Design Document: Assignment 1
CSE130: Principles of Computer System Design

Rahul Mahendru

1 Overview

Assignment 1 is an implementation of a "simple single-threaded HTTP server". The server will be designed to respond to simple GET, PUT and HEAD commands.

1.1 Goals and Objectives

The goal of the assignment is to create a "simple single-threaded HTTP server", which can respond to simple "GET and PUT commands to read and write files respectively". The program will also implement the HEAD command which will provide information on the existing files. The program will build a `httpserver` executable using `make`.

1.2 Statement of Scope

The HTTP request consists of a line of request composed of ASCII text followed by an empty line. The first line of the request consists of the action and one or multiple header arguments, which is a key-value pair "separated by a semi-colon(:)". The only header that we are interested in, for this assignment, is the `Content-Length` header, which indicates how much data follows the request. The new lines are encoded as `\r\n`, the end of requests as `\r\n\r\n` and all the filenames are preceded by a `/`.

Specifically, the above mentioned syntax is only required for the PUT command, while the GET and HEAD commands do not require any headers in the requests. The commands respond with a "response", "which is a response line followed by necessary header and optionally followed by data". The server is expected to handle any errors that may arise without crashing. Specifically, the server needs to handle the following error codes: `200(OK), 201(Created), 400(Bad Request), 403(Forbidden), 404(Not Found) and 500(Internal Server Error)`.

1.3 Major Constraints

Some of the constraints that need to be followed while implementing the functionality of the program are as follows:

- Valid resource names must be up to 27 ASCII characters long. "They must only consist of the upper letters of the (English) alphabet, the digit 0-9, dash (-) and underscore (_)." Any character besides these 64 characters is invalid.
- `Content-Length` header is always present in a PUT request.
- The request will not be longer than 4KiB, though the data may be longer.
- All file I/O is done using file descriptors `read()` and `write()`. The buffer is not allocated more than 32KiB of space.
- `FILE *` operations must not be used.

2 System Architecture and Design

This section elaborates on the procedures, functions and structures used in order to achieve efficient functionality. The program is to be implemented in the C language.

2.1 Setup HTTP Server Connection

In order to set up and use a HTTP server connection at address X and port Y, the `socket, bind, listen, accept, recv, send, read, write and close` functions are used. The functions are used from the starter code. The procedure to be followed is:

1. Create the socket:

```
server_sockd = socket(AF_INET, SOCK_STREAM, 0);
```

2. Identify the socket:

```
ret = bind(server_sockd,
           (struct sockaddr *) &server_addr,
           addrlen);
```

3. Wait for an incoming connection on the server

```
ret = listen(server_sockd, NUM_OF_QUEUED_CONNECTIONS);
// Loop for client requests
client_sockd = accept(server_sockd,
                      &client_addr,
                      &client_addrlen);
```

4. Send and receive messages

```
// recv() to recieve messages, and use
// send() to send the response
read_http_response() -> parse request
process_request() -> do PUT, GET or HEAD
if (error_code encountered)
    send_error_response()
```

5. Close the socket

```
close(client_sockd);
close(server_sockd);
```

2.2 Receive and process HTTP message

2.2.1 read_http_response

The message is received from the client. Then the request is parsed and the data is stored in the associated `httpObject`'s associated fields. This is followed by checking if the request values are valid.

```
bytes = recv(client_socket, buffer)
parse(buffer) -> httpObject(message) // Header
if (check_method() == false)
    send 400 bad request;
if (check_filename() == false)
    send 400 bad request;
if (check_httpversion() == false)
    send 400 bad request;
Store content-length header if present
check for bad content length: send 400 bad request

// For PUT request
if content-length > 0 {
//parse the data following the request
buff = strstr(buffer, "\r\n\r\n");
```

```
message->buffer <-- buff;
buffer_bytes = len(buff)
set status code = 200
```

In order to check the validity of the request:

1.
```
check_method(){
if (method != PUT, GET or HEAD)
    send 400 bad request
}
```

2.
```
check_filename(){
if (filename[0] != '/') return false
if (filename > 27 ASCII characters) return false
check if filename contains any characters other
than (A-Z),(a-z),(0-9),(-) and (_)
}
```

3.
```
check_httpversion(){
if (httpversion != 1.1)
    send 400 bad request
check if filename contains any characters other
}
```

2.2.2 process_request

The message `httpObject` is processed by calling the specific method to perform the requested action. It processes the requested method and constructs and sends responses back to the client.

1. `server_get` function in order to implement the `GET` function

```
server_get() {
    if (file doesnt exist) send 404 error
    if (file == directory) send 403 error
    if (file permissions are invalid) send 403 error
    open(file, RD_ONLY)
    if (open fails) send 404 error
    if (file and request == valid) :
        set content-length
        send response 200 OK
    loop{
```

```
            read (file -> buffer);
            send (buffer -> client);
            }
        }
        return;
    }
```

2. `server_head` function in order to implement the `HEAD` function

```
server_head() {
    //check validity of file
    if (file doesnt exist) send 404 error
    if (file == directory) send 403 error
    set content-length;
    send 200 OK response with content length
    return
}
```

3. `server_put` function to implement the `PUT` function

```
server_put() {
    if (file exists and == directory)
        send 403 error
    open (file , O_RDWR)
    if (open fails) send 403 error
    if opened :
        if (buffer has data):
            write(file <- buffer);
        loop {
            recv(client_sockd -> buffer)
            write(file <- buffer)
        }
    send 201 created request with content-length = 0
    return;
}
```

2.2.3 send_error_response

After reading and processing requests, If any error arises, an error response is sent according to the error code.

```
send_error_response() {
```

```
    check for error status code:
        send corresponding error response
}
```