP request

The format of an HTTP request message is:

- An HTTP request line containing a method, a request URI, and the HTTP protocol version
- Zero or more HTTP header lines
- A blank line (i.e. a CRLF by itself)

The line ending used in HTTP is CRLF, which is represented as `\r\n` in C.

Below is an example HTTP request message sent by the Google Chrome browser to an HTTP web server running on localhost (`127.0.0.1`) on port 8000 (the CRLF's are written out using their escape sequences):

```
GET /hello.html HTTP/1.0\r\n
Host: 127.0.0.1:8000\r\n
Connection: keep-alive\r\n
```

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\r\n

User-Agent: Chrome/45.0.2454.93\r\n

Accept-Encoding: gzip,deflate,sdch\r\n

Accept-Language: en-US,en;q=0.8\r\n

\r\n
```

Header lines provide information about the request. For a deeper understanding, open your web browser's developer tools, then click on the "Network" tab and look at the headers sent when you request any webpage. Here are some HTTP request header types:

- **Host**: contains the hostname part of the URL of the HTTP request (e.g. `inst.eecs.berkeley.edu` or `127.0.0.1:8000`)
- **User-Agent**: identifies the HTTP client program, takes the form `Program-name/x.xx`, where `x.xx` is the version of the program. In the above example, the Google Chrome browser sets User-Agent as `Chrome/45.0.2454.93` (Or at least this was the idea back in the early days of the web. Now the User-Agent is generally an unholy mess. If you're curious as to why this is the case, the history behind it is amusing).

## Structure of an HTTP response

The format of an HTTP response message is:

- An HTTP response status line containing the HTTP protocol version, the status code, and a human-readable description of the status code
- Zero or more HTTP header lines
- A blank line (i.e. a CRLF by itself)
- The body (i.e. content) requested by the HTTP request

Here is an example HTTP response with a status code of 200 and a body consisting of an HTML file (the CRLF's are written out using their escape sequences):

```
HTTP/1.0 200 OK\r\n

Content-Type: text/html\r\n

Content-Length: 84\r\n

\r\n

<html>\n

<body>\n

<h1>Hello World</h1>\n
```

```
<p>\n
Let's see if this works\n
</p>\n
</body>\n
</html>\n
```

### Status lines

Typical status lines might be `HTTP/1.0 200 OK` (as in our example above), `HTTP/1.0 404 Not Found`, etc.

The status code is a three-digit integer, and the first digit identifies the general category of response. (For those curious, more information can be found here):

- 1xx indicates an informational message only
- 2xx indicates success
- 3xx redirects the client to another URL
- 4xx indicates an error in the client
- 5xx indicates an error in the server

### Header lines

Header lines provide information about the response. Here are some HTTP response header types:

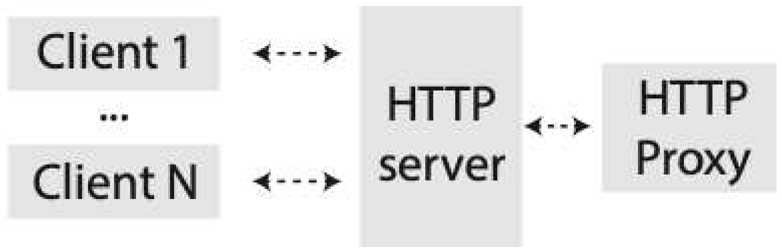
- **Content-Type**: the MIME type of the data attached to the response, such as `text/html` or `text/plain`
- **Content-Length**: the number of bytes in the body of the response

## Server outline

From a network standpoint, your basic HTTP web server should implement the following.

1. Create a listening socket and bind it to a port
2. Wait for a client to connect to the port
3. Accept the client and obtain a new connection socket
4. Read in and parse the HTTP request
5. Do one of two things: (determined by command line arguments)
    - Serve a file from the local file system, or yield a 404 Not Found
    - Proxy the request to another HTTP server. When using a proxy, the HTTP server serves requests by streaming them to a remote HTTP server (proxy). Responses from the proxy are

sent back to clients.



The httpserver will be in *either* file mode *or* proxy mode; it does not do both things at the same time.

6 Send the appropriate HTTP response header and attached file/document back to the client (or an error message)

The skeleton code already implements steps 2-4.

## Usage

Below is a description of how to invoke `http_server` from the shell. The argument parsing step has already been implemented for you:

```
./httpserver --help

Usage: ./httpserver --files any_directory_with_files/ [--port 8000 --num-threads 5]

       ./httpserver --proxy inst.eecs.berkeley.edu:80 [--port 8000 --num-threads 5]
```

The available options are:

`--files`

> Selects a directory from which to serve files. You should be serving files from the `hw-http/` folder (e.g. if you are currently in the `hw-http/` folder, you should just use `--files www/`.

`--proxy`

> Selects an “upstream” HTTP server to proxy. The argument can have a port number after a colon (e.g. `inst.eecs.berkeley.edu:80`). If a port number is not specified, port 80 is the default.

`--port`

> Selects which port the HTTP server listens on for incoming connections. Used in both files mode and proxy mode. If a port number is not specified, port 8000 is the default. If you want to use a port number between 0 and 1023, you will need to run your HTTP server as root. These ports are

> the “reserved” ports, and they can only be bound by the root user. You can do this by running `sudo http_server_rs --port PORT --files www/`.

`--num-threads`

> Indicates the number of threads in your thread pool that are able to concurrently serve client requests. This argument is initially unused and it is up to you to use it properly.

Running `make` will give you 4 executables: `httpserver`, `forkserver`, `threadserver`, and `poolserver`.

### How to test your server

In order to test your server you will need at least two different terminals, both of which are `ssh`-ed to the Workspace. Using `tmux` for this task can make it easier but it is not necessary.

In one of the terminals you should start your server, so it can start listening for connections. To do so you can run `./httpserver --files www`. You can change this command using the flags above and you can also modify the arguments to the `--files` flag. Also, once the other types of servers have been implemented, you will have to use the appropriate executable depending on which server you want to test.

*Note: You can specify the port you want the server to use for listening for connections using the `--port` flag as it is mentioned above. If there are some issues, with the basic command above, you can try changing the port your server is using using the `--port` flag. If the issues persist, please see the Troubleshooting section below.*

On the other terminal you should be able to use one of the `curl` commands that can be seen below. This will send a request to the server that you started in the other terminal.

### Accessing your server

To run your server, you should `ssh` into your Workspace and then run the commands as they are shown.

You can send HTTP requests with the `curl` program. An example of how to use `curl` is:

```
curl -v http://0.0.0.0:8000/
curl -v http://0.0.0.0:8000/index.html
curl -v http://0.0.0.0:8000/path/to/file
```

You can also open a connection to your HTTP server directly over a network socket using netcat (`nc`) and typing out your HTTP request (or piping it from a file).

