
ECE 369: COMPUTER NETWORKS

FINAL REPORT

May 1, 2021

We certify that this work is our own and that we did not plagiarize it.

Michael Berek _____

Michael Bisbano _____

Jacob Merola _____

Contents

1	Introduction	4
1.1	Github Repository	4
2	Understanding Socket Programming	5
2.1	Experiment 1: Client/Server Framework	5
2.2	Experiment 2: Multithreaded Server	6
2.3	Experiment 3: Timer Client	7
3	Design Choice: Client/Server vs Peer2Peer	9
3.1	Experiment 4: Peer2Peer Framework	9
3.1.1	UDP Peer2Peer	9
3.1.2	TCP Peer2Peer	10
4	Socket Integration	11
4.1	Experiment 5: Client Timer in C	11
5	Network App	13
5.1	Proposal	13
5.2	Approach	13
5.3	Problems discovered after proposal acceptance	15
5.4	Alternative solutions	15
5.5	System Design	15
5.6	Demonstration Screenshots	17
6	Conclusion	18
7	References	18

List of Figures

1	TCPServer.py and TCPClient.py demonstration	5
2	UDPServer.py and UDPClient.py demonstration	6
3	TCPServerMultithread.py with 2 simultaneous clients	7
4	UDPClientTimer.py demonstration	8
5	UDPClientTimer.py timeout	9
6	UDP Peer to Peer	10
7	TCP Peer to Peer	11
8	UDPServer and UDPClientTimer in C	12
9	UDPClientTimer timeout	13

10	Top-level diagram	14
11	System diagram	14
12	Screenshots of Web App working	17

Listings

1	TCPServer.py	19
2	TCPClient.py	19
3	UDPServer.py	21
4	UDPClient.py	21
5	TCPServerMultithread.py	22
6	UDPClientTimer.py	24
7	UDPpeer.py	26
8	TCPpeer.py	30
9	UDPServer.c	38
10	UDPClientTimer.c	41
11	MiddleGround.py	43
12	index.html	44
13	PostLogin.php	48

Abstract

In short summary, this project was a process of building up to creating a network app. This consisted of understanding socket programming with experiments 1 through 3, doing peer2peer shown in experiment 4, setting up the UDPClientTimer.py to port to C, which led into the network app developed by the group. The network app developed is a file-sharing program used to zip files from the M:/ drive on the UMass Dartmouth servers and make them accessible through a web server. HTML and CSS were used to create the web server itself while PHP scripts were used to flow between the pages and make downloads accessible, with some added python scripts to move files around. Since the group lacked sufficient access to be able to access the M:/ drive due to the firewall, it was instead decided to use a personal Raspberry Pi with a mock M:/ drive. The results were achieved in this case, where the files from the M:/ drive were zipped and sent to the web server, from which the user could safely download the zipped file onto their personal computer with no issues.

1 Introduction

The purpose of this network framework was to be able to take the concepts learned through the first five experiments and apply them to create and develop a network app. The network app that the group designed was to make accessing the M:/ drive at UMass Dartmouth possible without having to go through the VMware servers and having to log into the school's VPN. The purpose of this project if done successfully would be very useful to students that have to frequently access the M:/ drive as well, since it would remove the need to take the steps necessary to do so at this moment in time. The VMware can be slow and a pain to deal with sometimes when accessing the servers off campus. Eliminating a reason to use it with a simple solution as the one this group provides, the ability to download files off a simple web server would make a small but important improvement in a student's studies.

1.1 Github Repository

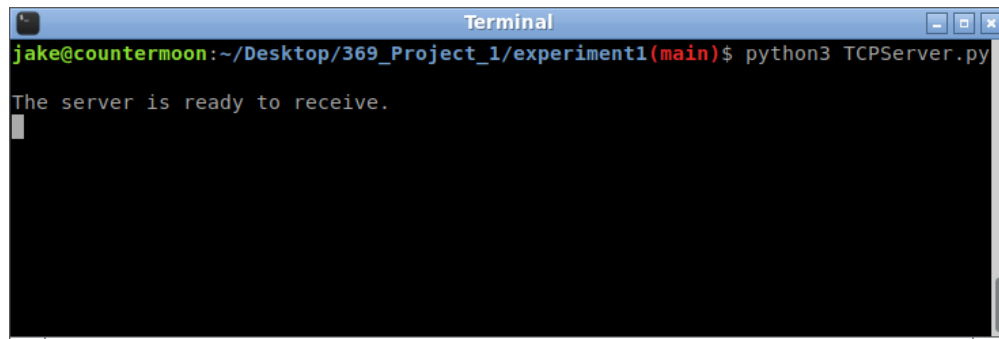
In addition to the code listings provided in the Appendix, a Github repository of this project can be found at the following address:

https://www.github.com/mbisbano1/369_Project_1

2 Understanding Socket Programming

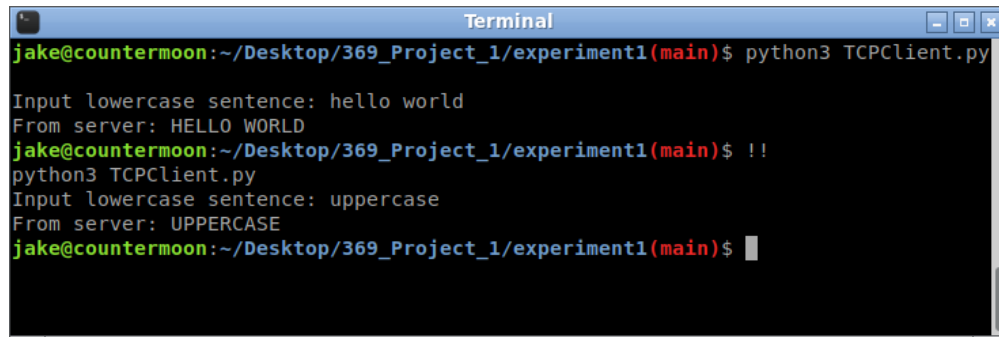
2.1 Experiment 1: Client/Server Framework

Experiment 1 required the group to develop both a TCP and UDP implementation of a client/server network. This portion of the project was not very difficult at all, as both of these protocols were assigned as a homework assignment earlier in the semester. The TCP implementation is shown in Figure 1. The program listings for TCPServer.py and TCPClient.py are provided in Listings 1 and 2 respectively, located in the Appendix.

A terminal window titled "Terminal" with a blue title bar. The prompt is "jake@countermoon:~/Desktop/369_Project_1/experiment1(main)\$". The command "python3 TCPServer.py" has been executed, and the output is "The server is ready to receive." followed by a cursor.

```
jake@countermoon:~/Desktop/369_Project_1/experiment1(main)$ python3 TCPServer.py
The server is ready to receive.
```

Server

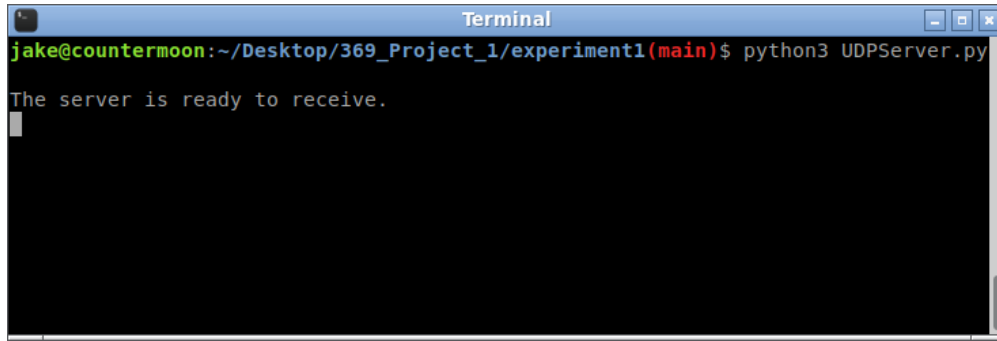
A terminal window titled "Terminal" with a blue title bar. The prompt is "jake@countermoon:~/Desktop/369_Project_1/experiment1(main)\$". The command "python3 TCPClient.py" has been executed. The output shows two interactions: first, "Input lowercase sentence: hello world" followed by "From server: HELLO WORLD"; second, "Input lowercase sentence: uppcase" followed by "From server: UPPERCASE". The prompt is now "jake@countermoon:~/Desktop/369_Project_1/experiment1(main)\$" with a cursor.

```
jake@countermoon:~/Desktop/369_Project_1/experiment1(main)$ python3 TCPClient.py
Input lowercase sentence: hello world
From server: HELLO WORLD
jake@countermoon:~/Desktop/369_Project_1/experiment1(main)$ !!
python3 TCPClient.py
Input lowercase sentence: uppcase
From server: UPPERCASE
jake@countermoon:~/Desktop/369_Project_1/experiment1(main)$
```

Client

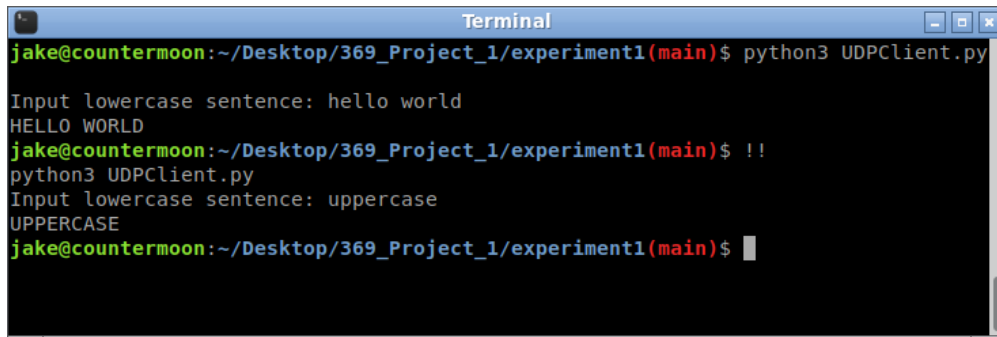
Figure 1: TCPServer.py and TCPClient.py demonstration

The UDP client/server programs have the same output as the TCP client/server. The only difference between the two protocols to an end-user is the filename. However, the underlying code is different, as the UDP program does not establish a connection with the server. The UDP server and client programs are shown in Figure 2. The program listings for UDPServer.py and UDPClient.py are provided in Listings 3 and 4 respectively, located in the Appendix.

