

Socket Programming

Computer Networks CS4330

Christopher Hille, Anita Eluwa, Alex Montes, Amanda Villarreal

April 27, 2021

Table of Contents

Introduction.....	1
Project Specification.....	2
Design.....	4
User Manual.....	9
Summary and Conclusion	19
Future Work.....	20
Code Listing.....	21

Introduction

In this Socket Programming project, we implement five computer networking related programs meant to test and apply our understanding of the overall concepts learned throughout CS4330. The project assignment was to complete the development for a Web Server, UDP Pinger, Mail Client, Multi-Threaded Web Proxy, and an ICMP Ping given the skeleton code for each program to start us off. Our programs are written in the programming language Python as the assignment requires, and implemented using the IDE PyCharm and Spyder as they are convenient tools for developing Python applications.

In this project documentation, we discuss 1) the specifics of what each program will do in the Project Specification section, 2) the basic design of our programs including block diagrams and software modules in the Design section, 3) any algorithms used in the development of each program in the Algorithms section, 4) how the user can run and utilize each program in the User Manual section, 5) an overall description of what we learned and concluded for our project in the Summary and Conclusion section, 6) what we could do in the future to improve the project in the Future Work section, 7) a work cited page in the References section, and 8) a complete listing of our source code for each project in the Code Listing section.

Project Specifications

All five Socket Programming projects in conjunction make up our entire product. With the completion of all five programs, our product will be able to perform the individual functionalities of each of the five programs. However, the project will not run each program together, instead, each program is executed individually. In more detail, each project is listed below with its functionality and what it will allow our system to do:

I. Web Server

In this first program, we developed a simple Web server in Python that is capable of processing only one request. Specifically, Our Web server will (i) create a connection socket when contacted by a client (browser), (ii) receive the HTTP request from this connection, (iii) parse the request to determine the specific file being requested, (iv) get the requested file from the server's file system, (v) create a HTTP response message consisting of the requested file preceded by header lines, and (vi) send the response over the TCP connection to the requesting browser. If a browser requests a file that is not present in our server, our server will return a "404 Not Found" error message. When the user runs this server, they will be able to send requests from browsers running on different hosts.

II. UDP Pinger

In this second program, we wrote a client ping program that can (i) send a simple ping message to a server, (ii) receive a corresponding pong message from the server, and (iii) determine the Round Trip Time (RTT) delay between when the client sent the ping message and received the pong message. The functionality of the client and server in this program is similar to the functionality of a standard ping program in modern operating systems which use ICMP. However, in this program, we use a nonstandard UDP based pinger. This program is able to send 10 ping messages to a target server over UDP and the RTT for a sent ping message and received pong message will be computed by the client. Any packet sent by the client or server has a chance of getting lost considering that UDP is an unreliable protocol, thus, our client cannot wait indefinitely for a pong message. If there is no reply to a client ping received within one second, our client assumes the reply was lost. Additionally, optional exercise 1 was completed which furthered the program by adding the calculation of the round-trip time for each packet and printing it out individually. It also reports the minimum, maximum, and average RTTs at the end of all pings from the client and finally calculates and displays the packet loss rate (in percentage).

III. Mail Client

In this third program, we developed a simple mail client designed to send emails to any recipient. Specifically in this program, our client will (i) establish a TCP connection with a mail server using SSL, (ii) dialogue with the mail server using the SMTP protocol, (iii) send an email

message to a recipient through the mail server, and (iv) close the TCP connection with the mail server. Upon completion, the client is able to send emails to different user accounts.

IV. Multi-Threaded Web Proxy

In this fourth program, we developed a web proxy that can (i) receive an HTTP request for an object from a browser, (ii) generate a new HTTP request for the same object, and (iii) send it to the origin server. When the proxy receives the corresponding HTTP response with the object from the origin server, it will create a new HTTP response, including the object, and send it to the client. Since the server is multi-threaded, it will be able to handle multiple requests at the same time. Upon completion, this web proxy is able to handle different browser's request Web objects.

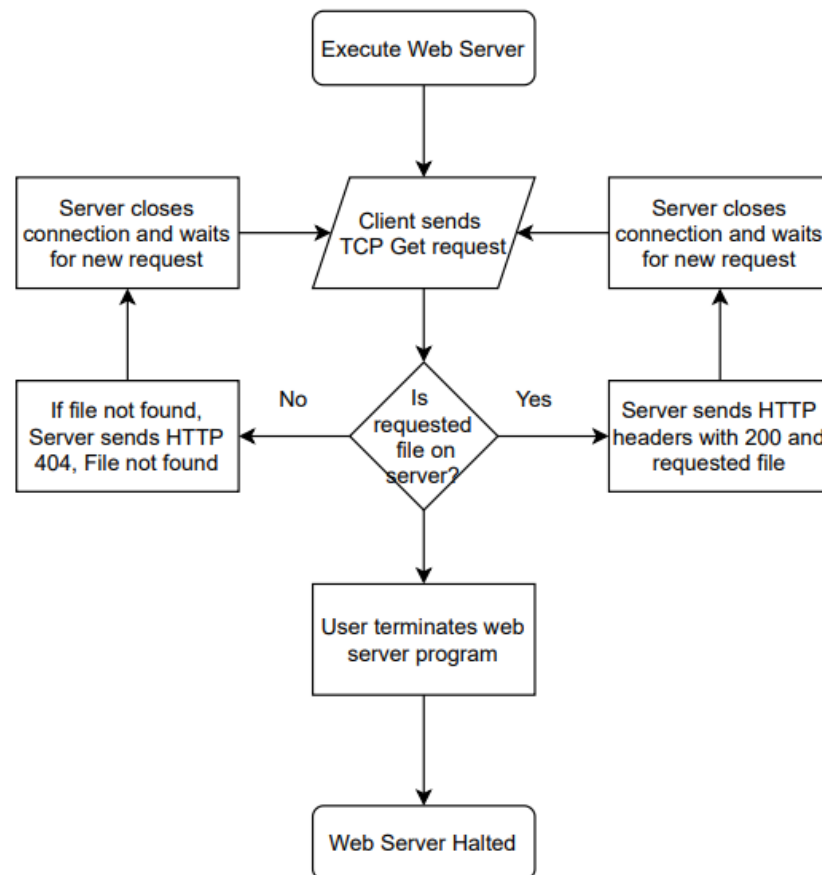
V. ICMP Ping

In this fifth program, we wrote our own ping application using the ICMP protocol's request and reply messages. This application builds the client side of the ping as the server side's functionality is built into most operating systems. This program is able to (i) send ping requests to a specified host separated by approximately one second, and (ii) wait up to one second to receive a reply after each packet is sent. If the one second limit goes by without a reply from the server, the client assumes that either the ping message or pong message was lost in the network.

Design

In this section, each of the five socket programs will be shown through flow diagrams that will help explain their individual designs and processes. Each program is based on its respective design, and follows the flow of the diagram in order to carry out its functionality. Since the section only reviews design, a brief overview of each program's design will be included under each diagram. Each program's actual flow and functionalities will be detailed in the User Manual section.

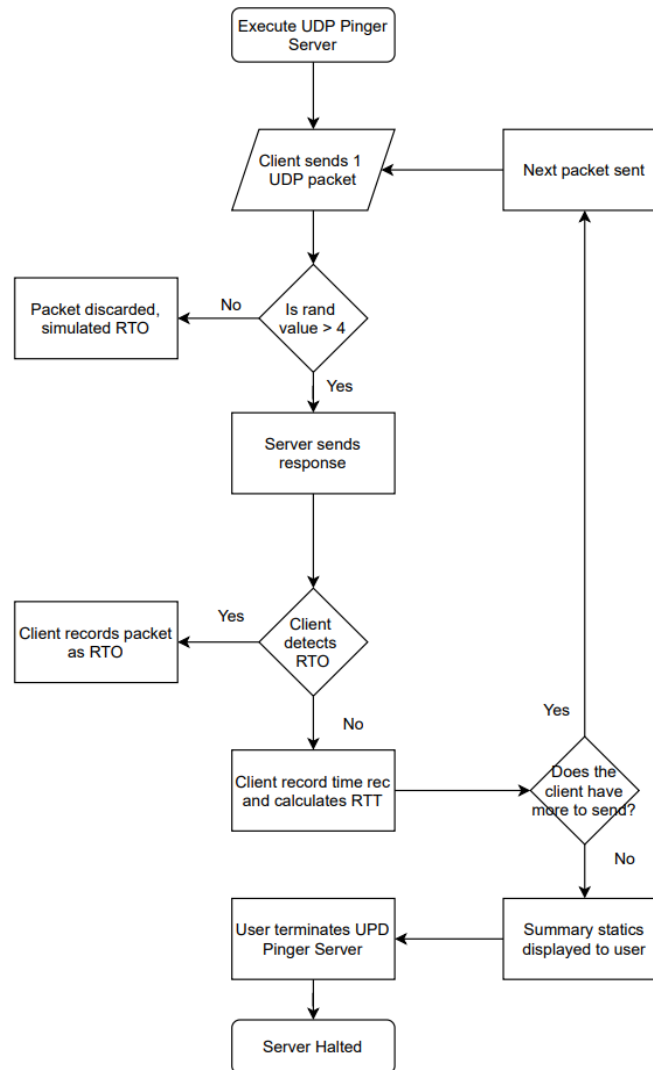
Web Server Flow Diagram



Shown above is the program flow for the web server program. This program has a central while loop that will be in a constant run state, listening for requests for web pages. Once the code is started, it is dependent on a request from a client. After receiving the request, the program will store the senders IP, Port and message in variables. It will next slice the message variable for the file name located after the last /. It will then search the file directory for that filename. If found, it will then build a HTTP 200 response and include the appropriate headers and file. Finally, the server will return the connection to the client and the file will be displayed.

