|  |  |
| --- | --- |
| **ISHWAR BHAT**  **DEEPAK PARAMESHWARAN**  **HYUNWOOK PARK** | **Professor: Dr. Jim Helm** |
| **IFT 520: Principles of Telecommunication Technology** | **Project 2** |

**Title of Report: Building a Multi-threaded Web Server**

**Summary:**

This project saw us develop a Web Server in two steps: Accept and print the client’s HTTP Request; send an HTTP response to the client. In the end, we had built a multi-threaded web server capable serving multiple clients in parallel.

***Checklist (items included):***

|  |  |  |
| --- | --- | --- |
| **1. Objectives** |  | **4. Screenshots** |
| **2. Introduction & Background** |  | **5. Source Code** |
| **3. Procedure and Approach** |  | **6. Conclusions** |

**Key Results:**

Successful deployment of Web Server has been achieved using Python.

**Key Conclusions (technical):**

In this project, we developed a Web server in two steps. We built a multi-threaded Web server that is capable of processing multiple simultaneous service requests in parallel. We can demonstrate that our Web server can deliver the home page to a Web browser, as well as other objects like images and text.

In the first part of the project, we developed a multithreaded web server that can display the contents of the HTTP GET message (generated using a standard web browser).

In the second part of the project, we built upon the server and we could respond to the client’s request with the requested objects.

**I. DELIVERABLES INCLUDED:**

1. Copy of completed code used in your server system for both the server and the client.

2. Written summary of your experiences in working this project and include what you learned as a part of this exercise. Describe the structure of your program and include screenshots of your work, as appropriate. Limit this written overview to seven pages. This does not include the code, which is a required delivery item for this project.

3. Include a Wireshark printout with the message sequences.

**II. INTRODUCTION:**

Web server is an application which runs on server machine and listens request from various clients on specific port and gives some specific response for each request. Web servers and client machines use sockets for transferring data between each other. In general if we construct simple web server using single execution instance then it will be able to serve only one client at time, so if multiple clients want to connect to the server then other clients may need to wait. To resolve this problem we can construct multithreaded server, which will create and execute a single thread for each individual request (may be from same client machine or different client machines).

What we will be doing in this project:

1. Establish the listen socket

2. Listen for a TCP Connection request

3. Construct an object to process the HTTP request message.

4. Get a reference to the socket’s input and output streams.

5. Set up input stream filters.

6. Get the request line of the HTTP request message.

**III. APPROACH:**

We decided to use an AWS (Amazon Web Services) EC2 instance running Ubuntu 14.04 as our machine/hardware for the Web Server. Also, we used Python 2.7 as the programming language for two reasons: 1. Simple and easy to develop applications; and 2 Available by default in our selected machine.

**Part A: Basic Multithreaded Web Server**

We started by implementing a simple Web Server which accepts HTTP GET requests, and print the request headers in our console.

* To begin with, we used BaseHTTPServer module for implementing our HTTP web server. In this module, we used HTTPServer class to create and listens at the HTTP socket, dispatching the requests to a handler.
* For handling the requests, we used SimpleHTTPRequestHandler class from SimpleHTTPServer module.
* The do\_GET() method in the Handler maps the request to a local file.

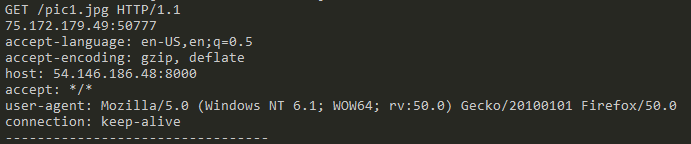
**Multithreading**

* Python provides SocketServer module to allow network servers to process requests asynchronously. In particular, we used ThreadingMixIn class to create a separate process or thread to handle each request.
* The serve\_forever() method of the SocketServer class keeps handling requests until we explicitly shutdown the server with a KeyboardInterrupt.

