Brandon Vo

Homework 3

Part 1: netcat TCP chat

We will begin with netcat—a TCP/UDP utility which comes preinstalled on most UNIX systems. As quoted from the man page (open a terminal and type man netcat), "The nc (or netcat) utility is used for just about anything under the sun involving TCP or UDP. It can open TCP connections, send UDP packets, listen on arbitrary TCP and UDP ports, do port scanning, and deal with both IPv4 and IPv6."

We will use netcat to create a two-way chat over a network. The purpose of this section is to see an example of how easy it can be to implement a TCP chat over a network.

Let R2 be the server and KALI the client. Type the following commands:

R2 (Server):

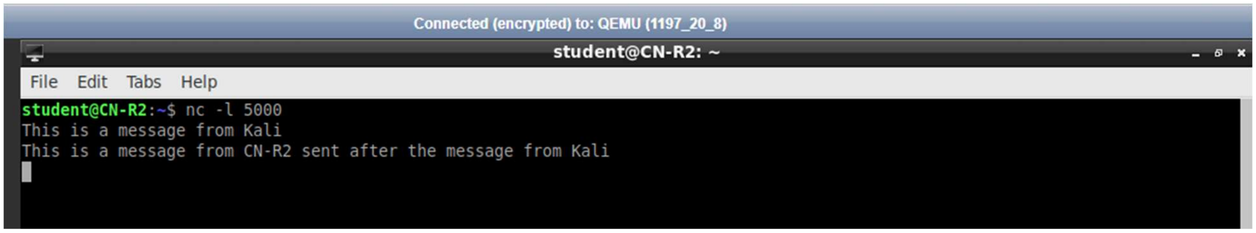
nc -l <port> // use any port that is not reserved (e.g., 5000)

KALI (Client):

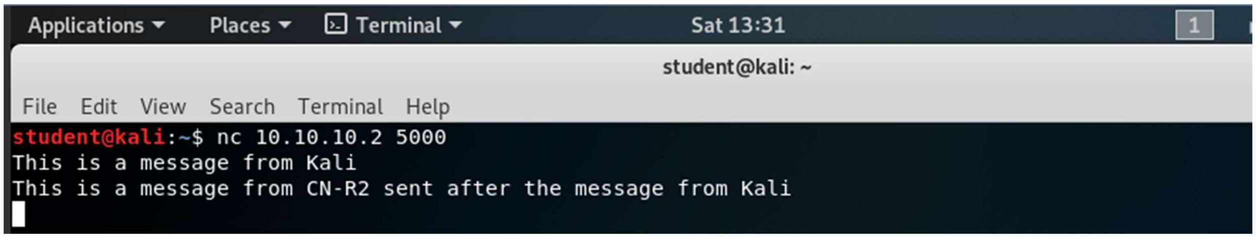
nc <server hostname or IP address> <port>

On R2, type a message in the terminal window and press enter. It should be viewable in R3's terminal. Type a message in R3's terminal window and press enter. It should be viewable in R2's terminal. Type a few messages on each side to make a conversation.

Press CTRL-C in both R2 and KALI to close the connection.



```
student@CN-R2:~$ nc -l 5000
This is a message from Kali
This is a message from CN-R2 sent after the message from Kali
```



```
student@kali:~$ nc 10.10.10.2 5000
This is a message from Kali
This is a message from CN-R2 sent after the message from Kali
```

The only problem with this chat is that it is hard to tell who sent each message. Our goal is for the chat window to behave like this:

R2 types: “Hi, my name is R2”

Output in R2 and KALI terminal: “R2: Hi, my name is R2”

KALI types: “Hi R2, my name is KALI. Nice to meet you.”

Output in R2 and KALI terminal: “KALI: Hi R2, my name is KALI. Nice to meet you.

Add a username (“R2” or “KALI”) so that each message can be identified by its sender.

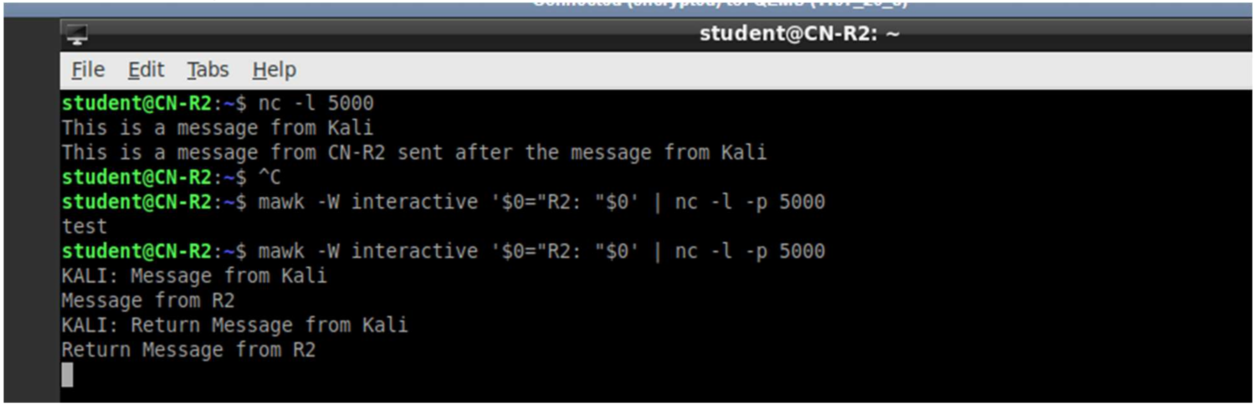
R2 (Server):

mawk -W interactive '$0="R2: "$0' | nc -l -p <port_number>

KALI (Client):

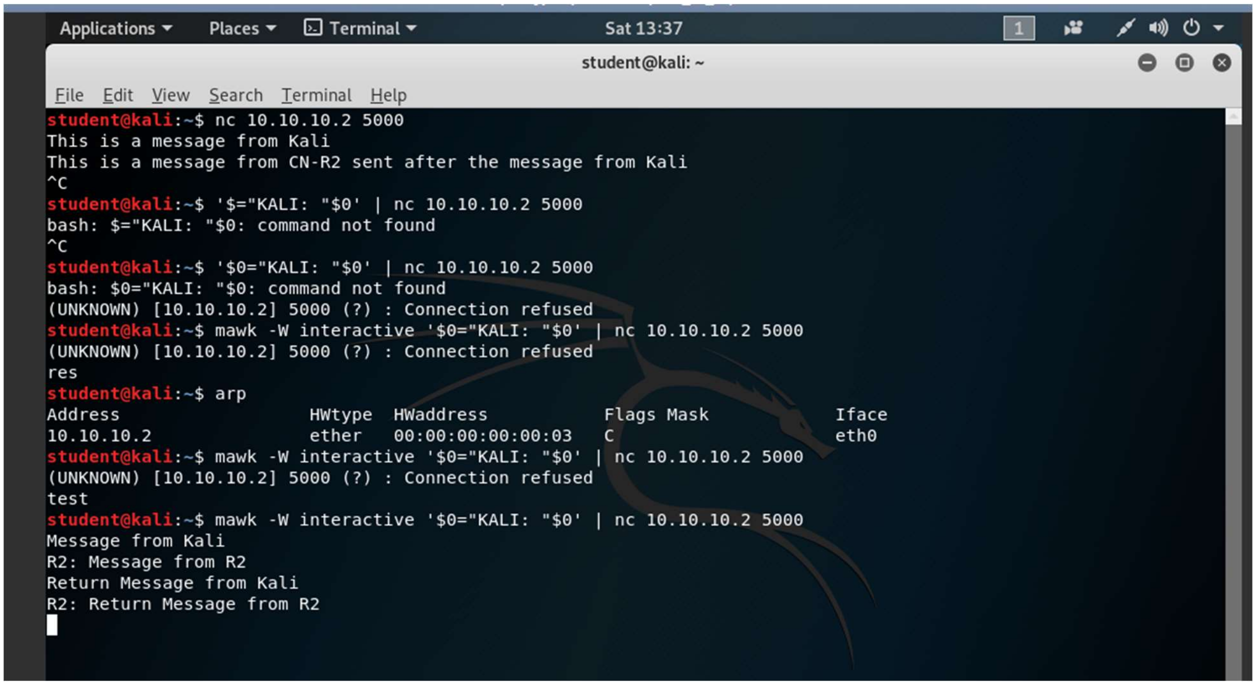
mawk -W interactive '$0="KALI: "$0' | nc <server IP> <port_number>

Create a short conversation between R2 and KALI and take a screenshot of each terminal window showing the chat.



```
student@CN-R2:~$ nc -l 5000
This is a message from Kali
This is a message from CN-R2 sent after the message from Kali
student@CN-R2:~$ ^C
student@CN-R2:~$ mawk -W interactive '$0="R2: "$0' | nc -l -p 5000
test
student@CN-R2:~$ mawk -W interactive '$0="R2: "$0' | nc -l -p 5000
KALI: Message from Kali
Message from R2
KALI: Return Message from Kali
Return Message from R2
```