If the server cannot find the request file, a HTTP 404 response will be sent along with the body “Not Found”. The server will continue to serve requests as long as the python script remains in a running state.

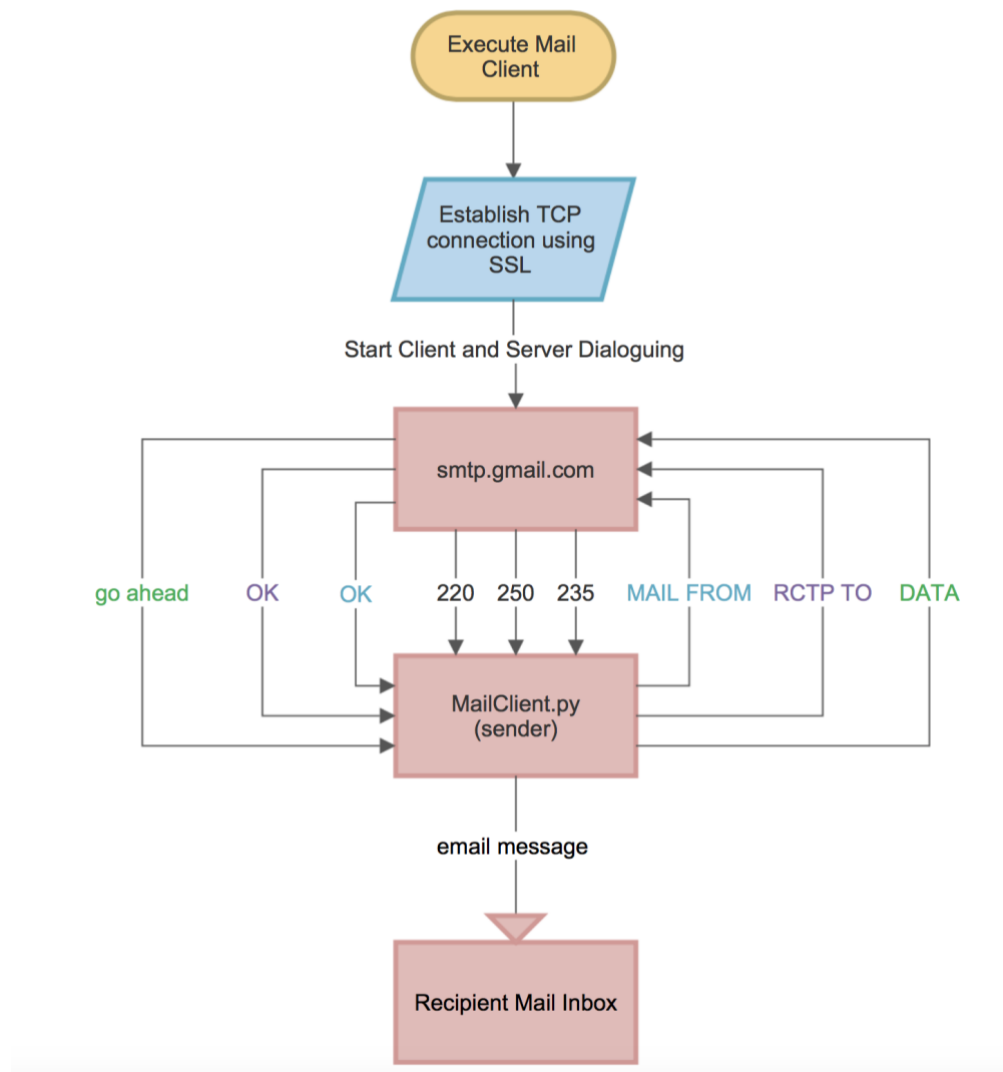
UDP Pinger Flow Diagram



Shown above is the program flow for the UDP Pinger Server and UDP Client. The server will remain in a constant while loop, listening for new UDP requests. The server also has a random variable for each loop that will be used to simulate a dropped packet. After the server has been started, a client will start its code. 10 UDP packets will be sent one at a time to the server. The time sent is recorded. The server will receive 1 UDP packet and begin to process it. It will record the senders information, IP and port, along with the message. The server will then check the random value, if less than 4 the packet will be discarded, if greater than 4 the server will then send a pong message back to the client it received the packet from. This will continue

for the remaining 9 packets the client will send and then the server will go back to a ready state. On the client side, once the packets are received back from the server the received time is subtracted from the sent time and the Round Trip Time (RTT) time will then be recorded. The client has a timeout set to 1 sec to help simulate the dropped packet. Once all 10 packets have been sent and the ones returned recorded, the client program will print out the results of the set of packets sent. The Max, Min, Avg RTT's, Number of packets dropped and percentage of dropped packets will be displayed. The client program will terminate.

