CS425 - Project 1 Report

Nishant Gupta, nishgu@iitk.ac.in, 13447

August 19, 2016

HTTP File Server

1 Implemented Options

1.1 Mandatory

- Supports GET method
- Supports Persistent Connections and multiple clients
- Provides Content-length and content-Type fields in response
- Provides appropriate Status-Code and Response-Phrase values in response to errors.

1.2 Optional

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

2 Optional feature details

2.1 Server port and root directory initialization :

You can optionally give port with `-p` and root with `-r`. Default values are: `PORT=9576` and `ROOT=./`
eg: `bin/server -p 9577 -r test/`

2.2 Date and Server fields :

Server is sent as Alchemist and Date is sent in the format `Thu, 18 Aug 2016 18:52:47 GMT`.

2.3 POST Method :

In, POST request, message-body is written to `abs-path` of URI relative to root directory. Html displaying `Content written Succesfully` is returned in response. See Appendix

2.4 HEAD Method :

Only headers of a GET request are returned in HEAD request.
See Appendix

2.5 Hyperlinked directory :

A clickable list of all the filenames is is returned on GET request for a directory. Spaces in filenames and directories are not supported.

3 Test results

3.1 Multiple Clients and Persistent connections

See Multiple clients and persistent connections of Appendix for Verification image and log. It can be seen in chrome network inspector image that multiple requests complete at almost same time. This means that server supports multiple connections.
Also Stderr log of opening `index.html` in firefox shows that multiple requests were served to same client, which proves that persistent connections work and the clients requests are intermingled, which proves that multiple simultaneous connections work.

3.2 Webpage provided with project

Google Chrome Version 52.0.2743.116 (64-bit) was used for testing in this part

3.2.1 GET and Directory listing

These are Screenshots of Google Chrome showing index.html and directory listing working correctly.

3.2.2 POST request

After submitting form on `post_test.html`, following is the content of `test/post_test_file.txt`

```
TextEntry_1=Hola&TextEntry_2=This+is+some+text+for+testing+the+POST+method.%0D%0A&Item=Item_1
```

3.3 Testing using command line

httpie : A CLI HTTP client was used.

3.3.1 POST request

I generated a random file of 10^6 bytes and sent it using `httpie`, and compared the saved file to original file. They were same. POST request section of Appendix verifies that.

3.3.2 GET request

I made a GET request using httpie and it ran succesfully as can be seen in GET Request section of Appendix.

3.3.3 HEAD request

I made a HEAD request using httpie and it ran succesfully as can be seen in HEAD request section of Appendix.

3.4 Status Codes

I made different types of requests that would yield different status codes using httpie. They can be seen in Status Codes section of Appendix.

- I recognize `GET, HEAD, POST, PUT, OPTIONS, CONNECT, DELETE`. Only three of them are implemented, others give `501 : Not implemented`
- If anything other than these seven comes, it is BAD request and `400 : Bad Request` is returned. Bad request is also returned if any protocol other than HTTP/1.0 or HTTP/1.1 is requested
- I have put my entire parser in a try catch block, if parser fails I send `500 : Internal Server Error` in catch block.
- If Parser succeeds, I process corresponding request and then `404` or `200` is sent by corresponding method.