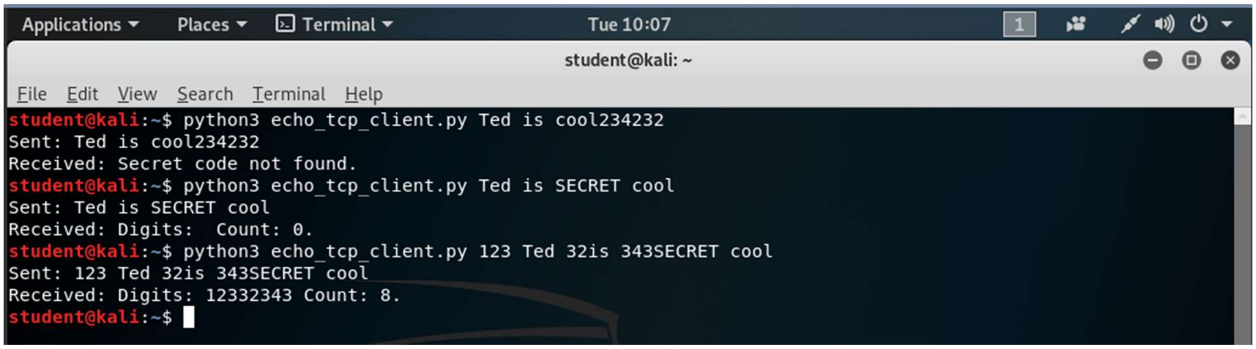
Brandon Vo



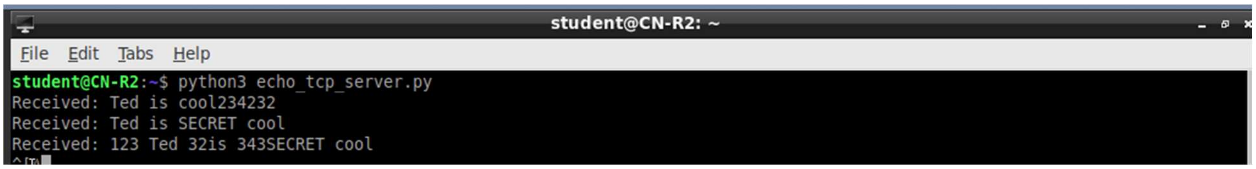
```
student@kali:~$ nc 10.10.10.2 5000
This is a message from Kali
This is a message from CN-R2 sent after the message from Kali
^C
student@kali:~$ '$="KALI: "$0' | nc 10.10.10.2 5000
bash: $="KALI: "$0: command not found
^C
student@kali:~$ '$0="KALI: "$0' | nc 10.10.10.2 5000
bash: $0="KALI: "$0: command not found
(UNKNOWN) [10.10.10.2] 5000 (?) : Connection refused
student@kali:~$ mawk -W interactive '$0="KALI: "$0' | nc 10.10.10.2 5000
(UNKNOWN) [10.10.10.2] 5000 (?) : Connection refused
res
student@kali:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.10.10.2               ether   00:00:00:00:00:03   C                     eth0
student@kali:~$ mawk -W interactive '$0="KALI: "$0' | nc 10.10.10.2 5000
(UNKNOWN) [10.10.10.2] 5000 (?) : Connection refused
test
student@kali:~$ mawk -W interactive '$0="KALI: "$0' | nc 10.10.10.2 5000
Message from Kali
R2: Message from R2
Return Message from Kali
R2: Return Message from R2
```

Part 2: Client-server with secret code

You should write two files for this part: **echo_tcp_server.py** (on R2) and **echo_tcp_client.py** (on KALI). The client should send a string to the server, and the server should receive it. If the string contains the secret code "SECRET", the server should return all the digits in the string as well as the total number of digits. If the string does not contain the secret code, the server should respond with the message, "Secret code not found." The client should receive the output. The output format and behavior should match the example below.



```
student@kali:~$ python3 echo_tcp_client.py Ted is cool234232
Sent: Ted is cool234232
Received: Secret code not found.
student@kali:~$ python3 echo_tcp_client.py Ted is SECRET cool
Sent: Ted is SECRET cool
Received: Digits:  Count: 0.
student@kali:~$ python3 echo_tcp_client.py 123 Ted 32is 343SECRET cool
Sent: 123 Ted 32is 343SECRET cool
Received: Digits: 12332343 Count: 8.
student@kali:~$
```

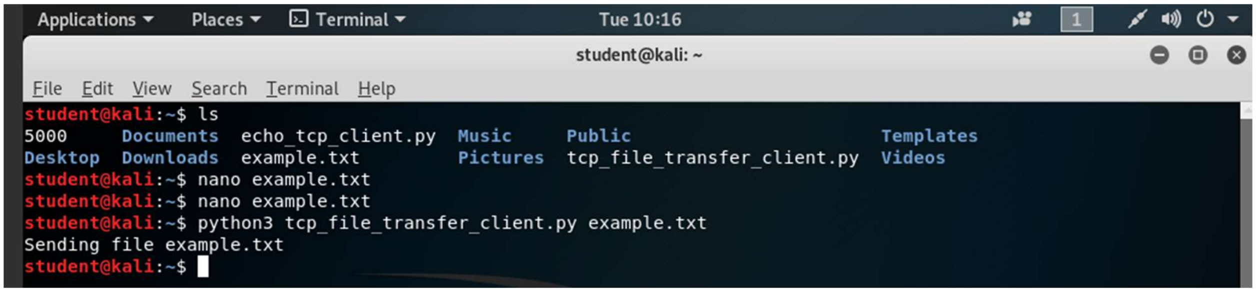


```
student@CN-R2:~$ python3 echo_tcp_server.py
Received: Ted is cool234232
Received: Ted is SECRET cool
Received: 123 Ted 32is 343SECRET cool
^[[A
```

Brandon Vo

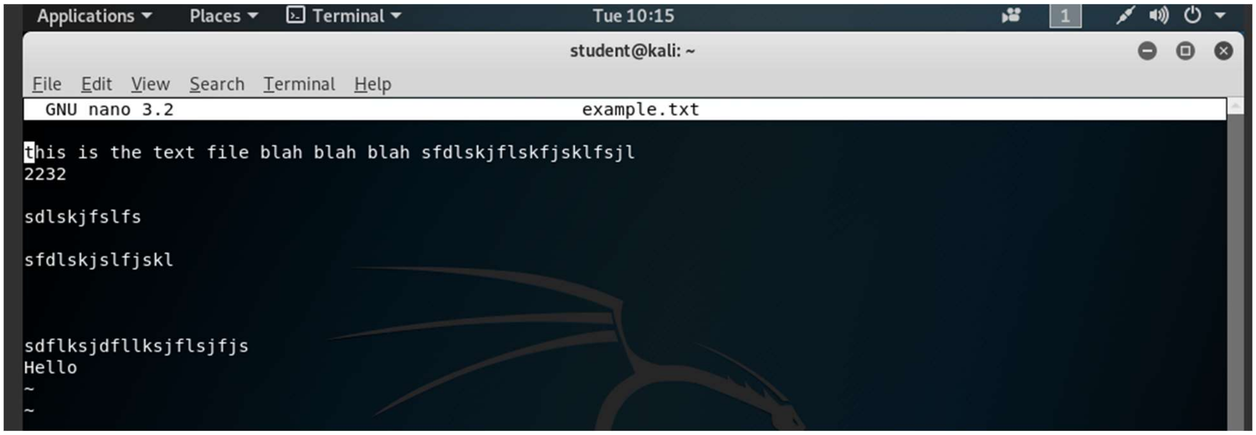
Part 3: Client-server with file transfer

You should write two files for this part: **tcp_file_transfer_server.py** (on R2) and **tcp_file_transfer_client.py** (on KALI). The client should create a file (any text file with some content will suffice) and send the data in this file to the server. The server should receive the data and write it to a file. The resulting file should be exactly the same as the file on the client side. Once the transfer is complete, the connection should be closed. The output format and behavior should match the example below.



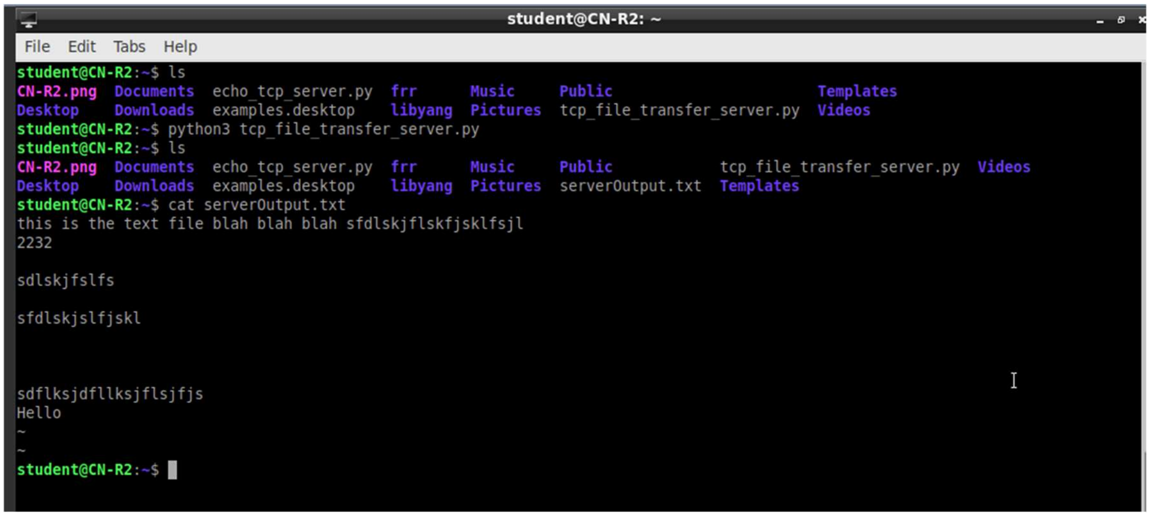
```
student@kali:~$ ls
5000     Documents  echo_tcp_client.py  Music     Public                      Templates
Desktop  Downloads  example.txt         Pictures  tcp_file_transfer_client.py  Videos
student@kali:~$ nano example.txt
student@kali:~$ nano example.txt
student@kali:~$ python3 tcp_file_transfer_client.py example.txt
Sending file example.txt
student@kali:~$ 
```

Contents of example.txt



```
  GNU nano 3.2                              example.txt

this is the text file blah blah blah sfdlskjflskfjsklfsjl
2232

sdlskjfslfs

sfdlskjslfjskl



sdflksjdfllksjflsjfjs
Hello
~
~
```