A terminal window titled "Terminal" with a blue title bar. The prompt is "jake@countermoon:~/Desktop/369_Project_1/experiment1(main)\$". The command "python3 UDPServer.py" has been executed, and the output is "The server is ready to receive." followed by a cursor.

```
jake@countermoon:~/Desktop/369_Project_1/experiment1(main)$ python3 UDPServer.py
The server is ready to receive.
```

Server

A terminal window titled "Terminal" with a blue title bar. The prompt is "jake@countermoon:~/Desktop/369_Project_1/experiment1(main)\$". The command "python3 UDPClient.py" has been executed. The output shows two interactions: first, "Input lowercase sentence: hello world" followed by "HELLO WORLD"; second, "Input lowercase sentence: uppercase" followed by "UPPERCASE".

```
jake@countermoon:~/Desktop/369_Project_1/experiment1(main)$ python3 UDPClient.py
Input lowercase sentence: hello world
HELLO WORLD
jake@countermoon:~/Desktop/369_Project_1/experiment1(main)$ !!
python3 UDPClient.py
Input lowercase sentence: uppercase
UPPERCASE
jake@countermoon:~/Desktop/369_Project_1/experiment1(main)$
```

Client

Figure 2: UDPServer.py and UDPClient.py demonstration

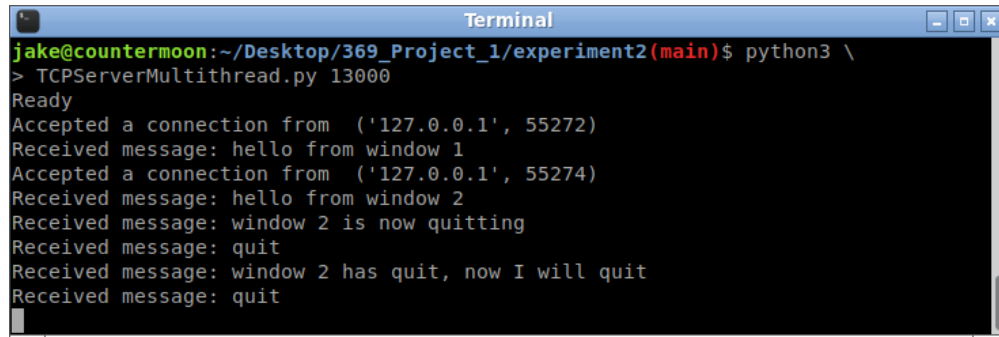
2.2 Experiment 2: Multithreaded Server

The TCP and UDP client/servers featured in Experiment 1 are single-threaded, meaning a server can only accept one client connection. The objective of Experiment 2 is to develop a multithreaded server that can manage multiple client connections simultaneously. This can be accomplished using Python’s `threading` module.

TCPServerMultithread.py is featured in Listing 5, located in the Appendix. The TCP client from Experiment 1 is reused (Listing 2). A demonstration of the multithreaded server is shown in Figure 3. The following list shows the order in which the commands were executed for this test. This order was carefully chosen to show that the server was accepting commands from both clients simultaneously:

1. Server is started (top window in Figure 3)
2. Window 1 opens connection (middle window in Figure 3)
3. Window 1 says “hello from window 1”
4. Window 2 opens connection (bottom window in Figure 3)
5. Window 2 says “hello from window 2”

6. Window 2 says “window 2 is now quitting”
7. Window 2 quits
8. Window 1 says “window 2 has quit, now I will quit”
9. Window 1 quits

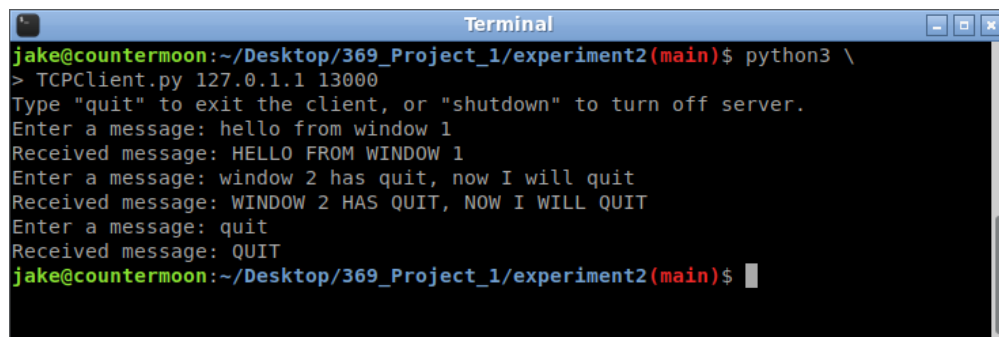


```

jake@countermoon:~/Desktop/369_Project_1/experiment2(main)$ python3 \
> TCPServerMultithread.py 13000
Ready
Accepted a connection from ('127.0.0.1', 55272)
Received message: hello from window 1
Accepted a connection from ('127.0.0.1', 55274)
Received message: hello from window 2
Received message: window 2 is now quitting
Received message: quit
Received message: window 2 has quit, now I will quit
Received message: quit

```

Server

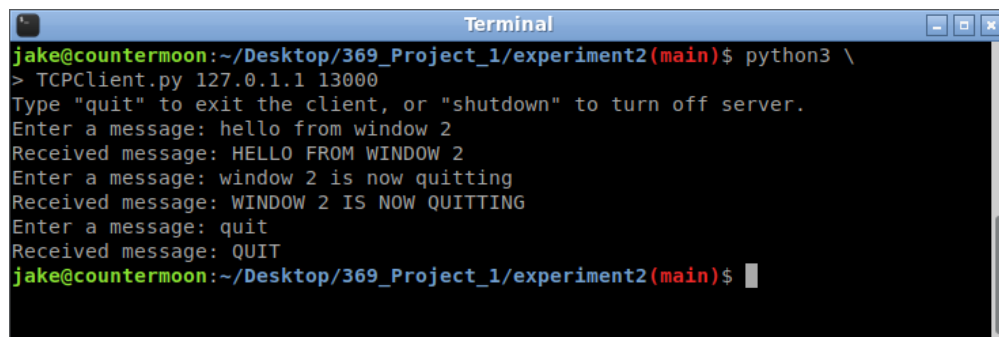


```

jake@countermoon:~/Desktop/369_Project_1/experiment2(main)$ python3 \
> TCPClient.py 127.0.1.1 13000
Type "quit" to exit the client, or "shutdown" to turn off server.
Enter a message: hello from window 1
Received message: HELLO FROM WINDOW 1
Enter a message: window 2 has quit, now I will quit
Received message: WINDOW 2 HAS QUIT, NOW I WILL QUIT
Enter a message: quit
Received message: QUIT
jake@countermoon:~/Desktop/369_Project_1/experiment2(main)$

```

Window 1



```

jake@countermoon:~/Desktop/369_Project_1/experiment2(main)$ python3 \
> TCPClient.py 127.0.1.1 13000
Type "quit" to exit the client, or "shutdown" to turn off server.
Enter a message: hello from window 2
Received message: HELLO FROM WINDOW 2
Enter a message: window 2 is now quitting
Received message: WINDOW 2 IS NOW QUITTING
Enter a message: quit
Received message: QUIT
jake@countermoon:~/Desktop/369_Project_1/experiment2(main)$

```

Window 2

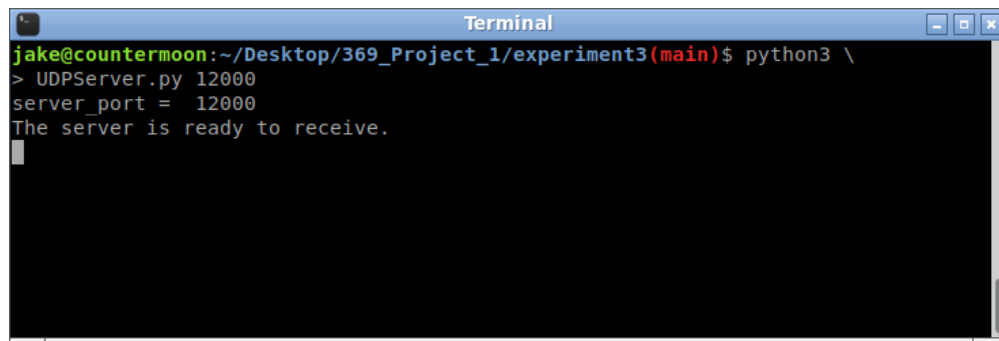
Figure 3: TCPServerMultithread.py with 2 simultaneous clients

2.3 Experiment 3: Timer Client

Experiment 3 required the group to develop a UDP client which times how long it takes to receive a reply from the server. This is accomplished using Python’s `time` module.

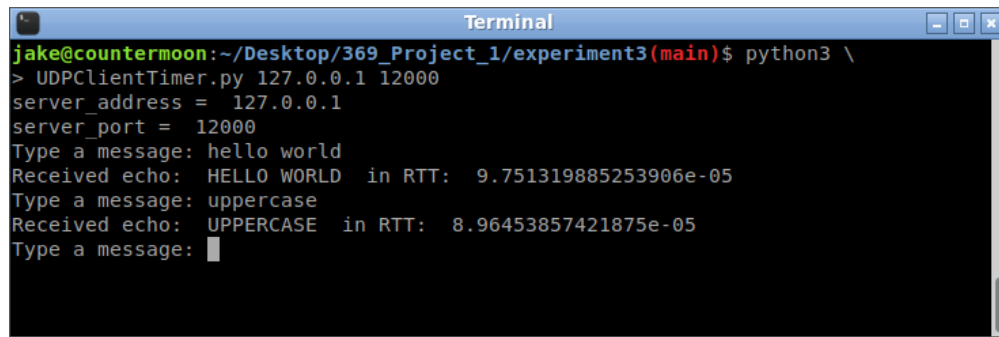
When the message is sent to the server, the start time is recorded using `Ti=time.time()` \rightarrow . When the message is received, the end time is recorded using `Tj=time.time()`. The transmission time can then be displayed using `str(Tj - Ti)`. The `time` module overloads `str()`, allowing the time to be displayed in scientific notation. This same approach is used in the C implementation of this program provided in Experiment 5.

