CS425 - Project 1 Report

Nishant Rai, 13449, nishantr@iitk.ac.in
August 19th, 2016

A. Options Implemented:

A.1. Mandatory Options:

- Supports the GET method
- Server supports persistent connections
- Server provides Content-Length and Content-Type fields in its response
- Server attempts to provide appropriate Status-Code and Response-Phrase values in response to errors.

A.2. Optional Features:

- Allow server port and document base directory to be initialized at start-up
- Include the Date and Server fields in the Response message header
- Respond with a directory listing if a directory is the requested resource

B. Optional Feature specifications:

A.1. Server port and root directory:

Port and base directory (Absolute path required) are optional and if desired should be passed in the following order, port and base directory.
e.g. .server 10000 /home/nishant/Documents/Academics/CS425A/Assignments/project1/code
Default port is 10000, default dir is ./

A.2. Date and Server response:

Date is sent as “Date: Fri, 19 Aug 2016 16:40:47 GMT”. Server is myServer/1.0

A.3. Directory structure:

In case the directory contains an index.html file, it attempts to open it first. Otherwise, a list of all the filenames is returned on a request for a directory.

C. Test Results:

A.1. Persistent connections:

We can see that log in firefox when loading the page shows completion of multiple requests at similar times. Which is only possible in case we handle multiple requests simultaneously.

A.2. Other results:

The web page used is the one provided with the project.

Google Chrome (64-bit) and FireFox was used for testing it (Only firefox results are shown).

GET Requests and Directory listing: Screenshots of the Firefox test are shown below. The directory listing is also shown.

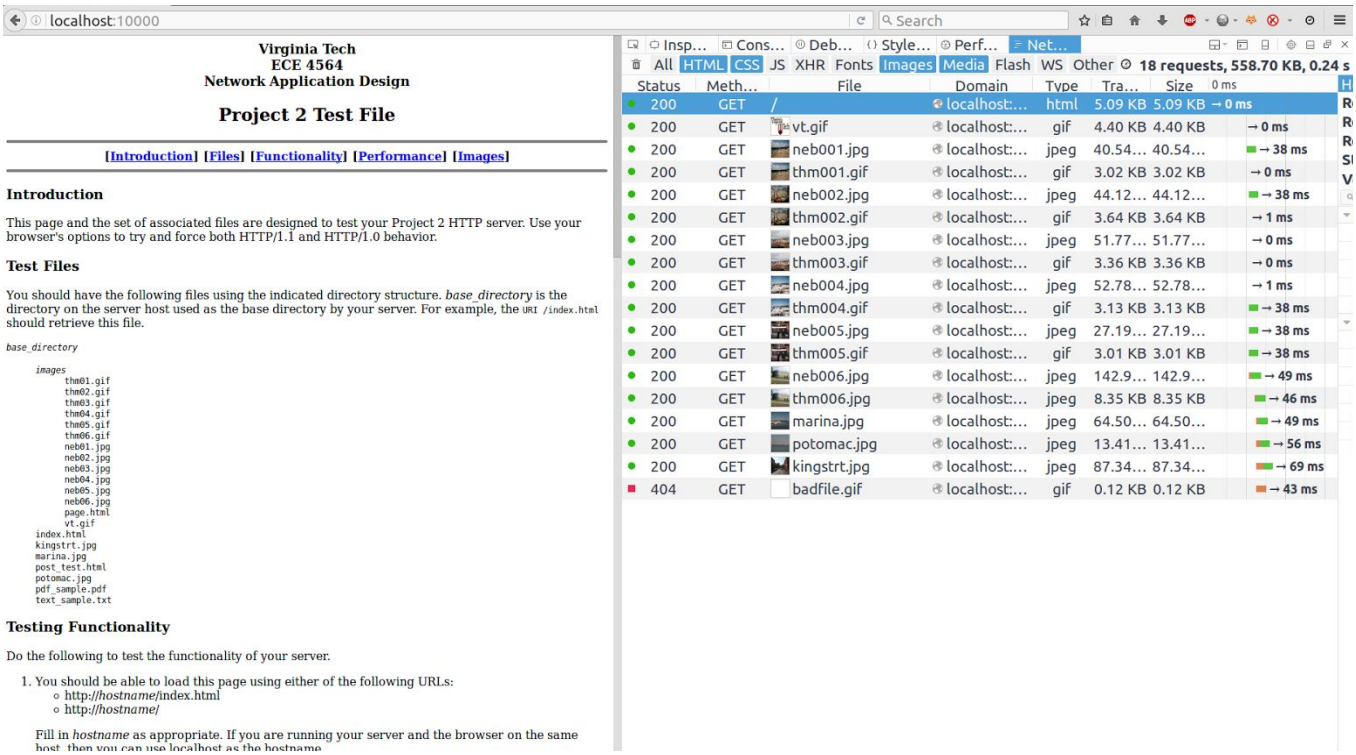
Status Codes :