Output is written into the text file serverOutput.txt



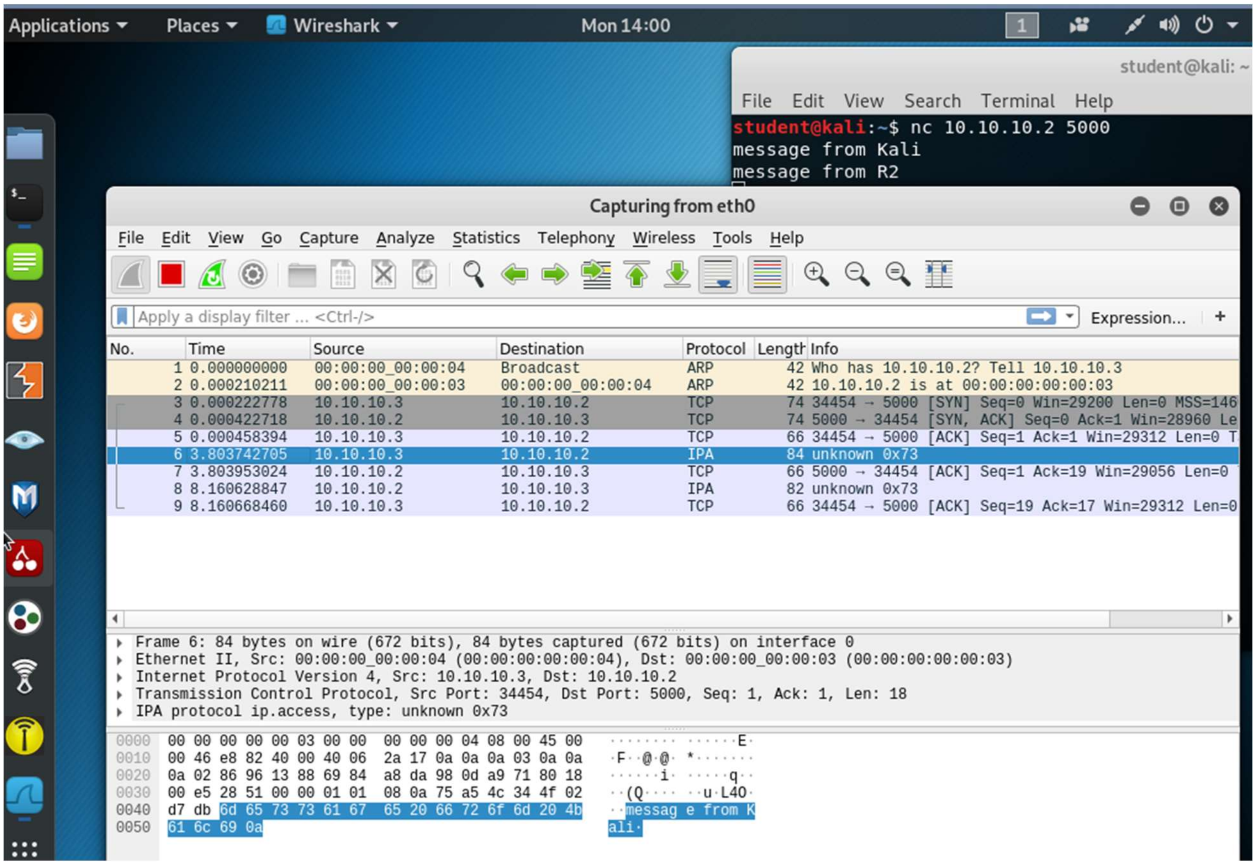
```
student@CN-R2:~$ ls
CN-R2.png  Documents  echo_tcp_server.py  frr      Music     Public                      Templates
Desktop    Downloads  examples.desktop    libyang  Pictures  tcp_file_transfer_server.py  Videos
student@CN-R2:~$ python3 tcp_file_transfer_server.py
student@CN-R2:~$ ls
CN-R2.png  Documents  echo_tcp_server.py  frr      Music     Public             tcp_file_transfer_server.py  Videos
Desktop    Downloads  examples.desktop    libyang  Pictures  serverOutput.txt  Templates
student@CN-R2:~$ cat serverOutput.txt
this is the text file blah blah blah sfdlskjflskfjsklfsjl
2232

sdlskjfslfs

sfdlskjslfjskl



sdflksjdfllksjflsjfjs
Hello
~
~
student@CN-R2:~$ 
```

Brandon Vo

Part 4: Questions

A) In netcat, you specified the port on which the server should listen but did not specify the port the server should use to send a message to the client. Which client port does your netcat server send to? Use Wireshark to answer the question and include a screenshot.

The client has a port of 34454.


Applications
Places
Wireshark
Mon 14:00
student@kali: ~
File Edit View Search Terminal Help
Capturing from eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
+

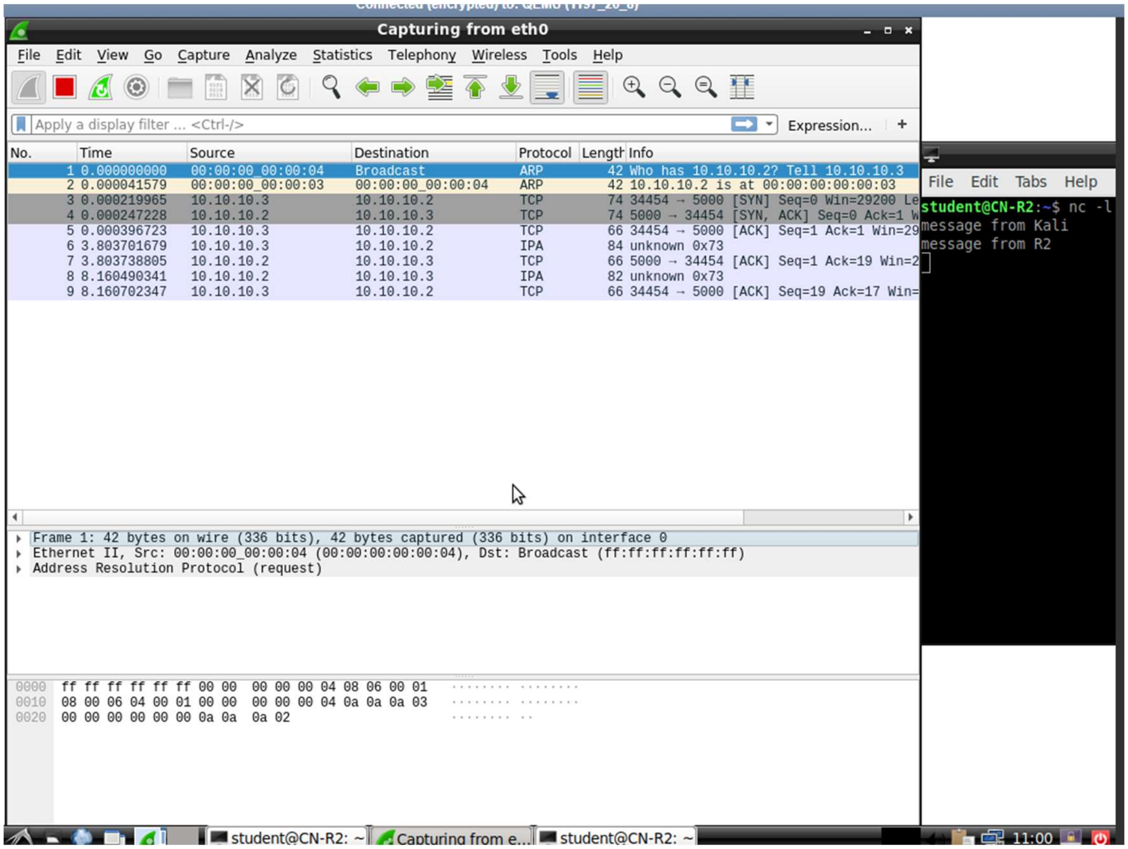
```
student@kali:~$ nc 10.10.10.2 5000
message from Kali
message from R2
```

```
No.  Time         Source             Destination        Protocol Length Info
   1 0.000000000  00:00:00_00:00:04  Broadcast          ARP      42 Who has 10.10.10.2? Tell 10.10.10.3
   2 0.000210211  00:00:00_00:00:03  00:00:00_00:00:04  ARP      42 10.10.10.2 is at 00:00:00:00:00:03
   3 0.000222778  10.10.10.3         10.10.10.2         TCP      74 34454 → 5000 [SYN] Seq=0 Win=29200 Len=0 MSS=146
   4 0.000422718  10.10.10.2         10.10.10.3         TCP      74 5000 → 34454 [SYN, ACK] Seq=0 Ack=1 Win=28960 Le
   5 0.000458394  10.10.10.3         10.10.10.2         TCP      66 34454 → 5000 [ACK] Seq=1 Ack=1 Win=29312 Len=0 T
   6 3.803742705  10.10.10.3         10.10.10.2         IPA      84 unknown 0x73
   7 3.803953024  10.10.10.2         10.10.10.3         TCP      66 5000 → 34454 [ACK] Seq=1 Ack=19 Win=29056 Len=0
   8 8.160628847  10.10.10.2         10.10.10.3         IPA      82 unknown 0x73
   9 8.160668460  10.10.10.3         10.10.10.2         TCP      66 34454 → 5000 [ACK] Seq=19 Ack=17 Win=29312 Len=0
```

```
Frame 6: 84 bytes on wire (672 bits), 84 bytes captured (672 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:04 (00:00:00:00:00:04), Dst: 00:00:00_00:00:03 (00:00:00:00:00:03)
Internet Protocol Version 4, Src: 10.10.10.3, Dst: 10.10.10.2
Transmission Control Protocol, Src Port: 34454, Dst Port: 5000, Seq: 1, Ack: 1, Len: 18
IPA protocol ip.access, type: unknown 0x73
```

```
0000  00 00 00 00 00 03 00 00  00 00 00 04 08 00 45 00   ········ ······E·
0010  00 46 e8 82 40 00 40 06  2a 17 0a 0a 0a 03 0a 0a   ·F··@·@· *·······
0020  0a 02 86 96 13 88 69 84  a8 da 98 0d a9 71 80 18   ······i· ·····q··
0030  00 e5 28 51 00 00 01 01  08 0a 75 a5 4c 34 4f 02   ··(Q···· ··u·L4O·
0040  d7 db 6d 65 73 73 61 67  65 20 66 72 6f 6d 20 4b   ··messag e from K
0050  61 6c 69 0a                                        ali·
```