3.5 Date and Server fields

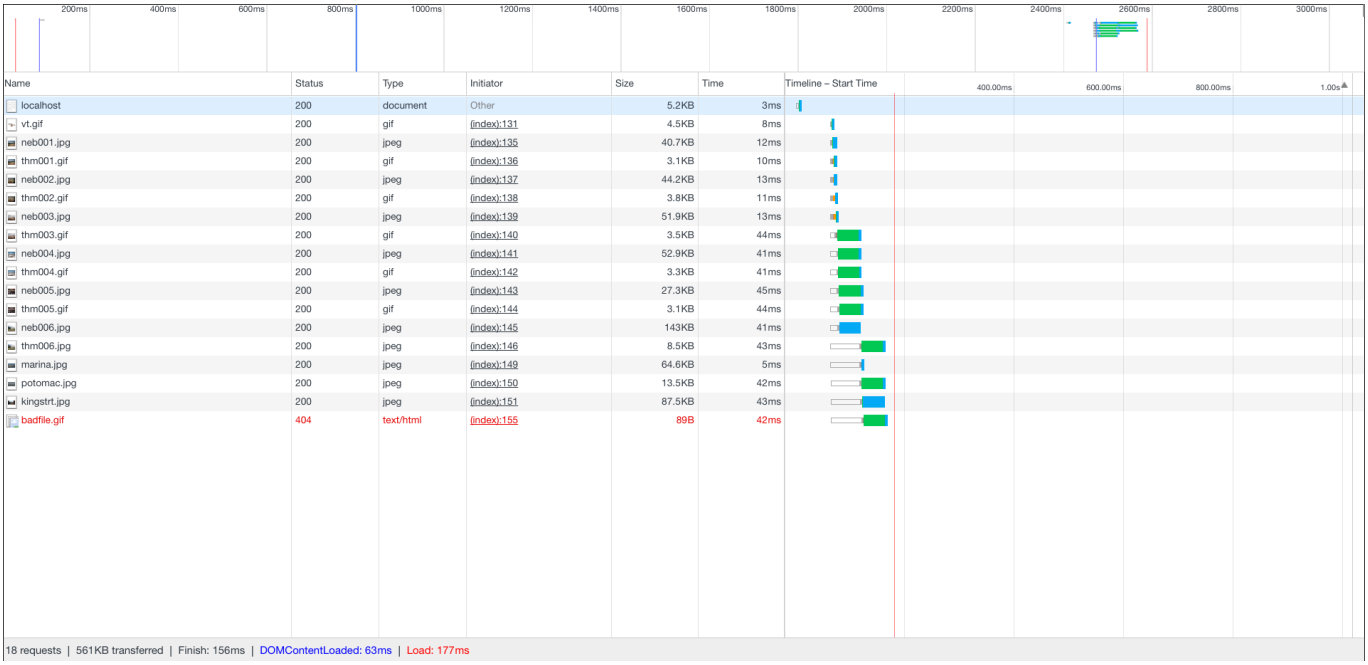
It can be seen in any request in Appendix that Server and Appendix Headers are returned.

3.6 Summary

- I have not supported spaces and other special characters in resources requested, so it will give 404.
- Also, in hyperlinked directory listing, link will be wrong if there are spaces or other special characters in the name.
- Everything else is working as it should.
- I had no way to check Internal Server Error.
- All the other Status codes are working as they should.
- Multiple clients and persistent connections are working correctly, I have not implemented timeout.

4 Appendix

4.1 Multiple clients and persistent connections

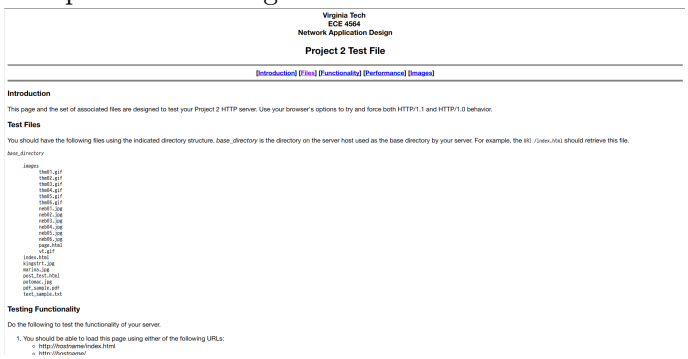


Following is the condensed stderr log of opening `index.html` in firefox.

```
Connected to client 6
Request 1 from Client 5
Request 2 from Client 5
Connected to client 6
Connected to client 7
Connected to client 8
Connected to client 9
Connected to client 10
Request 1 from Client 10
Request 1 from Client 9
Request 1 from Client 8
Request 1 from Client 6
Request 1 from Client 7
Request 3 from Client 5
Request 2 from Client 8
Request 2 from Client 10
Request 2 from Client 9
Request 2 from Client 7
Request 2 from Client 6
Request 3 from Client 6
Request 4 from Client 5
Request 3 from Client 8
Request 4 from Client 6
Request 5 from Client 5
Request 5 from Client 6
Closed Client
Closed Client
Closed Client
Closed Client
Closed Client
Closed Client
```

4.2 GET Request :

GET request made using Chrome



Virginia Tech
ECE 4564
Network Application Design

Project 2 Test File

[Introduction] [Files] [Functionality] [Performance] [Images]

Introduction

This page and the set of associated files are designed to test your *Project 2* HTTP server. Use your browser's options to try and force both HTTP/1.1 and HTTP/1.0 behavior.

