Akshita Mittel
CS13B1040

14 September 2015

HTTP Server and Client

CS3041: Computer Networks, Assignment 2

1. SERVER:

The HTTP server was created using a TCP socket that read the URL from the browser and parsed the corresponding GET message. The path provided in the GET message was used to retrieve the corresponding file, if it was present in the servers directory, else and error message was raised. The implementation of the server code is as follows:

The HOST is defined to be the localhost. An attempt is made to connect to PORT 80, if this fails we try to connect to PORT 8000, which is in the permitted range. However, if both the ports aren't accessible, we quit using sys.exit(0).

If the socket connects to the host and port we send the socket, namely the server to the function connect(server), where it tries to host the page that has been requested by the browser.

A maximum queue size of 3 is permitted. If the socket receives a connection from the browser, it accepts and receives the GET request. It then parses the GET request as follows:

1. The request is of the form "GET /path HTTP/1.1\r\n\r\n"
2. The data is split according to the spaces:
3. Segment[0] = GET
4. Segment[1] = /path
5. Segment[2] = HTTP…

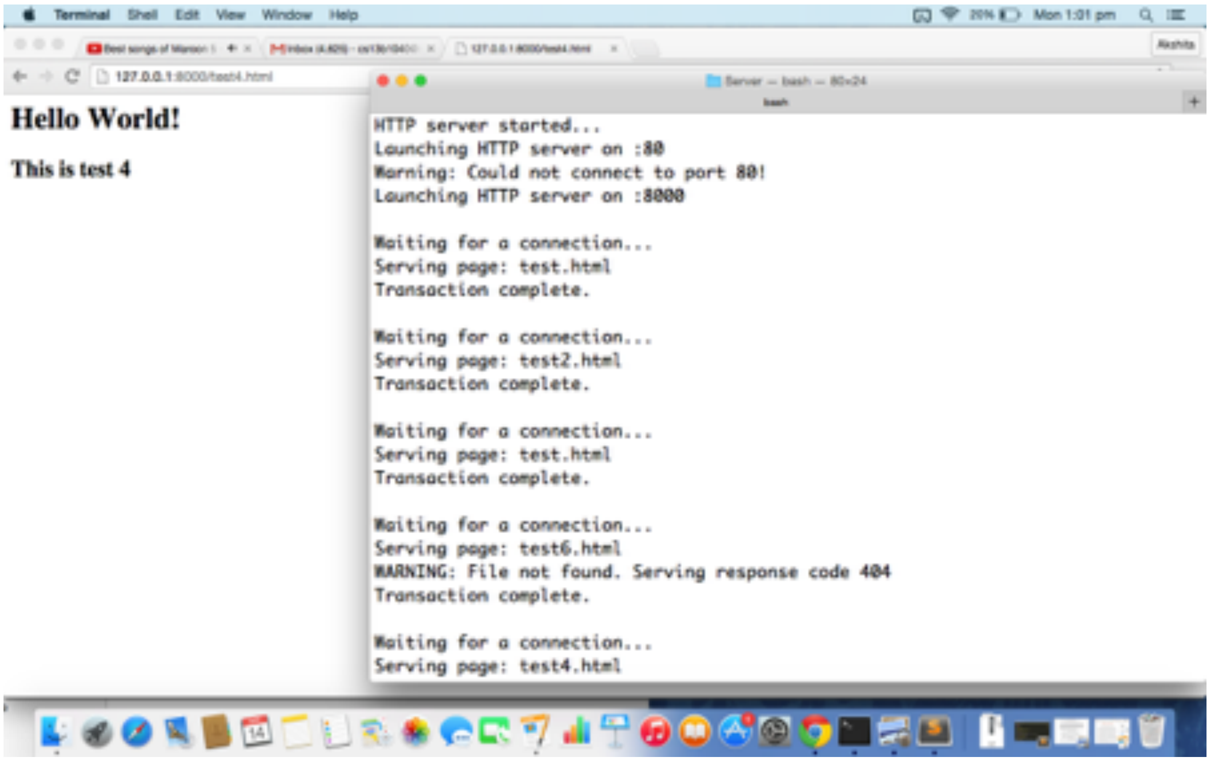
If the message received is of the type GET, then we proceed to get the path specified by the message, else we ignore it and send an incorrect message.

