Design Document: HTTP server

Julia Sales
CruzID: jesales

1 Goals

The goal of this program is to implement a HTTP server that will respond to GET and PUT commands. These commands will correlate with read and write files. The HTTP server will store files in a directory persistently. Our program receives and sends files via http. The server will respond back with messages like 200 and 404.

2 Design

There are six parts to this design. First, we have to create a socket, which will create the connection between the server and the client. We then need to read the data from the connection. From this read, we need to determine which command this is calling (GET or PUT?). After that we need to code accordingly to which status code will appear based on the conditions. We also need to figure out how to deal with invalid file names and file names that are valid but not not exist. We need to figure out the errors for these.

There are four parts to this assignment. First, we have to create a socket, which will create the connection between the server and the client. We then need to parse the header. After parsing the header, we need to determine what we need to do if the request is a GET or a PUT.

2.1 Setting up the Socket

Creating a sockaddr_in

```
1. struct hostent *hent = gethostbyname(SERVER_NAME_STRING /* eg “localhost” */);
2. struct sockaddr_in addr;
3. memcpy(&addr.sin_addr.s_addr, hent->h_addr, hent->h_length);
4. addr.sin_port = htons(PORT_NUMBER);
5. addr.sin_family = AF_INET;
```

Creating a socket

```
1. int sock = socket(AF_INET, SOCK_STREAM, 0);
2. if no connection, when sock is 0
3.       |  Error: In socket, no connection
```

Socket Setup for Server

Note: "cl" is now another socket that you can call read/recv and write/send on to communicate with the client

```
1.  int enable = 1;
2.  setsockopt(sock, SOL_SOCKET, SO_REUSEADOR, &enable, sizeof(enable));
3.  bind(sock, (struct sockaddr *)&addr, sizeof(addr));
4.      | if not being able to bind - Error: cannot bind
5.  listen(sock, 0);
6.  int cl = accept(sock, NULL, NULL);
7.  if cl is < 0
8.      | Error: Cannot accept
```

2.2 Parse Header

```
1.  create buffer with size 327868
2.  header1 = GET or PUT <- the request
3.  header2 = the 27 ascii string <- the file
4.  header3 = HTTP/1,1
5.  header4 = "Content-length: %d"
6.  int valread = read(cl, buffer, size of buffer) <- buffer = header
7.  sscanf(buffer, "%s %s %s", header1, header2, header3); <- how we parse the header
8.  char line = strtok(strdup(buffer), newline);
9.  while line is not null
10.     | if the first line
11.         | sscanf(line, "%s %s %s", header1, header2, header3);
12.     | if line contains "Content-length: "
13.         | sscanf(line, "%s %d", header4, content-length);
14.     | line = strtok(NULL, new line);
15.
```

2.3 GET

GET will only print out the status codes, 400,404, 200, and 500. GET does not deal with the status codes 201 and 403 because GET is not writing to the server.

```
1.  if the request command is “GET”
2.      |  open the file <- file
3.      |  if there is no target <- if only a slash
4.          |  write(cl, HTTP/1.1 400 Bad request);
5.          |  Content-length: 0
6.      |  if file is not found <- if the open file returns -1
7.          |  write(cl, HTTP/1.1 404 Not Found);
8.          |  Content-length: 0
9.      |  if file is found <- if open file is not equal to -1
10.         |  read file = read(file, file buffer, buffer size);
11.         |  get Content-length (2.3b)
12.         |  write(cl, HTTP/1.1 200 OK);
13.         |  sprintf(new buffer, content-length: %d, content-length);
14.         |  write(cl, content-length);
15.         |  while file size is >= size of buffer <- handling large files
16.             |  write(cl, file contents);
17.             |  read file again starting at file
18.         |  write(cl, file contents at file buffer)
19.     |  else
20.         |  write(cl, HTTP/1.1 500 Internal Server Error);
21.         |  Content-length: 0
```

2.3(b) get Content-length

```
1.  struct stat st;
2.  stat(target, st);
3.  content-length = st.st_size;
```

2.4 PUT

PUT deals with all the status codes because it deals with writing to a file
else 400 bad request
if falls through bad request, return 500

```
1.   if file contains "/"
2.       |  write(cl, HTTP/1.1 403 Forbidden);
3.       |  Content-length: 0
4.  else
5.       |  open(header2, O_RDWR|O_CREAT|O_TRUNC, 644);
6.       |  if no file
7.           |  write(cl, HTTP/1.1 400 Bad request);
8.           |  Content-length: 0
9.       |  else
10.          |  write(cl, HTTP/1.1 100 Continue, strlen(HTTP/1.1 100 Continue));
11.          |  read(cl, file buffer, buffer size);
12.          |  write(file descriptor, file buffer, content length);
13.
14.    How to deal with large files:
```