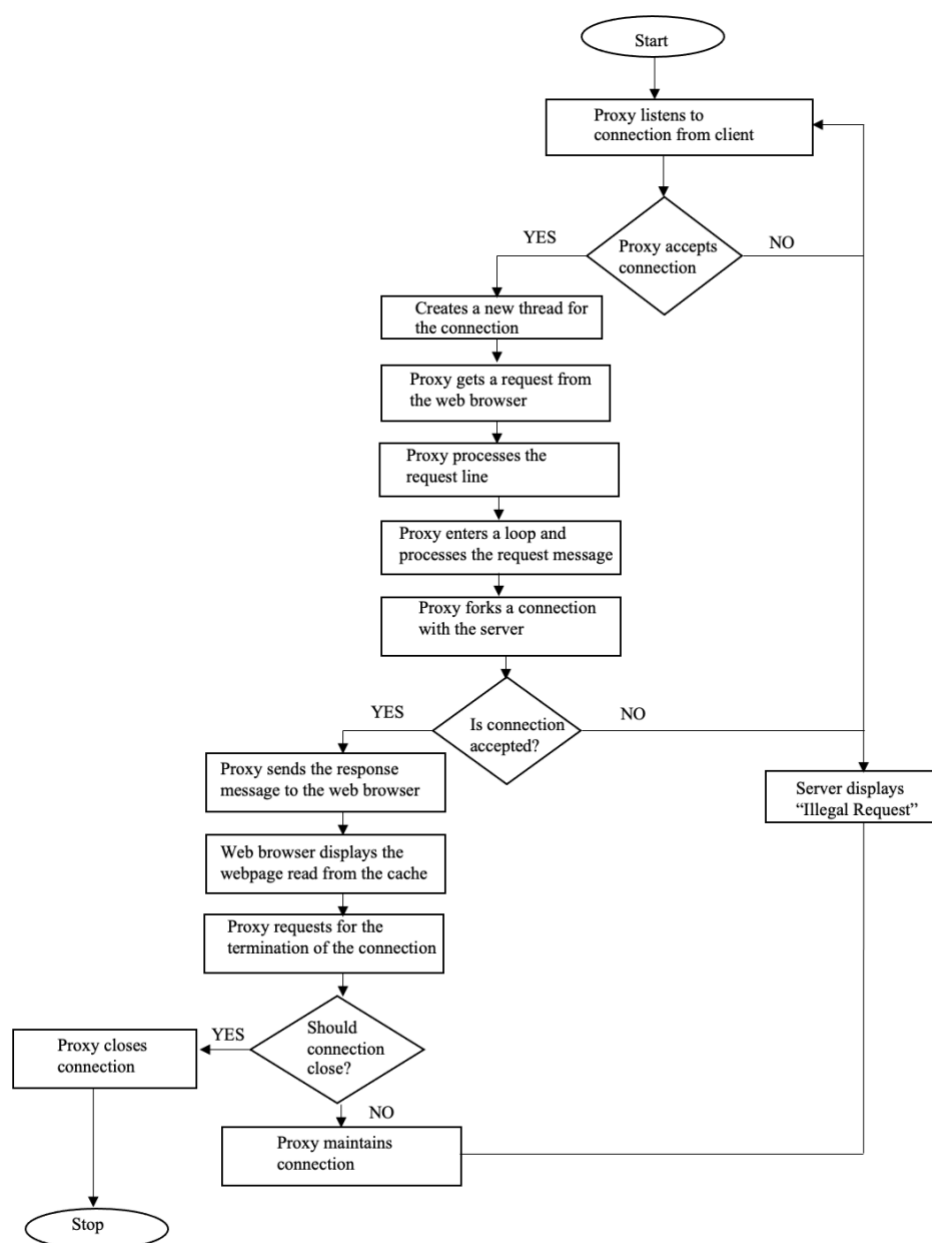
Mail Client Flow Diagram



Shown above is the flow diagram for the Mail Client. We can see this program takes a more linear design approach, with a small loop in the center. Looking at the flow diagram, once the Mail Client is executed, it will first establish a TCP connection with the desired mail server using SSL, and then immediately begin dialoguing. The mail server will send codes 220, 250, and 235 to MailClient.py to signal that it is recognized, accepted, and connected. Now the main

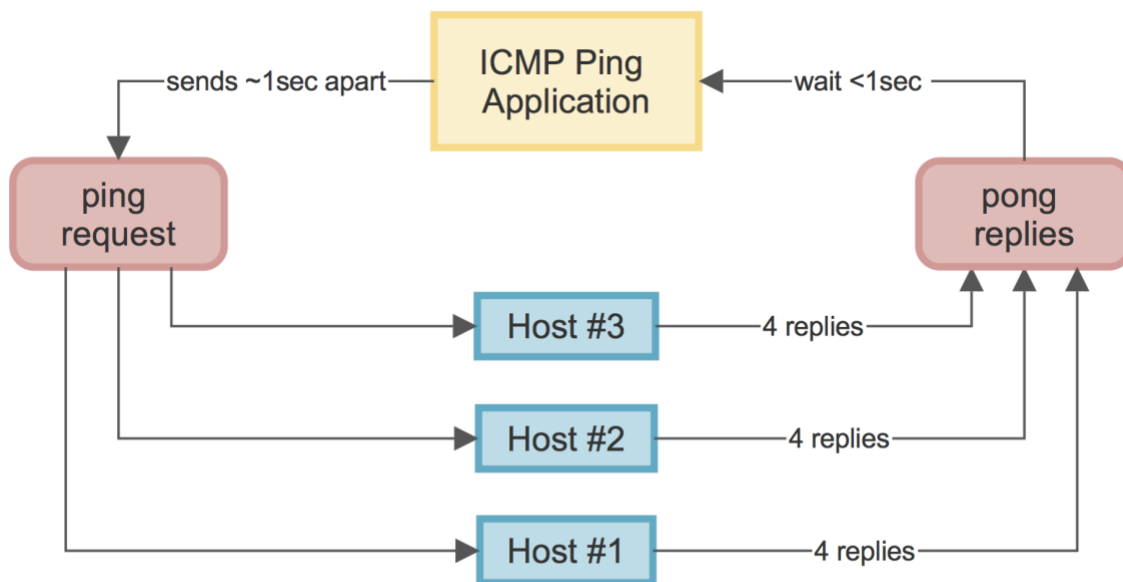
dialoguing process can begin. This dialoguing process is represented by the very brief loop between smtp.gmail.com and MailClient.py. While there is no actual looping in the program, the cyclical arrows in this loop strive represent that the MailClient.py has sent commands (MAIL FROM, RCTP TO, and DATA) to the mail server, and receives the respective response from the mail server (OK, OK, go ahead). After the dialoguing between the mail client and mail server is complete, the mail client sends the email message to the desired recipient, and the email goes into the recipient's inbox.

Multi-threaded Web Proxy Flow Diagram



The figure shown above is the flow diagram of the web proxy server. The proxy is open and awaits connection from the client. Now when a connection has been made, the proxy accepts the connection. Else, server generates an output “Illegal request”. But if the connection was accepted, the proxy then creates a new thread for the connection and gets a request from the web browser. The proxy then enters a while loop while processing the request. The proxy then forks a connection with the server. If the connection is accepted, the proxy sends the response message to the web browser and the web browser then displays the webpage read from the cache in the directory. But if the connection is not accepted, the server provides an output “illegal request”. The proxy then requests for the termination of the connection. If yes, the connection is closed. But if no, the proxy maintains the connection.

ICMP Ping Flow Diagram

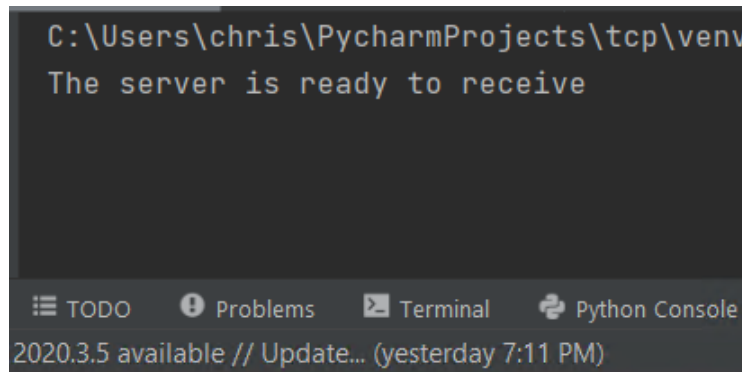


The diagram above shows the flow of the ICMP program. Here we can see the ICMP Ping application will execute, and immediately send ping requests to three different hosts, each ping request sent to each host is sent approximately one second apart. Each host can receive the ping requests if they are available, and will send four pong replies back to the ICMP Ping application. The application will wait up to one second to receive each ping reply. This is the basic flow of the ICMP Ping program, and will be further demonstrated in an example in the User Manual section below.

User Manual

Web Server

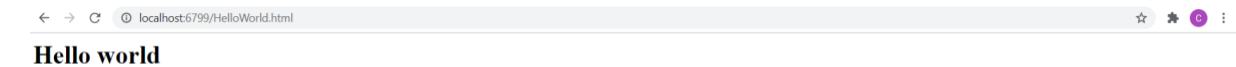
The web server consists of one python .py file named webserver. This file can be run from any device that has the latest Python Virtual Machine (VM) installed. Typically the VM is installed when a user installs an appropriate Integrated Development Environment (IDE). Once the IDE is installed, the user will then navigate to the scripts file location in the IDE. Once opened, the user will then run the file. This will start the web server, listening on the port identified in the program. Once a successful initialization of the program, the message the server is ready to receive, will be displayed in the output or console window.



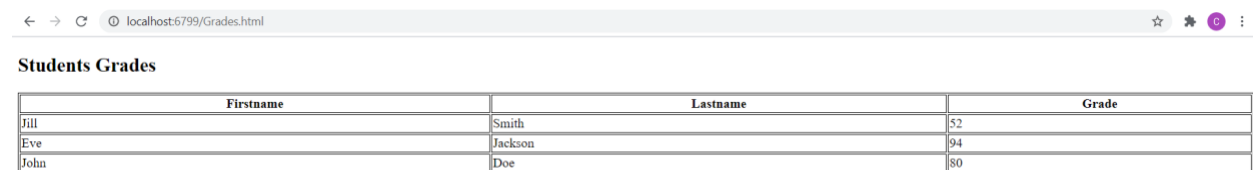
```
C:\Users\chris\PycharmProjects\tcp\venv
The server is ready to receive
```

TODO Problems Terminal Python Console
2020.3.5 available // Update... (yesterday 7:11 PM)

Next the user will then open a web browser and enter <http://localhost:6799/HelloWorld.html>, which will display the file HelloWorld.html as shown here:



Similarly if we visit <http://localhost:6799/Grades.html>, we are served the grades page:



← → ↻ localhost:6799/Grades.html ☆ ⚙ ⌵

Students Grades

Firstname	Lastname	Grade
Jill	Smith	52
Eve	Jackson	94
John	Doe	80

Finally to show the page that displays if we try to visit a page not hosted on our web server, the return will be a 404 - Not found as shown below:



Lastly, the following output is expected upon successful retrieval and connection to the client from the server output:

```
The server is ready to receive
GET /test.html HTTP/1.1
Host: localhost:6799
Connection: keep-alive
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="90", "Google Chrome";v="90"
sec-ch-ua-mobile: ?0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4430.85 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
```

UDP Pinger

The UDP Pinger relies on two separate python files, a client and a server. Similarly to the web server, the installation of a python IDE is recommended, otherwise the Python VM will be sufficient. To start the application, open the UDPServer.py in the IDE and then select run. In the console or output terminal window of the IDE, the following output will be expected:

A screenshot of a terminal window. The command prompt shows the path 'C:\Users\chris\PycharmProjects\tcp\venv\Scripts\python'. The output of the command is 'ready to receive'.

