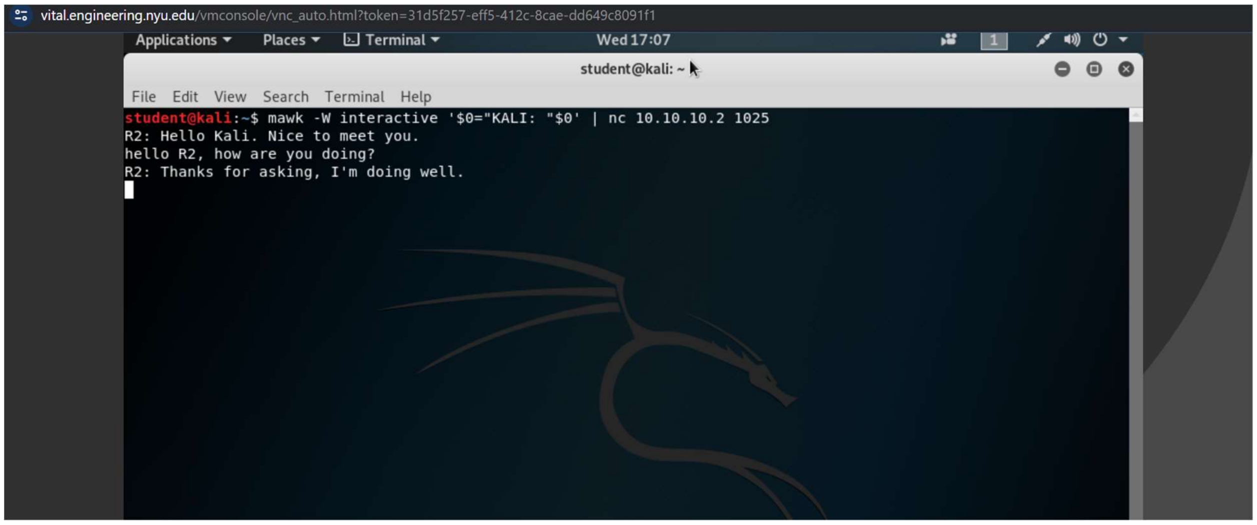
CN Sockets Lab

Submissions

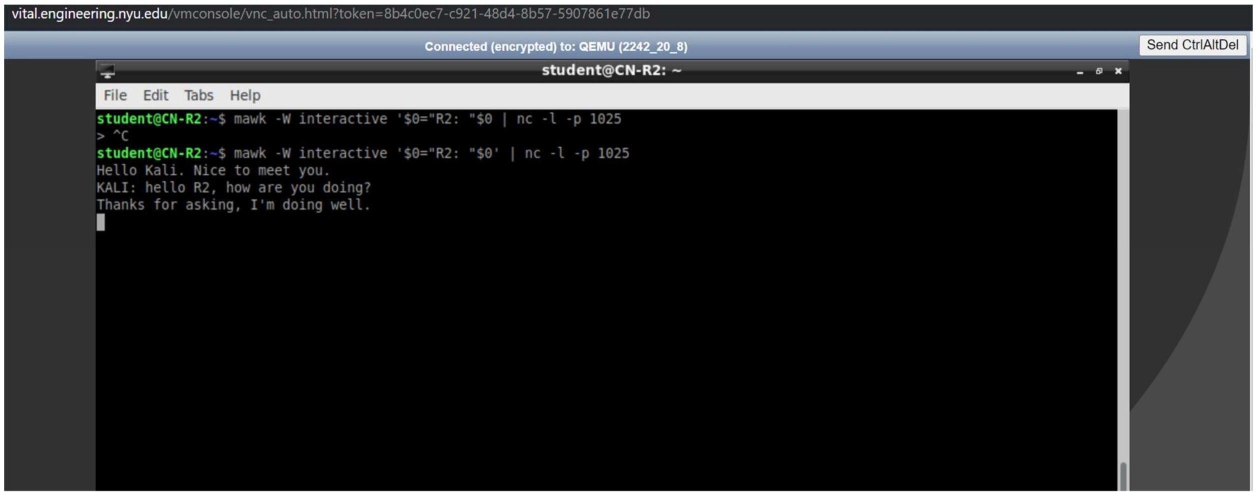
1. Screenshots of R2 and KALI showing the netcat TCP chat