Brandon Vo


Capturing from eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
+
File Edit Tabs Help
student@CN-R2: ~
Capturing from e...
student@CN-R2: ~
11:00

```
No.  Time         Source             Destination        Protocol Length Info
   1 0.000000000  00:00:00_00:00:04  Broadcast          ARP      42 Who has 10.10.10.2? Tell 10.10.10.3
   2 0.000041579  00:00:00_00:00:03  00:00:00_00:00:04  ARP      42 10.10.10.2 is at 00:00:00:00:00:03
   3 0.000219965  10.10.10.3         10.10.10.2         TCP      74 34454 → 5000 [SYN] Seq=0 Win=29200 Le
   4 0.000247228  10.10.10.2         10.10.10.3         TCP      74 5000 → 34454 [SYN, ACK] Seq=0 Ack=1 W
   5 0.000396723  10.10.10.3         10.10.10.2         TCP      66 34454 → 5000 [ACK] Seq=1 Ack=1 Win=29
   6 3.803701679  10.10.10.3         10.10.10.2         IPA      84 unknown 0x73
   7 3.803738805  10.10.10.2         10.10.10.3         TCP      66 5000 → 34454 [ACK] Seq=1 Ack=19 Win=2
   8 8.160490341  10.10.10.2         10.10.10.3         IPA      82 unknown 0x73
   9 8.160702347  10.10.10.3         10.10.10.2         TCP      66 34454 → 5000 [ACK] Seq=19 Ack=17 Win=
```

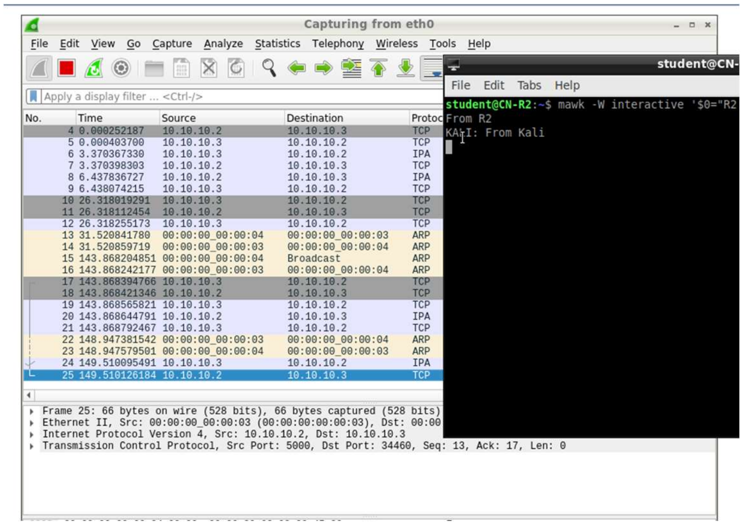
```
Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:04 (00:00:00:00:00:04), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
```

```
0000  ff ff ff ff ff ff 00 00  00 00 00 04 08 06 00 01   ........ ........
0010  08 00 06 04 00 01 00 00  00 00 00 04 0a 0a 0a 03   ........ ........
0020  00 00 00 00 00 00 0a 0a  0a 02                     ........ ..
```

```
student@CN-R2:~$ nc -l
message from Kali
message from R2
```

When using mawk -W along with netchat:

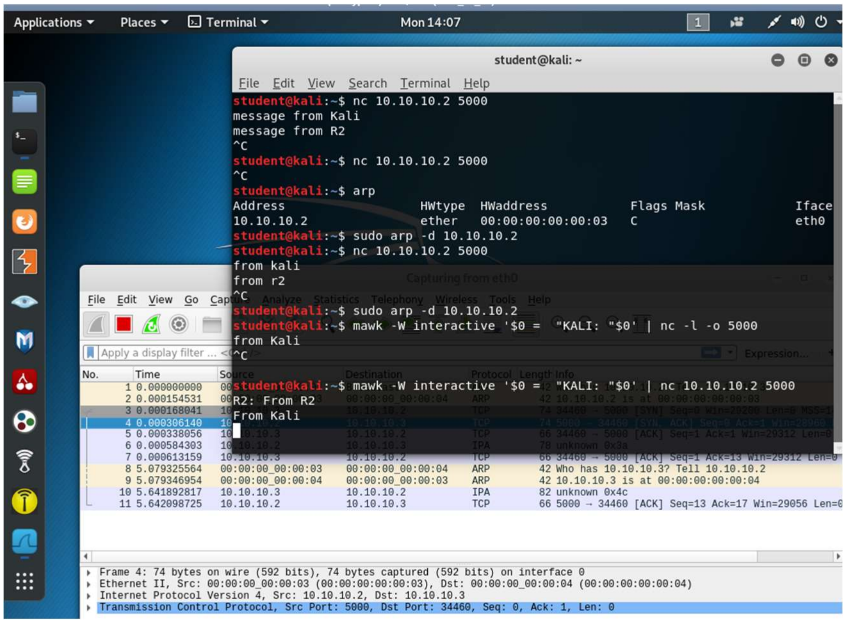
The port number becomes 34460. This is because the operating system will automatically assign the client a randomly available port number to use before establishing a connection with the server.



Capturing from eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
student@CN-
File Edit Tabs Help

```
No.  Time           Source              Destination         Protoc
  4 0.000252187    10.10.10.2          10.10.10.3          TCP
  5 0.000403700    10.10.10.3          10.10.10.2          TCP
  6 3.370367330    10.10.10.3          10.10.10.2          IPA
  7 3.370398303    10.10.10.2          10.10.10.3          TCP
  8 6.437836727    10.10.10.2          10.10.10.3          IPA
  9 6.438074215    10.10.10.3          10.10.10.2          TCP
 10 26.318019291   10.10.10.3          10.10.10.2          TCP
 11 26.318112454   10.10.10.2          10.10.10.3          TCP
 12 26.318255173   10.10.10.3          10.10.10.2          TCP
 13 31.520841780   00:00:00_00:00:04   00:00:00_00:00:03   ARP
 14 31.520859719   00:00:00_00:00:03   00:00:00_00:00:04   ARP
 15 143.868204851  00:00:00_00:00:04   Broadcast           ARP
 16 143.868242177  00:00:00_00:00:03   00:00:00_00:00:04   ARP
 17 143.868394766  10.10.10.3          10.10.10.2          TCP
 18 143.868421346  10.10.10.2          10.10.10.3          TCP
 19 143.868565821  10.10.10.3          10.10.10.2          TCP
 20 143.868644791  10.10.10.2          10.10.10.3          IPA
 21 143.868792467  10.10.10.3          10.10.10.2          TCP
 22 148.947381542  00:00:00_00:00:03   00:00:00_00:00:04   ARP
 23 148.947579501  00:00:00_00:00:04   00:00:00_00:00:03   ARP
 24 149.510095491  10.10.10.3          10.10.10.2          IPA
 25 149.510126184  10.10.10.2          10.10.10.3          TCP

▸ Frame 25: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
▸ Ethernet II, Src: 00:00:00_00:00:03 (00:00:00:00:00:03), Dst: 00:00
▸ Internet Protocol Version 4, Src: 10.10.10.2, Dst: 10.10.10.3
▸ Transmission Control Protocol, Src Port: 5000, Dst Port: 34460, Seq: 13, Ack: 17, Len: 0
```

```
student@CN-R2:~$ mawk -W interactive '$0="R2
From R2
KALI: From Kali
```



Applications
Places
Terminal
Mon 14:07
1
student@kali: ~
File Edit View Search Terminal Help
File Edit View Go Capture
Apply a display filter ...
Expression...

```
student@kali:~$ nc 10.10.10.2 5000
message from Kali
message from R2
^C
student@kali:~$ nc 10.10.10.2 5000
^C
student@kali:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.10.10.2               ether   00:00:00:00:00:03   C                     eth0
student@kali:~$ sudo arp -d 10.10.10.2
student@kali:~$ nc 10.10.10.2 5000
from kali
from r2
^C
student@kali:~$ sudo arp -d 10.10.10.2
student@kali:~$ mawk -W interactive '$0 =  "KALI: "$0' | nc -l -o 5000
from Kali
^C

student@kali:~$ mawk -W interactive '$0 = "KALI: "$0' | nc 10.10.10.2 5000
R2: From R2
From Kali
```

```
No.  Time          Source              Destination         Protocol Length Info
  1 0.000000000   00
  2 0.000154531   00                  00:00:00_00:00:04   ARP      42 10.10.10.2 is at 00:00:00:00:00:03
  3 0.000168041   10                  10.10.10.2          TCP      74 34460 → 5000 [SYN] Seq=0 Win=29200 Len=0 MSS=1
  4 0.000306140   10                  10.10.10.3          TCP      74 5000 → 34460 [SYN, ACK] Seq=0 Ack=1 Win=28960
  5 0.000338056   10.10.10.3          10.10.10.2          TCP      66 34460 → 5000 [ACK] Seq=1 Ack=1 Win=29312 Len=0
  6 0.000584303   10.10.10.2          10.10.10.3          IPA      78 unknown 0x3a
  7 0.000613159   10.10.10.3          10.10.10.2          TCP      66 34460 → 5000 [ACK] Seq=1 Ack=13 Win=29312 Len=0
  8 5.079325564   00:00:00_00:00:03   00:00:00_00:00:04   ARP      42 Who has 10.10.10.3? Tell 10.10.10.2
  9 5.079346954   00:00:00_00:00:04   00:00:00_00:00:03   ARP      42 10.10.10.3 is at 00:00:00:00:00:04
 10 5.641892817   10.10.10.3          10.10.10.2          IPA      82 unknown 0x4c
 11 5.642098725   10.10.10.2          10.10.10.3          TCP      66 5000 → 34460 [ACK] Seq=13 Ack=17 Win=29056 Len=0

▸ Frame 4: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:03 (00:00:00:00:00:03), Dst: 00:00:00_00:00:04 (00:00:00:00:00:04)
▸ Internet Protocol Version 4, Src: 10.10.10.2, Dst: 10.10.10.3
▸ Transmission Control Protocol, Src Port: 5000, Dst Port: 34460, Seq: 0, Ack: 1, Len: 0
```

B) Briefly explain your code from Part 2 and Part 3. In your explanation, focus not on the syntax but on the TCP communication establishment and flow.

The server creates a socket and with the specifications that it will run IPv4 and a TCP socket stream. The server's socket will also do a system call to bind to a specified, open port number then wait for any incoming requests to that port number.

The client will initiate its own socket with specifications to use IPv4 and TCP socket streams through AF_INET and SOCK_STREAM. The client connects to the specified server IP address and port number using connect(). The server will see the incoming request, and the client and server will initiate a 3-way handshake[1].