We are now ready to open and run the client python portion. This can be done on the same host as the server, however for a more realistic experience and results, it is recommended that another host on the network be used. Open a Python IDE, then the UDPPingerClient.py file. Executing the file, either by running the IDE will not produce a ready to receive output, rather will execute the programs function of sending 10 UDP packets to the server on the designated port. The expected output will be as follows:

```

In [1]: runfile('C:/Users/C/UDPPingerClient.py', wdir='C:/Users/C')
Message Sent To Server: Ping
-----
Message Received From Server: pong
Round Trip Time (RTT) for Packet [1] 0.99s
-----
Message Received From Server: pong
Round Trip Time (RTT) for Packet [2] 1.99s
-----
Message Received From Server: pong
Round Trip Time (RTT) for Packet [3] 1.01s
-----
Message Received From Server: pong
Round Trip Time (RTT) for Packet [4] 0.98s
-----
Message Received From Server: pong
Round Trip Time (RTT) for Packet [5] 1.0s
-----
Request Time Out (RTO) for Packet [6]
-----
Message Received From Server: pong
Round Trip Time (RTT) for Packet [7] 1.37s
-----
Request Time Out (RTO) for Packet [8]
-----
Request Time Out (RTO) for Packet [9]
-----
Message Received From Server: pong
Round Trip Time (RTT) for Packet [10] 33.93s
-----
Optional Excercise 1
Max RTT: 33.93s
Min RTT: 0.98s
Average RTT: 5.89s
Packets lost: 3
Packet Loss is: 30.0%
-----

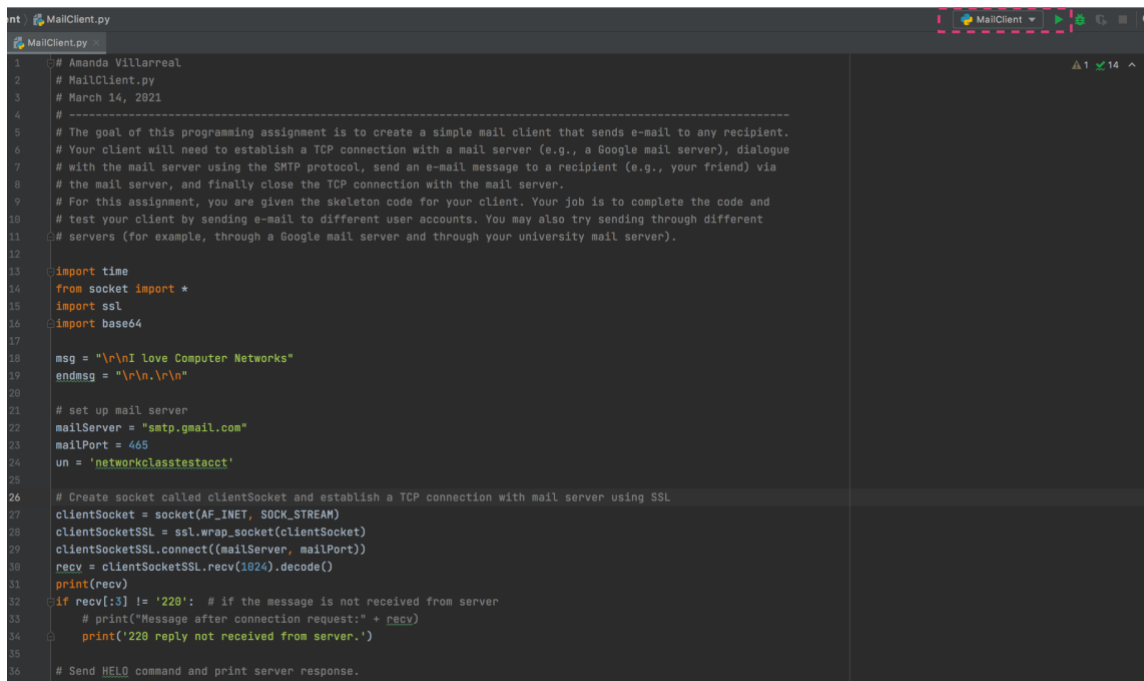
```

The program displays in the output section, the Round Trip Time (RTT) for each packet and whether or not it was received. In the case it isn't, a message of Request Time Out (RTO) is displayed. After all 10 packets are finished sending and received, the statistics of Max, Min and average RTT are displayed align with how many packets were lost and then the percentage lost.

Mail Client

Opening our file MailClient.py in any IDE that supports Python, the user can press the “run” button to execute the program. With the way our mail client is set up, no user input is required as the program has everything predefined and set up. Running the program, the user is able to use the mail client to establish connection with the mail server smtp.gmail.com using

mail port 465. From here, the program uses our mail account networkclasstestacc@gmail.com as the mail sender. The mail sender must be this account as was set up for our program and authenticated using its username and password. The mail receiver in our program is defined to be amandavillarreal99@yahoo.com for testing purposes, but can be changed to any recipient by changing the email defined in the variable rcptTo. When the user executes the program, they should expect to see the correct dialogue between the mail server and client, as well as an email 1) sent to the defined recipient (amandavillarreal99@yahoo.com), 2) from the correct sender (networkclasstestacc@gmail.com), 3) with the subject “SMTP mail client testing”, and 4) with the body of the email containing the date and time. Each expected output just mentioned will be reviewed below through screenshot demonstrations. Now looking at MailClient.py, the user presses the “run” button as shown below.



```

1 # Amanda Villarreal
2 # MailClient.py
3 # March 14, 2021
4 # -----
5 # The goal of this programming assignment is to create a simple mail client that sends e-mail to any recipient.
6 # Your client will need to establish a TCP connection with a mail server (e.g., a Google mail server), dialogue
7 # with the mail server using the SMTP protocol, send an e-mail message to a recipient (e.g., your friend) via
8 # the mail server, and finally close the TCP connection with the mail server.
9 # For this assignment, you are given the skeleton code for your client. Your job is to complete the code and
10 # test your client by sending e-mail to different user accounts. You may also try sending through different
11 # servers (for example, through a Google mail server and through your university mail server).
12
13 import time
14 from socket import *
15 import ssl
16 import base64
17
18 msg = "\r\nI Love Computer Networks"
19 endmsg = "\r\n.\r\n"
20
21 # set up mail server
22 mailServer = "smtp.gmail.com"
23 mailPort = 465
24 un = 'networkclasstestacc'
25
26 # Create socket called clientSocket and establish a TCP connection with mail server using SSL
27 clientSocket = socket(AF_INET, SOCK_STREAM)
28 clientSocketSSL = ssl.wrap_socket(clientSocket)
29 clientSocketSSL.connect((mailServer, mailPort))
30 rcv = clientSocketSSL.recv(1024).decode()
31 print(rcv)
32 if rcv[:3] != '220': # if the message is not received from server
33     # print("Message after connection request:" + rcv)
34     print('220 reply not received from server.')
35
36 # Send HELO command and print server response.

```

Running the program gives us the following output shown below. Here, the user can see that the mail server smtp.gmail.com is recognized (code 220), the mail server has replied (code 250), and the request to use the mail server has been accepted (235). Next, the output displays the data received from the mail server after each MAIL FROM, RCPT TO, and DATA command, and it is clear the user is successfully dialoguing with the mail server. Lastly, the dialoguing is complete, the connection with the mail server is closed, and the email has been sent from the sender email address to the recipient address as mentioned above.

```

"/Users/amandavillarreal/Desktop/St. Mary's/Spring 2021/Computer Networks/ComputerNetworks
220 smtp.gmail.com ESMTP 93sm919778otr.31 - gsmt
p

250 smtp.gmail.com at your service

235 2.7.0 Accepted

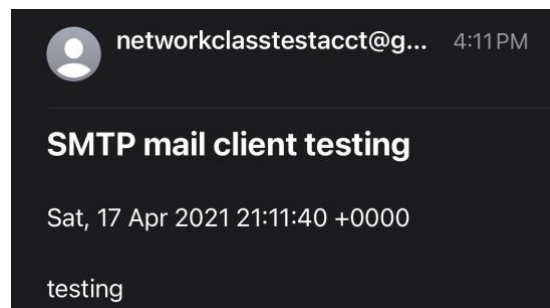
('After MAIL FROM command: ', u'250 2.1.0 OK 93sm919778otr.31 - gsmt
p\r\n')
('After RCPT TO command: ', u'250 2.1.5 OK 93sm919778otr.31 - gsmt
p\r\n')
('After DATA command: ', u'354 Go ahead 93sm919778otr.31 - gsmt
p\r\n')
Response after sending message body: 250 2.0.0 OK 1618693900 93sm919778otr.31 - gsmt
p

221 2.0.0 closing connection 93sm919778otr.31 - gsmt
p

Process finished with exit code 0

```

Last, after running MailClient.py, the user should check the recipient email to see if the email has gone through successfully. In this case, the inbox for amandavillarreal99@yahoo.com is checked for the email. As seen by the image below, the email has been sent successfully to the correct recipient and from the correct sender. The user can also see that the email correctly contains (1) the defined recipient amandavillarreal99@yahoo.com, (2) the defined sender networkclasstestacct@gmail.com, (3) the subject “SMTP mail client testing”, and (4) the date and time the email was sent as the body.