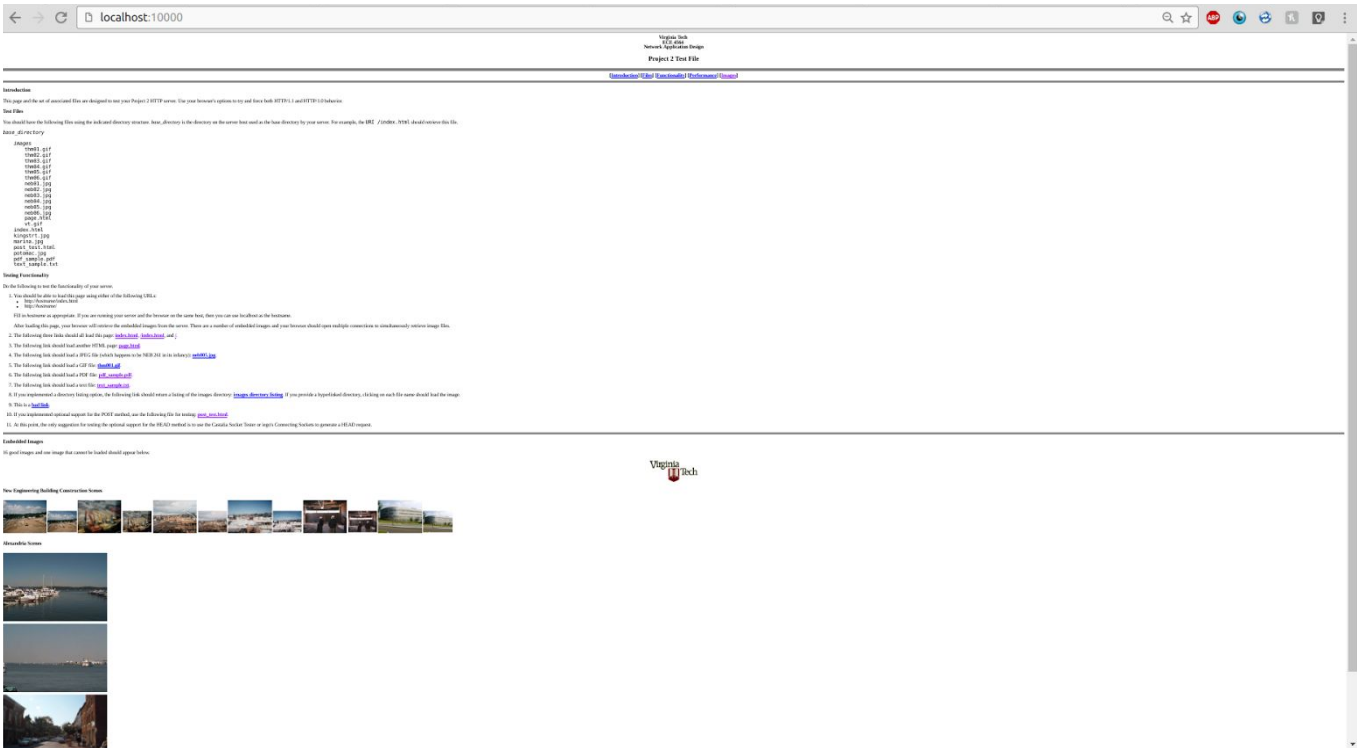
We recognize GET requests, otherwise 501 error is given.
It is a BAD request if the if any protocol other than HTTP/1.0 or HTTP/1.1 is requested.