The demonstration of `UDPClientTimer.py` is shown in Figure 4, and the corresponding code is provided in Listing 6. The delay for the first message was 97.51 microseconds, and the delay for the second message was 89.65 microseconds. This variation in round-trip time is likely due to varying activity on the network.



```
Terminal
jake@countermoon:~/Desktop/369_Project_1/experiment3(main)$ python3 \
> UDPServer.py 12000
server_port = 12000
The server is ready to receive.
```

Server

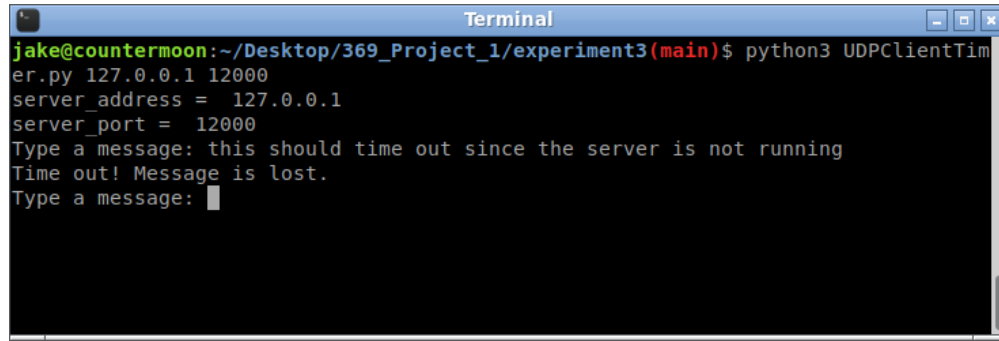


```
Terminal
jake@countermoon:~/Desktop/369_Project_1/experiment3(main)$ python3 \
> UDPClientTimer.py 127.0.0.1 12000
server_address = 127.0.0.1
server_port = 12000
Type a message: hello world
Received echo: HELLO WORLD in RTT: 9.751319885253906e-05
Type a message: uppercase
Received echo: UPPERCASE in RTT: 8.96453857421875e-05
Type a message: 
```

Client

Figure 4: `UDPClientTimer.py` demonstration

If the client does not receive a response within 1 second (chosen arbitrarily), it provides a timeout message to the user. This is shown in Figure 5.

A terminal window titled "Terminal" with a blue title bar. The prompt is "jake@countermoon:~/Desktop/369_Project_1/experiment3(main)\$". The command "python3 UDPClientTimer.py 127.0.0.1 12000" has been executed. The output shows the program setting "server_address = 127.0.0.1" and "server_port = 12000". It then prompts "Type a message:" and the user enters "this should time out since the server is not running". The program responds with "Time out! Message is lost." and then prompts "Type a message:" again, with a cursor visible.

```
jake@countermoon:~/Desktop/369_Project_1/experiment3(main)$ python3 UDPClientTimer.py 127.0.0.1 12000
server_address = 127.0.0.1
server_port = 12000
Type a message: this should time out since the server is not running
Time out! Message is lost.
Type a message: █
```

Figure 5: UDPClientTimer.py timeout

3 Design Choice: Client/Server vs Peer2Peer

3.1 Experiment 4: Peer2Peer Framework

Peer to peer frameworks, often shortened to P2P, are a powerful method of distributing the workload of a network application away from the standard client-server architecture and creating a more flexible system. The standard client-server application has a centralized server which is Always-On and accepts connections/datagrams from the application users, known as clients. P2P on the other hand, is an interconnection of all the users, in such a way that each user instantiates a server and client “side”. Below, a basic communication/chat application was developed with a P2P architecture using both UDP and TCP, to facilitate a better understanding of how P2P could be implemented for the Network App portion of this project.

3.1.1 UDP Peer2Peer

The UDP P2P implementation’s behavior is shown below in Figure 6, and the code is provided in Listing 7. The top and bottom halves of the figure are the same code running on two separate hosts in a local area network. The figure shows that by simply starting the code, the peers are able to find each-other on the network and communicate/transfer messages to each-other. This is done using Broadcast messages to “ping” the entire network for other peers, and then storing a local list of all other active peers addresses to communicate with.

```
mbisbano@Desktop:/mnt/d/Documents/GitHub/369_Project_1/experiment4$ python3 UDPpeer.py
Hostname is: Desktop
IP address is: 127.0.1.1
Broadcast Sent!
Starting Server Thread!
Starting Client Thread!
Both Threads Started
ServerSide is ready to receive.
Input message to send: Message Received from: ('192.168.0.172', 12000)
Acknowledged from: 127.0.1.1
Appended 192.168.0.172 to peersList
hello peer! From Desktop
message sent will be input message..
Sending to:
  peer: 192.168.0.172
  port: 12000
Message Received from: ('192.168.0.172', 12000)
Acknowledged from: 127.0.1.1
Input message to send: Message Received from: ('192.168.0.172', 56980)
Hello peer! From Home-Office
```

UDP Peer running on Desktop

```
mbisbano@Home-Office:/mnt/c/Users/Michael/Documents/GitHub/369_Project_1/experiment4$ python3 UDPpeer.py
Hostname is: Home-Office
IP address is: 127.0.1.1
Broadcast Sent!
Starting Server Thread!
Starting Client Thread!
Both Threads Started
ServerSide is ready to receive.
Input message to send: Message Received from: ('192.168.0.150', 61093)
  ECE369 Peer Active 127.0.1.1
Appended 192.168.0.150 to peersList
Message Received from: ('192.168.0.150', 52260)
hello peer! From Desktop
Hello peer! From Home-Office
message sent will be input message..
Sending to:
  peer: 192.168.0.150
  port: 12000
Message Received from: ('192.168.0.150', 12000)
Acknowledged from: 127.0.1.1
Input message to send: _
```

UDP Peer running on Home Office

Figure 6: UDP Peer to Peer

3.1.2 TCP Peer2Peer

The TCP P2P implementation's behavior is shown below in Figure 7, and the code is provided in Listing 8. Just as in the UDP version, the top and bottom halves of the figure are the same code running on two separate hosts in a local area network. The TCP implementation behaves very similarly to the UDP one, at least from the user's perspective. This is done using Broadcast messages to “ping” the entire network for other peers, and then threads are started to open connections to all the active peers on the network to create the inter-connected P2P architecture.

```

D:\Documents\GitHub\369_Project_1\C:/Users/bacon/AppData/Local/Programs/Python/Python37/python.exe d:/Documents/GitHub/369_Project_1/experiment4/TCPpeer.py
Hostname is: Desktop
IP address is: 192.168.0.150
Started the Broadcast Handler Thread, it will scan for other peers to connect to!
Broadcast Sent!
Wait 5 seconds for response:
You are the next Master! Be prepared!
Next Master thread started!
Started the Server thread
Master Address is: 192.168.0.172
Started the Client thread
The server is ready to receive 10 peers.
Input message to send to peers: Received Message from: 192.168.0.172
Message: Hello from HomeOffice
HI from Desktop
Manager set MessageQueued to True
Client Thread has Message Queued
Manager waiting for all client threads to send message
self.MessagesSent = 0
length of self.clientThreads = 1
From server: HI FROM DESKTOP
incremented MessagesSent value
Manager done waiting for client threads to send message!
Thread done waiting for all clients to send message
Input message to send to peers:

```

TCP Peer running on Desktop

```

C:\Users\Michael\Documents\GitHub\369_Project_1\C:/Users/Michael/AppData/Local/Microsoft/WindowsApps/python.exe c:/Users/Michael/Documents/GitHub/369_Project_1/experiment4/TCPpeer2.py
Hostname is: Home-Office
IP address is: 192.168.0.172
Started the Broadcast Handler Thread, it will scan for other peers to connect to!
Broadcast Sent!
Wait 5 seconds for response:
You are the Master on this Network! Congrats!
Master Thread Started!
Started the Server thread
The server is ready to receive 10 peers.
Started the Client thread
Input message to send to peers: Appended 192.168.0.150 to peersList
Hello from HomeOffice
Manager set MessageQueued to True
Manager waiting for all client threads to send message
self.MessagesSent = 0
length of self.clientThreads = 0
Manager done waiting for client threads to send message!
Starting new Client Thread: Addr = 192.168.0.150
Input message to send to peers: Hello from HomeOffice
Manager set MessageQueued to True
Manager waiting for all client threads to send message
self.MessagesSent = 0
length of self.clientThreads = 1
Client Thread has Message Queued
From server: HELLO FROM HOMEOFFICE
incremented MessagesSent value
Manager done waiting for client threads to send message!
Thread done waiting for all clients to send message
Input message to send to peers: Received Message from: 192.168.0.150
Message: HI from Desktop

```

TCP Peer running on Home Office

Figure 7: TCP Peer to Peer

Based off of the lessons learned implementing the P2P experiment codes in both UDP and TCP, the group decided to stick to the standard client-server architecture for the Network App, as the M-Drive files are on a centralized server anyways, so keeping the centralized nature of those files is much more efficient than spreading it out throughout a P2P system.

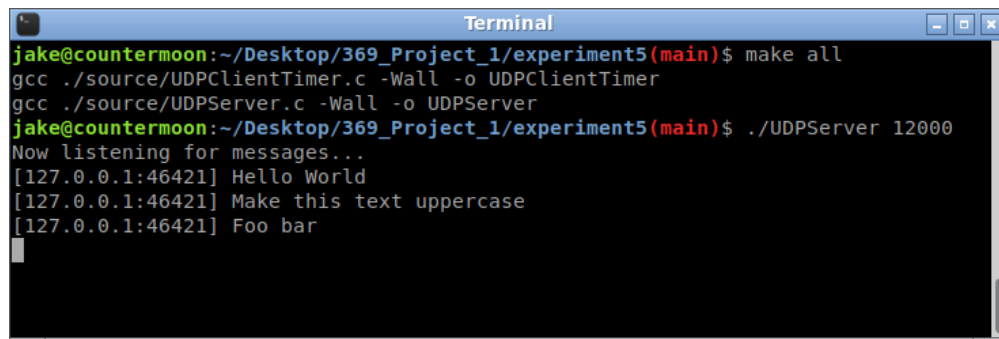
4 Socket Integration

4.1 Experiment 5: Client Timer in C

The UDP timer client written in Experiment 3 was re-written in C. This was slightly difficult, as C's procedural paradigm is not very conducive to the object-oriented approach taught in the course. One issue with socket programming in C is namespace pollution. In Python, importing a module with `import socket` requires each socket method to be prefaced with `socket`, such as `socket.close()` and `socket.bind()`. In C, header files `<sys/socket`

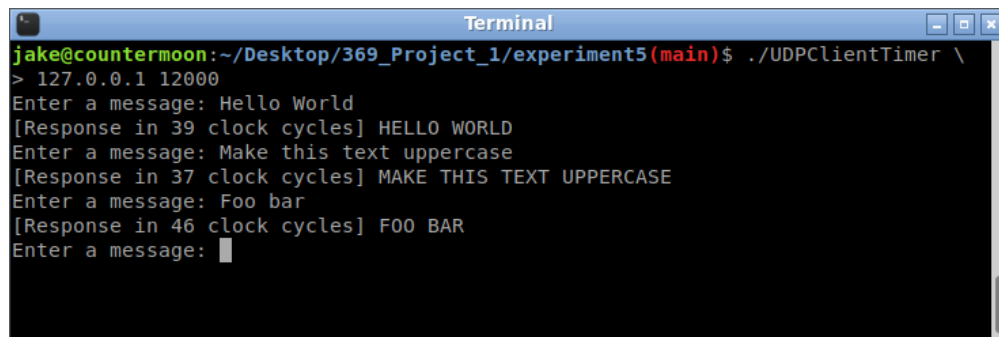
↪ `.h` and `<arpa/inet.h>` are placed in the global space. This makes it confusing to keep track of all the methods. For example, does `sendto()` belong to `<sys/socket.h>` or `<arpa/inet.h>`? This information is readily available on the Internet, but it adds an extra layer of confusion that is avoided using Python's multiple namespaces.

