Design Document: Assignment 1 Basic Web Server

Ryan McCrory
CruzID: rmccrory

1 Goals:

The goal of this assignment is to implement a simple single-threaded http server. The server will respond to simple GET and PUT commands to read and write (respectively) “files” named by 27-character ASCII names. The server will persistently store files in a directory on the server, so it can be restarted or otherwise run on a directory that already has files.

Notes:

- The only status codes you’ll need to implement are 200 (OK), 201 (Created), 400 (Bad Request), 403 (Forbidden), 404 (Not Found), and 500 (Internal Server Error).
- Use a port number above 1024
- Test get using: `curl -s http://127.0.0.1:8888/blah --request-target ABCDEFarqdeXYZxyzf012345-ab`
- Test put using: `curl -s -T my_local_file http://127.0.0.1:8888/blah --request-target ABCDEFarqdeXYZxyzf012345-ab`

2 Design:

There will be several parts to the design. First, I will do the command line parsing in the main method, which I will use to name the IP address and port number. Next, I will open a network socket on the server. Then, using curl on another terminal, the client sends a request to the server that either asks to send a file from client to server (PUT) or fetch a file from server to client (GET).

2.1 MAIN:

The first argument of httpserver is the address of the http server to contact, which may be a hostname or an IP address. The second, optional, argument to httpserver is the port number on which to listen. If there’s no second argument, assume the standard HTTP port, port 80. I will open a socket using the code given by professor miller. Once the socket is opened, we accept the connection from the client with an infinite loop, and then wait for a get or put request. “The client, curl, sends a HTTP request. The request contains a method (like GET, POST, HEAD etc), a number of request headers and sometimes a request body. The HTTP server responds with a status line (indicating if things went well), response headers and most often also a response body. The "body" part is the plain data you requested, like the actual HTML or the image etc" (quote is from assignments specs). When we set client = accept(socket, NULL, NULL); and call the parse function with the client -> parse(client);

2.2 Parse:

We call the parse function with the client file descriptor we got from accept(). We receive a header from the client and we parse it using sscanf() to determine if it is a GET or PUT request, and to determine the name of the file from the client, and the name of the file on the server. Next we make sure this is a valid request header, so we check if the resource is 27 valid ascii characters. We then call either the get() or put() function depending on what we determine the client to be doing. The parameters we pass get() and put() are the client file descriptor, the file name, and the number of bytes we are reading, which is 1024.

Input: client file descriptor

```
parse(int cl){
    //read incoming data
    Int bytes = 1024;
    Char *buffer = (char *) malloc(bytes * sizeof(char));
    read(cl, buffer, bytes);
    //declare variables
    Char *action = (char *) malloc(bytes * sizeof(char));
    Char *file_name = (char *) malloc(bytes * sizeof(char));
    //parse data using scanf
    sscanf(buffer, "%s %s \n", action, file_name);
    //error checking
    If (strlen(file_name) !=  27){
        send(400 Bad Request error);
    }
    If (file_name is not all ascii characters) {
        send(400 Bad Request error);
    }
    //call put or get depending on what action* is
    If (action is "PUT"){
        put(cl, file_name, bytes);
    Else if (action is “GET”){
        get(cl, file_name, bytes);
    } else {
        send(400 Bad Request error);
    }
    //free memory
    free(buffer);
    free(action);
    free(file_name);
}
```

Algorithm 1: Parse

2.3: PUT

When there is a put request, parse will call this function. This will use a similar algorithm to asgn0, which will read the contents of the file we are trying to put. Instead of writing it to std out like in asgn0, we will be creating and writing it to a new file, which will have the same name as the request target curl specifies. After we are finished writing the file, we send the client a response header, indicating if this was successful or not.

Input: client file descriptor, file name, a number of bytes (1024)

```
put(int cl, char *file_name, int bytes){
        Char *buffer = (char *) malloc(bytes * sizeof(char));
        Int read_bytes = recv(cl, buffer, bytes, 0);
        //create the new file with the proper permissions
        Int new_file = open(file_name, O_CREAT | O_WRONLY,  0644);
        //write to the new file, same code as dog() in asgn0
        While (read_bytes > 0){
                write(new_file, buffer, read_bytes);
                Read_bytes = recv(cl, buffer, bytes, 0);
        }
        //send response header to client
        If (successful) {
                Send (successful header message);
        } else {
```

```
        Send (unsuccessful header message);
    }
    //close file and free memory
    close(new_name);
    free(buffer);
}
```

Algorithm 2: PUT

2.4: GET

When there is a get request, parse will call this function. It is also similar to dog() from asgn0. First, we check to see if the file the client is trying to get exists on the server side. We send the client a response header letting it know if it does or not, and the content length of this file. Then, if it exists, we open the local file on the server side that the client is trying to get. Then we read this file and place its contents in a buffer, which we send to the client. This is done in a loop that runs until all bytes of the file have been read and sent.

Input: client file descriptor, file name, a number of bytes (1024)

```
get(int cl, char *file_name, int bytes){
    //open file on server side and send response
    int local_file = open(file_name, O_RDWR);
    if (local_file == -1){
        warn("%s", file_name);
        send(response header indicating fail);
    }
    send(response header indicating success);
    //read and write data, similar to dog() from asgn0
    char *buffer = (char *) malloc(bytes * sizeof(char));
    int read_bytes = read(local_file, buffer, bytes);
    while (read_bytes > 0){
        send(cl, buffer, read_bytes, 0);
        read_bytes = read(local_file, buffer, bytes);
    }
    //free memory
    free(buffer);
}
```

Algorithm 3: GET