Design Document: HTTP Server

Eric Tang
etang12

1 Goals

The goal of this assignment is to create a functioning HTTP server that can handle PUT, GET, and HEAD requests. These requests consist of a header with information for classification purposes, and a body that contains any kind of information. A PUT request is used by a client to send a file over to a server where it will be saved. A GET request is used by a client to retrieve a file from a server. A HEAD request functions similarly to a GET request, but slightly differs in the fact that it only retrieves the response from the server and the file is not sent. All three requests send a response from the server to the client that provides information on the files being sent and received.

2 Design and Implementation

2.0 Design Flow My thought process going into the program was simple. I first needed to determine what kind of information was being sent to the server by the client. This highly depended on the steps I would take in receiving and reading the message from the client. After reading the request and understanding what it was asking for, my program would need to move onto processing the request. It would need to carefully execute the operations that the request was asking for. Finally, after the operation had been carried out by the server, it would then need to send a response back to the client. I chose to follow a 3-step plan:
Read message and determine what kind of request it is ------->
Process the request and execute the operations asked by the client ------->
Send appropriate response back to the client by the server -------->
end

2.1 Main Function The main function of this program works to set up the necessary requirements to open a socket and start processing requests between a client and server. The port number will be inputted through the command line by the user which the socket will open. The server socket is set up using the struct sockaddr_in which uses fields like sin_family, sin_port,

sin_addr.s_addr. After initializing and creating the server socket, it must be bound to the server address using bind. All that's left is to listen for incoming connections since the server socket is ready. Next, the server needs a client to connect to. A client socket is created in similar fashion to the server socket but toned down. The client has an address created for it from a struct sockaddr. From there, the client socket is established and accepted using the accept function. Now that the main objects (sockets) are created and functioning, the main function is complete where it also calls other functions to read and process HTTP requests.

```
Function main(argc, argv):
   Process sockaddr_in with server information;
   Create the server socket;
   Bind server address to the server socket;
   Listen for incoming connections;
   Process sockaddr for client;
   Initialize struct httpObject;
   while(true){
         client socket accepts connection from server;
         read http request;
         process and send http request;
         clear statically allocated memory;
         close client socket;
   }
end
```

2.2 read_http_request The read_http_request function in my program is a very essential piece of code. It is used to receive a HTTP request from the client and determine what kind of request it is specifically (PUT, GET, HEAD) and split it into parts to allow for parsing of the data which significantly reduces the complexity of the program. For context, a HTTP request comes in different formats depending on the type of method being used. "PUT /filename HTTP version\r\nContent Length: X\r\n\r\n", is a valid header for a PUT request. The body follows after the double CRLF where data about the file is stored. HEAD and GET requests share the same formatting of a request with the exception of their method name while being different from a PUT request because they do not contain extra header information about Content-Length and such; "METHOD /filename HTTP version\r\n\r\n". In all three requests, the double CRLF signals the end of a header. This function is required to search through these types of requests and find any abnormalities within them. If an invalid request is found through an extensive error checking process, a multitude of responses are sent back to the client depending on what kind of error was pinpointed. The entire program depends on the header being parsed correctly in order

to assure proper execution to send a correct response to the client. There are many edge cases to deal with in this function, just to list them:

only ASCII characters that range from a-z, A-Z, 1-9, -, _, are allowed for a file name
- this would warrant a response of 400 Bad Request

deal with situations where body may be found within the header
- this would warrant a response of 400 Bad Request

header can be sent separately, so function needs to confirm that full header has been received
- this would warrant a response of 400 Bad Request

header cannot be larger than the 4096 buffer size
- this would warrant a response of 400 Bad Request

filename cannot be longer than 28 characters
- this would warrant a response of 400 Bad Request

The requests are first received by using the recv() function to read a file descriptor which is the client socket. From here, the bytes received from the client will allow the function to parse the request sent. The requests are parsed using string functions, mainly strstr() and strtok() to split up and segregate individual parts of the request to easily differentiate and store them in an httpObject struct (2.3) using sscanf().

```
void read_http_request(ssize_t client_sockd, struct httpObject* message){
    Receive message from client socket and store into message->buffer;
    Use strstr() and strtok() to split up the message,use sscanf() to store in various message fields;
    Error check for the different problems listed earlier and send proper response back to client;
    end
}
```