Test Files

You should have the following files using the indicated directory structure. *base_directory* is the directory on the server host used as the base directory by your server. For example, the URI `/index.html` should retrieve this file.

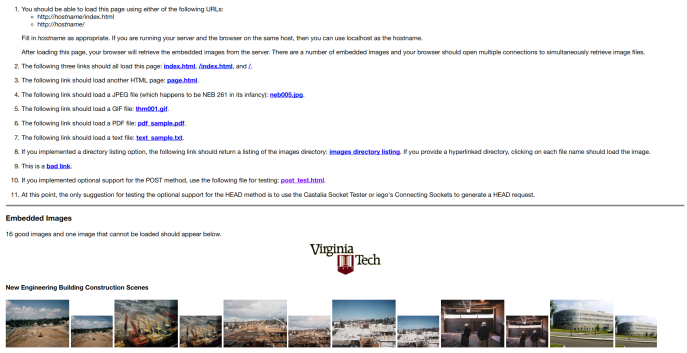
base_directory

```
    images
        thm01.gif
        thm02.gif
        thm03.gif
        thm04.gif
        thm05.gif
        thm06.gif
        neb01.jpg
        neb02.jpg
        neb03.jpg
        neb04.jpg
        neb05.jpg
        neb06.jpg
        page.html
        vt.gif
    index.html
    kingstr1.jpg
    marlin.jpg
    post_test.html
    potomac.jpg
    pdf_sample.pdf
    text_sample.txt
```

Testing Functionality

Do the following to test the functionality of your server.

1. You should be able to load this page using either of the following URLs:
   - http://hostname/index.html
   - http://hostname/



1. You should be able to load this page using either of the following URLs:
   - http://hostname/index.html
   - http://hostname/

   Fill in *hostname* as appropriate. If you are running your server and the browser on the same host, then you can use localhost as the hostname.

   After loading this page, your browser will retrieve the embedded images from the server. There are a number of embedded images and your browser should open multiple connections to simultaneously retrieve image files.
2. The following three links should all load this page: **index.html**, **/index.html**, and **/**.
3. The following link should load another HTML page: **page.html**.
4. The following link should load a JPEG file (which happens to be NEB 261 in its infancy): **neb005.jpg**.
5. The following link should load a GIF file: **thm001.gif**.
6. The following link should load a PDF file: **pdf_sample.pdf**.
7. The following link should load a text file: **text_sample.txt**.
8. If you implemented a directory listing option, the following link should return a listing of the images directory: **images directory listing**. If you provide a hyperlinked directory, clicking on each file name should load the image.
9. This is a **bad link**.
10. If you implemented optional support for the POST method, use the following file for testing: post_test.html.
11. At this point, the only suggestion for testing the optional support for the HEAD method is to use the Castalia Socket Tester or iego's Connecting Sockets to generate a HEAD request.

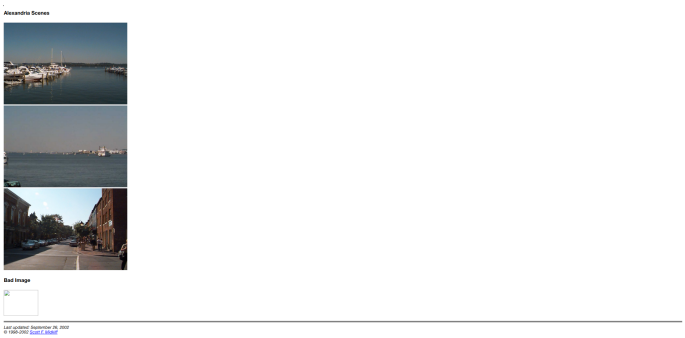
Embedded Images

16 good images and one image that cannot be loaded should appear below.