Multi-Threaded Web Proxy

In a multi-threaded web proxy, The server will be able to handle multiple simultaneous service requests in parallel. This means that the Web server is multi-threaded. In the main thread, the server listens to a fixed port. When it receives a TCP connection request, it sets up a TCP connection through another port and services that request in a separate thread. The version of Python used to run this program is Python 2.7. Please see below the source code used in executing this program.

```

Proxy server proxyserver.py
1 from socket import *
2 import sys
3
4 port = 8888
5 max_connections = 5
6
7 if len(sys.argv) < 1:
8     print('Usage : "python ProxyServer.py server_ip"\n[server_ip : It is the IP Address Of Proxy Server]')
9     sys.exit(2)
10
11 # Create a server socket, bind it to a port and start listening
12 tcpSerSock = socket(AF_INET, SOCK_STREAM)
13 tcpSerSock.bind(('', port))
14 tcpSerSock.listen(max_connections)
15
16 while 1:
17     # Start receiving data from the client
18     print('Ready to serve...')
19
20     tcpCliSock, addr = tcpSerSock.accept()
21     print('Received a connection from:', addr)
22
23     message = tcpCliSock.recv(1024)
24     # print(message)
25
26     # Extract the filename from the given message
27     # print(message.split()[2])
28     filename = message.split()[1].partition("/")[2]
29     # filename = message.split()
30     print("Line 29: ", filename)
31
32     fileExist = "false"
33     filetoUse = "/" + filename # .replace("/", "")
34
35     try:
36         # Check whether the file exist in the cache
37

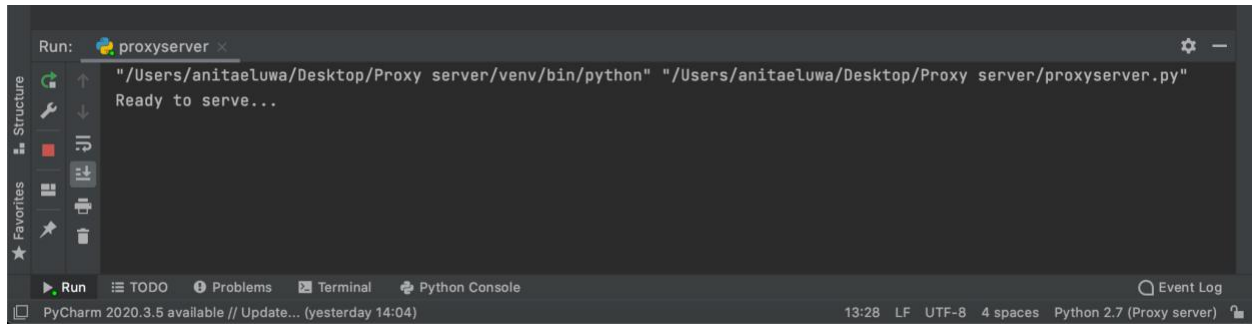
```

```

Proxy server proxyserver.py
76 # Create a temporary file on this socket and ask port 80 for the file requested by the client
77 fileobj = c.makefile('r', 0)
78 fileobj.write("GET " + "http://" + filename + " HTTP/1.0\n\n")
79
80 # Show what request was made
81 print("GET " + "http://" + filename + " HTTP/1.0")
82
83 # Read the response into buffer
84 buff = fileobj.readlines()
85
86 # Create a new file in the cache for the requested file.
87 # Also send the response in the buffer to client socket and the corresponding file in the cache
88 # if (filename[-1:] == '/'):
89     filename = filename[:-1]
90     print('****')
91     print(filename)
92     # tmpFile = open("./" + filename.replace("/", ""), "wb")
93     tmpFile = open("./" + filename, "wb")
94     for line in buff:
95         tmpFile.write(line)
96         tcpCliSock.send(line)
97
98     # tmpFile.write(response)
99     # tmpFile.close()
100     # print(tmpFile)
101     # tcpCliSock.send(tmpFile.encode())
102     tcpCliSock.close()
103 except Exception as e:
104     print(str(e))
105     print("Illegal request")
106 else:
107     # HTTP response message for file not found
108     tcpCliSock.send("HTTP/1.0 404 Not Found\r\n".encode())
109     tcpCliSock.send("Content-Type:text/html\r\n".encode())
110     pass
111 # Close the client and the server sockets
112 tcpCliSock.close()

```

Running this program, it gives us an output “ready to serve” which means that the TCP connection is ready to receive a request from the web server.



```
Run: proxyserver x
"/Users/anitaeluwa/Desktop/Proxy server/venv/bin/python" "/Users/anitaeluwa/Desktop/Proxy server/proxyserver.py"
Ready to serve...
```

PyCharm 2020.3.5 available // Update... (yesterday 14:04) 13:28 LF UTF-8 4 spaces Python 2.7 (Proxy server)

Now we will pass this through the local host of google.com. The port number used in the program is 8888.

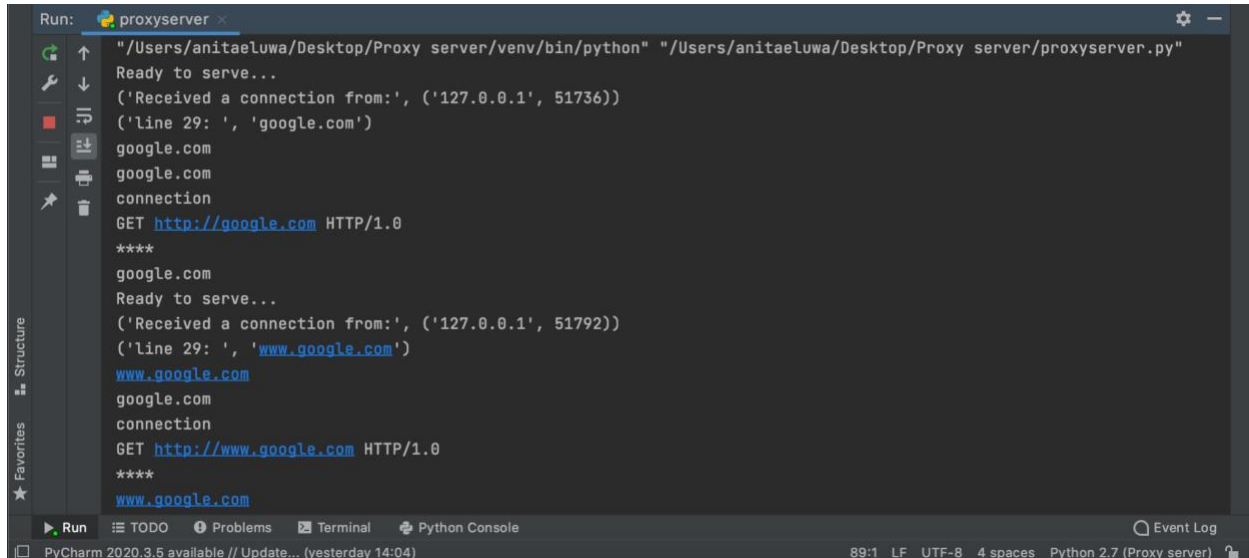


After passing this through the local host of google, using: localhost:8888/google.com, the client receives the request and generates the web page of google.