In the 3-way handshake, the client will send a request to initiate a connection to the server. The server will receive the request and send an acknowledgement packet to the client, informing the client that the server is able to establish a connection. The client will receive the acknowledgement response from the server and send its own acknowledgement response to the server. When the server receives the acknowledgement from the client, then the connection between the host and client has been created[2].

TCP handshake is completed after the server's socket's accept() call. The server will create a new socket dedicated towards responding to any requests from this specific client machine. The server and client are now ready to communicate to each other.

Part 2:

The client will encode its echo message into a byte-sized buffer before sending it to the server. The client will create a TCP packet and transmit the buffer through the client's socket. The server receives the message and decodes the message back into a string. The server will check the string to see if the word SECRET is found.

If SECRET is found, then it will check and record any the digits as well as the total amount of digits found in the string and then encode a message into a byte-sized buffer. The server will send this buffer in a TCP packet to the client.

If SECRET was not found, then the server returns a TCP packet that informs the client that the secret code was not found.

The client will receive the message from the server and print out the response on its own machine before closing its own socket.

The server closes its socket with the client, ending the TCP connection.

Part 3:

The client will read from the text file specified by the user and send the contents of the file through TCP packets. The server will receive the file contents received from the TCP packets. The server will check the buffer received from the TC packet to see if the contents can be written to file.

If the contents cannot be written to the file, then the server will just close the connection. If the contents can be written to file, then the server will open/create an empty text file with the text file name specified by the code. The contents from the TCP packet will be copied into the server's text file. Once completed, the server will close the file, close the socket with the client,

[1] Kurose & Ross, pg. 165

[2] https://www.techopedia.com/definition/10339/three-way-handshake

close the server's TCP socket, and then exit the python file. The client will, in turn, close its own connection socket as well.

C) What does the socket system call return?

If the socket encounters an error or fails in execution, it returns an error number to show the error and a value of -1. Otherwise, it will return the value of the file descriptor as a non-negative integer[3]. The file descriptor is the lowest-numbered file descriptor not currently open for the process[4].

D) What does the bind system call return? Who calls bind (client/server)?

The server calls bind.

TCP bind will return -1 if an error is encountered and sets the error number. If the bind request was successful, it returns 0[5].

The server has to call an explicit bind request in order to assign an address to the socket and refer it to the file descriptor[6]. The client, however, does not need to call bind because the kernel will automatically initiate an implicit bind request when the client connects to the server[7].

E)) Suppose you wanted to send an urgent message from a remote client to a server as fast as possible. Would you use UDP or TCP? Why? (Hint: compare RTTs.)

TCP has a higher RTT time due to the mandatory three-way handshake process needed. TCP requires a minimum of 2 RTT because it is connection oriented. The first RTT is overhead needed to set up the connection while the second RTT is needed to transmit the message itself.

UDP is a connectionless service, so it does not bother with any handshaking protocol. It will send as much data at any rate, meaning that UDP will transmit the message in only one RTT[8].

[3] https://pubs.opengroup.org/onlinepubs/009604599/functions/socket.html#:~:text=RETURN%20VALUE,set%20to%20indicate%20the%20error.

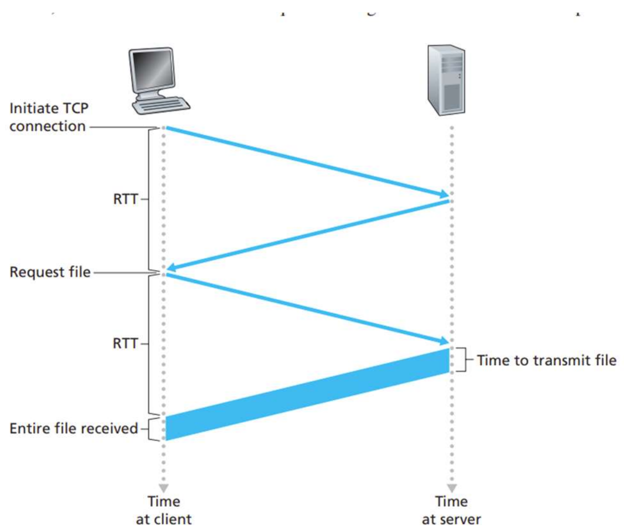
[4] https://man7.org/linux/man-pages/man2/socket.2.html

[5] https://man7.org/linux/man-pages/man2/bind.2.html#RETURN_VALUE

[6] https://linux.die.net/man/2/bind

[7] https://developer.ibm.com/technologies/systems/articles/au-tcpsystemcalls/#bind

[8] Kurose and Ross, pg. 102



F) What is Nagle’s algorithm? What problem does it aim to solve and how?

Nagle’s algorithm is a method to increase the efficiency of a TCP stream by attempting to transmit as many full-sized TCP packets as possible.

When there are multiple small packets under TCP’s maximum byte limit, the server will send out one small packet first. While waiting for the packet to be acknowledged, the server will accumulate the smaller packets and buffer them into one larger packet. When the server receives the acknowledgement from the first packet or if enough packets have accumulated to completely fill the maximum size for a TCP segment, then it will transmit the TCP segment containing the multiple smaller packets. This makes it so that only one packet will be outstanding and make sure that as much data is being sent out in as few RTTs as possible[9][10][11].

G) Explain one potential scenario in which delayed ACK could be problematic.

Delayed ACKs conflicts with Nagle’s algorithm extremely poorly. As Nagle mentioned, the 200 ms Ack delay was a solution used to reduce overhead. While every other protocol uses computed time as a guideline, Delayed ACKs is the only solution that uses a fixed time measurement[12].

Nagle’s algorithm depends on receiving an ACK to send data while Delayed ACKs want to send more full packets if it can. Combining both will lead to stalling as the client will delay its acknowledgement while the server will delay its packet transmission[13].

If a client using delayed ACKs sends a few packets to a server using Nagle’s algorithm, then it can risk leading to a deadlock between the two. If the client and/or server has issues transmitting packets or transmits multiple packets before reading for any incoming packets, then it could lead

[9] Tanenbaum, pg. 566

[10] https://news.ycombinator.com/item?id=9048947 (John Nagle)

[11] https://www.lifewire.com/nagle-algorithm-for-tcp-network-communication-817932

[12] https://news.ycombinator.com/item?id=9045125 (John Nagle)

[13] https://www.extrahop.com/company/blog/2016/tcp-nodelay-nagle-quickack-best-practices/#1

to a scenario where the server is waiting on an acknowledgement from the client before sending out more data. The client, however, may not be able to respond because it is too busy waiting to receive any data to piggyback an acknowledgement on.

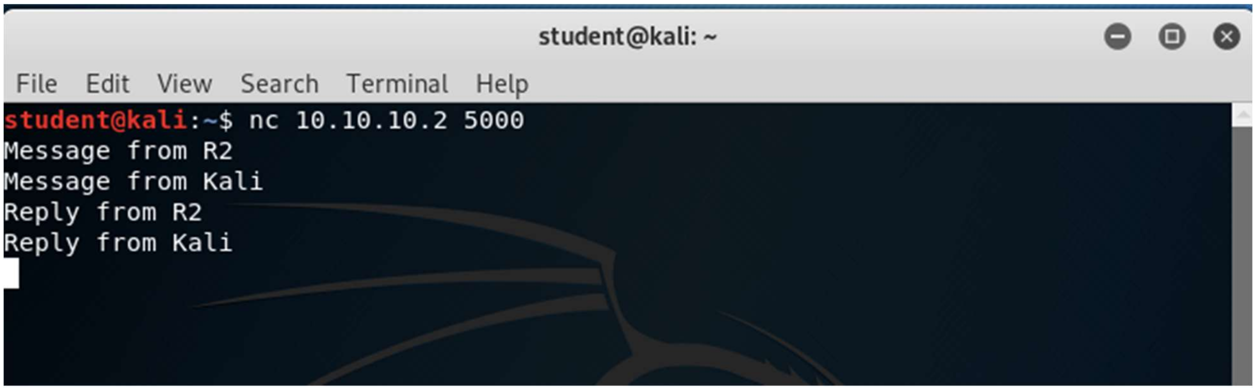
The client is unable to give the signal to the server to send more data, leading to both the client and server waiting for a response from each other[14]. This leads to a deadlock between the client and server. If there is a time out feature, then both systems will be bottlenecked and would be transmitting at the maximum amount of time allotted by the timeout system[15].

[14] https://serverfault.com/questions/834326/questions-about-nagle-vs-delayed-ack
[15] Tanenbaum pg. 567

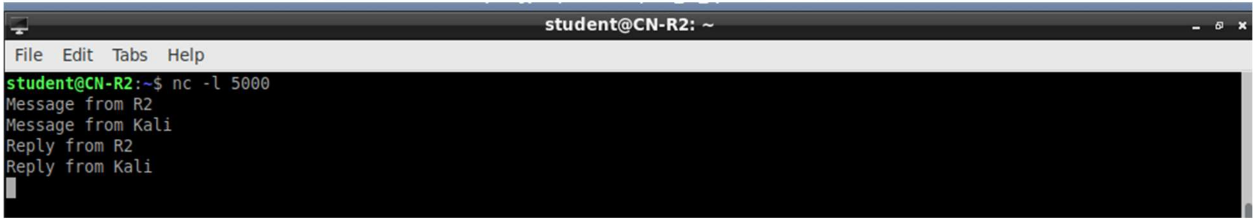
Brandon Vo

Submissions

1. Screenshots of R2 and KALI showing the netcat TCP chat

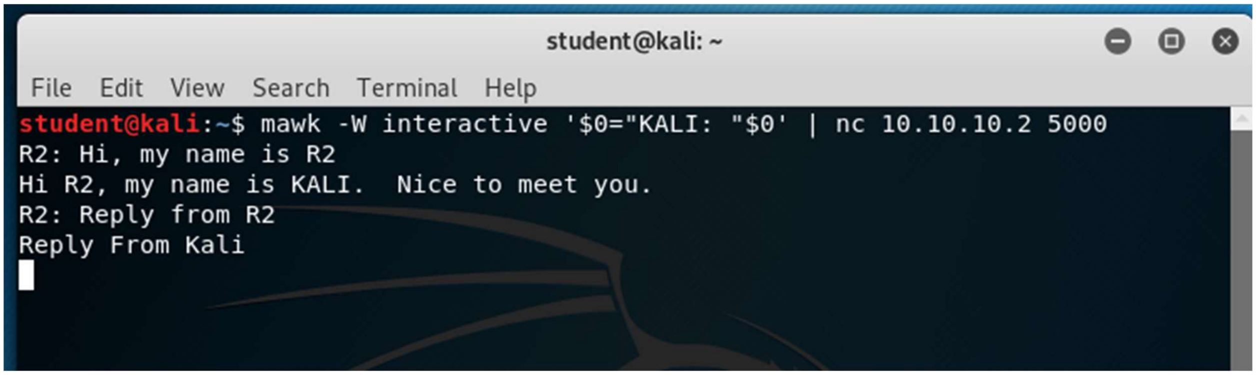


```
student@kali:~$ nc 10.10.10.2 5000
Message from R2
Message from Kali
Reply from R2
Reply from Kali
```