Virginia Tech

New Engineering Building Construction Scenes

- neb004.jpg
- thm005.gif
- neb003.jpg
- neb001.jpg
- page.html
- neb005.jpg
- vt.gif
- neb002.jpg
-
- .
- ss
- neb006.jpg
- thm001.gif
- thm006.jpg
- thm002.gif
- thm004.gif
- thm003.gif



Alexandria Scenes

Bad Image

Last updated: September 26, 2002

GET request made using httpie

```
 ~/c/n/project1 $ http GET  :9577/post_test_file.txt Connection:close -v
GET /post_test_file.txt HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: close
Host: localhost:9577
User-Agent: HTTPie/0.9.4



HTTP/1.1 200 OK
Connection: Close
Content-Length: 93
Content-Type: text/plain
Date: Thu, 18 Aug 2016 18:52:47 GMT
Server: Alchemist

TextEntry_1=Hola&TextEntry_2=This+is+some+text+for+testing+the+POST+method.%0D%0A&Item=Item_1
```

4.3 POST request :

POST Request made using httpie

```
 ~/c/n/project1 $ base64 /dev/urandom | head -c 1000000 >! file.txt
 ~/c/n/project1 $ cat file.txt | http POST :9577/file.txt -p=Hhb
POST /file.txt HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 1000000
Content-Type: application/json
Host: localhost:9577
User-Agent: HTTPie/0.9.4

HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 36
Content-Type: text/html
Date: Fri, 19 Aug 2016 13:31:22 GMT
Server: Alchemist

<h1>Content written Succesfully</h1>

 ~/c/n/project1 $ diff file.txt test/file.txt
 ~/c/n/project1 $
```

4.4 HEAD request :

HEAD request made using httpie

```
 ~/c/n/project1 $ http HEAD  :9577/ Connection:close -v
HEAD / HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: close
Host: localhost:9577
User-Agent: HTTPie/0.9.4



HTTP/1.1 200 OK
Connection: Close
Content-Length: 5211
Content-Type: text/html
Date: Thu, 18 Aug 2016 18:57:32 GMT
Server: Alchemist
```

4.5 Status Codes :

Following are some of httpie requests that showcase diffrent status codes

- Not Implemented

```
 ~/c/n/project1 $ http PUT  :9577/fajlsdfj -v
PUT /fajlsdfj HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 0
Host: localhost:9577
User-Agent: HTTPie/0.9.4


HTTP/1.0 501 Not Implemented
Connection: keep-alive
Content-Length: 29
Content-Type: text/html
Date: Fri, 19 Aug 2016 13:30:57 GMT
Server: Alchemist

<h1>501: Not Implemented</h1>
```

- Bad Request

```
 ~/c/n/project1 $ http PUT  :9577/fajlsdfj -v
PUT /fajlsdfj HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 0
Host: localhost:9577
User-Agent: HTTPie/0.9.4


HTTP/1.0 501 Not Implemented
Connection: keep-alive
Content-Length: 29
Content-Type: text/html
Date: Fri, 19 Aug 2016 13:30:57 GMT
Server: Alchemist

<h1>501: Not Implemented</h1>
```

- Not Found

```
 ~/c/n/project1 $ http GET  :9577/fajlsdfj -v
GET /fajlsdfj HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:9577
User-Agent: HTTPie/0.9.4


HTTP/1.0 404 Not Found
```

```
Connection: keep-alive
Content-Length: 23
Content-Type: text/html
Date: Fri, 19 Aug 2016 13:35:36 GMT
Server: Alchemist

<h1>404: Not Found</h1>
```