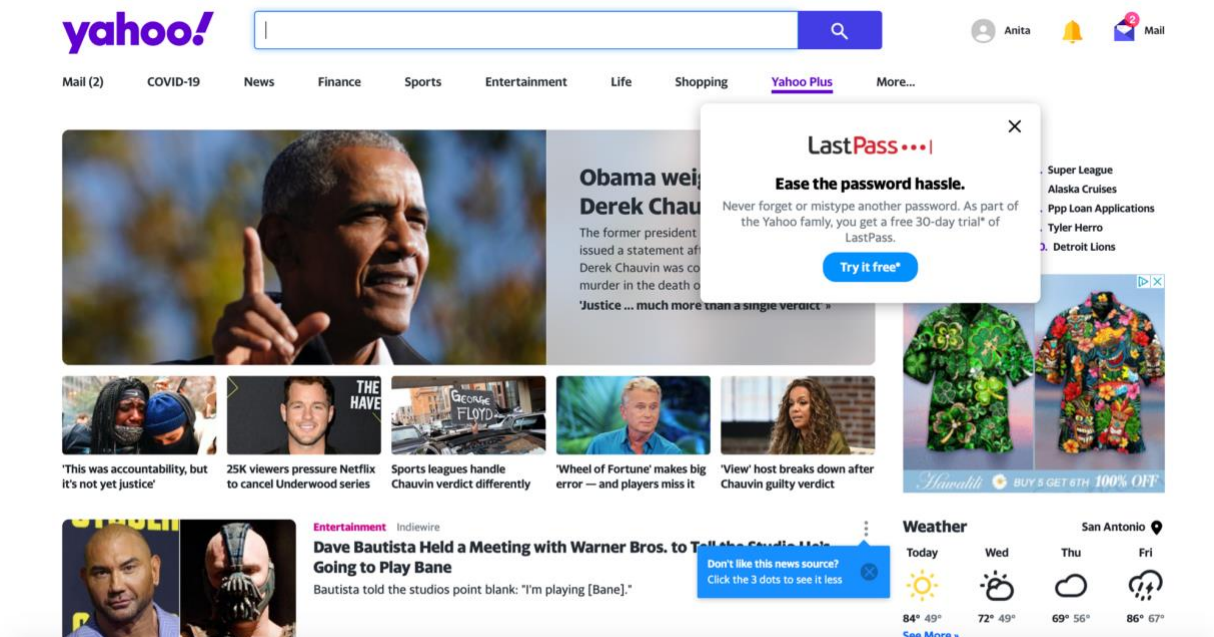
301 Moved

The document has moved [here](#).

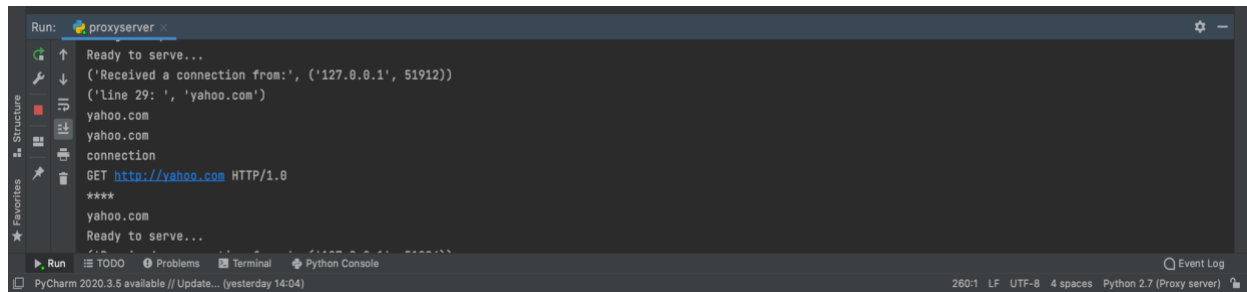
After clearing the cache of my web browser, while the server is still running, i sent the request the second time and the output “301 moved, the document has moved here” is received because i cleared my cache. When the link is clicked, it then reads it now from the cache and produces the google web page.



```
Run: proxyserver x
"/Users/anitaeluwa/Desktop/Proxy server/venv/bin/python" "/Users/anitaeluwa/Desktop/Proxy server/proxyserver.py"
Ready to serve...
('Received a connection from:', ('127.0.0.1', 51736))
('line 29: ', 'google.com')
google.com
google.com
connection
GET http://google.com HTTP/1.0
****
google.com
Ready to serve...
('Received a connection from:', ('127.0.0.1', 51792))
('line 29: ', 'www.google.com')
www.google.com
google.com
connection
GET http://www.google.com HTTP/1.0
****
www.google.com
```



Here, we passed the request to yahoo.com using the URL localhost:8888/yahoo.com.

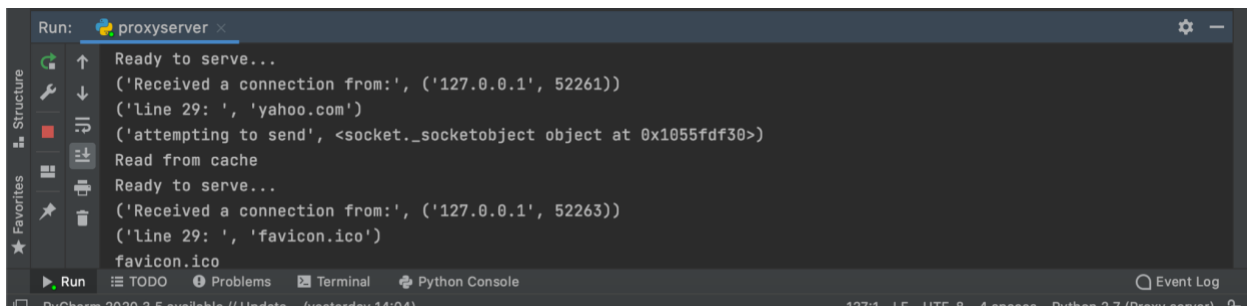


```

Run: proxyserver
Ready to serve...
('Received a connection from:', ('127.0.0.1', 51912))
('line 29: ', 'yahoo.com')
yahoo.com
yahoo.com
connection
GET http://yahoo.com HTTP/1.0
****
yahoo.com
Ready to serve...

```

The above image shows that the server has gotten a response received from yahoo.com. And the below shows the output received from the second request to yahoo.com and this time reading from the cache.



```

Run: proxyserver
Ready to serve...
('Received a connection from:', ('127.0.0.1', 52261))
('line 29: ', 'yahoo.com')
('attempting to send', <socket._socketobject object at 0x1055fdf30>)
Read from cache
Ready to serve...
('Received a connection from:', ('127.0.0.1', 52263))
('line 29: ', 'favicon.ico')
favicon.ico

```

ICMP Ping

The ICMP Ping application is a ping application that sends requests to a specified host, separated by approximately one second. Each message contains a payload of data that includes a timestamp. After sending each packet, the application waits up to one second to receive a reply. If one second goes by without a reply from the server, then the client may assume that either the ping packet or the pong packet was lost in the network (or that the server is down). This application does not exactly follow the official specification in RFC 1739 in order to keep it simple. This application focuses on the client side, as the functionality needed on the server side is built into almost all operating systems. To run the ICMP Ping project, open the ICMP.py file onto any IDE that supports python (adjustments may have to be made depending on the IDE), and click the run button. No input is required to see the predefined servers as the program is ready to use upon opening the file. If the user wishes to ping to different hosts, they may head to the very bottom of the code and change the print statement to input the name of the host and change the IP addresses to match the host server they would like to test.

```

print("Ping to Washington DC:")
ping("23.81.0.59")

print("\nPing to Buenos Aires, Argentina:")
ping("131.255.7.26")

print("\nPing to Frankfurt, Germany:")
ping("195.201.213.247")

```

The figure above tells the program to ping to the specified host. Upon running the program, the user should expect to see four replies from one server (if a reply is received). Each reply will be printed one second after the other and after the final reply, the application will attempt to reach the next host (if there is one). The figure below shows the output pinging to Washington DC, Buenos Aires, and Frankfurt.

```

Ping to Washington DC:
Pinging 23.81.0.59 using Python:

0.07727217674255371
0.08306550979614258
0.08045554161071777
0.0789332389831543

Ping to Buenos Aires, Argentina:
Pinging 131.255.7.26 using Python:

0.2905232906341553
0.22238373756408691
0.22992753982543945
0.22217988967895508

Ping to Frankfurt, Germany:
Pinging 195.201.213.247 using Python:

0.2263627052307129
0.2277822494506836
0.23229527473449707
0.14934086799621582

Process finished with exit code 0

```

Summary and Conclusion

In summary, our project group applied the networking principles and learning objectives from the class to each of the five programming problems. In addition to applying our knowledge of computer networking and socket programming, our group also practiced our industry skills of debugging, testing, and diagram making. Each group member also got a great opportunity to practice their knowledge, or even just start learning, the Python programming language, and add this language as experience to the languages they know in the field of Computer Science. Overall, these foundational principles will help us grow not only as a student, but as a future leader in tomorrow's Information Technology workforce.