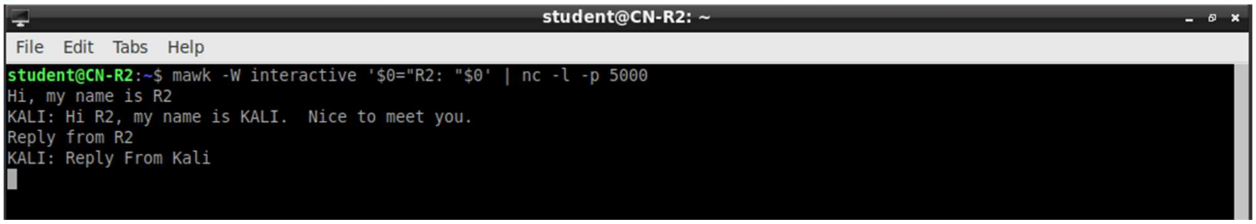


```
student@CN-R2:~$ nc -l 5000
Message from R2
Message from Kali
Reply from R2
Reply from Kali
```

With usernames



```
student@kali:~$ mawk -W interactive '$0="KALI: "$0' | nc 10.10.10.2 5000
R2: Hi, my name is R2
Hi R2, my name is KALI.  Nice to meet you.
R2: Reply from R2
Reply From Kali
```

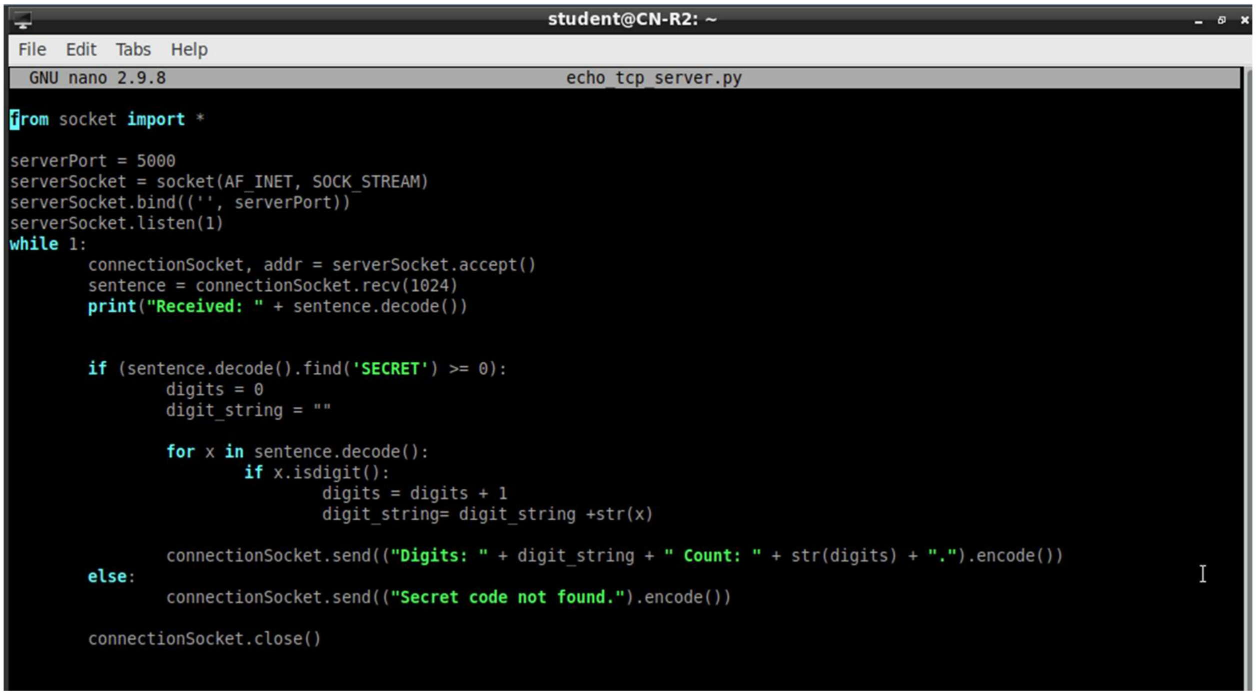


```
student@CN-R2:~$ mawk -W interactive '$0="R2: "$0' | nc -l -p 5000
Hi, my name is R2
KALI: Hi R2, my name is KALI.  Nice to meet you.
Reply from R2
KALI: Reply From Kali
```

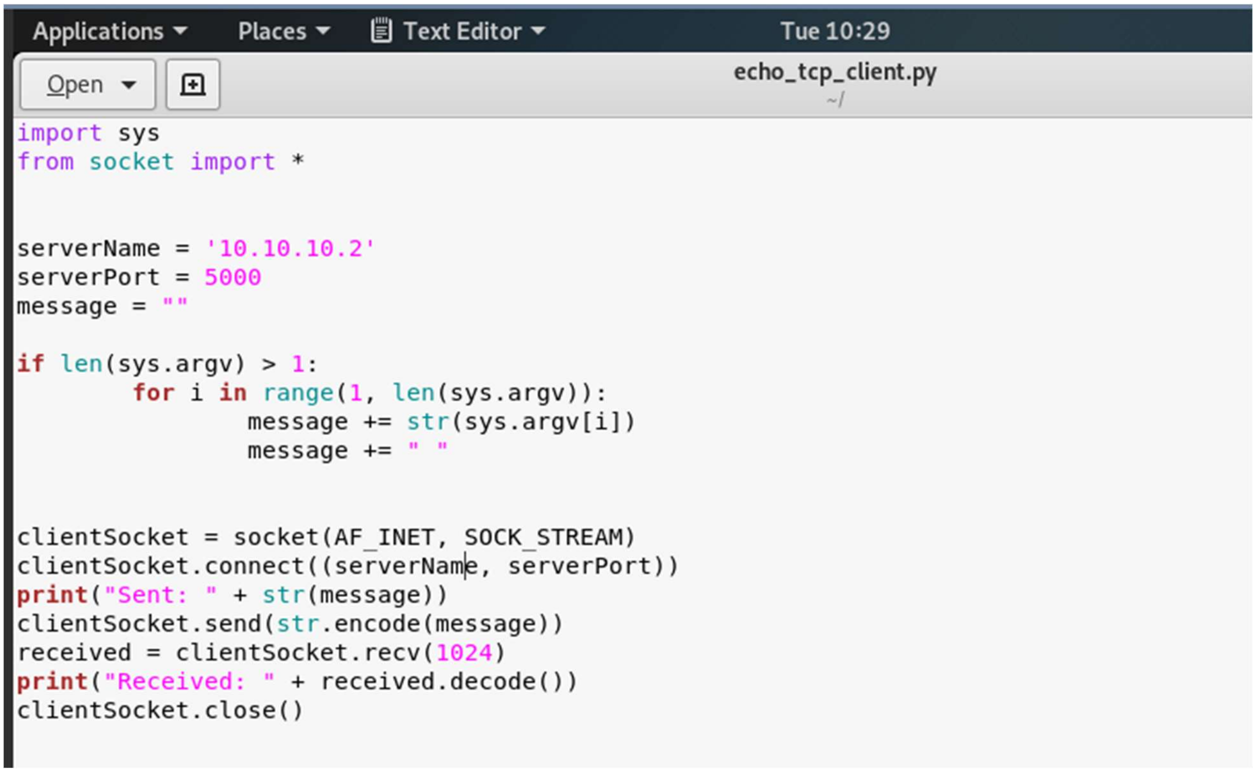
2. Screenshots of echo_tcp_server.py and echo_tcp_client.py

(showing all code)