Screenshot of chat seen in Kali


vital.engineering.nyu.edu/vmconsole/vnc_auto.html?token=31d5f257-eff5-412c-8cae-dd649c8091f1
Applications
Places
Terminal
Wed 17:07
student@kali: ~
File Edit View Search Terminal Help

```
student@kali:~$ mawk -W interactive '$0="KALI: "$0' | nc 10.10.10.2 1025
R2: Hello Kali. Nice to meet you.
hello R2, how are you doing?
R2: Thanks for asking, I'm doing well.
```

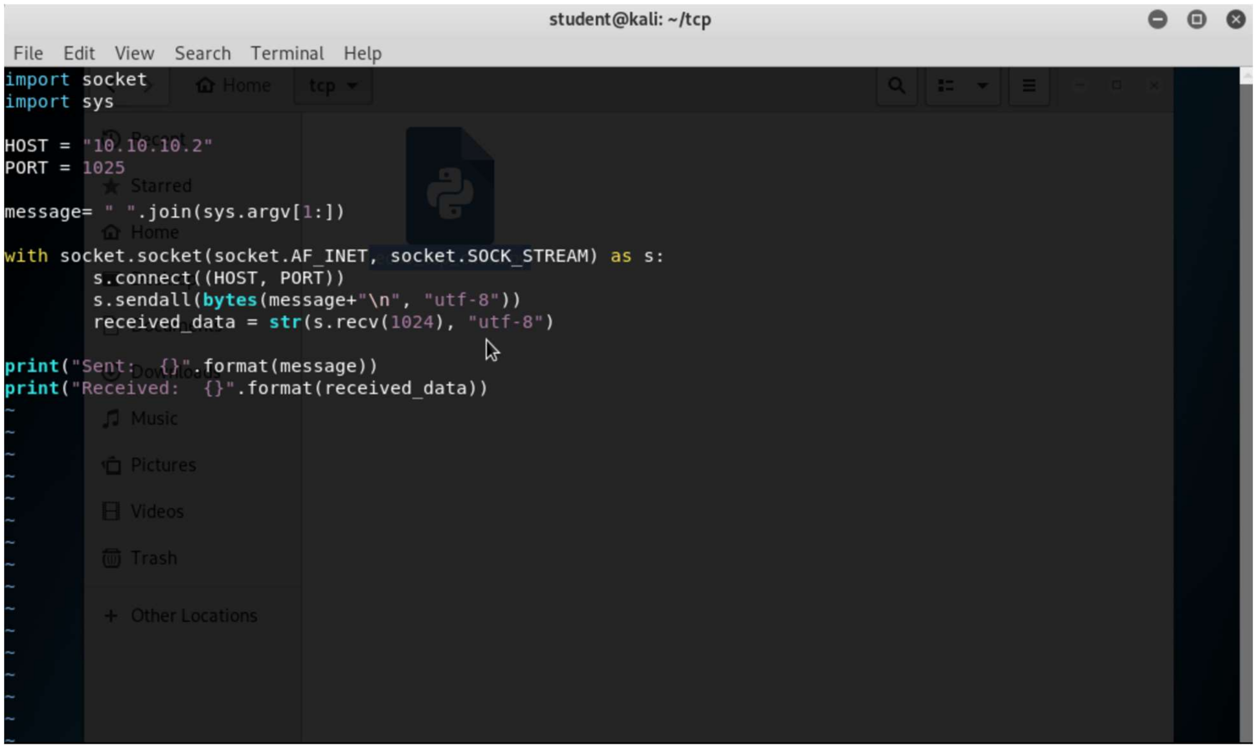
Screenshot of chat seen in R2


vital.engineering.nyu.edu/vmconsole/vnc_auto.html?token=8b4c0ec7-c921-48d4-8b57-5907861e77db
Connected (encrypted) to: QEMU (2242_20_8)
Send CtrlAltDel
student@CN-R2: ~
File Edit Tabs Help

```
student@CN-R2:~$ mawk -W interactive '$0="R2: "$0 | nc -l -p 1025