Future Work

One small way our project can be improved in the future is by bringing all five programs together into one program. All five programs were developed for this project individually, meaning each program has its own executable Python file. It would be a convenient improvement in the future to compile all five programs into one executable. This one program would give the user the option to choose which one of five programs they would like to run, run the selected program from within this compilation program, and keep asking which program to run until the user chooses to terminate.

Each of the five programs could also be modified individually to improve on their functionality. For example, the Mail Client could be improved by allowing for user input. This would entail allowing the user to input the sender, the recipient, as well as the email subject and body to make for a much more realistic mail client experience.

For the web server project, the following functionality should be evaluated and implemented: enable multithreading to handle multiple requests simultaneously. By doing so, this program could serve as a cheap and effective alternative to a Windows IIS or REHL Weblogic server which requires licenses and provides additional security risks. It could be used in smaller/limited use cases which wouldn't require the server to maintain or manage thousands of connections per second.

For the UDP Pinger project, the following should be evaluated and implemented: UDP heartbeat. The Heartbeat can be used to check if an application is up and running and to report one-way packet loss. The client sends a sequence number and current timestamp in the UDP packet to the server, which is listening for the Heartbeat (i.e., the UDP packets) of the client. Upon receiving the packets, the server calculates the time difference and reports any lost packets. This could provide useful information to a local area network as to the status of host on the network in a single application. Most Graphical User Interfaces (GUI) that can provide this functionality can be costly and be difficult to learn. This option provides a free, quickly implementable solution.

For the Multi-threaded Web Proxy Server, the following should be re-evaluated and taken into serious consideration: The security of the proxies. Proxies can break Transport Layer Security (TLS) encryption that protects end-users browsing the internet. The security features of the HTTPS should be improved to prevent interception. Hence, the security community needs to improve network security and identity validation to eliminate the need for problematic proxies.

Lastly, the ICMP Ping can be improved on by implementing the optional exercises. The program can be modified to report the maximum, minimum, and average RTT's at the end of all pings and calculate the packet loss rate, and parse the ICMP error codes and display the corresponding error results.

Code Listing

Web Server

```

from socket import *
import datetime
import sys

# Create a TCP server socket
# (AF_INET is used for IPv4 protocols)
# (SOCK_STREAM is used for TCP)

serverSocket = socket(AF_INET, SOCK_STREAM)

# Assign a port number
serverPort = 6799
# Bind the socket to server address and server port
serverSocket.bind("", serverPort)
# Listen to at most 1 connection at a time
serverSocket.listen(1)

# Server should be up and running and listening to the incoming connections

while True:
    print('The server is ready to receive')

    # Set up a new connection from the client
    connectionSocket, addr = serverSocket.accept()

    try:
        message = connectionSocket.recv(1024).decode()
        if len(message) > 0:
            # Extract the path of the requested object from the message
            # The path is the second part of HTTP header, identified by [1]
            filename = message.split()[1]
            # Because the extracted path of the HTTP request includes
            # a character '\', we read the path from the second character
            f = open(filename[1:])
            # Store the entire content of requested file in a temporary
            buffer

            outputdata = f.read()
            #build the HTTP header
            now = datetime.datetime.now()
            statusLine = "HTTP/1.1 200 OK\r\n"
            headerInfo = {"Date":now.strftime("%Y-%m-%d%H:%M:%S"),
                           "Content-Type":"text/html",
                           "Charset=": "uuiltf-8",
                           "Content-Length": len(outputdata),
                           "Keep-Alive": "timeout=%d,Max%d"%(10,100),

```

```

        "Connection": "Keep-Alive:{}".format(
            headerLines = "\r\n".join("%s:%s"%(item, headerInfo[item]) for
item in headerInfo)
            HTTPResponse = statusLine + headerLines + "\r\n\r\n"

            # Send the HTTP response header line to the connection socket
            connectionSocket.send(HTTPResponse.encode())

            # Send the content of the requested file to the connection socket
            for i in range(0, len(outputdata)):
                connectionSocket.send(outputdata[i].encode())
            connectionSocket.send("\r\n".encode())

            # Close the client connection socket
            connectionSocket.close()

    except IOError:
        # Send HTTP response message for file not found

        connectionSocket.send("HTTP/1.1 404 Not Found\r\n\r\n".encode())
        connectionSocket.send("<html><head></head><body><h1>404 Not
Found</h1></body></html>\r\n".encode())
        # Close the client connection socket
        connectionSocket.close()

serverSocket.close()
sys.exit() # Terminate the program after sending the corresponding data

```

UDP Pinger

(Client)

```

import time
from socket import *

serverName = '192.168.103.148'
#serverName = '127.0.0.1'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
clientSocket.settimeout(1)
message = 'Ping'
rtt=[]
print ("Message Sent To Server:", message)
print('-----')
for i in range(10):

    sendTime = time.time()

```



```

message = 'PING ' + str(i + 1) + " " + str(time.strftime("%H:%M:%S"))
clientSocket.sendto(message.encode(), (serverName, serverPort))
try:
    data, server = clientSocket.recvfrom(1024)
    recdTime = time.time()
    rtt.append(recdTime - sendTime)

    #print(recdTime - sendTime, i)
    print ("Message Received From Server:", data.decode())
    print ("Round Trip Time (RTT) for Packet", [i+1],
str(round(rtt[i]*1000,2))+ 's')
    print('-----')

except timeout:
    print ('Request Time Out (RTO) for Packet', [i+1])
    rtt.append(None)
    print('-----')
#Optional Exercise 1
clean = [x for x in rtt if x != None]
if len(clean) > 0:
    print("Optional Exercise 1")
    print("Max RTT:", str(round(max(clean)*1000, 2))+ 's')
    print("Min RTT:", str(round(min(clean)*1000, 2))+ 's')
    print("Average RTT:", str(round(sum(clean)/len(clean)*1000, 2))+ 's')
print("Packets lost:", 10-len(clean))
print("Packet Loss is:", str(round(((10-len(clean))/10), 2)*100)+'%')
print('-----')

```

(Server)

```

# UDPPingerServer.py
# We will need the following module to generate randomized lost packets
import random
import time
from socket import *

# Create a UDP socket
# Notice the use of SOCK_DGRAM for UDP packets
serverSocket = socket(AF_INET, SOCK_DGRAM)
# Assign IP address and port number to socket
serverSocket.bind(('', 12000))
while True:
    print("ready to receive")
    # Generate random number in the range of 0 to 10
    rand = random.randint(0, 10)
    # Receive the client packet along with the address it is coming from
    message, address = serverSocket.recvfrom(1024)
    # Capitalize the message from the client
    message = 'pong'

```

```

    # If rand is less is than 4, we consider the packet lost and do not
    respond
    if rand < 4:
        continue
    # Otherwise, the server responds
    serverSocket.sendto(message.encode(), address)
    print(address, message)

```

Mail Client

```

import time
from socket import *
import ssl
import base64

msg = "\r\nI love Computer Networks"
endmsg = "\r\n.\r\n"

# set up mail server
mailServer = "smtp.gmail.com"
mailPort = 465
un = 'networkclasstestacct'

# Create socket called clientSocket and establish a TCP connection with mail
server using SSL
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocketSSL = ssl.wrap_socket(clientSocket)
clientSocketSSL.connect((mailServer, mailPort))
recv = clientSocketSSL.recv(1024).decode()
print(recv)
if recv[:3] != '220': # if the message is not received from server
    # print("Message after connection request:" + recv)
    print('220 reply not received from server.')

# Send HELO command and print server response.
helloCommand = 'HELO Alice\r\n'
clientSocketSSL.send(helloCommand)
recv1 = clientSocketSSL.recv(1024).decode()
print(recv1)
if recv1[:3] != '250': # if message not received
    print('250 reply not received from server.')

username = "networkclasstestacct@gmail.com"
password = '1qazxsw2!QAZXSW@'
base64_str = ("\"x00"+username+"\"x00"+password).encode()
base64_str = base64.b64encode(base64_str)
authMsg = "AUTH PLAIN ".encode()+base64_str+"\r\n".encode()

