COM SCI 118 Spring 2019
Computer Network Fundamentals

Project 1: Web Server Implementation Using BSD Sockets

Jesse Catalan (204785152)
Ricardo Kuchimpos (704827423)
University of California, Los Angeles

Web Server Implementation Using BSD Sockets

Abstract

We designed and developed a web server written in C that uses BSD sockets for networking and is capable of serving simple GET requests. The server listens for incoming HTTP requests from a client (i.e., a web browser), parses the necessary information from the header, and sends an HTTP response containing the requested file to the client. In addition, the web server also dumps the HTTP request to the console, allowing us to inspect the incoming requests. In this implementation, there are two status codes, 200 for successful requests and 404 for requests that cannot be satisfied due to a missing file. From our testing, we observed that the server successfully handles small and large files (upwards to 100 MB) of different media types, correctly serves the requested file (even if the filename has a space or there is a difference in casing), and displays a 404 page for requested files that do not exist on the server.

1. Background

1.1. HTTP

The hypertext transfer protocol is the set of rules governing internet communication between servers and clients. In a simplified view, clients initiate communication with the server and request a set of files while the server responds to requests by delivering those files or raising an error. However, under this basic framework is a strictly defined system of messages enabling connection, file transfers, and error recognition.

HTTP is built upon the Transfer Control Protocol. When the client wishes to begin communication with the server, the client is actually initiating a TCP connection with the server at the server's specified port. On the server end, the specified port waits for client connections and accepts the connection, with a notification sent back to the client upon success. Once this basic TCP handshake has resolved, the client is free to request object files from the server, starting with the index file. HTTP's request format is as follows:

```
GET /filename.extension HTTP/1.1
Host: 127.0.0.1:33623
Keep-Alive: 300
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh;
Intel Mac OS X 10_14_4)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/73.0.3683.103 Safari/537.36
Accept:
text/html,application/xhtml+xml,applic
ation/xml;q=0.9,image/webp,image/apng,
*/*;q=0.8,application/signed-exchange;
v=b3
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
```

The request message begins with the request line, which indicates the method type (GET), the file's URL, and the HTTP version. What follows is a sequence of header lines terminated with a line with a single `CRLF`. The information detailed by each header is as follows [1]:

Host	the server's IP address or domain name
Connection	whether the TCP connection should be kept alive or closed after transmission
Keep-Alive	how long the connection should be kept alive for
User-Agent	the client's browser and operating name and version
Accept	which content types the client is able to understand

Accept-Encoding	which compression types the client is able to understand
Accept-Language	which human languages the client is able to understand
Upgrade-Insecure-Requests	when set to 1, the server sends authenticated and encrypted responses

Upon reception, the server is able to parse and handle the client's request. In the case that the client wants a file back, the server sends a response message with the asked for content:

```
HTTP/1.1 404 Not Found
Content-Type: text/html
Content-Length: 156

<!DOCTYPE html>
<html>
    <head>
        <title>Error 404</title>
    </head>
    <body>
            <h1>404. File not found on
this server.</h1>
    </body>
</html>
```

The response message begins with a status line indicating the HTTP version, status code, and status phrase. In cases where the requested file was sent to the client successfully, the status code is 200 and the status phrase is "OK". If the server fails to find the requested file, it will instead send a 404 status code with the phrase "Not Found". The status line is followed by more header lines holding the following information:

Content-Type	the file's media or mime type
Content-Length	the size of the file in bytes

A `CRLF` line then separates the header lines from the file's actual content.

Upon sending the response, the server can either terminate the connection (non-persistent TCP) or keep it open (persistent TCP), as in this design.

1.2. BSD Sockets

An Internet socket represents a local endpoint between the server and its clients. This is typically in the form of a file descriptor in Unix systems. The Sockets API provides several functions to work with these. In order to have communication at all, a socket must be set up with a `socket()` system call. This provides a file descriptor to the newly-created socket. However, in order for this socket to be useful, it must be bound to an address, which in this case is the address of our server (IP and port number). This is accomplished with a call to `bind()`. Now the server is ready to begin listening on the socket. We then make a system call to `listen()` specifying the socket file descriptor, and a backlog which is the maximum number of requests that can wait in the queue. Finally, the server can use the accept() system call to accept a connection on the socket, returning a file descriptor for the accepted socket. We can then use this new socket file descriptor to send and receive data between the client and the server.

2. Manual for Setting up the Server

To compile our source code, simply run `make` from the command line. This will compile the source code in *webserver.c* into an executable named *webserver*. To run the executable, use `./webserver` in the command line, which will set the local host as the server and search for requests on port number 33623.