In the true condition, we try to open the file specified by the user, if the file exists we call for an OK header, by sending the code 200 to the header file. To this header we append our file.

If the file does not exist, we get the header file corresponding to 404, and append a self made error webpage.

We send this message back through the connection and close it. Allowing the server to wait for another connection.

1. **The program being executed on the terminal**

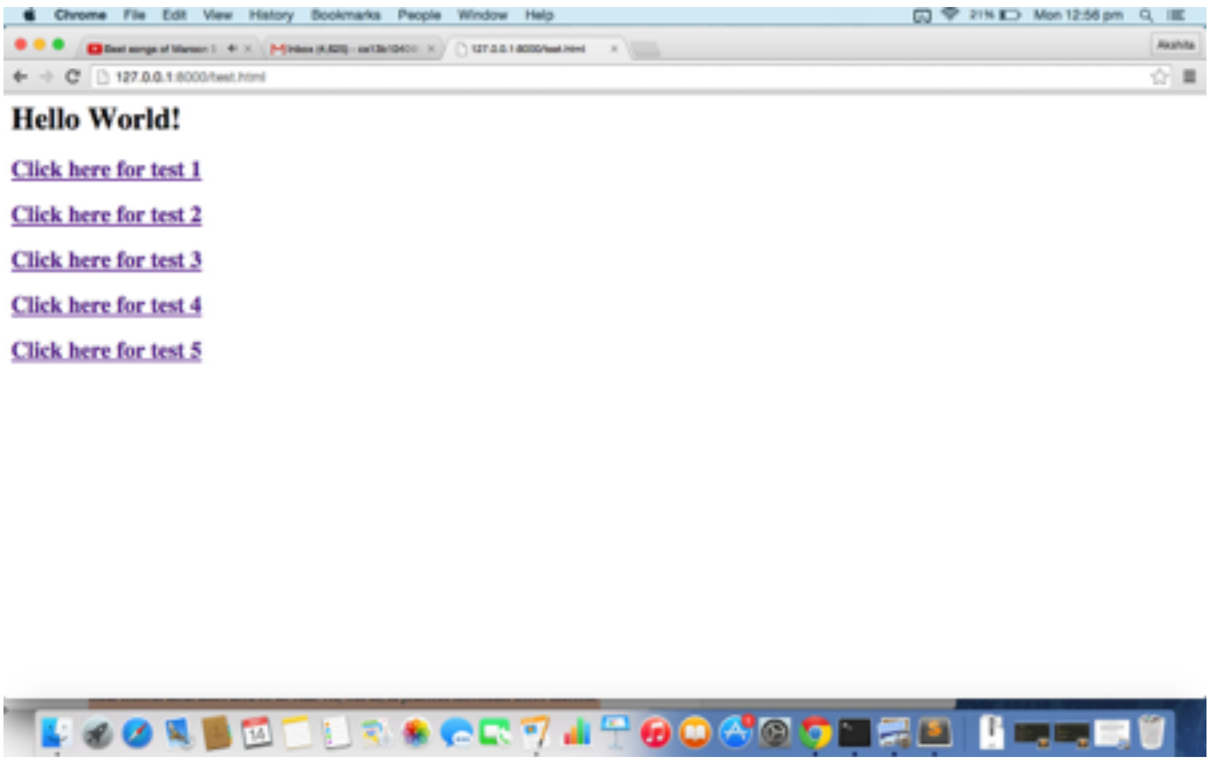


Hello World!