> ^C
student@CN-R2:~$ mawk -W interactive '$0="R2: "$0' | nc -l -p 1025
Hello Kali. Nice to meet you.
KALI: hello R2, how are you doing?
Thanks for asking, I'm doing well.
```

2. Screenshots of echo_tcp_server.py and echo_tcp_client.py (showing all code)

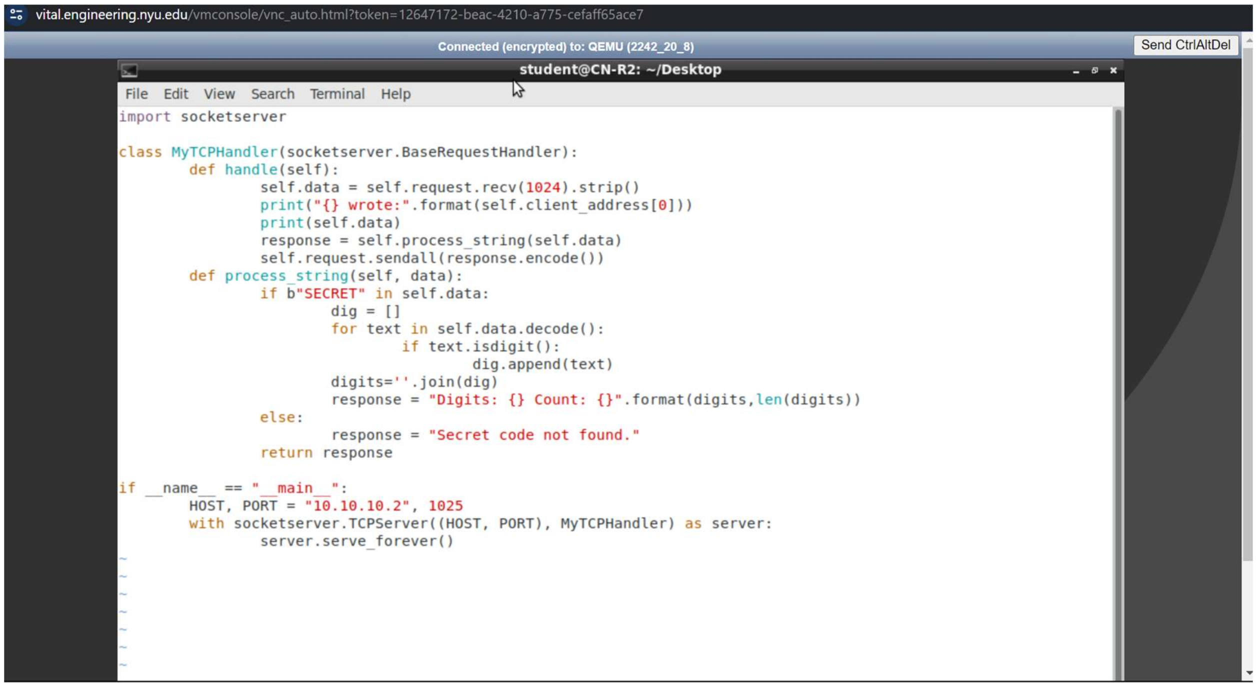
Code of echo_tcp_client.py in Kali



student@kali: ~/tcp

```python
import socket
import sys

HOST = "10.10.10.2"
PORT = 1025

message= " ".join(sys.argv[1:])

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((HOST, PORT))
        s.sendall(bytes(message+"\n", "utf-8"))
        received_data = str(s.recv(1024), "utf-8")

print("Sent:  {}".format(message))
print("Received:  {}".format(received_data))
```