Brandon Vo



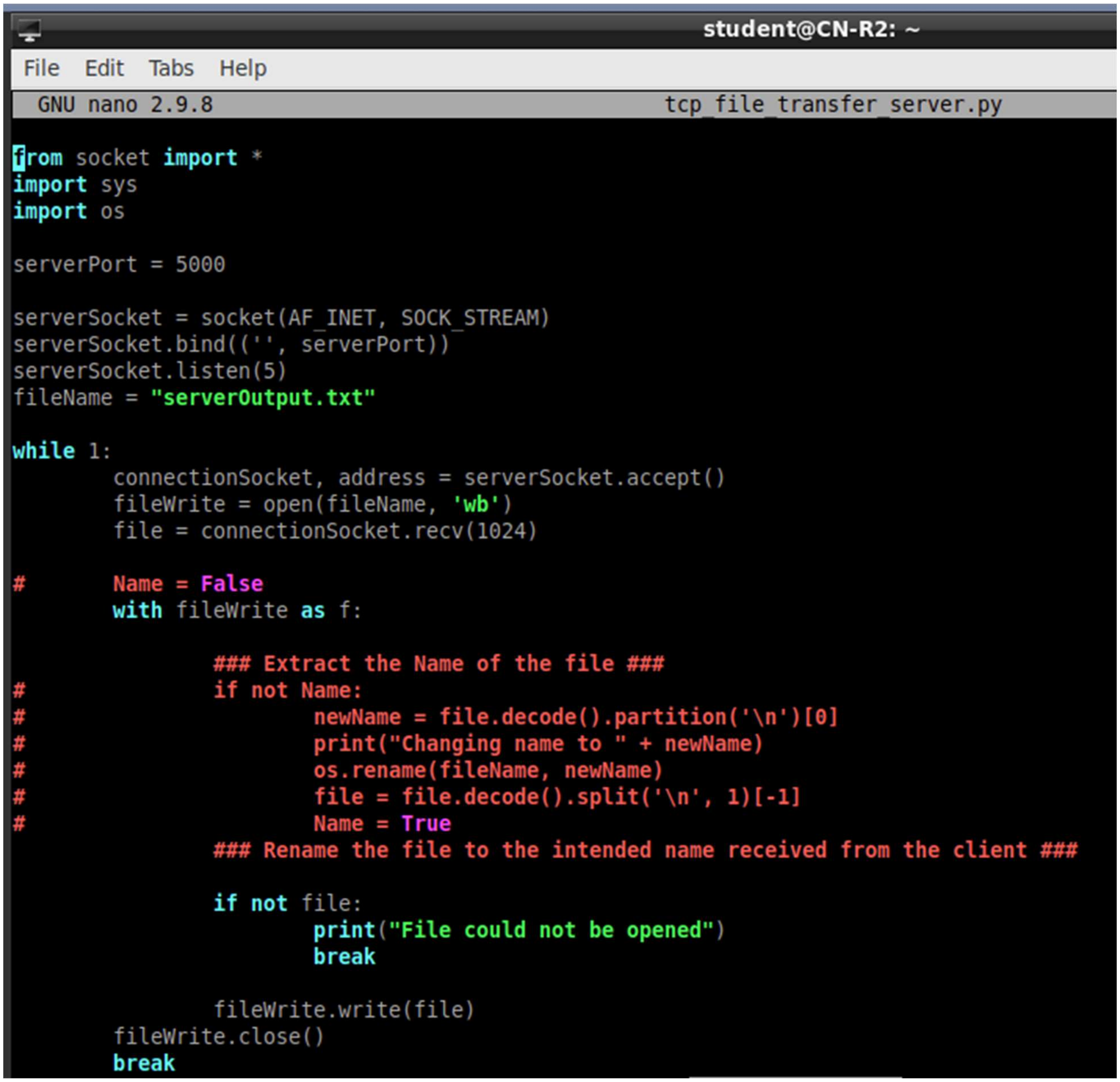
```
from socket import *

serverPort = 5000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(1)
while 1:
        connectionSocket, addr = serverSocket.accept()
        sentence = connectionSocket.recv(1024)
        print("Received: " + sentence.decode())


        if (sentence.decode().find('SECRET') >= 0):
                digits = 0
                digit_string = ""

                for x in sentence.decode():
                        if x.isdigit():
                                digits = digits + 1
                                digit_string= digit_string +str(x)

                connectionSocket.send(("Digits: " + digit_string + " Count: " + str(digits) + ".").encode())
        else:
                connectionSocket.send(("Secret code not found.").encode())

        connectionSocket.close()
```



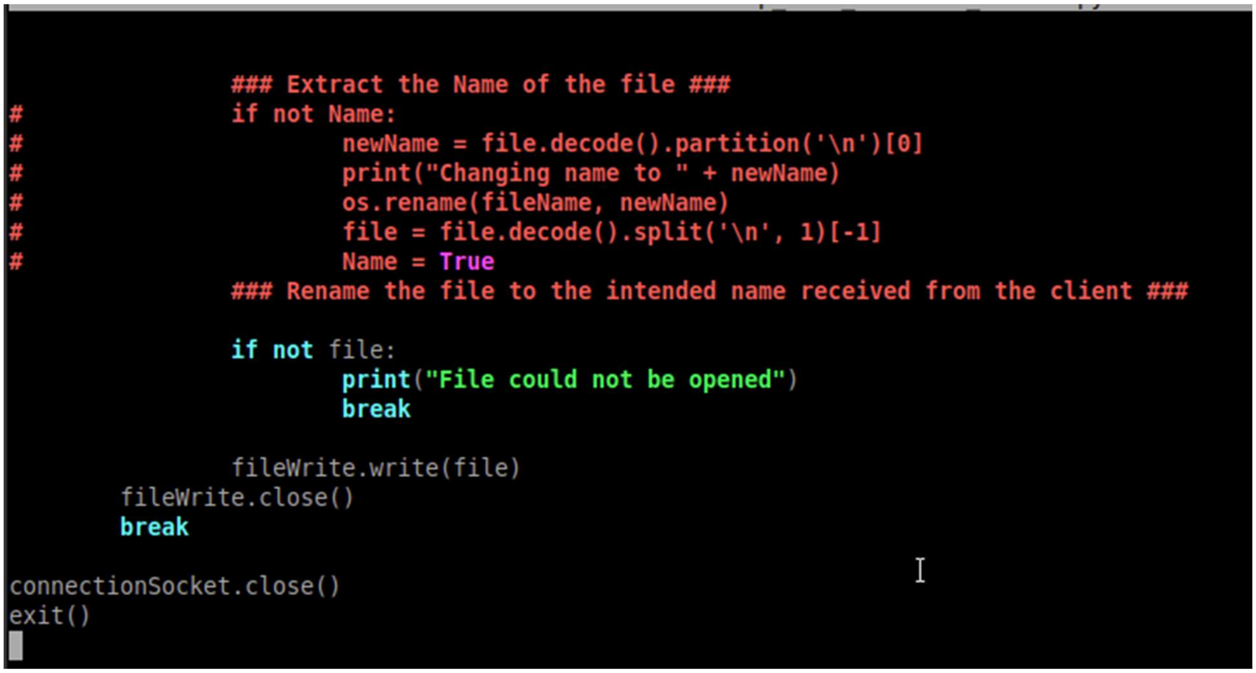
```
import sys
from socket import *


serverName = '10.10.10.2'
serverPort = 5000
message = ""

if len(sys.argv) > 1:
        for i in range(1, len(sys.argv)):
                message += str(sys.argv[i])
                message += " "


clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
print("Sent: " + str(message))
clientSocket.send(str.encode(message))
received = clientSocket.recv(1024)
print("Received: " + received.decode())
clientSocket.close()
```

3. Screenshots of tcp_file_transfer_server.py and tcp_file_transfer_client.py (showing all code)

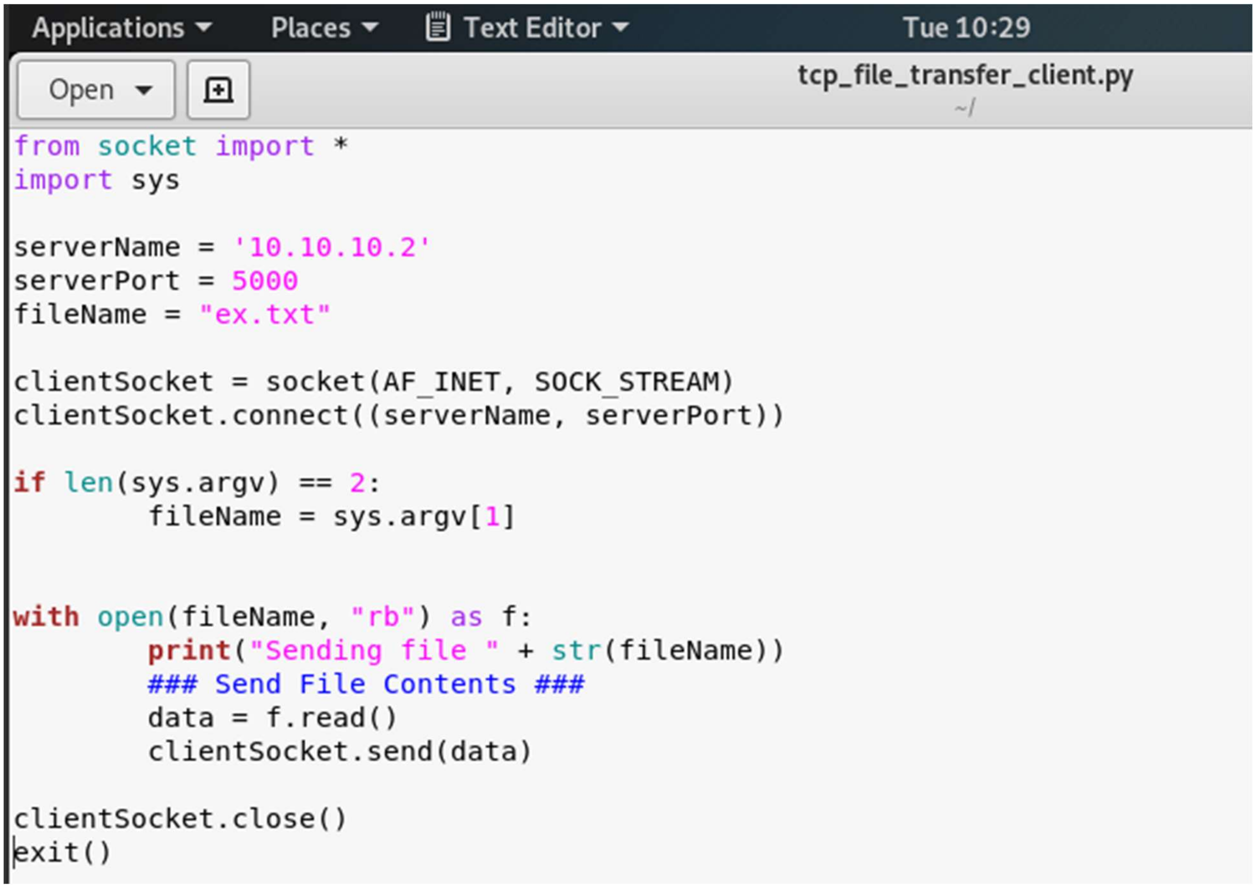
Brandon Vo



```
from socket import *
import sys
import os

serverPort = 5000

serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(5)
fileName = "serverOutput.txt"

while 1:
        connectionSocket, address = serverSocket.accept()
        fileWrite = open(fileName, 'wb')
        file = connectionSocket.recv(1024)

#       Name = False
        with fileWrite as f:

                ### Extract the Name of the file ###
#               if not Name:
#                       newName = file.decode().partition('\n')[0]
#                       print("Changing name to " + newName)
#                       os.rename(fileName, newName)
#                       file = file.decode().split('\n', 1)[-1]
#                       Name = True
                ### Rename the file to the intended name received from the client ###

                if not file:
                        print("File could not be opened")
                        break

                fileWrite.write(file)
        fileWrite.close()
        break
```

