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, etc.

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

<u>Creating a sockaddr_in</u>

```
1.  if argv[1] is not NULL then make it SERVER_NAME_STRING
2.  if argv[2] is not NULL then make it PORT_NUMBER
3.  if argv[1] is NULL print the error "Request is missing required `Host` header"
4.  if argv[2] is NULL print the error "Request is missing required `Port` header"
5.  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;
```

<u>Creating a socket</u>

```
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

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

2.2 Parse Header

```
create buffer with size 327868
header1 = GET or  PUT <- the request
header2 = the 27 ascii string <- the file
header3 = HTTP/1,1
header4 = “Content-length: %d”
int valread = read(cl, buffer, size of buffer) <- buffer = header
sscanf(buffer, “%s %s %s”, header1, header2, header3); <- how we parse the header
char line = strtok(strdup(buffer), newline);
while line is not null
    |  if the first line
        |  sscanf(line, “%s %s %s”, header1, header2, header3);
    |  if line contains “Content-length: “
        |  sscanf(line, “%s %d”, header4, content-length);
    |  line = strtok(NULL, new line);

```

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.

```
if the request command is “GET”
    | open the file <- file
    | if there is a “/“ in the target name
        | write(cl, HTTP/1.1 403 Forbidden);
        | Content-length: 0
    | if the file name is not 27 ascii characters long
        | write(cl, HTTP/1.1 400 Bad Request);
        | Content-length: 0
    | if file is not found <- if the open file returns -1
        | write(cl, HTTP/1.1 404 Not Found);
        | Content-length: 0
    | if file is found <- if open file is not equal to -1
        | read file = read(file, file buffer, buffer size);
        | get Content-length (2.3b)
        | write(cl, HTTP/1.1 200 OK);
        | sprintf(new buffer, content-length: %d, content-length);
        | write(cl, content-length);
        | while file size is >= size of buffer <- handling large files
            | write(cl, file contents);
            | read file again starting at file
        | write(cl, file contents at file buffer)
    | else
        | write(cl, HTTP/1.1 500 Internal Server Error);
        | Content-length: 0
```

2.3(b) get Content-length

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

2.4 PUT

PUT deals with all the status
codes except 404 because it creates a target file if one is not found. PUT puts the contents of one file into the target file.

```
if file contains “/“
    | write(cl, HTTP/1.1 403 Forbidden);
    | Content-length: 0
else if target name is not 27 ascii characters long
    | write(cl, HTTP/1.1 400 Bad Request);
    | Content-length: 0
else
    | fd = open(header2, O_RDWR);
    | if no file <- create a file
        | close first open and reopen it by creating new file and writing to it
        | fd = open(header2, O_RDWR|O_CREAT|O_TRUNC, 0664); <- this creates file
        | write(cl, HTTP/1.1 100 Continue);
        | putread = read(cl, file buffer, content length);
        | counter = content length
        | if the content length >= buffer size <- if this is a large file
            | write(fd, file buffer, putread);
            | while(putread > 0 && counter >= putread)
                | write(fd, file buffer, putread)
                | counter = counter - buffer size;
                | putread = read(cl, file buffer, putread);
        | write(fd, file buffer, putread);
        | write(cl, HTTP/1.1 201 Created);
        | Content-length: 0
        | close fd
    | else if a file does exist, then overwrite it
        | close first open and open it again writing over file;
        | fd = open(header2, O_RDWR|O_CREAT|O_TRUNC, 0664);
        | write(cl, HTTP/1.1 100 Continue);
        | putread = read(cl, file buffer, buffer size);
        | counter = content length
        | if the content length >= buffer size <- if this is a large file
            | while(putread > 0 && counter >= buffer size)
                | write(file descriptor, file buffer, putread);
                | putread = read(cl, file buffer, putread);
                | counter = counter - buffer size
        | write(file descriptor, file buffer, content length);
        | write(cl, HTTP/1.1 200 OK”);
        | Content-length: 0
        | close(fd);
    | else
        | write(cl, HTTP/1.1 500 Internal Service Error);
        | Content-length: 0
```

```
At the end of everything
close(cl);
```