The server will accept and respond to requests made from the local host's browser. To make requests to the server, run any browser on the host computer and use the following URL:

```
http://127.0.0.1:33623/filename.ext
```

The IP address in the URL ensures the request goes to the local host on port 33623. The client can then request for a file in the executable's directory by replacing

"filename.ext" with the desired filename and its appropriate extension.

For the testing purposes, the client can request from a small collection of sample files. These are:

error_404.html	404 error page; default page sent upon 404 error
screenshot.png	desktop screenshot of palm trees
screen shot.png	same desktop screenshot running VirtualBox
test.txt	small text file with test data
webpage.html	"Hello World" html file
bird404.png	image of bird used in the 404 error page

In the case that the user wants to get a 404 error, the user can request for a file not in the directory.

3. Design and Implementation

The entry point of the source code is the setup of the socket which will be used as the endpoint for communication, the binding of the socket to an address, opening the socket for listening, and accepting incoming connections from the queue. For every incoming request, the HTTP request header is parsed to extract the requested filename. Then the filename must be matched with a file on the server, not taking into account case sensitivity.

Once the file is found, the server assigns the Content-Type based on the extension, or if no extension is found, it is treated as a regular binary file. The server also determines the size of the file in bytes to set the Content-Length. However, if the requested file cannot be found on the server, a special "Not Found" page will be sent instead. Now that all the essential information to service the request has been extracted and processed, the HTTP response header is built from the status code (200 or 404 depending on whether the file could be successfully located), the content type, and the content length.

The server then sends this header, followed by a blank line, which in turn is followed by the body (the file contents). The body is stored in a byte buffer that is dynamically allocated in order to account for different file sizes. An advantage of this approach is that as opposed to using a large fixed size buffer, we can save a considerable amount of space when handling small files.

Once the server has responded to the client, the connection is then closed, meaning that a new connection must be opened for each request. For example, if there are other local objects such as images referenced on an HTML web page, the browser will send a separate GET request for each of those.

4. Testing

In order to ensure the integrity of the server, extensive testing was carried out. The simplest test case was requesting a small text file, consisting of only a few characters.

We also tested to make sure that an HTML page would properly render, and if it had linked images, that it would make separate requests for those as well. The browser also succeeded in displaying images served by the server. We also had to make sure that the requested files were being compared to the files available on the server without considering differences in casing. For example, a request to the file `SAMPLE.HTML` should match the file `sample.html` on the server.

We also needed to test large files. We used the following command to generate a ~100MB binary file filled with random bytes:

```
dd if=/dev/urandom of=largefile
bs=1048576 count=100
```

The `diff` program was then used to make sure that the file downloaded through the web browser exactly matched the file on the server. A difference in even one byte would have meant that the file got corrupted.

5. Challenges

Since we chose C as the language to implement the server in, we encountered several language-specific challenges. One of these was string processing. While C does provide a useful set of functions to do this, string processing was not as clean as it would otherwise be in a language like Python. For example, extracting the filename from the HTTP headers required additional work to account for filenames that contained spaces. However, when the HTTP request is sent for such a file, the space is encoded into a `%20` sequence. This led to difficulty in identifying which spaces in the GET line were part of the filename. One of the ideas we had was to create a Regular Expression that would match only the filename, but it was simpler to iteratively scan through the string using two pointers, one at the beginning and one at the end, to consume characters that were not part of the filename.

When testing with the Chrome browser, we noticed that sometimes the server received empty requests, which would cause the server to crash due to improper handling of zero-length requests. The fix was as simple as adding a conditional check on the request length. Interestingly, we could not reproduce this issue with Safari.

During testing, we also encountered issues where the port would still be in use after a crash or when the program encountered a Ctrl-C or Ctrl-Z. Sometimes this would require issuing a manual `SIGKILL` to the process if it was still running in the background. This could be fixed with more robust resource management.

6. Conclusion

Through the development of this project, we learned about the theoretical background of HTTP and BSD sockets and how to implement them in a C program to build a web server. The webserver we implemented is capable of serving simple GET requests and displaying the HTTP requests on the console to give us a better understanding of how a client such as a web browser interacts with a server. A possible improvement for the project is more robust error-handling, to handle failures to open files and possible null pointers.

7. References

[1] Guzel, Burak. “HTTP Headers for Dummies.” *Code Envato Tuts* , 2 Dec. 2009, https://code.tutsplus.com/tutorials/http-headers-for-dummies--net-8039