This is test 4

```
HTTP server started...
Launching HTTP server on :80
Warning: Could not connect to port 80!
Launching HTTP server on :8000

Waiting for a connection...
Serving page: test.html
Transaction complete.

Waiting for a connection...
Serving page: test2.html
Transaction complete.

Waiting for a connection...
Serving page: test.html
Transaction complete.

Waiting for a connection...
Serving page: test6.html
WARNING: File not found. Serving response code 404
Transaction complete.

Waiting for a connection...
Serving page: test4.html
```

2. **The main file**



Hello World!

Click here for test 1

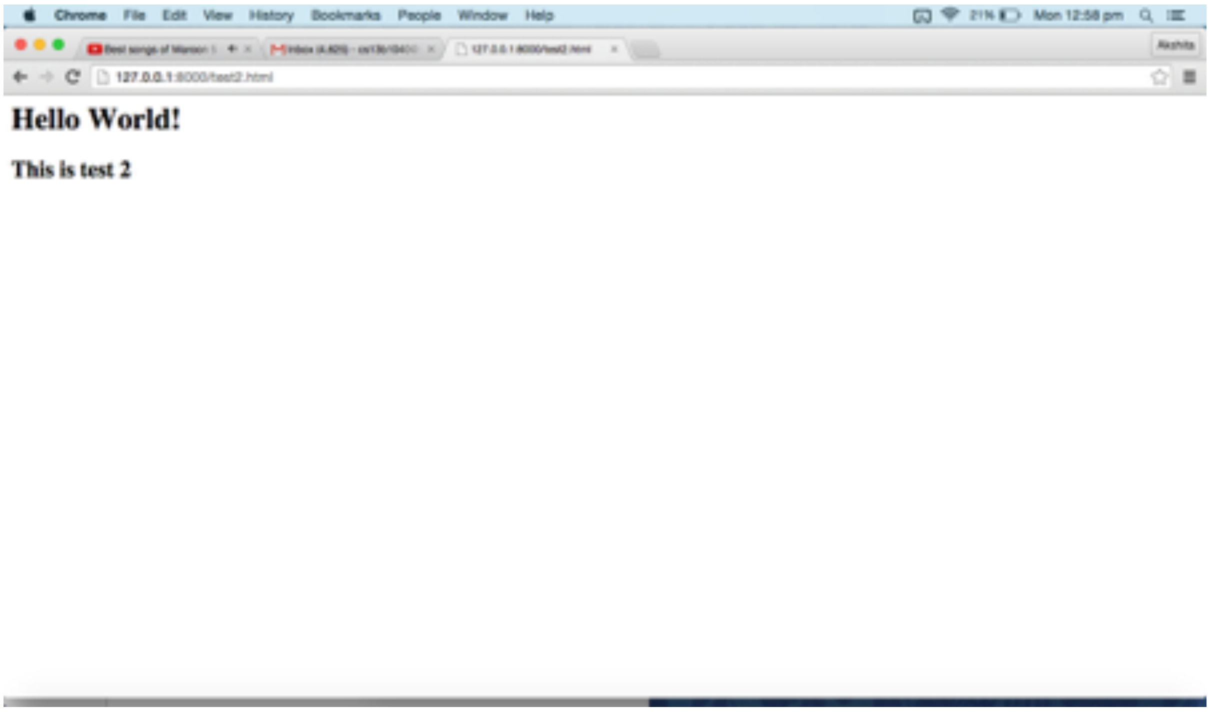
Click here for test 2

Click here for test 3

Click here for test 4

Click here for test 5

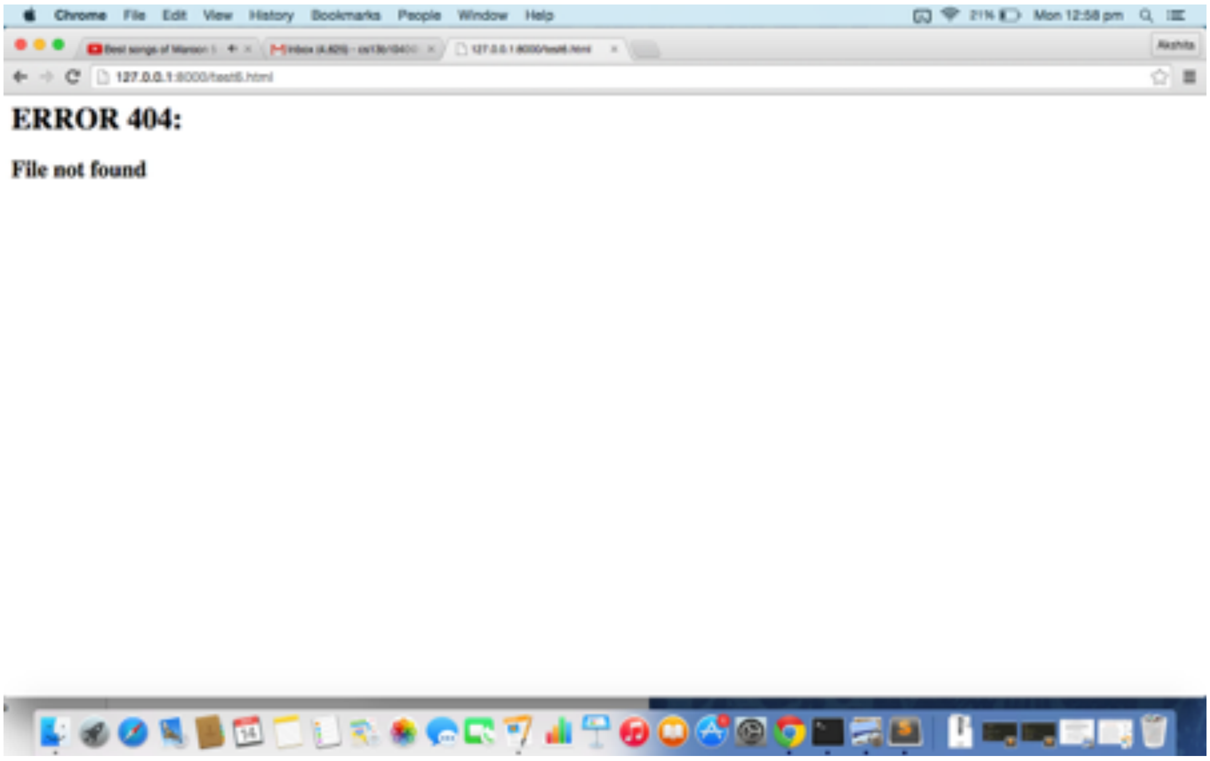
3. **A hyperlink from the main page**


Chrome File Edit View History Bookmarks People Window Help
Mon 12:58 pm
127.0.0.1:8000/test2.html

Hello World!

This is test 2

4. **A file which does not exist**


Chrome File Edit View History Bookmarks People Window Help
Mon 12:58 pm
127.0.0.1:8000/test6.html

ERROR 404:

File not found

2. CLIENT:

The client program takes as an argument the HOST, PORT, and the path of the file that has to be retrieved from the browser. It then establishes a connection by creating a socket using this information. The socket created is such that it can reuse the given address over again.

The next step is to create the header for the request. This should be in the correct protocol, which the browser will accept. The standard protocol could look something like this:

"GET /path.(html) HTTP/1.0\r\n\r\n"

GET is the request that we are sending to the browser.

The path and file that is requested (with the appropriate extension).