Code of echo_tcp_server.py in R2



vital.engineering.nyu.edu/vmconsole/vnc_auto.html?token=12647172-beac-4210-a775-cefaff65ace7

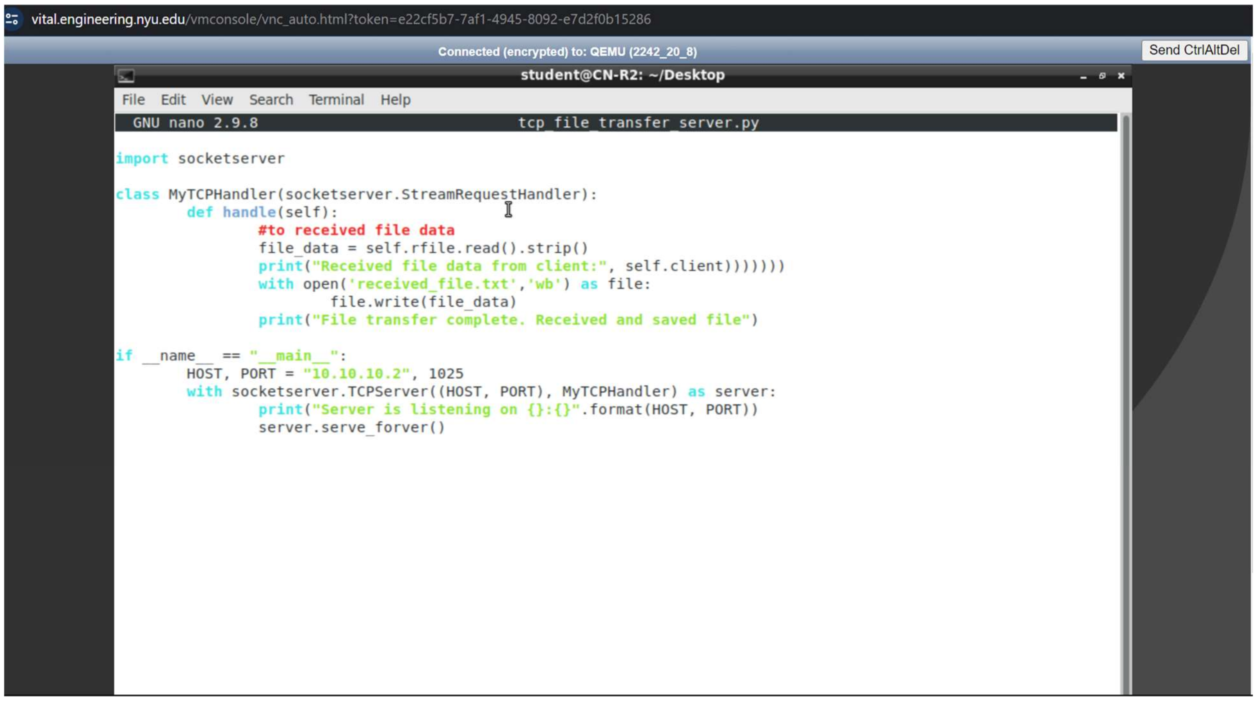
Connected (encrypted) to: QEMU (2242_20_8)

student@CN-R2: ~/Desktop

```python
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
        def handle(self):
                self.data = self.request.recv(1024).strip()
                print("{} wrote:".format(self.client_address[0]))
                print(self.data)
                response = self.process_string(self.data)
                self.request.sendall(response.encode())
        def process_string(self, data):
                if b"SECRET" in self.data:
                        dig = []
                        for text in self.data.decode():
                                if text.isdigit():
                                        dig.append(text)
                        digits=''.join(dig)
                        response = "Digits: {} Count: {}".format(digits,len(digits))
                else:
                        response = "Secret code not found."
                return response

if __name__ == "__main__":
        HOST, PORT = "10.10.10.2", 1025
        with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
                server.serve_forever()
```