4.6 Source code :

```
// AUTHOR: Nishant Gupta

#include <arpa/inet.h>
#include <assert.h>
#include <dirent.h>
#include <fcntl.h>
#include <netdb.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

#define CONNMAX 1000
#define BYTES 8096
#define min(a, b) (a < b) ? a : b

char *ROOT = getenv("PWD"), PORT[8];
int listenfd;

enum requestType { GET, POST, HEAD, BAD, UNIMPLEMENTED, ERROR };
void startServer(char *);
void serveClient(int);
requestType parseHeaders(char *, int &, char *, int &);
int respondHG(char *, int, requestType, int);
int respondPOST(char *, int, int, int);
int sendCommonHeaders(int, int, int, char *);
int sendNotFound(int, requestType);
char *generateDirectoryList(char *, int &);

// Parsing the command line arguments
int parseArgs(int argc, char *argv[]) {
  char c;

  while ((c = getopt(argc, argv, "p:r:")) ≠ -1)
    switch (c) {
    case 'r':
      ROOT = (char *)malloc(strlen(optarg));
      strcpy(ROOT, optarg);
      break;
    case 'p':
      strcpy(PORT, optarg);
```

```
      break;
    default:
      fprintf(stderr, "%s [-p PORT] [-r ROOT]\n", argv[0]);
      exit(1);
    }
  return 0;
}

int main(int argc, char *argv[]) {
  struct sockaddr_in clientaddr;
  socklen_t addrlen;
  int clientSock;
  strcpy(PORT, "9576");
  parseArgs(argc, argv);

  startServer(PORT);
  printf("Server started at port no. %s with root directory as %s\n", PORT,
         ROOT);

  while (1) {
    addrlen = sizeof(clientaddr);
    clientSock = accept(listenfd, (struct sockaddr *)&clientaddr, &addrlen);
    if (clientSock < 0)
      perror("accept() error");
    else {
      if (fork() == 0) {
        serveClient(clientSock);
        exit(0);
      }
    }
  }
  return 0;
}

// starts the server, One time call
void startServer(char *port) {
  struct addrinfo hints, *res, *p;
  int on = 1;
  // taken from `man 3 getaddrinfo`
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE; // to listen to all interfaces
  if (getaddrinfo(NULL, port, &hints, &res) ≠
      0) { // get a list of all interfaces to which socket is bound
    perror("getaddrinfo() error");
    exit(1);
  }
  // try to listen to atleast one
  for (p = res; p ≠ NULL; p = p->ai_next) {
    listenfd = socket(p->ai_family, p->ai_socktype, 0);
    if (listenfd == -1)
      continue;
    if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
      break;
```

```
  }
  freeaddrinfo(res);
  // Entire list traversed but could not bind to anyone
  if (p == NULL) {
    perror("socket() or bind()");
    exit(1);
  }

  // free the port as soon as program exits
  setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

  if (listen(listenfd, atoi(port)) ≠ 0) {
    perror("listen() error");
    exit(1);
  }
}

//Sends headers that are common in all thre requests, currently only GET and
//HEAD use it.
int sendCommonHeaders(int clientSock, int contentLength, int keepAliveStatus,
                      char *mimeType) {
  char tempBuffer[BYTES];
  send(clientSock, "HTTP/1.1 200 OK\r\n", 17, 0);
  send(clientSock, "Server: Alchemist\r\n", 19, 0);
  char dateBuffer[BYTES];
  time_t now = time(0);
  struct tm tm = *gmtime(&now);
  strftime(dateBuffer, sizeof(dateBuffer), "%a, %d %b %Y %H:%M:%S %Z", &tm);
  snprintf(tempBuffer, sizeof(tempBuffer), "Date: %s\r\n", dateBuffer);
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  snprintf(tempBuffer, BYTES, "Content-Length: %d\r\n", contentLength);
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  snprintf(tempBuffer, BYTES, "Content-Type: %s\r\n", mimeType);
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  if (keepAliveStatus == 1)
    snprintf(tempBuffer, BYTES, "Connection: Close\r\n\r\n");
  else
    snprintf(tempBuffer, BYTES, "Connection: keep-alive \r\n\r\n");
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  return 0;
}

// Send 404 Status
int sendNotFound(int clientSock, requestType curRequest) {
  char tempBuffer[1024];
  write(clientSock, "HTTP/1.0 404 Not Found\n", 23); // FILE NOT FOUND