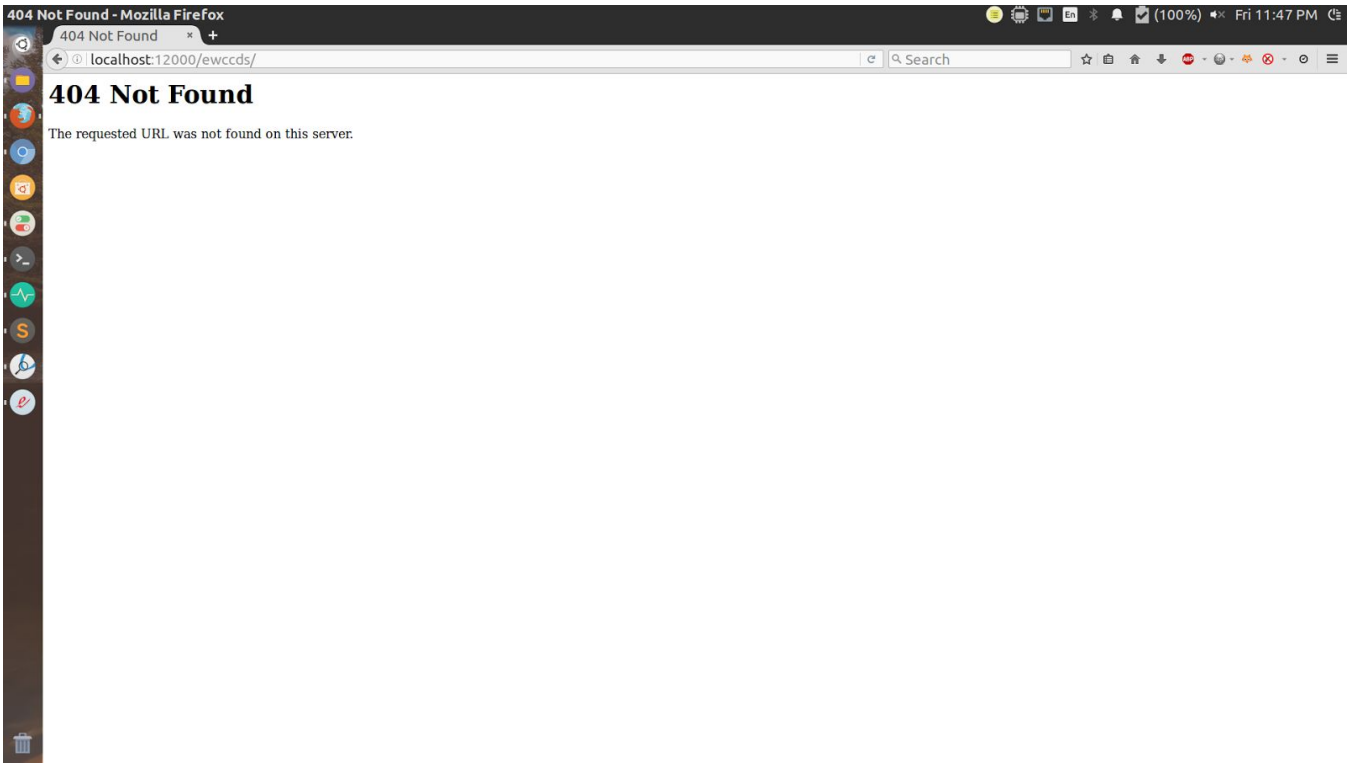
Example Response,

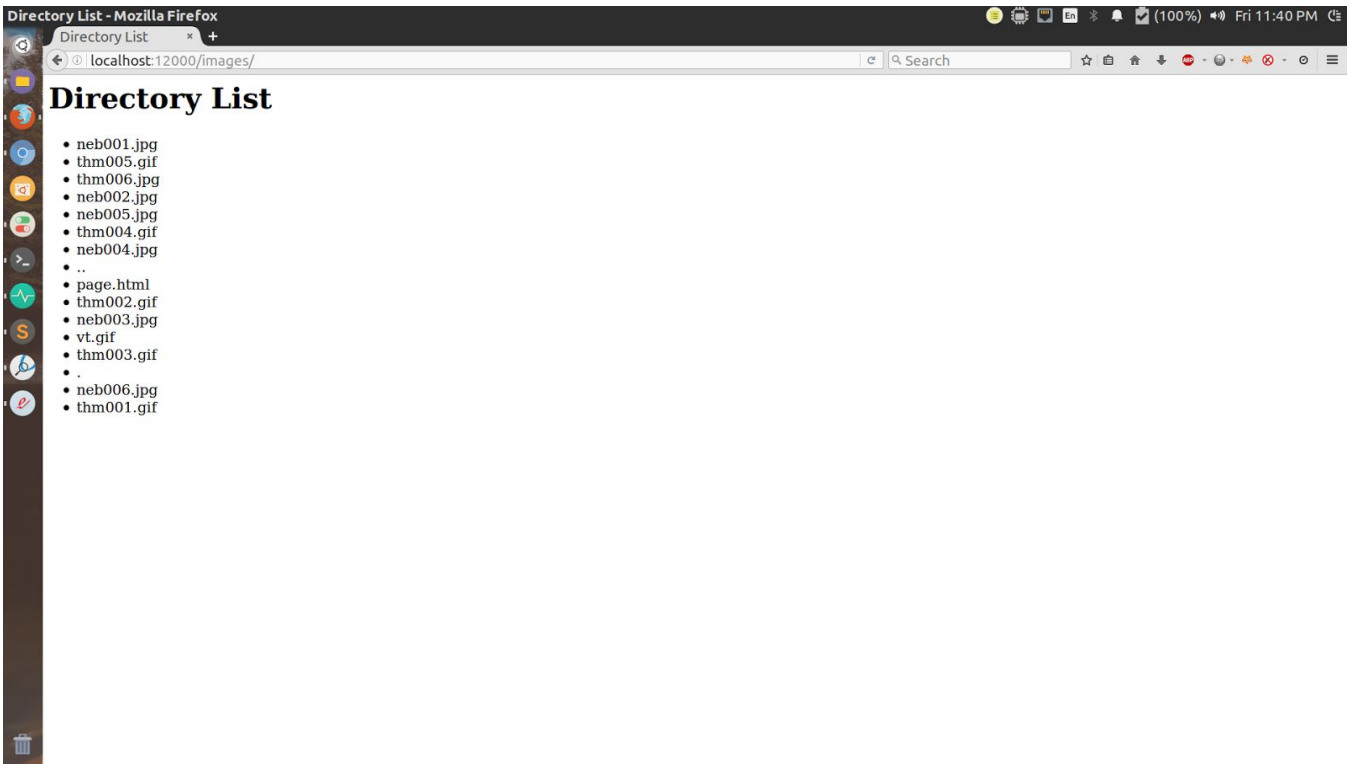
HTTP/1.1 404 Not Found
Date: Fri, 19 Aug 2016 18:18:27 GMT
Connection: Keep-Alive
Server: myServer/1.0
Content-Length: 125
Content-Type: image/gif

D. Images of Results:






404 Not Found - Mozilla Firefox
404 Not Found
localhost:12000/ewccds/
Search
(100%) Fri 11:47 PM
404 Not Found
The requested URL was not found on this server.


Directory List - Mozilla Firefox
Directory List
localhost:12000/images/
Search
(100%) Fri 11:40 PM
Directory List
neb001.jpg
thm005.gif
thm006.jpg
neb002.jpg
neb005.jpg
thm004.gif
neb004.jpg
..
page.html
thm002.gif
neb003.jpg
vt.gif
thm003.gif
.
neb006.jpg
thm001.gif

E. Source Code:

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cstring>
#include <string>
#include <sstream>
#include <vector>
#include <algorithm>
#include <fstream>

#include <dirent.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <signal.h>
#include <fcntl.h>

using namespace std;

#define MAX_CLIENTS 1000
#define BUF_SIZE 4096
#define DEBUG 1

typedef struct sockaddr_in socketAddr;
typedef struct sockaddr sockAddr;

string lowerStr(string str) {
	string ans(str);
	transform(str.begin(), str.end(), ans.begin(), ::tolower);
	// cout << "The lower case string is : " << ans << endl;
	return ans;
}

class Request {

	public:

		string crlf = "\r\n";
		int socketId;
		int port;
		int bufferSize = BUF_SIZE;
		char root[210];
		FILE* fp;
```