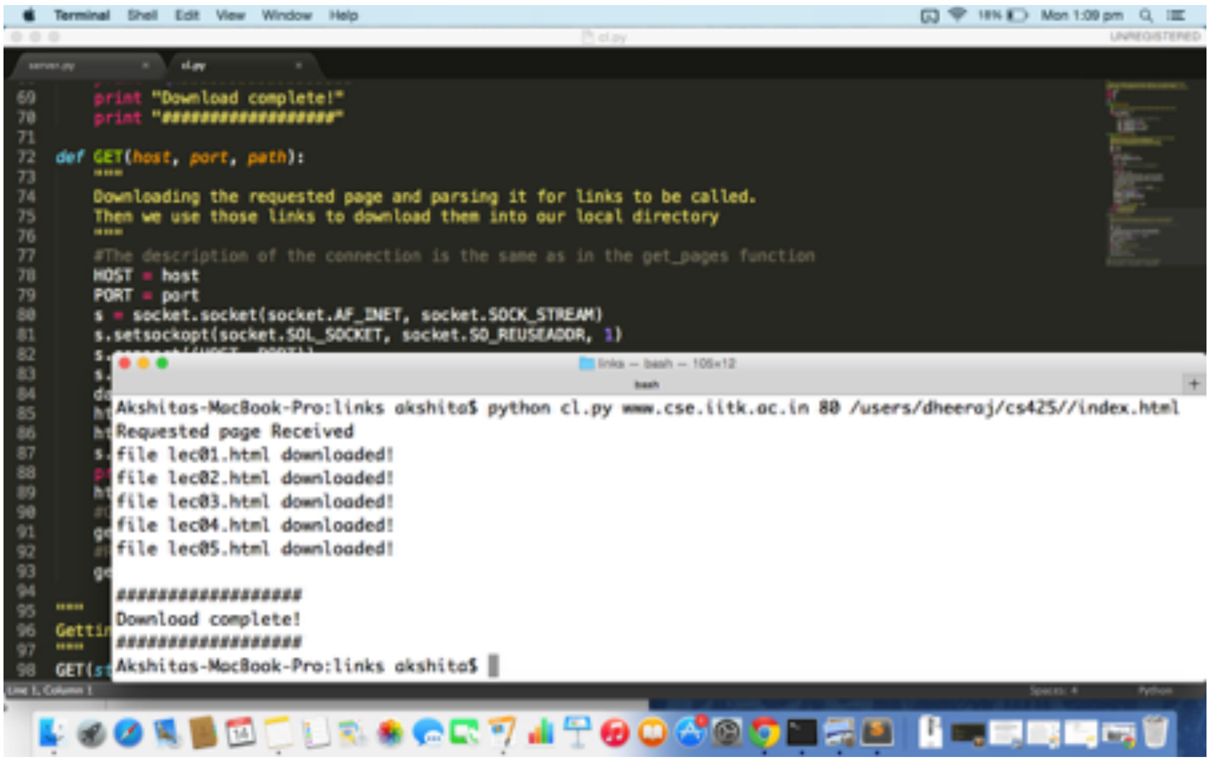
After getting that information we receive the data from the browser and store it into a file. We then send the name of the file into the "get_links" function where it attempts to parse and retrieve all the links in the page as follows:

For each line in the page, search for a line which has "href". Find the link enclosed in quotes succeeding the href. Extract that link using regular expressions, and only keep the required portion of the link. Append this link onto a list "links" that we will use later. Return to the main function.

From the main function we send the details about the host, port, and path to the "get_pages" function. Here we modify the path and append the link path, in order to obtain the page. Once again we establish a connection to the browser in a manner similar to the one that we adopted in the main function. We send headers to retrieve the file and store them in the clients directory. After each directory we tell the client that it has been downloaded.

We set a limit of "n" links, in this case 5 was chosen arbitrarily. So that a maximum of 5 files can be downloaded from each page.

The client receiving messages, whenever a file is downloaded:



```
    print "Download complete!"
    print "###################"

def GET(host, port, path):
    """
    Downloading the requested page and parsing it for links to be called.
    Then we use those links to download them into our local directory
    """
    #The description of the connection is the same as in the get_pages function
    HOST = host
    PORT = port
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

```
Akshitas-MacBook-Pro:links akshita$ python cl.py www.cse.iitk.ac.in 80 /users/dheeraj/cs425//index.html
Requested page Received
file lec01.html downloaded!
file lec02.html downloaded!
file lec03.html downloaded!
file lec04.html downloaded!
file lec05.html downloaded!

###################
Download complete!
###################
Akshitas-MacBook-Pro:links akshita$ 
```