Despite some initial confusion, server and client programs were successfully written in C. A makefile was written to automate the compilation process, allowing the terminal command `make all` to compile both `UDPServer.c` and `UDPClientTimer.c`. The `UDPServer` and `UDPClientTimer` executables are shown in Figure 8.



```
Terminal
jake@countermoon:~/Desktop/369_Project_1/experiment5(main)$ make all
gcc ./source/UDPClientTimer.c -Wall -o UDPClientTimer
gcc ./source/UDPServer.c -Wall -o UDPServer
jake@countermoon:~/Desktop/369_Project_1/experiment5(main)$ ./UDPServer 12000
Now listening for messages...
[127.0.0.1:46421] Hello World
[127.0.0.1:46421] Make this text uppercase
[127.0.0.1:46421] Foo bar
```

Server

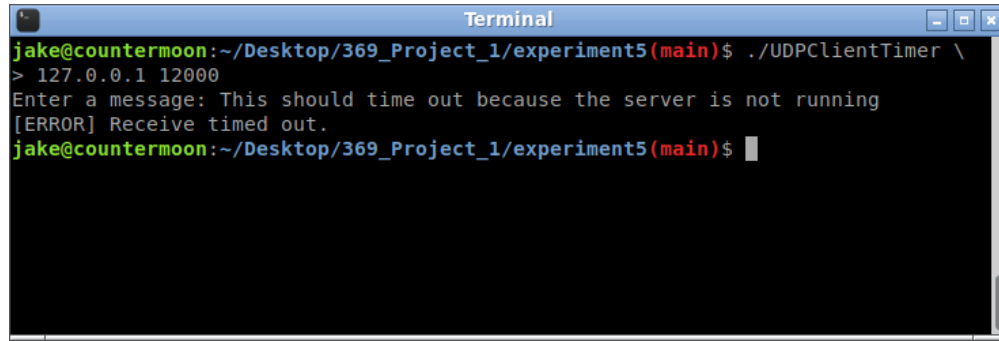


```
Terminal
jake@countermoon:~/Desktop/369_Project_1/experiment5(main)$ ./UDPClientTimer \
> 127.0.0.1 12000
Enter a message: Hello World
[Response in 39 clock cycles] HELLO WORLD
Enter a message: Make this text uppercase
[Response in 37 clock cycles] MAKE THIS TEXT UPPERCASE
Enter a message: Foo bar
[Response in 46 clock cycles] FOO BAR
Enter a message:
```

Client

Figure 8: `UDPServer` and `UDPClientTimer` in C

Header file `<sys/socket.h>` allows the socket timeout to be set using a special `timeval` struct. The timeout has been arbitrarily set to 10 seconds for this implementation. The timeout is shown in Figure 9.

A terminal window titled "Terminal" with a blue title bar. The prompt is "jake@countermoon:~/Desktop/369_Project_1/experiment5(main)\$". The user enters "./UDPClientTimer \", followed by "> 127.0.0.1 12000". The program outputs "Enter a message: This should time out because the server is not running" and "[ERROR] Receive timed out." The prompt returns to "jake@countermoon:~/Desktop/369_Project_1/experiment5(main)\$".

```
jake@countermoon:~/Desktop/369_Project_1/experiment5(main)$ ./UDPClientTimer \  
> 127.0.0.1 12000  
Enter a message: This should time out because the server is not running  
[ERROR] Receive timed out.  
jake@countermoon:~/Desktop/369_Project_1/experiment5(main)$
```

Figure 9: UDPClientTimer timeout

5 Network App

5.1 Proposal

Proposal as submitted to the Professor on March 22, 2021:

“Students in the engineering program frequently need access to the M:/ drive, but access can be more frustrating for students attending classes virtually or not using an engineering lab computer. Instead of having to remote desktop into the engineering server and email/onedrive a file to/from the M:/ drive, we want students to have a seamless web interface that handles file sharing to/from the M:/drive. Our objective for this class project is to develop a Raspberry Pi server that will access files from the engineering department’s M:/ drive. It will feature a web interface allowing easy access to the files after the user authenticates using their UMassD credentials; the interface will be read-only.”

5.2 Approach

Figure 10 shows a top-level diagram of the system. The Raspberry Pi will serve as the middle ground between the bottom-level engineering server (eng-svr-1) and the top-level user interface (web browser). More information about this top-middle-bottom approach is explained in the following paragraphs.

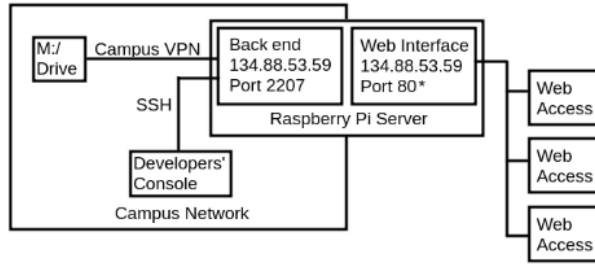


Figure 10: Top-level diagram

Figure 11 shows a more technical diagram of the system. The system consists of three levels. The top level is the web interface the user will use to login and access files. It will be implemented using HTML and CSS, as well as PHP calls to the middle level. The middle level, running on the Raspberry Pi, populates the directory listing and traverses the M:/ drive directories. It presents the directory listings to the upper level. The lower level, also running on the Raspberry Pi, accesses the files using SSH/SCP and provides authentication to the remote server. It presents the directory listings and files to the middle level.

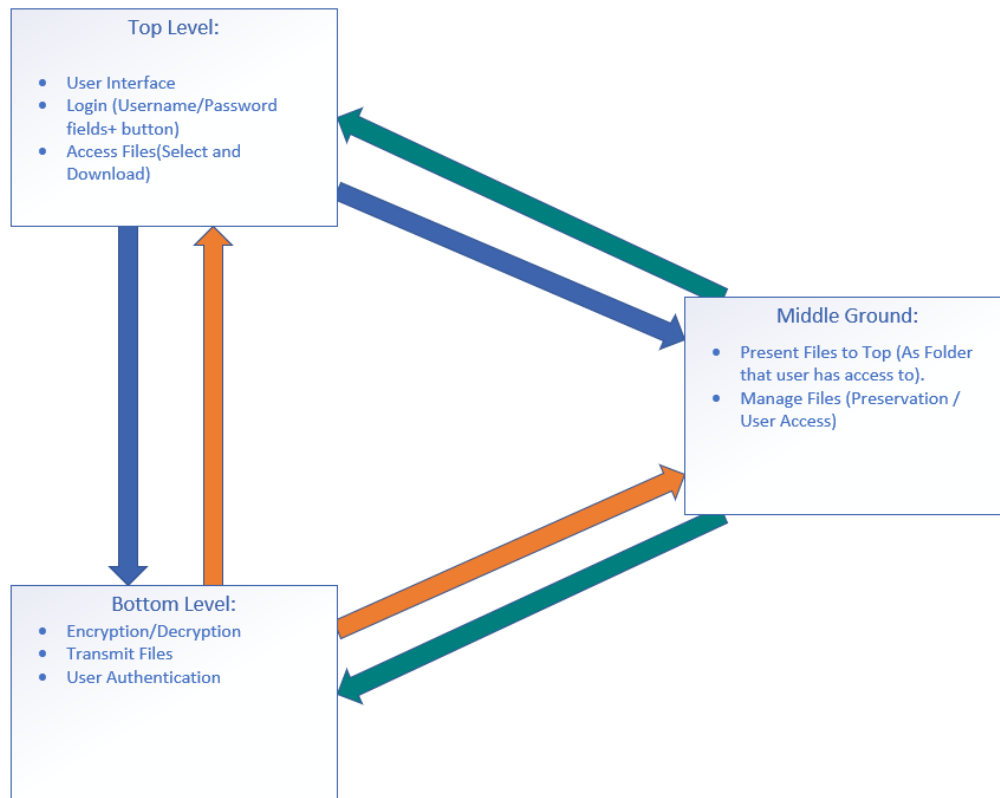


Figure 11: System diagram

5.3 Problems discovered after proposal acceptance

The main issue discovered after the group's proposal was accepted by the Professor was how to safely authenticate the user's UMassD credentials. The safest way to SSH/SCP to a remote server without sending a password is to generate an RSA public/private key pair. In a command-line interface, this is done by running `ssh-keygen` and following the prompts, which includes entering the password. Once this key is created, it can then be copied to the remote server using `ssh-copy-id`. The server can then be accessed without typing in a password. While this works well for CLI users, `ssh-keygen` cannot easily be abstracted through a GUI because its prompts are interactive. Additionally, the password would still need to travel from the GUI to `ssh-keygen` and `ssh-copy-id`. This renders this approach useless, as the entire goal was to prevent the password from being sent insecurely. To make matters worse, running `ssh-copy-id` on an ECE Raspberry Pi resulted in that team member's UMassD logon being locked! At this point, it was accepted that accessing the actual M:/ drive as proposed would be impossible.

5.4 Alternative solutions

Two alternative solutions were devised after the failure of the group's initial proposal. The first solution entails configuring a web interface on a school Raspberry Pi as proposed, but only to access a faux M:/ drive located on the Raspberry Pi. This would serve as a proof-of-concept of how the system would work if the actual M:/ drive weren't locked down. This eliminates the authentication issues mentioned in the previous sections, yet still allows the server to be accessed from anywhere in the world, as the Raspberry Pi's IP address (134.88.53.59) is public-facing. Unfortunately, this approach came to an end rather quickly as port 80 is locked on the Raspberry Pi. This was likely done as a security measure by ECE technicians. Methods of disabling the firewall, such as `sudo ufw allow 'Apache Full'`, did not work.

The second alternative consists of configuring a web server on a team member's Raspberry Pi connected to their own LAN. This allows that particular team member to have full control over the network, eliminating firewall problems. This is the approach the group chose. The only disadvantage of this approach is that the other group members could not access the server across LANs. Setting up a static IP address and port forwarding would allow this, however this introduces security risks and is difficult to implement correctly.