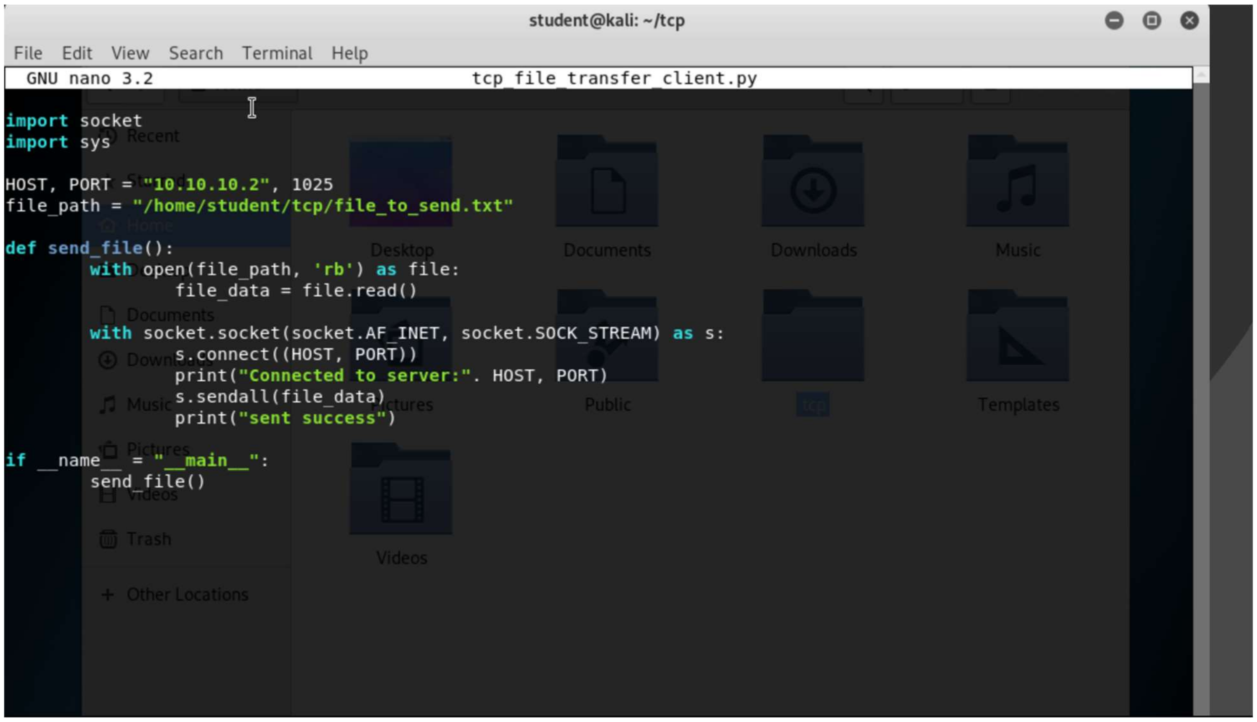
3. Screenshots of tcp_file_transfer_server.py and tcp_file_transfer_client.py (showing all code)

Code of tcp_file_transfer_server.py in R2


vital.engineering.nyu.edu/vmconsole/vnc_auto.html?token=e22cf5b7-7af1-4945-8092-e7d2f0b15286
Connected (encrypted) to: QEMU (2242_20_8)
Send CtrlAltDel
student@CN-R2: ~/Desktop
File Edit View Search Terminal Help
GNU nano 2.9.8
tcp_file_transfer_server.py