The connection basically happens over a 3-step process:

1. A RequestHandler class is created, subclassing SimpleHTTPRequestHandler  class.
2. The server class ThreadingHTTPServer is instantiated (as the variable httpd), passing it the server’s address (‘’ or ‘localhost’), port (8000) and the request handler.
3. The serve\_forever() method of the server class is called, to process one or multiple client requests.

In the do\_GET() method of the request handler, the headers dictionary contains a mapping of the header keys and values. This was printed to the console output, along with  command (GET), requested path (starting with /) and request\_version, like below:



Also, multiple simultaneous connections from different computers were set up and the multithreading was successfully verified.

**Part B: Request Analysis and Response**

Once we were able to successfully decode a client request; we began with parsing the request.

The path variable of the request handler contains the requested path from the client.

We tested different MIME types, as below:

* text (/, /index.html)
* image (/pic1.jpg, /arnold.gif)

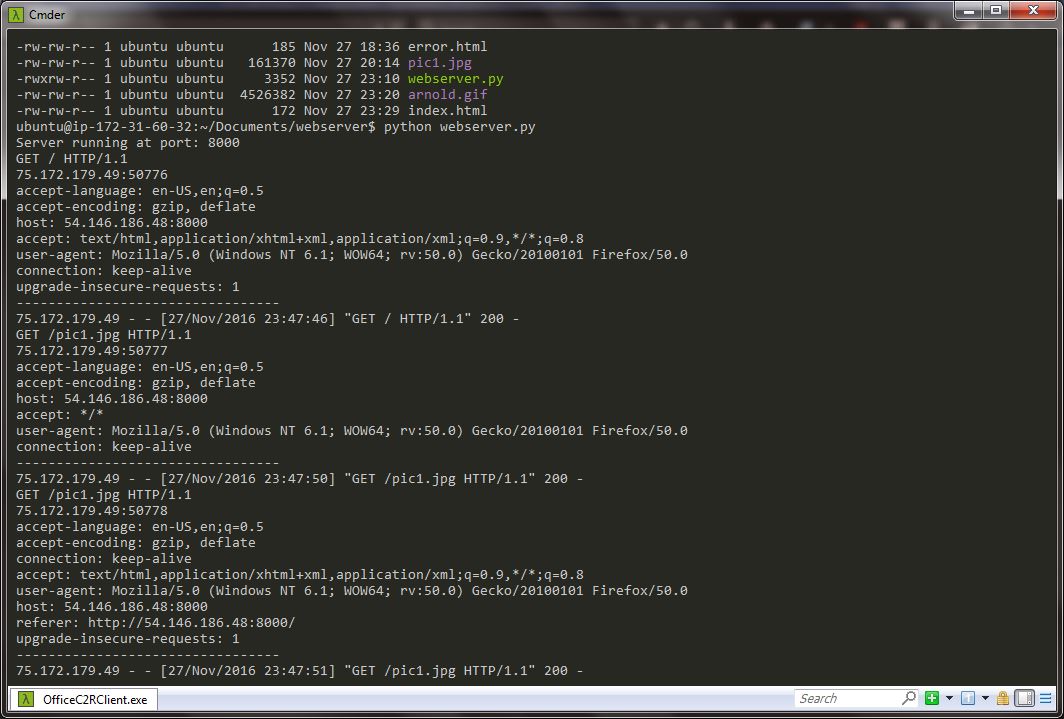
The program tests the ending part of the path string, and associates the corresponding MIME type. If the file is present in the server’s directory, a 200 OK Response along with the file is sent back to the client.

Exceptions (IOError) are handled while opening the requested file. If an unknown MIME type is found or a known MIME type but file not present, 404 Error Code with a custom error page (/error.html) is sent back to the client.