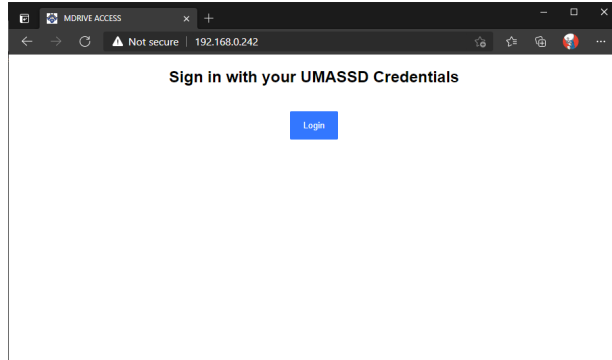
5.5 System Design

The final design of the group's Web App was mostly implemented in Python and PHP/HTML. Since the group decided to implement our app on a local network due to unforeseen complications with the campus network, the M:/ drive contents were simulated and

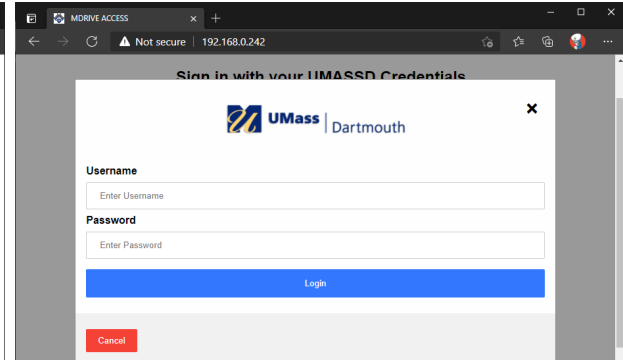
hosted on a local Raspberry Pi. These contents were offered to users accessing the Raspberry Pi through a simple http website. Users are able to login and request the server to package up the entirety of the M:/ drive folder as a .zip file. Once ready, the website alerts the user and allows them to download the .zip file. This behavior can be seen working in the screenshots of the web app below in Figure 12. The code used to implement this on the Raspberry Pi 4B with an Apache2 webserver is shown in the Github Repository (linked in Section 1.1) under the ZeroDrive folder, which is what the web app was jokingly nicknamed, as it is at-least one less than Microsoft's OneDrive.

Behind the scenes on the Raspberry Pi's server side of things, there is a Python script called `MiddleGround.py` (Listing 11) which imports the `shutil` library, which allows many filesystem functions to be automated within python, including zipping a directory and moving a file to a different directory. These functions within the python script are called by a PHP code called `PostLogin.php` (Listing 13) running on the webserver whenever a user clicks the "RE-ZIP the MDrive" button. This PHP code also presents the HTML code of the click button and other HTML functionality in it, eliminating the need for a discrete HTML file for the second page of the website. Within HTML, it is easy to create a downloadable resource, which is how the MDrive .zip file is made available for the user to download. As long as they click the "RE-ZIP" button before they download it, they can be confident that the files they receive are up to date. The home/login page of the webserver, `index.html` (Listing 12) was written with only HTML, and implemented a few advanced features of HTML formatting to provide a login page for users. In its present state, the login allows any credentials through, but creating some basic authentication that isn't held in plaintext would be the next step in developing this app given more time and resources.

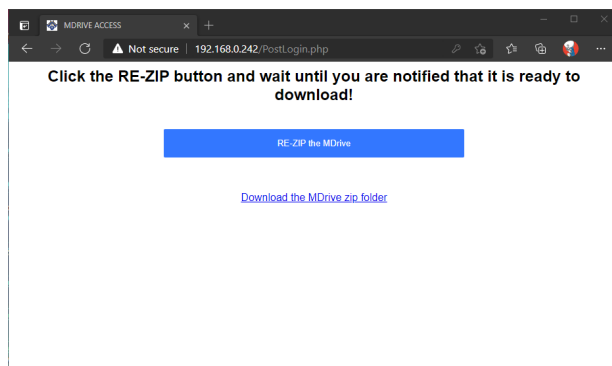
5.6 Demonstration Screenshots



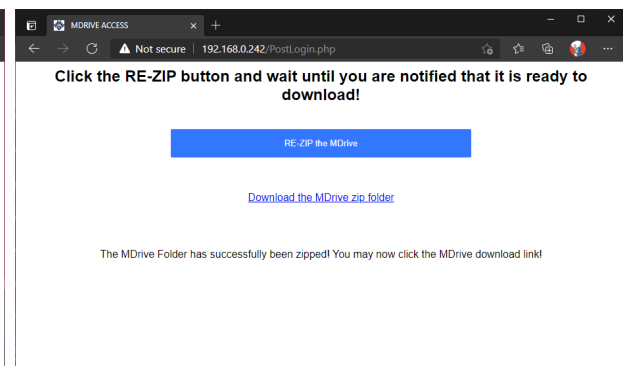
Home page of website



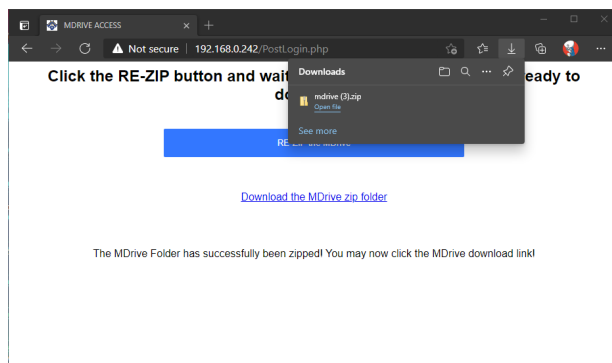
Login screen of website



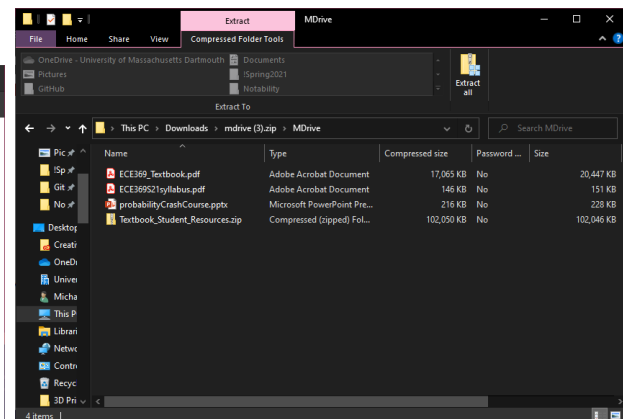
Rezip the Mdrive to update available contents



The mdrive is ready to download



Downloading the mdrive



The contents of simulated mdrive

Figure 12: Screenshots of Web App working

6 Conclusion

In conclusion, the lessons learned are planning ahead and making sure problems won't be run into when creating a project such as the one the group made. Since problems were run into accessing the M:/ drive, the group was unable to create the actual project desired. However, figuring this out ahead of time allowed the students to come up with a solution that allowed the concept to still be made successfully and developed. A couple improvements that could be made are the secure log-in hashing that hadn't been developed yet. This would make it much more difficult for hackers or malicious software to obtain user credentials from signing in. A second noticeable improvement that could be developed that the group didn't have time to develop within the time constraint would be automatically unzipping the zip file that has to be created. In theory this would remove the possibility of user error if the user accidentally forgets to click the unzip button after downloading the files.

7 References

- <https://www.w3schools.com/html>
- <https://www.w3schools.com/css>
- <https://www.w3schools.com/php>
- <https://docs.python.org/3/library/crypto.html>
- <https://docs.python.org/3/library/shutil.html>
- Course text (Kurose & Ross, "Computer Networking: A Top-Down Approach," 8e)

Appendix

Listing 1: TCPServer.py

```
from socket import *
serverPort=12000

class TCPServer:
    def __init__(self):
        try:
            serverSocket=socket(AF_INET, SOCK_STREAM)
            serverSocket.bind('', serverPort)
            serverSocket.listen(1)
            print("The server is ready to receive.")
            while 1:
                connectionSocket, addr = serverSocket.accept()
                sentence=connectionSocket.recv(1024).decode()
                capitalizedSentence=sentence.upper()
                connectionSocket.send(capitalizedSentence.encode())
                connectionSocket.close()

        except KeyboardInterrupt:
            print("Keyboard interrupt")
            exit(1)

TCPServer()
```

Listing 2: TCPClient.py

```
import socket
import sys
encoding="UTF-8"

class TCPClient:
    def __init__(self):
        if (len(sys.argv) != 3):
            print("Usage: python3 TCPClient.py server_address  

                 ↪ server_port")
            exit(1)

        serverAddress=sys.argv[1]
        try:
            serverPort=int(sys.argv[2])
        except ValueError:
```

```

    print("Error: input argument invalid.")
    exit(1)

if ((serverPort <= 1024) or (serverPort >= 65536)):
    print("Error: Server port value must be greater than 1024
    ↪ and less than 65536.")
    exit(1)

self.connect(serverAddress, serverPort)

def connect(self, serverAddress, serverPort):
    try:
        clientSocket=socket.socket(socket.AF_INET, socket.
        ↪ SOCK_STREAM)
        try:
            clientSocket.connect((serverAddress, serverPort))
        except ConnectionRefusedError:
            print("Error: Connection refused.")
            exit(1)

    print("Type \"quit\" to exit the client, or \"shutdown\" to
    ↪ turn off server.")

    while 1:
        message=input("Enter a message: ")
        clientSocket.send(message.encode(encoding))
        modifiedMessage=clientSocket.recv(1024)
        print("Received message: %s" % modifiedMessage.decode(
        ↪ encoding))

        if ((message == "quit") or (message == "shutdown")):
            break

    clientSocket.close()

except KeyboardInterrupt:
    print("Keyboard interrupt")
    if (clientSocket):
        clientSocket.close()
        print("Socket closed successfully.")

