Network Architecture and Protocol

Project 1 – A Client -Server Application

Date – 5th October 2017

Report by – Abhinuv Nitin Pitale (906039146)

Tools Used – Python 2.7.13, Command Prompt (Windows command Line utility), Git

Python libraries used - Socket, Re, CxFreeze, math, os

Operation System Tested on – Windows 10, Linux (Ubuntu)

Implementation

A. Server

Design Considerations:
BUFFERSIZE = 1024
Uses a polling type of server, which is always on and gets activated for incoming requests
Does not involve multithreading, hence only sequential sockets can be accepted.

Algorithm/Code Flow:

1. Get user input for selecting the port number for the server to be set-up on.
2. Create a TCP socket and bind it to the port.
3. Wait for the server to listen to incoming requests.
4. Accept the incoming socket and create a new connection socket for receiving and sending data.
5. Receive a HTTP query from the incoming socket. (Assumption : The incoming query length will be less than 1024 bits.)
6. Decode the query –
   a. Split the query into 3 parts
      i. Method Used (for eg – GET,POST
      ii. Filename Requested (for eg – docTest.txt)
      iii. HTTP version
   b. Check if the method used and the Http version is valid. If not, set the state as 400
   c. Check if the filename queried exists on the server. If not, set the state as 404
   d. If b and c is true, set the state as 200
   e. Return state and filename(if exists)
7. Create the Http Response to be sent over the server.
8. Check state and return the query to be sent over the server.
   a. If it is 400, output the following query

      HTTP/1.0 400 Bad Request

   b. If it is 404, output the following query

      HTTP/1.0 404 Not Found

   c. If it is 200,
      i. Read the file as binary.
      ii. Compute it's length
      iii. Send the query in this particular format-
      HTTP/1.0 404 Not Found
      Content-Length: *length*

      CONTENT
9. Send the data over the socket by computing it's length and sending it by parts, if the buffersize is overloaded.
10. Close the connection socket and go back to step 3.

B. Client

Design Considerations:

BUFFERSIZE = 1024

CodeFlow/Algorithm:

1. Query the user for host and port number.
2. Create a TCP socket connected to that port.
3. Display connection as success, if it passes step 2.
4. Query user for a filename to retrieve from the server.
5. Create the Http Request:

   query = GET /filename HTTP/1.0

6. Send the query via socket
7. Receive the server response and parse it
   a. Read the state from the response
      i. If 400, display 'Bad Request'
      ii. If 404, display 'File Not Found'
      iii. If 200, parse the data in the request.

         Open a file with the same name and store the *filecontent* in it.

8. Close the socket.

Error Checking implemented in the code:

1. If the user input is blank during querying for host and portnumber, a default port and localhost is specified.
2. Exception handling is implemented on both server and client side, which prevents the server from loading in case of a crash at the client side.

Testing Procedure

1. My client with the test server

a. Positive Testing

Server--

```
Command Prompt - java  HttpServer 8794

c:\Courses\Network Architechture and Protocols\Project 1\test Pack\package-for-student\sample-server>jav
a HttpServer 8794
0 connections served. Accepting new client...
1 connections served. Accepting new client...
```

Client--

```
Select Command Prompt
Minimize
c:\Courses\Network Architechture and Protocols\Project 1\Socket-Programming>python clientNew.py
Enter Server Host Name : localhost
Enter Server Port Number : 8794
Connected to server on port :  8794
Input File Name to query : docTest.txt
File  docTest.txt  created
```

➔ File Correctly received over the connection.

b. Negative testing

i. No localhost port specified.

```
Command Prompt
Minimize
c:\Courses\Network Architechture and Protocols\Project 1\Socket-Programming>python clientNew.py
Enter Server Host Name :
Enter Server Port Number :
Connected to server on port :  8794
Input File Name to query : docTest.txt
File  docTest.txt  created
```

➔ Even without specifying the port number/host, the client takes the default values and runs correctly.

ii. Incorrect Filename

```
Command Prompt
Minimize
c:\Courses\Network Architechture and Protocols\Project 1\Socket-Programming>python clientNew.py
Enter Server Host Name :
Enter Server Port Number :
Connected to server on port :  8794
Input File Name to query : pingu1
File Not Found!
```

➔ Displays error message without crashing!