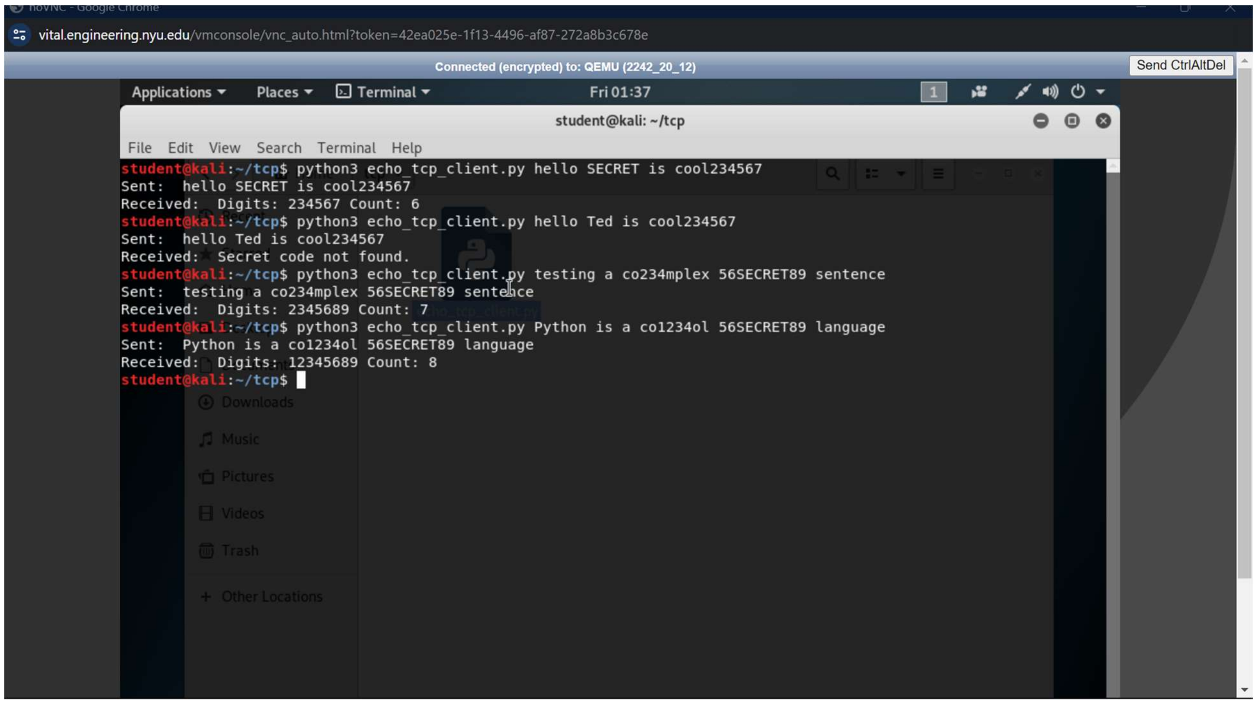
```
import socketserver

class MyTCPHandler(socketserver.StreamRequestHandler):
        def handle(self):
                #to received file data
                file_data = self.rfile.read().strip()
                print("Received file data from client:", self.client))))))
                with open('received_file.txt','wb') as file:
                        file.write(file_data)
                print("File transfer complete. Received and saved file")

if __name__ == "__main__":
        HOST, PORT = "10.10.10.2", 1025
        with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
                print("Server is listening on {}:{}".format(HOST, PORT))
                server.serve_forver()
```

Code of tcp_file_transfer_client.py in Kali


student@kali: ~/tcp
File Edit View Search Terminal Help
GNU nano 3.2
tcp_file_transfer_client.py

```
import socket
import sys

HOST, PORT = "10.10.10.2", 1025
file_path = "/home/student/tcp/file_to_send.txt"

def send_file():
        with open(file_path, 'rb') as file:
                file_data = file.read()

        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.connect((HOST, PORT))
                print("Connected to server:". HOST, PORT)
                s.sendall(file_data)
                print("sent success")

if __name__ = "__main__":
        send_file()
```