```

```

        except Exception as ex:
            print("Exception: %s" % ex)
            if (clientSocket):
                clientSocket.close()
                print("Socket closed successfully.")

TCPClient()

```

Listing 3: UDPServer.py

```

from socket import *
serverPort=12000

class UDPServer:
    def __init__(self):
        try:
            serverSocket=socket(AF_INET, SOCK_DGRAM)
            serverSocket.bind('', serverPort)
            print("The server is ready to receive.")
            while 1:
                message, clientAddress = serverSocket.recvfrom(2048)
                modifiedMessage=message.decode().upper()
                serverSocket.sendto(modifiedMessage.encode(),
                                    ↪ clientAddress)

        except KeyboardInterrupt:
            print("Keyboard interrupt")
            exit(1)

UDPServer()

```

Listing 4: UDPClient.py

```

from socket import *
serverName='localhost'
serverPort=12000

class UDPClient:
    def __init__(self):
        try:
            clientSocket=socket(AF_INET, SOCK_DGRAM)
            message=input("Input lowercase sentence: ")

```

```

        clientSocket.sendto(message.encode(), (serverName,
            ↪ serverPort))
        modifiedMessage, serverAddress = clientSocket.recvfrom
            ↪ (2048)
        print(modifiedMessage.decode())
        clientSocket.close()

    except KeyboardInterrupt:
        print("Keyboard interrupt")
        exit(1)

UDPClient()

```

Listing 5: TCPServerMultithread.py

```

import socket
import threading
import sys

encoding="UTF-8"

class TCPServerMultithread:
    def __init__(self):
        if (len(sys.argv) != 2):
            print("Usage: python3 TCPServerMultithread.py server_port")
            exit(1)

        try:
            serverPort=int(sys.argv[1])
        except ValueError:
            print("Error: input argument invalid.")
            exit(1)

        if ((serverPort <= 1024) or (serverPort >= 65536)):
            print("Error: Server port value must be greater than 1024
                ↪ and less than 65536.")
            exit(1)

        self.serverLoop(serverPort)

    def serverLoop(self, serverPort):
        try:

```

```

serverSocket=socket.socket(socket.AF_INET, socket.
    ↪ SOCK_STREAM)
host=socket.gethostname()
serverSocket.bind((host, serverPort))
print("Ready")
serverSocket.listen(5)
threadList=[]

while True:
    c, addr = serverSocket.accept()
    temp=threading.Thread(target=self.talkToClient, args=(c
        ↪ , addr ))
    temp.daemon=True
    temp.start()
    threadList.append(temp)

for thread in threadList:
    thread.join()

serverSocket.shutdown(0)
serverSocket.close()

except IOError:
    if (serverSocket):
        serverSocket.close()
    print("Error: Could not open socket.")
    exit(1)

except KeyboardInterrupt:
    print("Keyboard interrupt")
    if (serverSocket):
        serverSocket.close()
        print("Socket closed successfully.")
    exit(1)

def talkToClient(self, socket, address):
    print("Accepted a connection from ", address)
    watchdog=0

    while (watchdog < 50):
        watchdog+=1

```

```

        message=socket.recv(1024).decode(encoding)
        print("Received message: %s" % message)

        socket.send(message.upper().encode(encoding))

        if (message == "shutdown"):
            socket.close()
            break
        elif (message == "quit"):
            socket.close()
            break

TCPServerMultithread()

```

Listing 6: UDPClientTimer.py

```

# UDP Client Timer (simple echo timer code in Python)

# Import socket, system, and time modules
from socket import *
import sys, time

timeout = 1 # in second
global server_address
global server_port
if len(sys.argv) <= 2:
    #print 'Usage: "python UDPclient.py server_address server_port"'
    print('Usage: "python UDPclient.py server_address server_port"')
    #print 'server_address = Visible Inside: "eng-svr-1" or 2 or "
    ↪ localhost" or "127.0.0.1"'
    print('server_address = Visible Inside: "eng-svr-1" or 2 or "
    ↪ localhost" or "127.0.0.1"')
    #print '
    ↪ Visible Outside: IP address or fully
    ↪ qualified doman name'
    print('
    ↪ Visible Outside: IP address or fully
    ↪ qualified doman name')
    #print 'server_port = server socket port: #80GX'
    print('server_port = server socket port: #80GX')

    print('Using Default values for server_address and server_port')
    print('server_address = "localhost"')
    print('server_port = 12000')

```



```

server_address = 'localhost'
server_port = 12000
#sys.exit(2)
else:
    server_address = sys.argv[1]
    server_port = int(sys.argv[2])
    print('server_address = ', server_address)
    print('server_port = ', server_port)

# Create a UDP client socket: (AF_INET for IPv4 protocols, SOCK_DGRAM
    ↪ for UDP)
clientSocket = socket(AF_INET, SOCK_DGRAM)

# Client takes message from user input, sends it to the server, and
    ↪ receives its echo
while True:
    #message = raw_input ("Type a message: ")
    message = input("Type a message: ")
    # Set socket timeout as 1 second
    clientSocket.settimeout(timeout)
    try:
        # Record send_time
        Ti = time.time()
        # Send a UDP packet
        #clientSocket.sendto(message.encode(), (sys.argv[1], int(sys.
            ↪ argv[2])))
        clientSocket.sendto(message.encode(), (server_address,
            ↪ server_port))
        # Receive the server response
        modifiedMessage, serverAddress = clientSocket.recvfrom(1024)
        # Record receive_time
        Tj = time.time()
        # Print the RTT along with the echo message
        #print 'Received echo: ' , modifiedMessage, ' in RTT: ', str(Tj
            ↪ - Ti)
        print('Received echo: ' , modifiedMessage.decode(), ' in RTT: '
            ↪ , str(Tj - Ti))

```

```

except:
    # Server does not respond, so assume the packet is lost
    #print 'Time out! Message is lost.'
    print('Time out! Message is lost.')
if message == 'quit' or message == 'shutdown':
    #print 'Client quits!'
    print('Client quits!')
    break

    #print(
# Close the client socket
clientSocket.close()
sys.exit(0)

```

Listing 7: UDPpeer.py

```

import socket
import sys, threading, time

class UDPPeer:
    """UDP P2P Class/Object"""
    hostname = ''          # Each peer has it's own hostname
    addr = ''              # Each peer has it's own address
    peersList = []         # Each peer maintains a list of other peers
    ↪ addresses.add()
    ServerPort = 12000     # Each peer uses port 12000 for now?
    ClientPort = 12001     # Each peer has it's output/client side
    ↪ port at 12001 for now
    serverRunning = True   # Each peer has a status for it's server
    ↪ and client side
    clientRunning = True   # ...

    def quit(self):
        print("Quitting!")
        self.serverRunning = False
        self.clientRunning = False

    def serverSide(self):
        print("Starting Server Thread!")
        try:    # Try setting up the server
            serverS=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            serverS.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,

```

```

        ↪ 1)
serverS.bind('',self.ServerPort))
print("ServerSide is ready to receive.")
while self.serverRunning:
    message, clientAddress = serverS.recvfrom(2048)
    print("Message Received from: ", clientAddress)
    print(message.decode())
    if (clientAddress[0] not in self.peersList):
        self.peersList.append(clientAddress[0])
        print("Appended ", clientAddress[0], " to peersList
        ↪ ")
        time.sleep(1)
    elif (clientAddress in self.peersList) and (message.
    ↪ decode()[0:18] == ' ECE369 Peer Quit '):
        self.peersList.remove(clientAddress[0])
        print("Removed ", clientAddress[0], " from
        ↪ peersList")
    if (message.decode()[0:19] != 'Acknowledged from: '):
        #don't ack an ack you muppet
        ackMessage = 'Acknowledged from: ' + self.addr
        serverS.sendto(ackMessage.encode(), (clientAddress
        ↪ [0], self.ServerPort))

except KeyboardInterrupt:
    print("Keyboard Interrupt!")
    exit(1)
except Exception as ex:
    print("Exception: %s" % ex)
    exit(1)

def clientSide(self):
    print("Starting Client Thread!")

    try:
        clientS=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        clientS.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,
        ↪ 1)
        clientS.settimeout(3)    # set socket timeout as 1 second
        while self.clientRunning:
            self.waitingForInput = True
            typedInput=input("Input message to send: ")

```

```

self.waitingForInput = False
message = ''
if typedInput == 'Quit':
    print("message sent will be quit message..")
    message = ' ECE369 Peer Quit ' + self.addr
else:
    print("message sent will be input message..")
    message = typedInput
for peer in self.peersList:
    try:      # Try to send message to each peer in
        ↪ peersList
        print("Sending to: ")
        print(" peer: ", peer)
        print(" port: ", str(self.ServerPort))
        clientS.sendto(message.encode(), (peer, self.
            ↪ ServerPort))

        responseMessage, peerAddress = clientS.recvfrom
            ↪ (2048)
        print("Acknowledge Received from ", peerAddress
            ↪ )
        print(responseMessage)
    except KeyboardInterrupt:
        print("Keyboard Interrupt!")
        exit(1)
    except TimeoutError as ex:
        print("Timout Exception: %s" % ex)
        self.peersList.remove(peer)
        print("Removed ", peer, " from peersList")
    except Exception as ex:
        ex = ''
        #print("Exception: %s" % ex)
        #exit(1)
except KeyboardInterrupt:
    print("Keyboard Interrupt!")
    exit(1)
except Exception as ex:
    print("Exception: %s" % ex)
    #exit(1)
time.sleep(1)

```

```

def __init__(self):
    #self.addr = input("Enter your IP Address from ifconfig: eg.
    ↪ 192.168.0.150")
    self.hostname = socket.gethostname()
    self.addr = socket.gethostbyname(self.hostname)
    print("Hostname is: ", self.hostname)
    print("IP address is: ", self.addr)
    self.serverRunning = True
    try:    # Try sending broadcast to scan for other peers on
    ↪ network
        BroadcastS=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        BroadcastS.setsockopt(socket.SOL_SOCKET, socket.
        ↪ SO_BROADCAST, 1)
        BroadcastS.setsockopt(socket.SOL_SOCKET, socket.
        ↪ SO_REUSEADDR, 1)
        broadcastMessage = ' ECE369 Peer Active ' + self.addr
        #BroadcastS.sendto(broadcastMessage, ('255.255.255.255',
        ↪ 12000))
        BroadcastS.sendto(broadcastMessage.encode(), ('<broadcast>',
        ↪ 12000))
        print("Broadcast Sent!")
        BroadcastS.close
    except KeyboardInterrupt:
        print("Keyboard Interrupt!")
        exit(1)
    except Exception as ex:
        print("Exception: %s" % ex)
        exit(1)

    try:    # Try setting up threads for handling the client and
    ↪ server side of this peer
        # Create two new threads
        serverThread = threading.Thread(target=self.serverSide,
        ↪ args=())
        clientThread = threading.Thread(target=self.clientSide,
        ↪ args=())
        # Start the two threads
        serverThread.start()
        clientThread.start()
        print("Both Threads Started")
    except KeyboardInterrupt:

```

```

        print("Keyboard Interrupt!")
        exit(1)
    except Exception as ex:
        print("Exception: %s" % ex)
        exit(1)

UDPPeer()

```

Listing 8: TCPpeer.py

```

import socket
import sys, threading, time
import queue

# constant definitions
maxNumPeers = 10
TimeoutDelay = 5
masterPort = 42069
serverPort = 12000
clientPortStartingNum = 12001
peerFlag = 'ECE369 TCPP2P ACTIVE '

class TCPpeer:
    """TCP P2P Class/Object"""
    hostname = ''          # Each peer has it's own hostname
    addr = ''              # Each peer has it's own address
    peersList = []         # Each peer maintains a list of other peers
    ↪ addresses.add()
    newPeersList = []      # each peer has a new list of peers to
    ↪ compare
    peersConnectedList = [] # each peer has a flag for if a client is
    ↪ connected to it.
    serverRunning = False  # Each peer has a status for it's server
    ↪ and client side
    clientRunning = False  # ...
    numClients = 0

    clientThreads = []     # Each peer has a list of client threads
    ↪ running.

    isMaster = False