```
		// Constructor for Request
		Request(int id, char *baseDir) {
			socketId = id;
			strcpy(root, baseDir);
			// fp = fdopen(socketId, "r+b");
		}

		string getResponse(char *reqMsg, int flag);
		string getType(char *fileName);
		string getTime();
		void run();
		void end();

};

void Request::end() {
	// fclose(fp);
	// if (DEBUG)

	if (DEBUG)
		cout << "The connection has been closed!\n";

	shutdown(socketId, SHUT_RDWR);
	close(socketId);
}

string Request::getType(char *fileName) {
	string tmpFile(fileName);
	string extName = tmpFile.substr(tmpFile.find_last_of(".") + 1);
	string contType = "";

	if (DEBUG) {
		cout << "The extension is : ";
		cout << extName << endl;
	}

	if ((extName == "html") || (extName == "htm"))
		contType += "text/html";
	else if (extName == "txt")
		contType += "text/plain";
	else if ((extName == "jpeg") || (extName == "jpg"))
		contType += "image/jpeg";
	else if (extName == "gif")
		contType += "image/gif";
	else if (extName == "pdf")
		contType += "Application/pdf";
	else
		contType += "Application/octet-stream";
```

```
	if (DEBUG) {
		cout << "The type is " << contType << endl;
	}

	return contType;
}

string Request::getTime() {
	char buf[110];

	time_t now = time(0);
	struct tm tm = *gmtime(&now);
	strftime(buf, sizeof(buf), "%a, %d %b %Y %H:%M:%S %Z", &tm);
	string timeMsg = "Date: ";
	timeMsg.append(buf);

	if (DEBUG) {
		cout << "The time is:\n";
		cout << timeMsg << endl;
		cout << timeMsg.length() << endl;
	}

	return timeMsg;
}

string Request::getResponse(char *reqMsg, int flag) {

	char *requests[3], filePath[210], sendMsg[4100];
	string statMsg = "";
	string resMsg = "";
	string conLenMsg = "Content-Length: ";
	string conTypeMsg = "Content-Type: ";
	string timeMsg = getTime();
	string serverMsg = "Server: myServer/1.0";
	string connectMsg = "Connection: Keep-Alive";
	if (flag == 0) {
		connectMsg = "Connection: Close";
	}
	timeMsg += "\n" + connectMsg + "\n" + serverMsg;
	string finMsg = "";
	int numBytes;

	requests[0] = strtok(reqMsg, " \t\n");

	if (requests[0] != NULL) {

		if (strncmp(requests[0], "GET\0", 4) == 0) {

			requests[1] = strtok (NULL, " \t");
			requests[2] = strtok (NULL, " \t\n");
```

```
		if (DEBUG) {
			printf("%s\n", requests[0]);
			printf("%s\n", requests[1]);
			printf("%s\n", requests[2]);
		}

		if ((strncmp(requests[2], "HTTP/1.1", 8) != 0) && (strncmp(requests[2], "HTTP/1.0", 8) != 0)) {
			statMsg = "HTTP/1.1 400 Bad Request\n\n";
			statMsg += timeMsg + "\n" + conLenMsg + "\n" + conTypeMsg + "\n\n";
			write(socketId, statMsg.c_str(), statMsg.length());
			end();
		}
		else {

			int isDir = 0;
			string fileName(requests[1]);

			if (fileName.back() == '/') {
				fileName += "index.html";                                    // Default behavior
				isDir = 1;
			}

			cout << "File requested is " << fileName << endl;

			requests[1] = (char *) fileName.c_str();
			conTypeMsg += getType(requests[1]);

strcpy(filePath, root);
strcpy(filePath + strlen(root), requests[1]);

			FILE* fd = fopen(filePath, "rb");
			if (fd != NULL) {

				// Get the size of the file
				fseek(fd, 0L, SEEK_END);
				int fileSize = ftell(fd);
				rewind(fd);

				conLenMsg += to_string(fileSize);

				statMsg = "HTTP/1.1 200 OK\n";
				statMsg += timeMsg + "\n" + conLenMsg + "\n" + conTypeMsg + crlf + crlf;
				write(socketId, statMsg.c_str(), statMsg.length());

				resMsg = "";
				memset(sendMsg, '\0', bufferSize);
				while ((numBytes = fread(sendMsg, 1, bufferSize, fd)) > 0) {
					write (socketId, sendMsg, numBytes);
					resMsg += string(sendMsg);
```