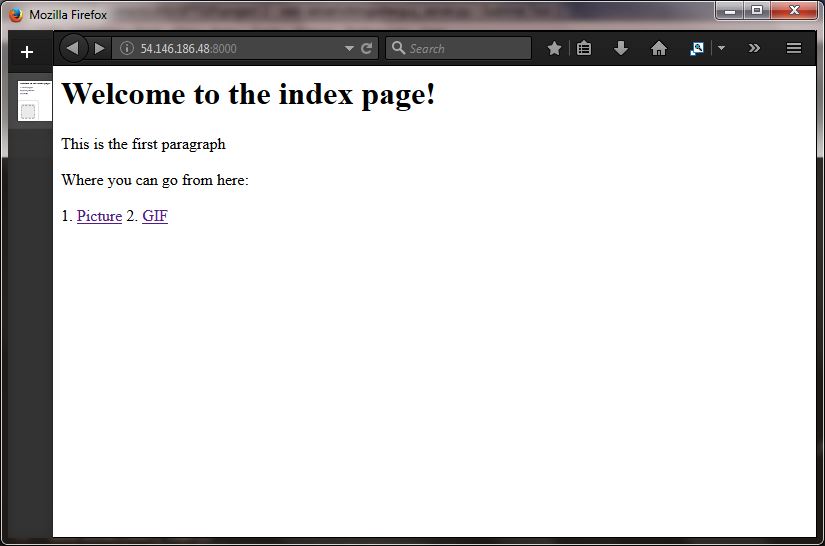
Since the server keeps running indefinitely, a keyboard Interrupt (Ctrl + C) is required to manually terminate the server.

**IV. SCREENSHOTS**

The following figure shows the port number where the server is running on, and the HTTP GET request from the client (Part A)

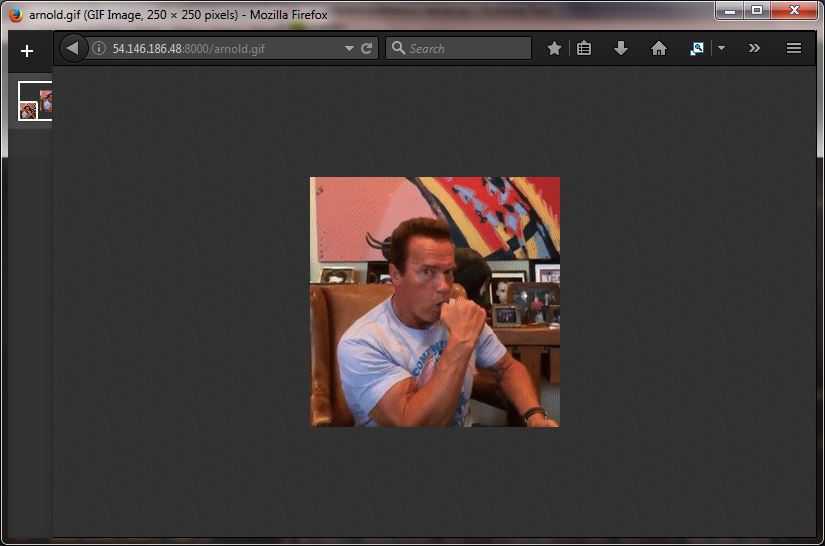
****

The following figure demonstrates that our Web server is capable of delivering the home page to a Web browser.