```

```

isNextMaster = False
masterTimeOut = False

newPeers = False

wantToShutdown = False
readyToShutdown = False
peerRunning = False

Message = ''
MessageQueued = False
MessagesSent = 0

#peerScanRunning = False # Each peer has a status for if it is
    ↪ still accepting UDP broadcast messages to map other peers in
    ↪ network

def Master(self):    # The master maintains the peersList and checks
    ↪ for peers that have timed out
    try:
        MasterS = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        MasterS.bind('', masterPort)
        while(self.isMaster):
            try:
                message, clientAddress = MasterS.recvfrom(2048)
                rxPeerFlag = message.decode()[0:21]

                if ((rxPeerFlag == peerFlag) and (clientAddress[0]
                    ↪ not in self.peersList)):
                    # update peersList, set newPeers flag
                    self.peersList.append(clientAddress[0])
                    print("Appended ", clientAddress[0], " to
                        ↪ peersList")
                    self.newPeers = True

                    # delay a tiny bit (200mS)
                    time.sleep(0.2)

                    # send peersList to new peer! Welcome to the

```

```

        ↪ fun!
        peersListStr = ' '.join(self.peersList)
        #for peer in self.peersList:
        MasterS.sendto(peersListStr.encode(),
            ↪ clientAddress)

    else:
        print("Sorry")
        print(rxPeerFlag + 'rx')
        print(peerFlag + 'df')

    except Exception as ex:
        print("Master While Exception: %s" %ex)
        sys.exit(1)

except KeyboardInterrupt:
    print("Keyboard Interrupt!")
    exit(1)

except Exception as ex:
    print("Master Exception: %s" % ex)
    sys.exit(1)

def nextMaster(self):    # does stuff to prepare to carry the torch
    a = 0

def broadcastHandler(self):
    try:    # Try sending broadcast to scan for other peers on
        ↪ network
        BroadcastS=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        BroadcastS.setsockopt(socket.SOL_SOCKET, socket.
            ↪ SO_BROADCAST, 1)
        BroadcastS.setsockopt(socket.SOL_SOCKET, socket.
            ↪ SO_REUSEADDR, 1)
        BroadcastS.settimeout(TimeoutDelay)
        broadcastMessage = peerFlag + self.addr
        #BroadcastS.sendto(broadcastMessage, ('255.255.255.255',
            ↪ 12000))
        BroadcastS.sendto(broadcastMessage.encode(), ('<broadcast>',

```



```

    ↪ 42069))
print("Broadcast Sent!")
print("Wait", str(TimeoutDelay) ,"seconds for response:")
#self.peerScanRunning = True
#while self.peerScanRunning:
try:
    peerListMessage, masterAddressTuple = BroadcastS.
        ↪ recvfrom(2048)
    self.masterIPAddress = masterAddressTuple[0]
    self.masterPort = masterAddressTuple[1]

    # split message and store peers in peersList.
    self.peersList.append(peerListMessage.decode().split("
        ↪ "))
    self.isMaster = False
    if len(self.peersList) != 1:
        print("Closing the Broadcast Socket")
        BroadcastS.close
        sys.exit(0)
    else:
        self.isNextMaster = True
        print("You are the next Master! Be prepared!")

except KeyboardInterrupt:
    print("Keyboard Interrupt!")
    exit(1)
except socket.timeout:      # will timeout in 10 seconds!
    print("You are the Master on this Network! Congrats!")
    self.isMaster = True
    self.peersList.append(self.addr)      #append master
        ↪ address to first slot in peersList
    #start master thread!
    #master_thread = threading.Thread(target=self.Master,
        ↪ args=())
    #master_thread.start()
    #print("Master Thread Started!")
except Exception as ex:
    print("Broadcast Handler Internal Exception: %s" % ex)
    sys.exit(1)

```

```

except KeyboardInterrupt:
    print("Keyboard Interrupt!")
    exit(1)
except Exception as ex:
    print("Broadcast Handler Exception: %s" % ex)
    exit(1)
# do something, not implemented yet

def clientTask(self, targetServerAddr, targetServerPort):
    try:
        clientSocket=socket.socket(socket.AF_INET, socket.
            ↪ SOCK_STREAM)

        try:
            clientSocket.connect((targetServerAddr,
                ↪ targetServerPort))
        except Exception as e:
            print("Client Thread connect error: %s" %e)
            exit(1)

        while(True):
            if (self.MessageQueued):
                print("Client Thread has Message Queued")
                #sentence=input("Input lowercase sentence: ")
                try:
                    clientSocket.send(self.Message.encode())
                    modifiedMessage=clientSocket.recv(1024) #
                        ↪ blocking
                    print("From server: %s" % modifiedMessage.
                        ↪ decode())
                    self.MessagesSent = self.MessagesSent+1
                    print("incremented MessagesSent value")
                except Exception as e:
                    print("internal client Thread error: %s" %e)
                while (self.MessageQueued):
                    #wait till all clients send message before
                        ↪ exiting
                    a = 0

```

```

        print("Thread done waiting for all clients to send
        ↪ message")
        #clientSocket.close()

except KeyboardInterrupt:
    print("Keyboard interrupt")
    exit(1)

def clientSideManager(self):
    if(not self.isMaster):
        try:
            masterAddr = self.peersList[0][0]
            print("Master Address is: %s" %masterAddr)
            masterClient_thread = threading.Thread(target=self.
            ↪ clientTask, args=(masterAddr, serverPort))
            self.clientThreads.append(masterClient_thread)
            masterClient_thread.start()
            self.numClients = self.numClients + 1
            self.peersConnectedList.append(True)
        except Exception as ex:
            print("clientSideManager MasterClient Exception: %s" %
            ↪ ex)
            sys.exit(1)

while(self.peerRunning):
    if(self.newPeers):
        correctNumClients = len(self.peersList)-1
        self.indexInPeersList = self.peersList.index(self.addr)
        #startIndex = self.numClients

        for i in range(1, len(self.peersList)):
            if (i != self.indexInPeersList) and (
            ↪ correctNumClients >= len(self.
            ↪ peersConnectedList)):
                self.peersConnectedList.append(True)
                print("Starting new Client Thread: Addr = %s" %
                ↪ self.peersList[i])
                newClientThread = threading.Thread(target=self.
                ↪ clientTask, args=(self.peersList[i],
                ↪ serverPort))

```

```

        self.clientThreads.append(newClientThread)
        newClientThread.start()
        #hi = 2

    self.newPeers = False

    InputMessage = input("Input message to send to peers: ")
    self.Message = InputMessage
    self.MessagesSent = 0
    self.MessageQueued = True
    print("Manager set MessageQueued to True")

    print("Manager waiting for all client threads to send
    ↪ message")
    print("self.MessagesSent = ", self.MessagesSent)
    print("length of self.clientThreads = ", len(self.
    ↪ clientThreads))
    while(self.MessagesSent < len(self.clientThreads)):
        #wait here doing nothing till all clients Threads send
        ↪ message
        a = 0
    print("Manager done waiting for client threads to send
    ↪ message!")
    self.MessageQueued = False
    self.Message = ''
    self.MessagesSent = 0

    #for i in range(0, len(self.clientThreads)-1)

    #sentence=input("Input lowercase sentence: ")
    #clientSocket.send(sentence.encode())
    #modifiedSentence=clientSocket.recv(1024) #blocking
    #print("From server: %s" % modifiedSentence.decode())
    #clientSocket.close()

    #clientSocket=socket.socket(socket.AF_INET, socket.SOCK_STREAM)

def serverSide(self):
    try:

```

```

serverSocket=socket.socket(socket.AF_INET, socket.
    ↪ SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(maxNumPeers)
print("The server is ready to receive " + str(maxNumPeers)
    ↪ +" peers.")
while 1:
    connectionSocket, addr = serverSocket.accept()
    sentence=connectionSocket.recv(1024).decode()
    capitalizedSentence=sentence.upper()
    print("Received Message from: %s" %addr[0])
    print("Message: %s" %sentence)
    connectionSocket.send(capitalizedSentence.encode())
    #connectionSocket.close()

except KeyboardInterrupt:
    print("Keyboard interrupt")
    exit(1)
except Exception as ex:
    print("Server Exception: %s" %ex)

def __init__(self):
    self.hostname = socket.gethostname()
    self.addr = socket.gethostbyname(self.hostname)
    print("Hostname is: ", self.hostname)
    print("IP address is: ", self.addr)
    #self.serverRunning = True

broadcastHandler_thread = threading.Thread(target=self.
    ↪ broadcastHandler, args=())
broadcastHandler_thread.start()
print("Started the Broadcast Handler Thread, it will scan for
    ↪ other peers to connect to!")

# wait for broadcastHandler thread to determine role
broadcastHandler_thread.join()

if self.isMaster:
    master_thread = threading.Thread(target=self.Master, args
    ↪ =())
    master_thread.start()

```

```

        print("Master Thread Started!")
    if self.isNextMaster:
        nextMaster_thread = threading.Thread(target=self.nextMaster
        ↪ , args=())
        nextMaster_thread.start()
        print("Next Master thread started!")

    self.peerRunning = True

    server_thread = threading.Thread(target=self.serverSide, args
    ↪ =())
    server_thread.start()
    print("Started the Server thread")

    client_thread = threading.Thread(target=self.clientSideManager,
    ↪ args=())
    client_thread.start()
    print("Started the Client thread")

TCPpeer()

```

Listing 9: UDPServer.c

```

#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define DEFAULT_PORT 12000
#define DEFAULT_IP "127.0.0.1"
#define MAX_MESSAGE_LENGTH 2048
#define CR printf("\n")

int serverLoop(int serverPort)
{
    int socketDescriptor;
    unsigned long int i;

```

```

char receivedClientMessage[MAX_MESSAGE_LENGTH];
struct sockaddr_in server_addr, client_addr;
unsigned int clientStructLength=sizeof(client_addr);

/*clear string*/
for (i=0; i<MAX_MESSAGE_LENGTH; i++)
{
    receivedClientMessage[i]='\0';
}

/*open socket*/
socketDescriptor=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (socketDescriptor < 0)
{
    printf("Error: could not create socket."); CR;
    return(1);
}

/*set socket library settings*/
server_addr.sin_family=AF_INET;
server_addr.sin_port=serverPort;
server_addr.sin_addr.s_addr=inet_addr(DEFAULT_IP);

/*bind*/
if (bind(socketDescriptor, (struct sockaddr*)&server_addr, sizeof(
    ↪ server_addr)))
{
    printf("Error: could not bind to port."); CR;
    return(1);
}

printf("Now listening for messages..."); CR;
while(1)
{
    /*listen*/
    if (recvfrom(socketDescriptor, receivedClientMessage, sizeof(
        ↪ receivedClientMessage), 0, \
        (struct sockaddr*)&client_addr, &clientStructLength) < 0)
    {
        printf("Error: could not receive data from client."); CR;
        return(1);
    }
}