  send(clientSock, "Server: Alchemist\r\n", 19, 0);
  char dateBuffer[BYTES];
  time_t now = time(0);
  struct tm tm = *gmtime(&now);
  strftime(dateBuffer, sizeof(dateBuffer), "%a, %d %b %Y %H:%M:%S %Z", &tm);
  snprintf(tempBuffer, sizeof(tempBuffer), "Date: %s\r\n", dateBuffer);
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  char *errMesg = (char *)"<h1>404: Not Found</h1>";
  snprintf(tempBuffer, BYTES, "Content-Length: %d\n", (int)strlen(errMesg));
```

```
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  snprintf(tempBuffer, BYTES, "Content-Type: text/html\r\n");
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  send(clientSock, "Connection: keep-alive\n\n", 24, 0);
  if (curRequest ≠ HEAD)
    send(clientSock, errMesg, strlen(errMesg) + 1, 0);
  return 0;
}

// Send 400 Status
int sendBadRequest(int clientSock, requestType curRequest) {
  char tempBuffer[1024];
  write(clientSock, "HTTP/1.0 400 Bad Request\r\n", 26);
  send(clientSock, "Server: Alchemist\r\n", 19, 0);
  char dateBuffer[BYTES];
  time_t now = time(0);
  struct tm tm = *gmtime(&now);
  strftime(dateBuffer, sizeof(dateBuffer), "%a, %d %b %Y %H:%M:%S %Z", &tm);
  snprintf(tempBuffer, sizeof(tempBuffer), "Date: %s\r\n", dateBuffer);
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  char *errMesg = (char *)"<h1>400: Bad Request</h1>";
  snprintf(tempBuffer, BYTES, "Content-Length: %d\n", (int)strlen(errMesg));
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  snprintf(tempBuffer, BYTES, "Content-Type: text/html\r\n");
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  send(clientSock, "Connection: keep-alive\r\n\r\n", 26, 0);
  if (curRequest ≠ HEAD)
    send(clientSock, errMesg, strlen(errMesg) + 1, 0);
  return 0;
}


// Send 500 status
int sendInternalError(int clientSock, requestType curRequest) {
  char tempBuffer[1024];
  write(clientSock, "HTTP/1.0 500 Internal Server Error\r\n", 36);
  send(clientSock, "Server: Alchemist\r\n", 19, 0);
  char dateBuffer[BYTES];
  time_t now = time(0);
  struct tm tm = *gmtime(&now);
  strftime(dateBuffer, sizeof(dateBuffer), "%a, %d %b %Y %H:%M:%S %Z", &tm);
  snprintf(tempBuffer, sizeof(tempBuffer), "Date: %s\r\n", dateBuffer);
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  char *errMesg = (char *)"<h1>500: Internal Server Error</h1>";
  snprintf(tempBuffer, BYTES, "Content-Length: %d\n", (int)strlen(errMesg));
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  snprintf(tempBuffer, BYTES, "Content-Type: text/html\r\n");
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  send(clientSock, "Connection: close\r\n\r\n", 26, 0);
  if (curRequest ≠ HEAD)
    send(clientSock, errMesg, strlen(errMesg) + 1, 0);
  exit(1);
  return 0;
}
```

```
// Send 501 status
int sendNotImplemented(int clientSock, requestType curRequest) {
  char tempBuffer[1024];
  write(clientSock, "HTTP/1.0 501 Not Implemented\r\n", 30); // FILE NOT FOUND
  send(clientSock, "Server: Alchemist\r\n", 19, 0);
  char dateBuffer[BYTES];
  time_t now = time(0);
  struct tm tm = *gmtime(&now);
  strftime(dateBuffer, sizeof(dateBuffer), "%a, %d %b %Y %H:%M:%S %Z", &tm);
  snprintf(tempBuffer, sizeof(tempBuffer), "Date: %s\r\n", dateBuffer);
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  char *errMesg = (char *)"<h1>501: Not Implemented</h1>";
  snprintf(tempBuffer, BYTES, "Content-Length: %d\n", (int)strlen(errMesg));
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  snprintf(tempBuffer, BYTES, "Content-Type: text/html\r\n");
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  send(clientSock, "Connection: keep-alive\r\n\r\n", 26, 0);
  if (curRequest ≠ HEAD)
    send(clientSock, errMesg, strlen(errMesg) + 1, 0);
  return 0;
}

// generates directory listing for hyperlinked directory part
char *generateDirectoryList(char *path, int &size) {
  DIR *dir;
  int idx = 0, written;
  char *retVal = (char *)malloc(99999);
  struct dirent *ent;
  if ((dir = opendir(path)) ≠ NULL) {