2.3 httpObject struct The httpObject struct is built to store the important information from a HTTP request. It makes it simpler to keep track of certain components of a request, and provides easy access to all these components for use in other functions. The organization that this struct provides is immensely helpful in implementing the other functions. An httpObject is comprised of six things:

char method[5] - To store the method type
char filename[29] - To store the file name
char httpversion[9] - To store the http version
ssize_t content_length - To store the length of the body
int status_code - To store the an identifying status code (200, 201, 400, 403, 404, 500)
uint8_t buffer[4097] - To store the buffer used the actual contents of the request (header)

2.4 if_exists && get_file_size && is_regular_file I decided to put all three of these functions under one section in the documentation because they all rely on the same C function to work properly. They all utilize stat() and each returns a different result necessary by another function in the program. stat() provides a range of functionality which is perfect for this program. It gives the program important information about a file that has many applications and uses. These functions that use stat() are pretty self explanatory functions but I felt the need to include them as they all have their own purpose. if_exists is used to determine if a file exists or not. get_file_size returns the size of a file. is_regular_file determines if a file is regular, meaning that it isn't a directory or anything else. These functions are used in the processing portion of the program, and each play their own role in helping achieve the right outcome from the processing of the http request. I broke them up into separate functions so that I would not have to call stat() multiple times in one function, and make the code more readable.

```
int if_exists (char *filename){
    create stat struct;
    return stat() call on file name; //if 0 file exists else doesn't exist
}

int get_file_size (char *filename){
    create stat struct;
    return size of filename with stat() call
}

int is_regular_file (char *filename){
    create stat struct;
    return S_ISREG() on stat() to determine is regular file;
}
```

2.5 process_request The process_request function is another major part of the program. This function is where all the messages received and parsed in read_http_request are sent in order to actually execute the request. Processing the request is a bit complicated as there are still many cases to consider and error responses that need to be checked before and during the actual processing. There are a couple of big issues to look out for: 404 (Not Found), 403 (Forbidden), and 500 (Internal Server Error). These errors must be sent back to the client when the request asks to do an operation that is out of the server's control. A 404 code happens when the file requested by the client does not exist. A 403 code occurs when the file requested by the client cannot be accessed due to having no permissions. A 500 code occurs during the failure of a

syscall within the function (read() or write() fails) such that the program actually fails. Now, the functionality of this function depends on what kind of request is parsed and sent.
If it is a PUT request, then the client wishes to send a file and store it in the server. In this case, this function will determine if the method name is PUT. Here, it will check if the file exists and determine an appropriate status code. If the file exists but has no write permissions, then a response to the client is sent with a 403 Forbidden. If not then the status code is set to 200 Created to acknowledge that the file will be overwritten. If the file doesn't exist, then the status code is set to 201 Created to acknowledge the creation of a new file. The file is then opened using open() to create it if it doesn't exist, or overwrite it. The file is then received using recv() and stored in a buffer to eventually be written. The contents of the file are then written to the file opened and stored within the server. A response message is sent back to the client to let them know if the file had been created or overwritten.
If it is a GET request, then the client wishes to receive a file from the server and display it. It follows the logic of a PUT request similarly in which it checks if a file exists, if it has permissions and sends a correct response after checking. However, GET reads from the file to get the data from it. First, the client sends a response to the client determining if it is a successful request or not (200 or 201 status code) and the content-length. From there, it writes the entirety of the data to the client, so that the information would be printed to the client's terminal.
If it is a HEAD request, then the client wishes to only receive the header information about the file, but not receive the contents of the file. It follows the same logic as a GET except it does not implement the writing portion of the code.

```
void process_request(ssize_t client_sock, struct httpObject* message){
    if the method is PUT{
        check if file exists then check if it has write permissions{
            if no write permissions then send 403 response else set status code to 200;
        }
        if file does not exist then set status code to 201 and continue;
        open the file to store in server and start receiving the data from the body of the request;
        write all the data from the body into the file;
        send appropriate response back to the client with correct status code;
    }
    if the method is GET{
        check if file exists, if it doesn't exist then send 404 Not Found response to client;
        check if file is regular, if it isn't then send 403 Forbidden response to client;
        check if file has read permissions, if it doesn't then send 403 Forbidden response to client;
        open the file from the server and start reading the data from the file;
        write all the data from the file and send it to the client along with a proper response;
    }
```

```
    if the method is HEAD{
        check if file exists, if it doesn't then send 404 Not Found response to client;
        check if file has read permissions, if it doesn't then 403 Forbidden response to the client;
        open the file from the server and start reading the data from the file;
        send a response back to the client containing the header information about the file;
    }
end;
}
```

Program is implemented and finished.