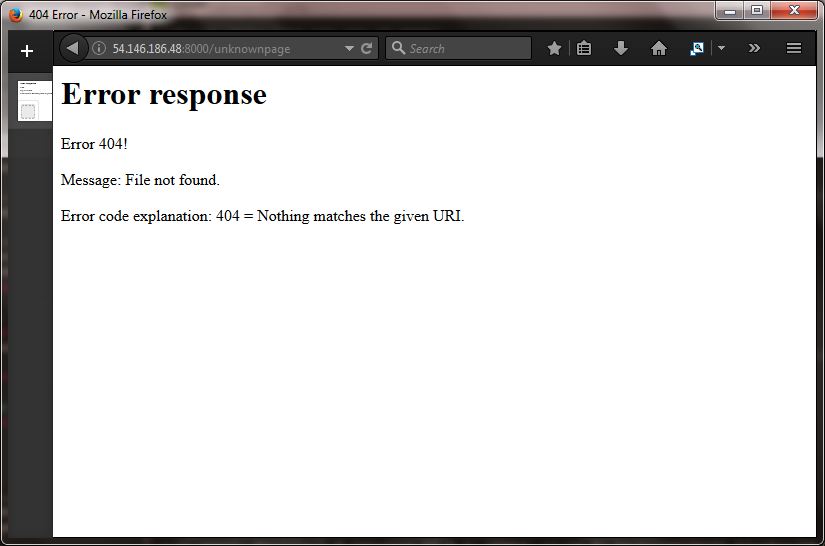


Testing different MIME types with .jpg and .gif:

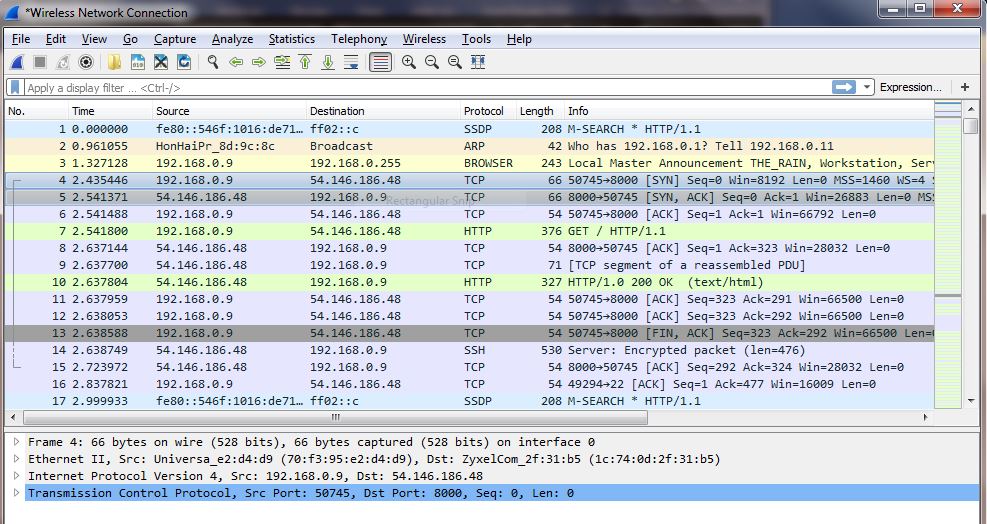




Testing error message for invalid request:



Wireshark Printout of message sequences:



**V. CONCLUSION**

With this project, we learnt how a server handles request from clients. We learnt about the structure of an HTTP GET request and how it is sent from client to a server. Also, we learnt about the usage of ports to set up a TCP connection (port 80 used by HTTP, and why ports below 1024 shouldn’t be used by other than root or privileged users). Multithreading demonstrated the true capabilities of the server – serving multiple clients simultaneously.

We also gained knowledge of the various MIME types, and how these requests (as well as exceptions) are handled by the server.

In all, this project was a great learning experience!

**VI. CODE**

1. **import** BaseHTTPServer
2. **import** SimpleHTTPServer
3. **from** SocketServer **import** ThreadingMixIn
4. **import** os
5. **from** os **import** getcwd
6. **import** socket
7. **import** errno
9. CURR\_DIR = getcwd()