```
> nc -v 0.0.0.0 8000
Connection to 0.0.0.0 8000 port [tcp/*] succeeded!
> (Now, type out your HTTP request here.)
```

After completing the GET request (directories) task, you can access your HTTP server through your web browser. You will need to use port forwarding to configure your local machine so that it can tunnel the connection to the Workspace when you try to perform an `ssh` into the Workspace. An example of how a typical port fowarding command should look like is provided below.

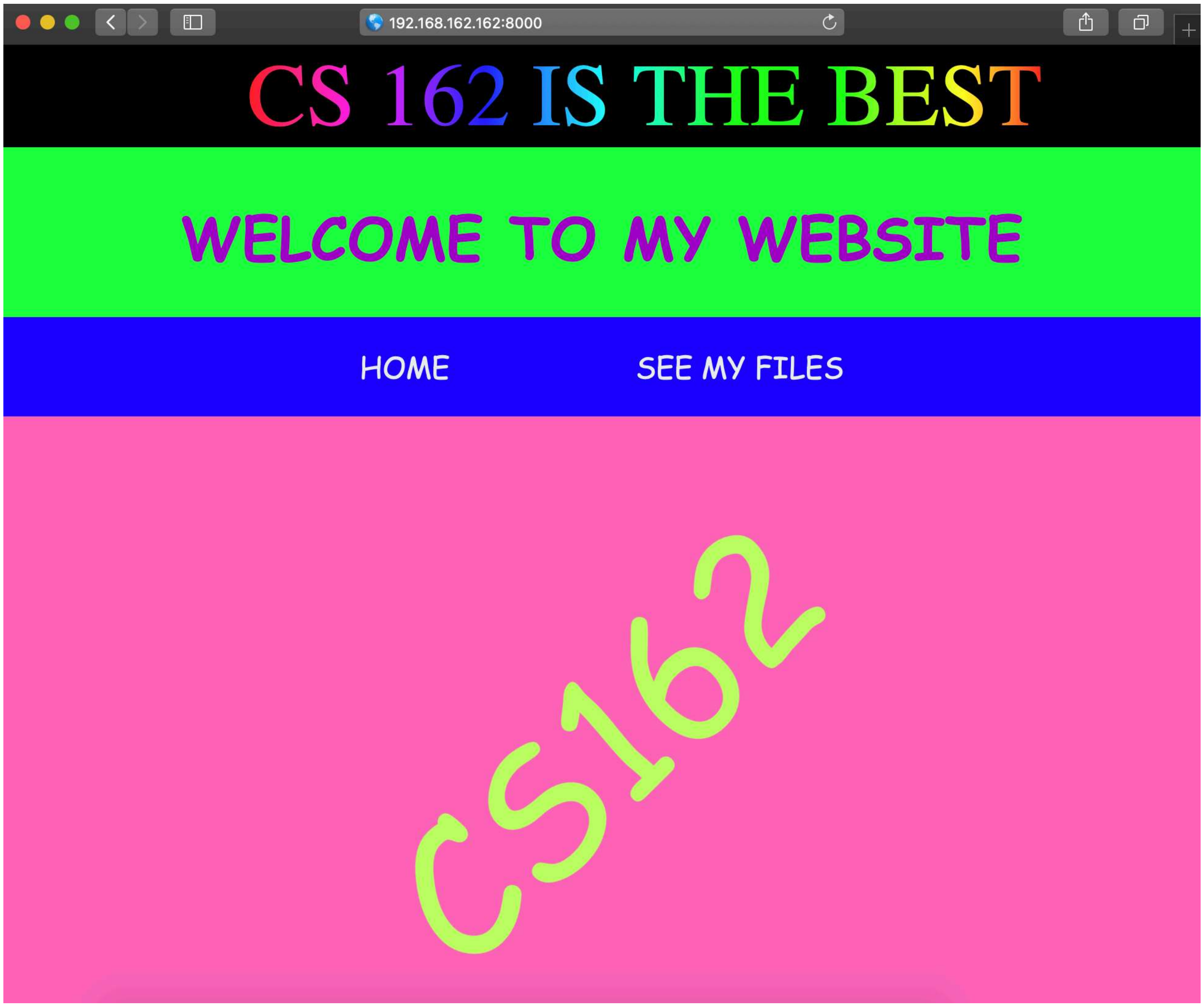
```
ssh -L local_port:destination_server_ip:remote_port ssh_server_hostname
```

In this case you should execute the command from your local machine and the command should be similar to the following:

```
ssh -L 8000:127.0.0.1:8000 workspace
```

*Note: If you are using port 8000 in your machine, you will have to choose another port to tunnel the connection. Any port higher than 1024 should work.*

After configuring your tunnel, you should be able to type it into your browser using your localhost connection and the port you used selected for the tunnel.



## Troubleshooting

### Failed to bind on socket: Address already in use

> This means you have an `httpserver` running in the background. This can happen if your code leaks processes that hold on to their sockets, or if you disconnected from your Workspace and never shut down your `httpserver`. You can fix this by running `pkill -9 httpserver`. If that doesn't work, you can specify a different port via `--port`.

### Failed to bind on socket: Permission denied

> If you use a port number that is less than 1024, you may receive this error. Only the root user can use the "well-known" ports (numbers 1 to 1023), so you should choose a higher port number (1024 to 65535).