```

```

    }

    /*have message, now send back uppercase version*/
    printf("[%s:%i] %s", inet_ntoa(client_addr.sin_addr), ntohs(
        ↪ client_addr.sin_port), receivedClientMessage);
    for(i=0; i<MAX_MESSAGE_LENGTH; i++)
    {
        if (receivedClientMessage[i] >= 'a' &&
            ↪ receivedClientMessage[i] <= 'z')
        {
            receivedClientMessage[i] -= 'a' - 'A';
        }
    }

    /*send uppercase string to client*/
    if (sendto(socketDescriptor, receivedClientMessage, sizeof(
        ↪ receivedClientMessage), 0, \
        (struct sockaddr*)&client_addr, clientStructLength) < 0)
    {
        printf("Error: could not send data to client."); CR;
        return(1);
    }
}

int main(int argc, char * argv[])
{
    unsigned int serverPort;

    if (argc == 2)
    {
        /*convert input string to int*/
        if (sscanf(argv[1], "%u", &serverPort) != 1)
        {
            printf("Error: Input value not an integer. Quitting."); CR;
            return 1;
        }
    }
    else
    {
        printf("Usage: ./UDPServer.c server_port"); CR;
    }
}

```



```

        printf("Using default server port value of %d", DEFAULT_PORT);
        ↪ CR;
        serverPort=DEFAULT_PORT;
    }

    return(serverLoop(serverPort));
}

```

Listing 10: UDPClientTimer.c

```

/*https://www.educative.io/edpresso/how-to-implement-udp-sockets-in-c*/
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <time.h>

#define CR printf("\n")
#define MAX_MESSAGE_SIZE 2048

int UDPClientTimer(char * serverAddress, int serverPort)
{
    unsigned long int i;
    int socketDescriptor;
    struct sockaddr_in server_addr;
    unsigned int serverStructLength=sizeof(server_addr);
    char messageFromServer[MAX_MESSAGE_SIZE];
    char messageToSend[MAX_MESSAGE_SIZE];
    clock_t startTime;
    clock_t totalTime;

    /*clear strings*/
    for(i=0; i<MAX_MESSAGE_SIZE; i++)
    {
        messageToSend[i]='\0';
        messageFromServer[i]='\0';
    }

    /*open socket*/
    socketDescriptor=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (socketDescriptor < 0)
    {
        printf("Error: could not create socket."); CR;
    }
}

```

```

    return(1);
}

/*set server attributes*/
server_addr.sin_family=AF_INET;
server_addr.sin_port=serverPort;
server_addr.sin_addr.s_addr=inet_addr(serverAddress);

while (1)
{
    /*get message*/
    printf("Enter a message: ");
    fgets(messageToSend, MAX_MESSAGE_SIZE, stdin);

    /*start timer*/
    startTime=clock();

    /*send to server*/
    if (sendto(socketDescriptor, messageToSend, sizeof(
        ↪ messageToSend), 0, \
        (struct sockaddr*)&server_addr, serverStructLength) < 0)
    {
        printf("Error: could not send message."); CR;
        return(1);
    }

    /*listen for response*/
    if (recvfrom(socketDescriptor, messageFromServer, sizeof(
        ↪ messageFromServer), 0, \
        (struct sockaddr*)&server_addr, &serverStructLength) < 0)
    {
        printf("Error: could not receive message."); CR;
        return(1);
    }

    /*end timer and calculate total time*/
    totalTime=(long int)(clock() - startTime);

    printf("[Response in %ld clock cycles] %s", totalTime,
        ↪ messageFromServer);
}

```

```

}

int main(int argc, char * argv[])
{
    unsigned int serverPort;

    if (argc == 3)
    {
        /*convert port number to int*/
        if (sscanf(argv[2], "%u", &serverPort) != 1)
        {
            printf("Error: input value invalid."); CR;
            return(1);
        }

        return(UDPClietTimer(argv[1], serverPort));
    }
    else
    {
        printf("Usage: ./UDPCliet server_address server_port"); CR;
        return(1);
    }
}

```

Listing 11: MiddleGround.py

```

#!/usr/bin/env python
import shutil
import os, sys

#Constant Definitions
CWD = os.getcwd()
source = CWD + '/MDrive'
destination = CWD + '/Downloadables/mdrive.zip'

#Function Definitions
def make_zip(source, destination):
    base = os.path.basename(destination)
    name = base.split('.')[0]
    format = base.split('.')[1]
    archive_from = os.path.dirname(source)
    archive_to = os.path.basename(source.strip(os.sep))

```

```

    print(source, destination, archive_from, archive_to)
    shutil.make_archive(name, format, archive_from, archive_to)
    shutil.move('%s.%s'%(name,format), destination)

#Beginning of code

make_zip(source, destination)

```

Listing 12: index.html

```

<!DOCTYPE html>
<html>

<head>
<title>MDRIVE ACCESS</title>
<meta name="viewport" content="width=device-width, initial-scale=1">
<style>

.buttoncontainer{
    text-align: center;
    padding-top: 10px;
    padding-bottom: 10px;

}

h2{text-align: center;}

body {font-family: Arial, Helvetica, sans-serif;}

/* Full-width input fields */
input[type=text], input[type=password] {
    width: 100%;
    padding: 12px 20px;
    margin: 8px 0;
    display: inline-block;
    border: 1px solid #ccc;
    box-sizing: border-box;
}

```

```

/* Set a style for all buttons */
button {
    background-color: #3377FF;
    color: white;
    padding: 14px 20px;
    margin: 8px 0;
    border: none;
    cursor: pointer;
    width: 100%;
}

button:hover {
    opacity: 0.8;
}

/* Extra styles for the cancel button */
.cancelbtn {
    width: auto;
    padding: 10px 18px;
    background-color: #f44336;
}

/* Center the image and position the close button */
.imgcontainer {
    text-align: center;
    margin: 24px 0 12px 0;
    position: relative;
}

img.avatar {
    width: 40%;
}

.container {
    padding: 16px;
}

span.psw {
    float: right;
    padding-top: 16px;
}

```

```

/* The Modal (background) */
.modal {
  display: none; /* Hidden by default */
  position: fixed; /* Stay in place */
  z-index: 1; /* Sit on top */
  left: 0;
  top: 0;
  width: 100%; /* Full width */
  height: 100%; /* Full height */
  overflow: auto; /* Enable scroll if needed */
  background-color: rgb(0,0,0); /* Fallback color */
  background-color: rgba(0,0,0,0.4); /* Black w/ opacity */
  padding-top: 60px;
}

/* Modal Content/Box */
.modal-content {
  background-color: #fefefe;
  margin: 5% auto 15% auto; /* 5% from the top, 15% from the bottom and
    ↳ centered */
  border: 1px solid #888;
  width: 80%; /* Could be more or less, depending on screen size */
}

/* The Close Button (x) */
.close {
  position: absolute;
  right: 25px;
  top: 0;
  color: #000;
  font-size: 35px;
  font-weight: bold;
}

.close:hover,
.close:focus {
  color: red;
  cursor: pointer;
}

```

```

/* Add Zoom Animation */
.animate {
  -webkit-animation: animatezoom 0.6s;
  animation: animatezoom 0.6s
}

@-webkit-keyframes animatezoom {
  from {-webkit-transform: scale(0)}
  to {-webkit-transform: scale(1)}
}

@keyframes animatezoom {
  from {transform: scale(0)}
  to {transform: scale(1)}
}

/* Change styles for span and cancel button on extra small screens */
@media screen and (max-width: 300px) {
  span.psw {
    display: block;
    float: none;
  }
  .cancelbtn {
    width: 100%;
  }
}
</style>
</head>
<body>

<h2>Sign in with your UMASSD Credentials</h2>

<div class="buttoncontainer">
  <button onclick="document.getElementById('id01').style.display='
    ↪ block'" style="width:auto;">Login</button>
</div>
<div id="id01" class="modal">

  <form class="modal-content animate" action="PostLogin.php" method="
    ↪ post">
    <div class="imgcontainer">

```

```

    <span onclick="document.getElementById('id01').style.display='
        ↪ none' " class="close" title="Close Modal">&times;</span>
    
</div>

<div class="container">
    <label for="uname"><b>Username</b></label>
    <input type="text" placeholder="Enter Username" name="uname"
        ↪ required>

    <label for="psw"><b>Password</b></label>
    <input type="password" placeholder="Enter Password" name="psw"
        ↪ required>

    <button type="submit">Login</button>

</div>

<div class="container" style="background-color:#f1f1f1">
    <button type="button" onclick="document.getElementById('id01').
        ↪ style.display='none' " class="cancelbtn">Cancel</button>

</div>
</form>
</div>

<script>
// Get the modal
var modal = document.getElementById('id01');

// When the user clicks anywhere outside of the modal, close it
window.onclick = function(event) {
    if (event.target == modal) {
        modal.style.display = "none";
    }
}
</script>

</body>
</html>

```


Listing 13: PostLogin.php

```
<html>
<body>
  <head>
    <title>
      MDRIVE ACCESS
    </title>
    <style>
      h2{text-align: center;}

      .container{
        text-align: center;
        padding-top: 10px;
        padding-bottom: 10px;
      }

      p{text-align: center;}

      body{font-family: Arial, Helvetica, sans-serif;}

      a{text-align: center;}

      button {
        background-color: #3377FF;
        color: white;
        padding: 14px 20px;
        margin: 8px 0;
        border: none;
        cursor: pointer;
        width: 50%;
      }
      button:hover {
        opacity: 0.8;
      }

    </style>

  </head>
  <h2>Click the RE-ZIP button and wait until you are notified that it
    ➞ is ready to download! </h2>
```

```

<form method="post">
  <div class="container">
    <button type="submit" value="RE-ZIP" name="RE-ZIP">RE-ZIP the
      ↪ MDrive </button>
    </div>
  </form>

  <p>
    <br>
    <a href="Downloadables/mdrive.zip" title="This is a link to the
      ↪ zipped m-drive">Download the MDrive zip folder</a>
    <br>
  </p>
  <br>

</body>
</html>

<?php
  if(isset($_POST['RE-ZIP']))
  {

    //echo"Hold on while we zip!<br>";
    $mDriveZip = shell_exec("sudo python3 /var/www/html/
      ↪ MiddleGround.py");
    //echo"<pre>$mDriveZip</pre>";
    echo"<br>";
    echo"<p>The MDrive Folder has successfully been zipped! You may
      ↪ now click the MDrive download link!</p>";
    echo"<br>";

  }

?>

```