CS 118 Computer Network Fundamentals

Project 1: Web Server Implementation Using BSD Sockets

Amit Mondal: (UID) 804746916
Cade Mallett: (UID) 804826951

1. Give a high-level description of your servers design.

The server opens a socket and then completes the address with a port number that is passed in as a command line argument to the program. The socket listens on the provided port for an HTTP request from a browser. Within an infinite loop, a call to accept() blocks until it receives an incoming TCP connection, and the HTTP requests that are received at the socket are read into a buffer and then dumped to stdout.

The request text is then parsed into a struct that contains field for the HTTP method, the version, and the request target, as well as two vectors. The first vector contains all the names of the different headers, while the second contains the information in those fields.

From this struct, it is simple to return the filename that is the target of the GET request. This filename is sanitized (to account for escaped spaces) and then passed along with the socket file descriptor to a function that sends the file. We use stat to get the file size and some basic string functions included in std::string to extract the file extension and store it in an enum. A struct representing the HTTP response is allocated, and if the file doesn't exist, the struct's header is simply a "404 File Not Found," and the body is a brief HTML message saying the file could not be found. If the file is found, we use the file extension to determine the "Content-Type" field for the response header, the header is sent, and then the file contents are read in using std::ifstream to a buffer, and then written in chunks using send(). Finally, we close the socket and block on the next call to accept(), thus waiting on the next incoming HTTP request.

2. What difficulties did you face and how did you solve them?

We initially faced difficulty handling the different types of files in the project. Particularly, it was unclear what a binary file was and how to handle it. Eventually, we decided to use the "Content-Type: application/octet-stream" header for binary files, which causes the browser to simply download binary files, which we figured would work for any binary data. We also had some minor issues keeping the server open for multiple HTTP requests because we did not realize that we needed to make a new call to accept() within our main loop. Using C++ also caused some minor issues, as some features required the -std=c++11 gcc flag, which was not supported on the old version of Ubuntu in the preset VM provided. Fortunately, the latest version of Ubuntu supported this without issue.

3. Include a brief manual in the report to explain how to compile and run your source code (if TAs cant compile and run your source code by reading your manual, the project is considered not to be finished).

To compile, simply type 'make' into the command line. The Makefile compiles with g++ and the flags "-g -Wall -Wextra -Wno-unused-parameter."

The Makefile also supports a 'clean' target to remove generated files and directories, as well as a 'dist' target to generate the submission tarball.

To run, compile and then type

```
$ ./server (some_valid_port_number)
```

Then, you can visit “localhost:some_valid_port_numer/{some_file}” from a browser, and the server will respond with the requested file if it exists, and will also print out the browser’s HTTP request.

4. Include and briefly explain some sample outputs of your client-server (e.g. in Part A you should be able to see an HTTP request). You do not have to write a lot of details about the code, but just adequately comment your source code. The cover page of the report should include the name of the course and project, your partners name, and student id

Here is a sample HTTP request sent from the browser and printed to the console by the server:

```
GET /test.jpg HTTP/1.1
Host: localhost:1200
Connection: keep-alive
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_2) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/63.0.3239.132 Safari/537.36
Upgrade-Insecure-Requests: 1
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
```

As you can see, the browser sends a GET request for the file ‘test.jpg’. It includes the HTTP version (1.1), the server host, whether or not the server should keep the TCP connection alive, and some information about the user agent. It also lists the formats it would accept in a response, as well as the encoding and language. A response sent by the server would have headers like this:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 67244
Content-Type: image/jpeg
```

And then the server would send the actual file data in the response body. If the file doesn’t exist, the response would look something like this:

```
HTTP/1.1 404 Not Found
```

<b>404 File Not Found </b>