CSE 130 - Asgn1 Design Document

Daniel Chen (dchen61@ucsc.edu)
Daniel Xiong (dxiong5@ucsc.edu)

1 Design

`httpserver.cpp` is meant to be a simple HTTP server that handles only GET and PUT commands. The server will create, store, read, and write files in the server directory. The server will also follow the HTTP protocol, which means it will parse HTTP headers.

First, `httpserver.cpp` will parse the command line arguments and get the specified hostname/ip address and the optional port number. Then, it will initialize the socket using the `socket`, `bind`, and `listen` syscall functions.

There may be multiple connection requests sent by a client, so the server handles them one by one, in the outer `while` loop, using the `accept` syscall function. For each connection, the server will receive (`recv`) requests and/or data from the client until no more bytes are sent. Once no more bytes are sent, the server will move onto the next connection in the queue.

Once the server receives a request, the server will parse the HTTP header and extract three things: the command type (PUT/GET), the file name, and the content length. If the request is a PUT request, the server will first open a file for writing, then receive and write data sent from the client until the content length is reached or until there are no more bytes sent. If the request is a GET request, the server will open a file for reading, then reads and sends the data to the client.

The server will also have to send HTTP responses to the client. If the server successfully does a PUT request, it will send a 201 response to the client. If the server successfully does a GET request, it will send a 200 response to the client. If the server receives a request from the client that is syntactically incorrect, it will send a 400 response. If the server cannot open the file specified due to permissions, it will send a 403 response. If the server cannot open the file because it doesn't exist, it will send a 404 response. Otherwise, all other errors will result in a 500 response.

2 Data Structures

2.1 `header`

`header` is a struct with members `char* command` (PUT/GET), `char* resource_name` (File name), and `int content_length` (Size of data). We create objects of type `header` to represent the HTTP request header.

2.2 buffer

buffer is a char array initialized with a size of 4KiB, intended to be used as a buffer for receiving data from the client.

3 Functions

3.1 struct header parseHeader(char buf[])

We use the parseHeader function to parse through a HTTP request header string (char buf[]) and extract the command (PUT/GET), resource_name (File name), and content_length (Size of data). If no content length is provided, content_length would be equal to -1. For the output, an object of type header is returned.

3.2 int send_response(int comm_fd, int response_num, int content_len, char* resource_name)

We use this function whenever we want to send a response. This function takes in as input the file descriptor from accepting a connection (int comm_fd), the response number we want to send (int response_num), the content length from the header (content_len), and the filename from the header (char* resource_name). The response numbers we can send are: 200, 201, 400, 403, 404, and 500. We compare the response number to one of the HTTP status codes and respond appropriately. The function will return 0 upon completion.

3.3 unsigned long getaddr(char* name)

We use this function to convert the hostname/ip address string given as a command line argument (input) into an address that is readable by the sockaddr object (output).

4 Testing

For testing, we first were using the nc command to make sure that the connection between the client and server was correctly made. We could type whatever we wanted to and the server would output the identical text, like an echo server. We also tested the server by switching between using a hostname and an IP address to make sure that the server could run on both. Then, we tested our parseHeader function to make sure we were correctly extracting the kind of request, the filename, and the content length if provided. We made up simple headers and printed what was extracted to stdout. After parsing the header, we made sure that our server was correctly sending the right responses in case the filename provided did not contain exactly 10 alphanumeric values.

Once we implemented code to handle PUT requests, this is where we started using the curl command. We first created several test files with different sizes. We created an empty file, a small file that contains less bytes than our buffer size, a large file that contains more

bytes than our buffer size, and a binary file. We made sure the server would save these properly. Then, we tested our server by reading/writing from files that exist in different directories, opening files with no read/write permissions, opening files that didn't exist, and stopping the connection in the middle of PUT requests. The server should keep running and act appropriately by either completing the request or sending the correct error response and then closing the connection with the client.

5 Assignment Question

1. *What happens in your implementation if, during a PUT with a Content-Length, the connection was closed, ending the communication early? This extra concern was not present in your implementation of dog. Why not?*

If the connection was closed during a PUT with a Content Length, our implementation just closes the connection and waits for another request. This extra concern wasn't present with dog because we weren't dealing with a client-server protocol. There was no need to worry about any connections or requests/responses sent between the client and server.