```

```

clientSocketSSL.send(authMsg)
recv_auth = clientSocketSSL.recv(1024).decode()
print(recv_auth)

# Send MAIL FROM command and print server response.
mailFrom = "MAIL FROM: <networkclasstestacct@gmail.com> \r\n" # must be from
this account
clientSocketSSL.send(mailFrom.encode())
recv1 = clientSocketSSL.recv(1024).decode()
print("After MAIL FROM command: ", recv1)
if recv1[:3] != '250': # if data is not received
    print('250 reply not received from server.')

# Send RCPT TO command and print server response.
rcptTo = "RCPT TO: <amandavillarreal99@yahoo.com> \r\n" # change as needed
clientSocketSSL.send(rcptTo.encode())
recv1 = clientSocketSSL.recv(1024).decode()
print("After RCPT TO command: ", recv1)
if recv1[:3] != '250': # data not received
    print('250 reply not received from server.')

# Send DATA command and print server response.
data = "DATA\r\n"
clientSocketSSL.send(data.encode())
recv1 = clientSocketSSL.recv(1024).decode()
print("After DATA command: ", recv1)
if recv1[:3] != '354': # if message not received (maybe 354)
    print('354 reply not received from server.')

# Send message data.
subject = "Subject: SMTP mail client testing \r\n\r\n"
clientSocketSSL.send(subject.encode())
message = "testing\r\n\r\n"
date = time.strftime("%a, %d %b %Y %H:%M:%S +0000", time.gmtime())
date = date + "\r\n\r\n"
clientSocketSSL.send(date.encode())
clientSocketSSL.send(message.encode())
clientSocketSSL.send(endmsg.encode())
recv_msg = clientSocketSSL.recv(1024).decode()
print("Response after sending message body: " + recv_msg)
if recv_msg[:3] != '250':
    print('250 reply not received from server.')

# close connection with mail server
# Send QUIT command and get server response.
quit = "QUIT\r\n"
clientSocketSSL.send(quit.encode())
message = clientSocketSSL.recv(1024).decode()
print (message)

```

```
clientSocketSSL.close()
```

Multi-Threaded Web Proxy

```
from socket import *
import sys

port = 8888
max_connections = 5

if len(sys.argv) < 1:
    print('Usage : "python ProxyServer.py server_ip"\n[server_ip : It is the
IP Address Of Proxy Server]')
    sys.exit(2)

# Create a server socket, bind it to a port and start listening
tcpSerSock = socket(AF_INET, SOCK_STREAM)
tcpSerSock.bind(('', port))
tcpSerSock.listen(max_connections)

while 1:

    # Start receiving data from the client
    print('Ready to serve...')

    tcpCliSock, addr = tcpSerSock.accept()
    print('Received a connection from:', addr)

    message = tcpCliSock.recv(1024)
    # print(message)
    # Extract the filename from the given message
    # print(message.split()[2])
    filename = message.split()[1].partition("/")[2]
    # filename = message.split()
    print("line 29: ", filename)

    fileExist = "false"
    filetouse = "/" + filename # .replace("/", "")

    try:
        # Check whether the file exist in the cache
        f = open(filetouse[1:], "r")
        outputdata = f.readlines()
        fileExist = "true"
        # ProxyServer finds a cache hit and generates a response message
        # resp = ""
```

```

# print(filetouse[1:], resp)
# print("line 47: ", resp)
# request = tcpCliSock.recv()

# tcpCliSock.connect((resp, 80))
# tcpCliSock.send(resp.encode())
tcpCliSock.send("HTTP/1.0 200 OK\r\n")
tcpCliSock.send("Content-Type:text/html\r\n")

print("attempting to send", tcpCliSock)
for s in outputdata:
    tcpCliSock.send(s)
# for s in range(0, len(outputdata)):
#     tcpCliSock.send(outputdata[s])

print('Read from cache')

# Error handling for file not found in cache
except IOError:
    if fileExist == "false":
        # Create a socket on the proxyserver
        c = socket(AF_INET, SOCK_STREAM)
        hostn = filename.replace("www.", "", 1) # hostn =
filename.split('/')[0].replace("www.", "", 1)
        print(filename)
        print(hostn)

        try:
            # Connect to the socket to port 80
            c.connect((hostn, 80))
            print("connection")
            # Create a temporary file on this socket and ask port 80 for
the file requested by the client
            fileobj = c.makefile('r', 0)
            fileobj.write("GET " + "http://" + filename + " HTTP/1.0\n\n")

            # Show what request was made
            print("GET " + "http://" + filename + " HTTP/1.0")
            # Read the response into buffer
            buff = fileobj.readlines()

            # Create a new file in the cache for the requested file.
            # Also send the response in the buffer to client socket and
the corresponding file in the cache
            # if (filename[-1:] == '/'):
            #     filename = filename[:-1]
            print('*****')
            print(filename)
            # tmpFile = open("./" + filename.replace("/", ""), "wb")

```

```

        tmpFile = open("./" + filename, "wb")
        for line in buff:
            tmpFile.write(line)
            tcpCliSock.send(line)

        # tmpFile.write(response)
        # tmpFile.close()
        # print(tmpFile)
        # tcpCliSock.send(tmpFile.encode())
        tcpCliSock.close()
    except Exception as e:
        print(str(e))
        print("Illegal request")
else:
    # HTTP response message for file not found

    tcpCliSock.send("HTTP/1.0 404 Not Found\r\n".encode())
    tcpCliSock.send("Content-Type:text/html\r\n".encode())
    pass
# Close the client and the server sockets
tcpCliSock.close()

```

ICMP Ping

```

import string
from socket import *
import os
import sys
import struct
import time
import select
import binascii

ICMP_ECHO_REQUEST = 8

def checksum(str_):
    # In this function we make the checksum of our packet
    str_ = bytearray(str_)
    csum = 0
    countTo = (len(str_) // 2) * 2
    count = 0

    while count < countTo:
        thisVal = (str_[count + 1]) * 256 + (str_[count])
        csum += thisVal
        csum &= 0xffffffff

```

```

        count += 2
    if countTo < len(str_):
        csum = csum + str_[-1]
        csum = csum & 0xffffffff

    csum = (csum >> 16) + (csum & 0xffff)
    csum = csum + (csum >> 16)
    answer = ~csum
    answer = answer & 0xffff
    answer = answer >> 8 | (answer << 8 & 0xff00)
    return answer

def receiveOnePing(mySocket, ID, timeout, destAddr):
    timeLeft = timeout
    while 1:
        startedSelect = time.time()
        whatReady = select.select([mySocket], [], [], timeLeft)
        howLongInSelect = (time.time() - startedSelect)
        if whatReady[0] == []: # Timeout
            return "Request timed out."

        timeReceived = time.time()
        recPacket, addr = mySocket.recvfrom(1024)

        # Fill in start
        icmp_header = recPacket[20:28]
        icmpType, code, myChecksum, p_id, sequence = struct.unpack('bbHHh',
icmp_header)
        if p_id == ID:
            return timeReceived - startedSelect
        # Fetch the ICMP header from the IP packet

        # Fill in end
        timeLeft = timeLeft - howLongInSelect
        if timeLeft <= 0:
            return "Request timed out."

def sendOnePing(mySocket, destAddr, ID):
    # Header is type (8), code (8), checksum (16), id (16), sequence (16)

    myChecksum = 0
    # Make a dummy header with a 0 checksum
    # struct -- Interpret strings as packed binary data
    header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, myChecksum, ID, 1)
    data = struct.pack("d", time.time())
    # Calculate the checksum on the data and the dummy header.
    myChecksum = checksum(header + data)

    # Get the right checksum, and put in the header

```

```

if sys.platform == 'darwin':
    # Convert 16-bit integers from host to network byte order
    myChecksum = htons(myChecksum) & 0xffff
else:
    myChecksum = htons(myChecksum)

header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, myChecksum, ID, 1)
packet = header + data
mySocket.sendto(packet, (destAddr, 1)) # AF_INET address must be tuple

# Both LISTS and TUPLES consist of a number of objects
# which can be referenced by their position number within the object.
def doOnePing(destAddr, timeout):
    icmp = getprotobyname("icmp")

    # SOCK_RAW is a powerful socket type. For more details:
    http://sockraw.org/papers/sock_raw
    mySocket = socket(AF_INET, SOCK_RAW, icmp)

    myID = os.getpid() & 0xFFFF # Return the current process i
    sendOnePing(mySocket, destAddr, myID)
    delay = receiveOnePing(mySocket, myID, timeout, destAddr)
    mySocket.close()
    return delay

def ping(host, timeout=1):
    # timeout=1 means: If one second goes by without a reply from the server,
    # client assumes that either the client's ping or the server's pong is
    lost
    dest = gethostbyname(host)
    print("Pinging " + dest + " using Python:")
    print("")
    # Calculate vars values and return them
    # vars = [str(round(packet_min, 2)), str(round(packet_avg, 2)),
    str(round(packet_max, 2)), str(round(stdev(stdev_var), 2))]

    # Send ping requests to a server separated by approximately one second
    for i in range(0, 4):
        delay = doOnePing(dest, timeout)
        print(delay)
        time.sleep(1) # one second
    return vars

#Pinging to different countries in separate continents
print("Ping to Washington DC:")
ping("23.81.0.59")

print("\nPing to Buenos Aires, Argentina:")

```



```
ping("131.255.7.26")

print("\nPing to Frankfurt, Germany:")
ping("195.201.213.247")
```