```
                                        memset(sendMsg, '\0', bufferSize);
                                }
                                fclose(fd);
                        }
                        else if (isDir && (fileName.find("index.html") != string::npos)) {
                                DIR *dir;
                                struct dirent *tmpDir;

                                resMsg = "<HEAD><TITLE>Directory List</TITLE></HEAD>\n<BODY><H1>Directory
List</H1><ul>";

                strcpy(filePath, root);
                requests[1][strlen(requests[1]) - 10] = '\0';
                                printf("The stripped fileName 1 : %s\n", requests[1]);
                strcpy(filePath + strlen(root), requests[1]);
                                printf("The stripped fileName : %s\n", filePath);
                                dir = opendir(filePath);
                                if (dir) {
                                        while ((tmpDir = readdir(dir)) != NULL) {
                                                // statMsg += "<li><a href=\"" + fileName +
string(tmpDir->d_name) + "\">" + string(tmpDir->d_name) + "</a></li>";
                                                resMsg += "<li>" + string(tmpDir->d_name) + "</li>";
                                                // printf("%s\n", tmpDir->d_name);
                                        }
                                        closedir(dir);
                                        // cout << endl;
                                        resMsg += "</ul></BODY>";
                                        statMsg = "HTTP/1.1 200 OK\n";
                                        conLenMsg += to_string(resMsg.size());
                                        statMsg += timeMsg + "\n" + conLenMsg + "\n" + conTypeMsg + crlf + crlf
+ resMsg;
                                }
                                else {
                                        statMsg = "HTTP/1.1 404 Not Found\n";
                                        resMsg = "<HEAD><TITLE>404 Not
Found</TITLE></HEAD>\n<BODY><H1>404 Not Found</H1>The requested URL was not found on this server.\n</BODY>";
                                        conLenMsg += to_string(resMsg.size());
                                        statMsg += timeMsg + "\n" + conLenMsg + "\n" + conTypeMsg + crlf + crlf
+ resMsg;
                                        // write(socketId, statMsg.c_str(), statMsg.length());
                                }

                                // cout << statMsg << endl;
                                write(socketId, statMsg.c_str(), statMsg.length());
                                // end();
                        }
                        else {
                                statMsg = "HTTP/1.1 404 Not Found\n";
                                resMsg = "<HEAD><TITLE>404 Not Found</TITLE></HEAD>\n<BODY><H1>404 Not
Found</H1>The requested URL was not found on this server.\n</BODY>";
```

```
                                        conLenMsg += to_string(resMsg.size());
                                        statMsg += timeMsg + "\n" + conLenMsg + "\n" + conTypeMsg + crlf + crlf + resMsg;

                                        write(socketId, statMsg.c_str(), statMsg.length());
                                        // end();
                                }
                        }
                }
                else {
                        statMsg = "HTTP/1.1 501 Not Implemented\n";
                        statMsg += timeMsg + crlf + crlf;
                        statMsg += "<HEAD><TITLE>501 Not Implemented</TITLE></HEAD>\n<BODY><H1>501 Not
Implemented</H1>The requested method was not implemented.\n</BODY>";
                        write(socketId, statMsg.c_str(), statMsg.length());
                        end();
                }
        }

        if (DEBUG) {
                cout << "The status : " << statMsg << endl;
                cout << "The content length : " << conLenMsg << endl;
                cout << "The content type : " << conTypeMsg << endl;
                cout << "The status string is: " << statMsg << endl;
        }

        if (DEBUG) {
                cout << "The response is _____\n";
                cout << statMsg << endl;
        }

        finMsg = statMsg + crlf + crlf + resMsg;

        return finMsg;
}

void Request::run() {

        char getMsg[BUF_SIZE + 1];
        char *cgetMsg;

        while (1) {

                string reqMsg = "";

                memset(getMsg, '\0', bufferSize);
                int numRead = 0, totRead = 0;
                while((numRead = read(socketId, getMsg, bufferSize)) > 0) {
                        reqMsg += string(getMsg);
                        getMsg[numRead] = '\0';
                        if (DEBUG) {
```