2. My server with test client

a. Positive Testing

Client

```
Command Prompt

C:\Courses\Network Architechture and Protocols\Project 1\test Pack\package-for-student\sample-client>jav
a HttpClient.class localhost 8794
Error: Could not find or load main class HttpClient.class

C:\Courses\Network Architechture and Protocols\Project 1\test Pack\package-for-student\sample-client>jav
a HttpClient localhost 8794
Please input file name: docTest.txt
Send out a bad request......
SET /docTest.txt HTTP/1.0

Receive response from server .....
        HTTP/1.0 400 Bad Request



Send out the right request.....
GET /docTest.txt HTTP/1.0

Receive response from server .....
        HTTP/1.0 200 OK
        Content-Length: 120



Closing I/O and quiting...
```

Server

```
Command Prompt - python  serverNew.py
Minimize
c:\Courses\Network Architechture and Protocols\Project 1\Socket-Programming>python serverNew.py
Enter Port Number to setup the server : 8794
Server started on port :  8794
SET /docTest.txt HTTP/1.0


GET /docTest.txt HTTP/1.0
Host: localhost
```

b. Negative Testing

```
Command Prompt

C:\Courses\Network Architechture and Protocols\Project 1\test Pack\package-for-student\sample-client>jav
a HttpClient localhost 8794
Please input file name: ghochu.html
Send out a bad request......
SET /ghochu.html HTTP/1.0

Receive response from server .....
        HTTP/1.0 400 Bad Request



Send out the right request.....
GET /ghochu.html HTTP/1.0

Receive response from server .....
        HTTP/1.0 404 Not Found


File does not exist on the server!



Closing I/O and quiting...
```

3. My server with my client

a. Positive Testing

Client

```
Command Prompt

c:\Courses\Network Architechture and Protocols\Project 1\Socket-Programming>python clientNew.py
Enter Server Host Name : localhost
Enter Server Port Number : 8794
Connected to server on port :  8794
Input File Name to query : docTest.txt
File  docTest.txt  created
```

Server

```
Command Prompt - python  serverNew.py

C:\Courses\Network Architechture and Protocols\Project 1\Socket-Programming>python serverNew.py
Enter Port Number to setup the server : 8794
Server started on port :  8794
GET /docTest.txt HTTP/1.0
```

b. Negative Testing

Nonexistent file queried!

```
Command Prompt

c:\Courses\Network Architechture and Protocols\Project 1\Socket-Programming>python clientNew.py
Enter Server Host Name :
Enter Server Port Number :
Connected to server on port :  8794
Input File Name to query : doesntwork.html
File Not Found!
```

Exception testing

1. Incorrect port/host connected - Client

```
Command Prompt

c:\Courses\Network Architechture and Protocols\Project 1\Socket-Programming>python clientNew.py
Enter Server Host Name : a
Enter Server Port Number : s
Exit Gracefully!
```

Testing Procedure Used

1. Manual Testing -> Input incorrect input for negative testing
2. Automated Testing -> Used for testing 400 Bad Format. Developed python script to send incorrect format.

Summary:

The following implementation of the Http protocol over TCP works for a typical single threaded system where the requests are sent sequentially.

Features Implemented

1. Request Format check
2. File handling
3. Exception handling for TCP socket errors and file I/O errors
4. Incorrect host/port correction
5. Server can handle multiple sequential requests

Features not implemented

1. A request (filename) greater than 1024 bytes cannot be processed
2. Multiple requests cannot be handled simultaneously by the server.