    /* print all the files and directories within directory */
    written = sprintf(retVal + idx, "<html><body><ul>\n");
    idx += written;
    while ((ent = readdir(dir)) ≠ NULL) {
      written = sprintf(retVal + idx, "<li><a href=\"%s%s\">%s</a></li>\n",
                        path + strlen(ROOT), ent->d_name, ent->d_name);
      idx += written;
    }
    written = sprintf(retVal + idx, "</ul></body></html>\n");
    idx += written;
    size = strlen(retVal);
    closedir(dir);
    return retVal;
  } else {
    /* could not open directory */
    // This should never happen, because we can stat the directory.
    // So we have exec permission.
    perror("Could not open directory");
    return NULL;
  }
}

// Write size bytes form buffer to clientSock
int writeBuffer(int clientSock, char *buffer, int size) {
  int totalWritten = 0, written = 0;
```

```
  while (totalWritten < size) {
    written = send(clientSock, buffer + totalWritten, size - totalWritten, 0);
    if (written <= 0)
      return -1;
    totalWritten += written;
  }
  return 0;
}

// respond to HEAD and GET requests
int respondHG(char *path, int clientSock, requestType curRequest,
              int keepAliveStatus) {
  int fd, bytes_read, isDirectory, size;
  char data_to_send[BYTES], *directoryList;
  char mimeType[20];
  struct stat statsBuf;
  printf("file: '%s'\n", path);

  if (stat(path, &statsBuf) == 0) {
    if (statsBuf.st_mode & S_IFDIR) {
      isDirectory = 1;
      directoryList = generateDirectoryList(path, size);
      strcpy(mimeType, "text/html");
    } else { // it is a file
      size = (int)statsBuf.st_size;
      // rest of the block runs file command in linux to get
      // mimetype. write the output of command to a file and
      // read it and delete it.
      char command[1024], tempFile[1024];
      snprintf(tempFile, 1024, "mimeT%d", clientSock);
      snprintf(command, 1024,
               "file --mime-type %s | sed -n 's/.*: \\(.*\\)$/\\1/p' > %s",
               path, tempFile);
      system(command);
      FILE *mimeF = fopen(tempFile, "r");
      fscanf(mimeF, "%s", mimeType);
      fclose(mimeF);
      snprintf(command, 100, "rm -f %s", tempFile);
      system(command);
    }
    //    printf("Mime Type : %s\n", mimeType);
    sendCommonHeaders(clientSock, size, keepAliveStatus, mimeType);
    if (curRequest == GET) {
      if (isDirectory)
        writeBuffer(clientSock, directoryList, size);
      else {
        fd = open(path, O_RDONLY);
        while ((bytes_read = read(fd, data_to_send, BYTES)) > 0)
          write(clientSock, data_to_send, bytes_read);
      }
    }
  } else { // If cannot stat, then as good as not found
    sendNotFound(clientSock, curRequest);
  }
  return 0;
```

```
}

// responds to POST request. strtok here is continued from parsing function
// with the same string. We read whatever payload was remaining in
// buffer, and keep recieving till we have recieved it entirely.
// Then, we write it to file.
int respondPOST(char *path, int clientSock, int postPayloadLength,
                int keepAliveStatus) {
  char postPayload[postPayloadLength], *initialPayload, tempBuffer[BYTES];
  initialPayload = strtok(NULL, "");
  int readTillNow = strlen(initialPayload), recieved;
  FILE *fd = fopen(path, "w");
  strcpy(postPayload, initialPayload);
  while (readTillNow < postPayloadLength) {
    recieved = recv(clientSock, tempBuffer,
                    min(BYTES, postPayloadLength - readTillNow), 0);
    strncpy(postPayload + readTillNow, tempBuffer,
            min(postPayloadLength - readTillNow, recieved));
    readTillNow += recieved;
  }
  fwrite(postPayload, 1, postPayloadLength, fd);
  fflush(fd);
  fclose(fd);
  char *successMesg = (char *)"<h1>Content written Succesfully</h1>";
  send(clientSock, "HTTP/1.1 200 OK\r\n", 17, 0);
  send(clientSock, "Server: Alchemist\r\n", 19, 0);
  char dateBuffer[BYTES];
  time_t now = time(0);
  struct tm tm = *gmtime(&now);