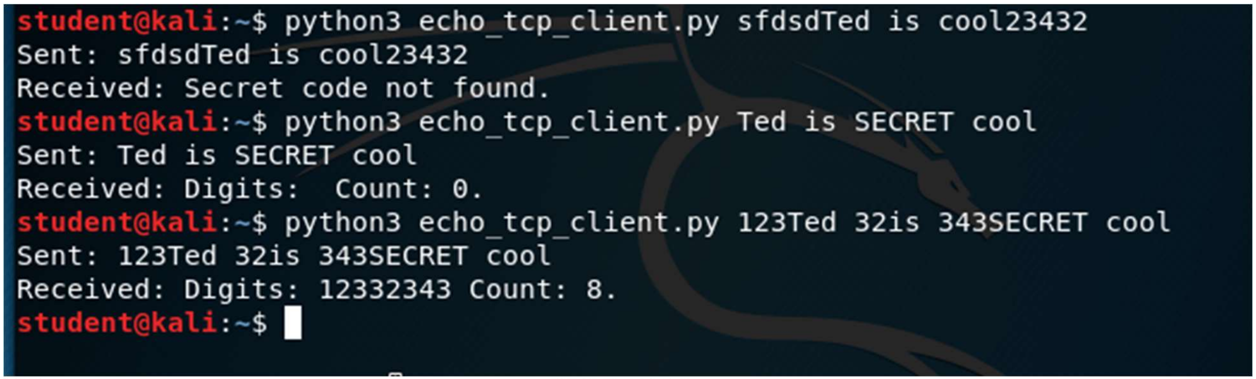


```
                ### Extract the Name of the file ###
#               if not Name:
#                       newName = file.decode().partition('\n')[0]
#                       print("Changing name to " + newName)
#                       os.rename(fileName, newName)
#                       file = file.decode().split('\n', 1)[-1]
#                       Name = True
                ### Rename the file to the intended name received from the client ###

                if not file:
                        print("File could not be opened")
                        break

                fileWrite.write(file)
        fileWrite.close()
        break

connectionSocket.close()
exit()
```

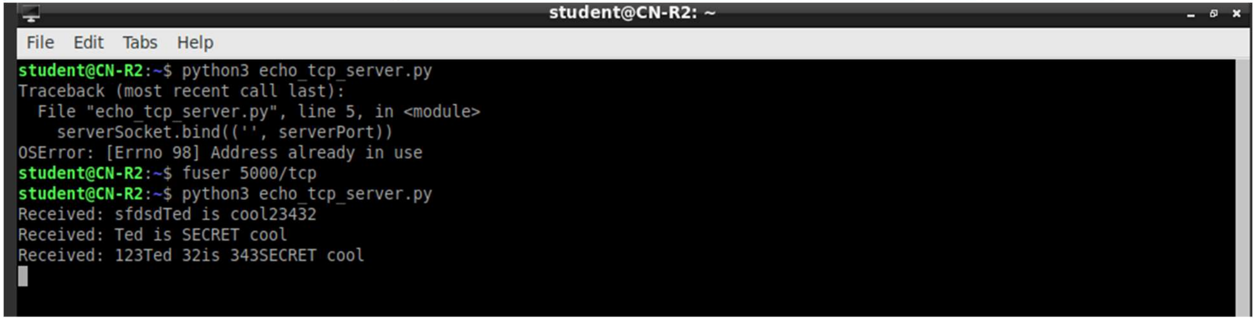


```
from socket import *
import sys

serverName = '10.10.10.2'
serverPort = 5000
fileName = "ex.txt"

clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))

if len(sys.argv) == 2:
        fileName = sys.argv[1]


with open(fileName, "rb") as f:
        print("Sending file " + str(fileName))
        ### Send File Contents ###
        data = f.read()
        clientSocket.send(data)

clientSocket.close()
exit()
```

4. Screenshots showing the behavior of Part 2. Make sure to include cases with and without the secret code.



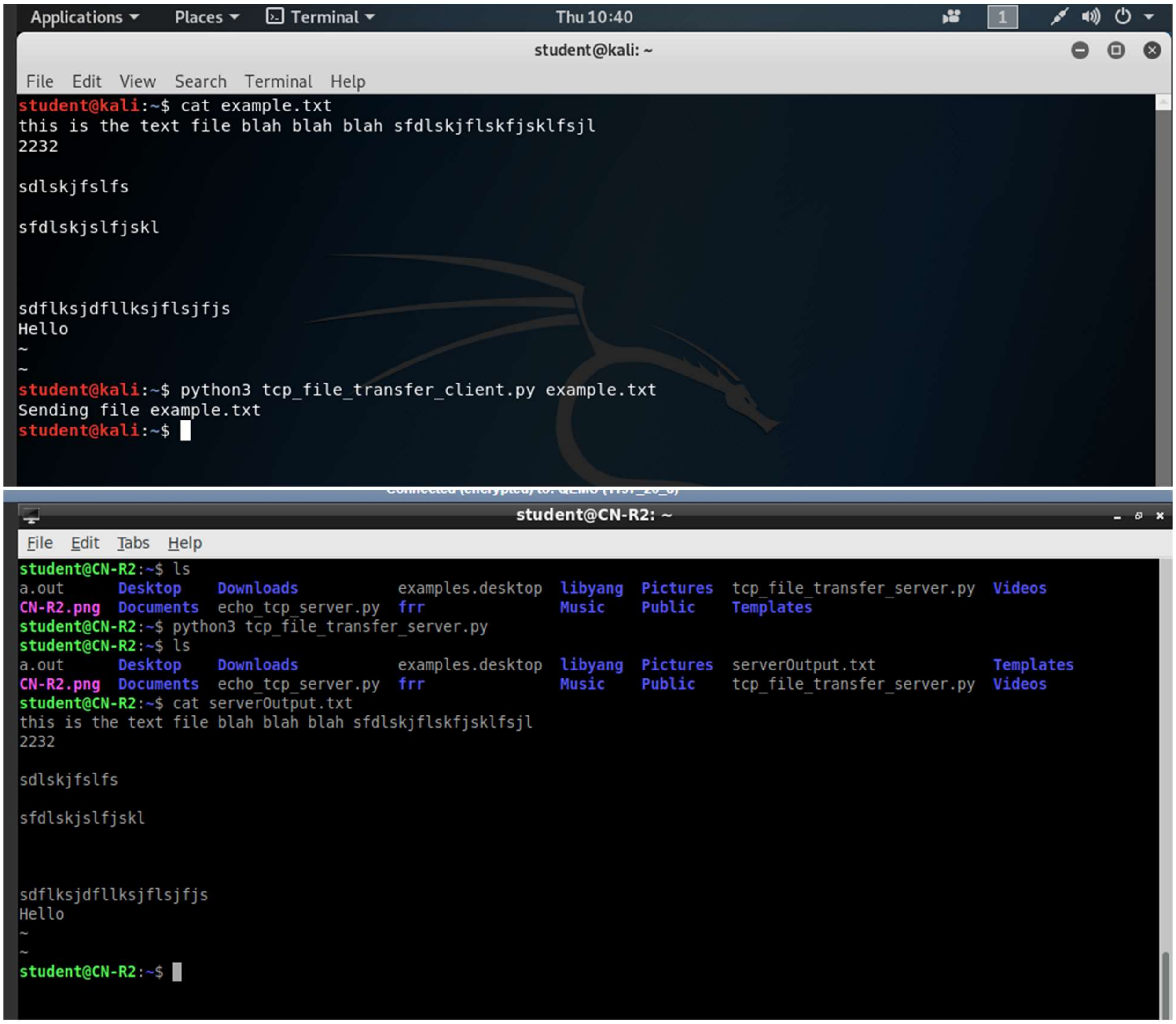
```
student@kali:~$ python3 echo_tcp_client.py sfdsdTed is cool23432
Sent: sfdsdTed is cool23432
Received: Secret code not found.
student@kali:~$ python3 echo_tcp_client.py Ted is SECRET cool
Sent: Ted is SECRET cool
Received: Digits:  Count: 0.
student@kali:~$ python3 echo_tcp_client.py 123Ted 32is 343SECRET cool
Sent: 123Ted 32is 343SECRET cool
Received: Digits: 12332343 Count: 8.
student@kali:~$ 
```



```
student@CN-R2:~$ python3 echo_tcp_server.py
Traceback (most recent call last):
  File "echo_tcp_server.py", line 5, in <module>
    serverSocket.bind(('', serverPort))
OSError: [Errno 98] Address already in use
student@CN-R2:~$ fuser 5000/tcp
student@CN-R2:~$ python3 echo_tcp_server.py
Received: sfdsdTed is cool23432
Received: Ted is SECRET cool
Received: 123Ted 32is 343SECRET cool
```

5. Screenshots showing the file transfer in Part 3: show the original file on KALI, the KALI terminal after transferring, and the transferred file on R2.

Brandon Vo



```
student@kali:~$ cat example.txt
this is the text file blah blah blah sfdlskjflskfjsklfsjl
2232

sdlskjfslfs

sfdlskjslfjskl



sdflksjdfllksjflsjfjs
Hello
~
~
student@kali:~$ python3 tcp_file_transfer_client.py example.txt
Sending file example.txt
student@kali:~$ 
```

```
student@CN-R2:~$ ls
a.out      Desktop    Downloads           examples.desktop  libyang  Pictures  tcp_file_transfer_server.py  Videos
CN-R2.png  Documents  echo_tcp_server.py  frr               Music    Public    Templates
student@CN-R2:~$ python3 tcp_file_transfer_server.py
student@CN-R2:~$ ls
a.out      Desktop    Downloads           examples.desktop  libyang  Pictures  serverOutput.txt             Templates
CN-R2.png  Documents  echo_tcp_server.py  frr               Music    Public    tcp_file_transfer_server.py  Videos
student@CN-R2:~$ cat serverOutput.txt
this is the text file blah blah blah sfdlskjflskfjsklfsjl
2232

sdlskjfslfs

sfdlskjslfjskl



sdflksjdfllksjflsjfjs
Hello
~
~
student@CN-R2:~$ 
```

6. Answers to questions 4a-4g