12. **class** ThreadingHTTPServer(ThreadingMixIn, BaseHTTPServer.HTTPServer):
13. **pass**
15. **class** RequestHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):
17. current\_path = CURR\_DIR
19. # Handle requests
20. **def** do\_GET(self):
21. # Print request header information
22. client\_host, client\_port = self.client\_address
24. **print** self.command, self.path, self.request\_version
25. **print** client\_host + ":" + str(client\_port)
27. **for** key **in** self.headers:
28. **print** key + ":", self.headers[key]
29. **print** "---------------------------------"
31. **if** self.path=="/":
32. self.path="/index.html"
34. # Check the file extension and set the mimetype
36. replyFlag = False
37. **if** self.path.endswith(".html"):
38. mimetype='text/html'
39. replyFlag = True
40. **if** self.path.endswith(".jpg"):
41. mimetype='image/jpg'
42. replyFlag = True
43. **if** self.path.endswith(".gif"):
44. mimetype='image/gif'
45. replyFlag = True
46. **if** self.path.endswith(".js"):
47. mimetype='application/javascript'
48. replyFlag = True
49. **if** self.path.endswith(".css"):
50. mimetype='text/css'
51. replyFlag = True
53. # Send the static file
54. **if** replyFlag:
56. **try**:
57. file\_path = self.path.lstrip(os.sep)
58. f = open(file\_path)
59. self.send\_response(200)
60. self.send\_header('Content-type',mimetype)
61. self.end\_headers()
62. self.wfile.write(f.read())
63. f.close()
65. **except**:
67. **print** 'Cannot open:', self.path
68. f = open('error.html')
69. self.send\_response(404)
70. self.send\_header('Content-type','text/html')
71. self.end\_headers()
72. self.wfile.write(f.read())
73. f.close()
74. **else**:
75. f = open('error.html')
76. self.send\_response(404)
77. self.send\_header('Content-type','text/html')
78. self.end\_headers()
79. self.wfile.write(f.read())
80. f.close()

83. **return**


87. **def** get\_server(port, remaining\_attempts=0, current\_path=None):
88. Handler = RequestHandler
89. **if** current\_path:
90. Handler.current\_path = current\_path
91. **while** remaining\_attempts >= 0:
92. **try**:
93. httpd = ThreadingHTTPServer(("", port), Handler)
94. **return** httpd
95. **except** socket.error as e:
96. **if** e.errno == errno.EADDRINUSE:
97. remaining\_attempts -= 1
98. port += 1
99. **if** e[0] == errno.EPIPE:
100. # remote peer disconnected
101. **print** "Detected remote disconnect"
102. **else**:
103. **raise**
105. **def** main():
106. PORT = 8000
107. current\_path = CURR\_DIR
109. httpd = get\_server(port=PORT, current\_path=current\_path)
110. **print** "Server running at port:", PORT
112. **try**:
113. httpd.serve\_forever()
114. **except** KeyboardInterrupt:
115. **pass**
117. **if** \_\_name\_\_ == "\_\_main\_\_" :
118. main()

Syntax highlighting done using http://www.planetb.ca/syntax-highlight-word

**VII. WIRESHARK PRINT**

Here are the summaries of a couple of TCP and HTTP packets, sniffed using Wireshark:

(**192.168.0.9** – Client **54.146.186.48**:**8000** – Server )



4 2.435446 192.168.0.9 54.146.186.48 TCP 66 50745→8000 [SYN] Seq=0 Win=8192 Len=0

MSS=1460 WS=4 SACK\_PERM=1

5 2.541371 54.146.186.48 192.168.0.9 TCP 66 8000→50745 [SYN, ACK] Seq=0 Ack=1 Win=26883

Len=0 MSS=1452 SACK\_PERM=1 WS=128

6 2.541488 192.168.0.9 54.146.186.48 TCP 54 50745→8000 [ACK] Seq=1 Ack=1 Win=66792

Len=0

7 2.541800 192.168.0.9 54.146.186.48 HTTP 376 GET / HTTP/1.1

10 2.637804 54.146.186.48 192.168.0.9 HTTP 327 HTTP/1.0 200 OK (text/html)

5303 30.538995 192.168.0.9 54.146.186.48 HTTP 379 GET /xxx HTTP/1.1

5306 30.626150 54.146.186.48 192.168.0.9 HTTP 340 HTTP/1.0 404 Not Found (text/html)

* Packets 4 – 6 show the setting up of a TCP connection (SYN, SYN-ACK and ACK)
* Packets 7 and 10: HTTP GET request and 200 OK response from the server
* Packets 5303 and 5306: HTTP GET request (invalid) and 404 Not Found response

[ Detailed capture file included along with the report ]