  strftime(dateBuffer, sizeof(dateBuffer), "%a, %d %b %Y %H:%M:%S %Z", &tm);
  snprintf(tempBuffer, sizeof(tempBuffer), "Date: %s\r\n", dateBuffer);
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  snprintf(tempBuffer, BYTES, "Content-Type: text/html\r\n");
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  snprintf(tempBuffer, BYTES, "Content-Length: %d\r\n",
           (int)strlen(successMesg));
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  if (keepAliveStatus == 1)
    snprintf(tempBuffer, BYTES, "Connection: Close\r\n\r\n");
  else
    snprintf(tempBuffer, BYTES, "Connection: keep-alive \r\n\r\n");
  send(clientSock, tempBuffer, strlen(tempBuffer), 0);
  send(clientSock, successMesg, strlen(successMesg), 0);
  return 1;
}

// Main function that is called after a fork().
// while loop on recv, calls, parseHeaders when it recieves
// some data, and then corresponding requestType handler
void serveClient(int clientSock) {
  char mesg[BYTES], path[BYTES];
  int rcvd, keepAliveStatus = 0, payloadLength, idx = 1;
  memset((void *)mesg, 0, BYTES);
  fprintf(stderr, "Connected to client %d\n", clientSock);
  while ((rcvd = recv(clientSock, mesg, BYTES - 1, 0)) > 0) {
```

```
    fprintf(stderr, "\nRequest %d from Client %d\n", idx++, clientSock);
    requestType currReq =
        parseHeaders(mesg, keepAliveStatus, path, payloadLength);
    if (currReq == GET || currReq == HEAD)
      respondHG(path, clientSock, currReq, keepAliveStatus);
    else if (currReq == POST)
      respondPOST(path, clientSock, payloadLength, keepAliveStatus);
    else if (currReq == UNIMPLEMENTED)
      sendNotImplemented(clientSock, currReq);
    else if (currReq == BAD)
      sendBadRequest(clientSock, currReq);
    else if (currReq == ERROR)
      sendInternalError(clientSock, currReq);
    if (keepAliveStatus)
      break;
  }
  fprintf(stderr, "Closed Client\n");
  close(clientSock);
}


//Parses request headers
//keepAliveStatus: 1 if connection is to be closed
//path: path of file to be fetched in GET or HEAD and
//      path of file to which payload is to be writeen in POST
//payloadLength: length of payload in case of POST, irrelevant in others
requestType parseHeaders(char *mesg, int &keepAliveStatus, char *path,
                         int &payloadLength) {
  char *reqline[3], *headerLine;
  try {
    requestType curRequest;
    keepAliveStatus = 0;
    reqline[0] = strtok(mesg, " \t");
    if (strncmp(reqline[0], "GET\0", 4) == 0)
      curRequest = GET;
    else if (strncmp(reqline[0], "HEAD\0", 5) == 0)
      curRequest = HEAD;
    else if (strncmp(reqline[0], "POST\0", 5) == 0)
      curRequest = POST;
    else if (strncmp(reqline[0], "PUT\0", 4) == 0 ||
             strncmp(reqline[0], "DELETE\0", 7) == 0 ||
             strncmp(reqline[0], "CONNECT\0", 8) == 0 ||
             strncmp(reqline[0], "OPTIONS\0", 8) == 0)
      return UNIMPLEMENTED;
    else
      return BAD;
    reqline[1] = strtok(NULL, " \t");
    reqline[2] = strtok(NULL, "\n");
    printf("%s %s %s", reqline[0], reqline[1], reqline[2]);
    if ((strncmp(reqline[2], "HTTP/1.1", 8) ≠ 0) &&
        (strncmp(reqline[2], "HTTP/1.0", 8) ≠ 0)) {
      return BAD;
    } else {
      if (strncmp(reqline[1], "/\0", 2) == 0) {
        if (curRequest == POST)
```

```
          reqline[1] = (char *)"/post_file_test.txt";
        else
          reqline[1] = (char *)"/index.html";
      }
      strcpy(path, ROOT);
      strcpy(&path[strlen(ROOT)], reqline[1]);
    }
    printf("\n");
    headerLine = strtok(NULL, "\n");
    while (strlen(headerLine) > 1) {
      printf("%s\n", headerLine);
      if ((strncmp(headerLine, "Connection: Close", 17) == 0) ||
          (strncmp(headerLine, "Connection: close", 17) == 0))
        keepAliveStatus = 1;
      if (strncmp(headerLine, "Content-Length", 14) == 0) {
        payloadLength = atoi(headerLine + 16);
      }
      headerLine = strtok(NULL, "\n");
    }
    return curRequest;
  } catch (...) {
    return ERROR;
  }
}
```