```
                        printf("Client reading %d\n", numRead);
                        printf("%s", getMsg);
                }
                if (reqMsg.find("\r\n\r\n") != string::npos)
                        break;
                memset(getMsg, '\0', bufferSize);
        }


        if (DEBUG) {
                cout << "The request is_____\n";
                cout << reqMsg << endl;
        }

        // sleep(0.25);

        int flag = 0;
        if (lowerStr(reqMsg).find("connection: close") == string::npos) {
                flag = 1;
        }

        if (numRead <= 0) {
                // Client not available anymore
                break;
        }

        cgetMsg = (char *) reqMsg.c_str();
        string resMsg = getResponse(cgetMsg, flag);

        if (DEBUG) {
                cout << "Now Here 2!\n";
                printf("%s\n", cgetMsg);
                cout << strlen(cgetMsg) << endl;
                cout << "Now Here 3!\n";
        }

        if (!flag) {
                break;
        }

    }

    end();
}

class Server {

    public:

        int portNum;
```

```
        int listenId;
        char rootDir[210];

        Server (int num, char *baseDir) {
                portNum = num;
                strcpy(rootDir, baseDir);
        }

        void startServer();
        void runServer();

};

void Server::startServer() {
        socketAddr serverAddr;

        // Socket Id to which we listen
        // Upon successful completion, socket() shall return a non-negative integer,
        // the socket file descriptor. Otherwise, a value of -1 shall be returned.
        listenId = socket(AF_INET, SOCK_STREAM, 0);

        if (listenId < 0) {
                printf("Could not retreive socket\n");
                return;
        }

        // For Internet family of IPv4 addresses we use AF_INET.
        // 'SOCK_STREAM' specifies that the transport layer protocol that we want should be reliable i.e it should have acknowledgement techniques. For example : TCP
        // The third argument is generally left zero to let the kernel decide the default protocol to use for this connection.
        // For connection oriented reliable connections, the default protocol used is TCP.

        printf("Succeeded in Socket retrieval\n");

        serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
        serverAddr.sin_family = AF_INET;
        serverAddr.sin_port = htons(portNum);

        /* Bind the address struct to the socket */
        // socket -> bind -> listen -> accept -> doStuff
        bind(listenId, (sockAddr*) &serverAddr, sizeof(serverAddr));

        // Listen with the max number of connections
        // On success, zero is returned. On error, -1 is returned
        if(listen(listenId, MAX_CLIENTS) < 0) {
                printf("Could not listen\n");
                return;
        }
}
```

```
void Server::runServer() {

	int port = portNum;
	int bufferSize = 1000;
	int connectId = 0, cnt = 0;
	int sockId = 0;

	startServer();

	while(1)
	{
		sockId = accept(listenId, (sockAddr*) NULL, NULL);
		// Message denoting new connection established
		if (DEBUG)
			printf("Connection established with client\n");

		if (sockId < 0)
			printf("No available slots for the client\n");
		else
		{
			if (DEBUG)
				printf("Creating new thread!\n");
			if (fork() == 0) {
				Request currRequest(sockId, rootDir);
				// Create another thread
				currRequest.run();
				exit(0);
			}
		}

		sleep(0.01);
		// Sleep for sometime
	}
}

int main(int argc, char* argv[]) {

	char baseDir[210];
	strcpy(baseDir, getenv("PWD"));
	int port = 10000;

	if (argc < 2) {
		printf("Enter the port number and root directory\n");
	}
	else if (argc < 3) {
		port = atoi(argv[1]);
		printf("Enter the port number and root directory. You could've entered the root directory too\n");
	}
	else {
		port = atoi(argv[1]);
```

```
                // memset(baseDir, '\0', strlen(baseDir));
                strcpy(baseDir, argv[2]);
        }

        if (baseDir[strlen(baseDir) - 1] == '/') {
                baseDir[strlen(baseDir) - 1] = '\0';
        }

        printf("The root directory is %s\n", baseDir);
        printf("The port number is %d\n", port);
        Server httpServer(port, baseDir);
        httpServer.runServer();

        return 0;

}
```