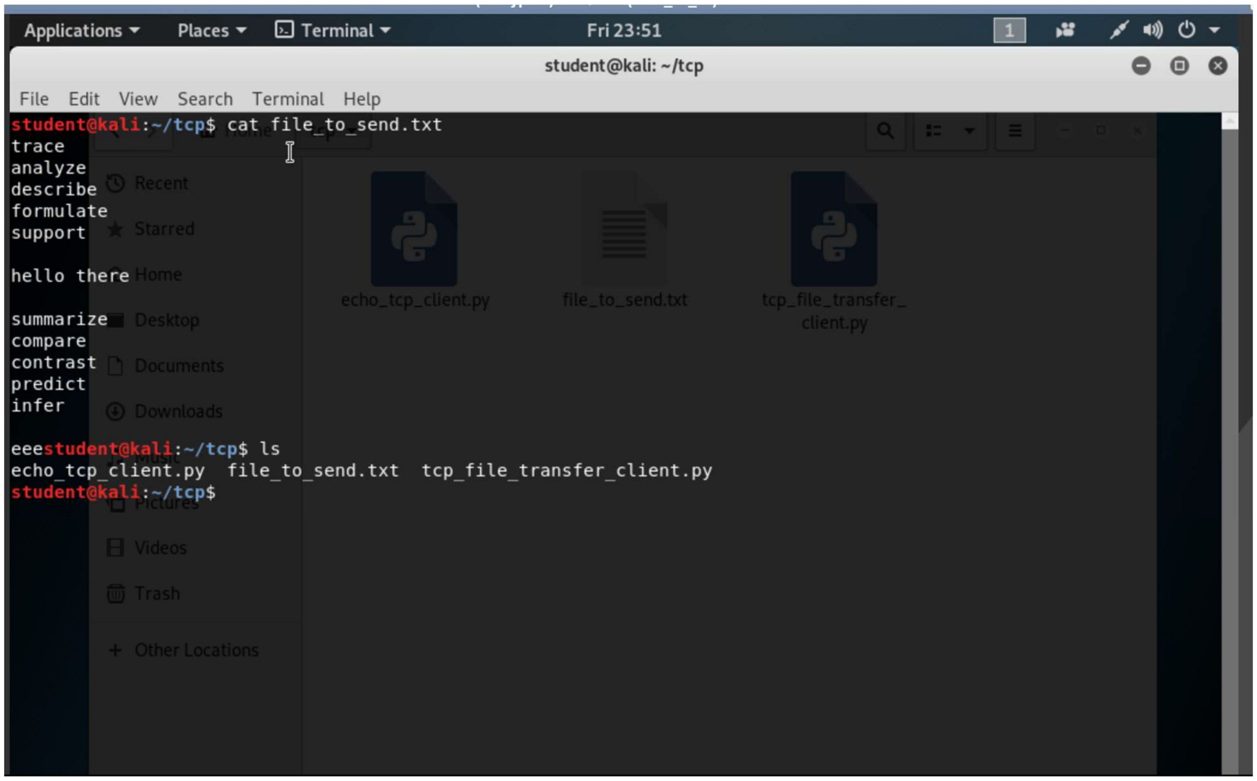
4. Screenshots showing the behavior of Part 2. Make sure to include cases with and without the secret code.



```
student@kali:~/tcp$ python3 echo_tcp_client.py hello SECRET is cool234567
Sent:  hello SECRET is cool234567
Received:  Digits: 234567 Count: 6
student@kali:~/tcp$ python3 echo_tcp_client.py hello Ted is cool234567
Sent:  hello Ted is cool234567
Received:  Secret code not found.
student@kali:~/tcp$ python3 echo_tcp_client.py testing a co234mplex 56SECRET89 sentence
Sent:  testing a co234mplex 56SECRET89 sentence
Received:  Digits: 2345689 Count: 7
student@kali:~/tcp$ python3 echo_tcp_client.py Python is a co1234ol 56SECRET89 language
Sent:  Python is a co1234ol 56SECRET89 language
Received:  Digits: 12345689 Count: 8
student@kali:~/tcp$ 
```

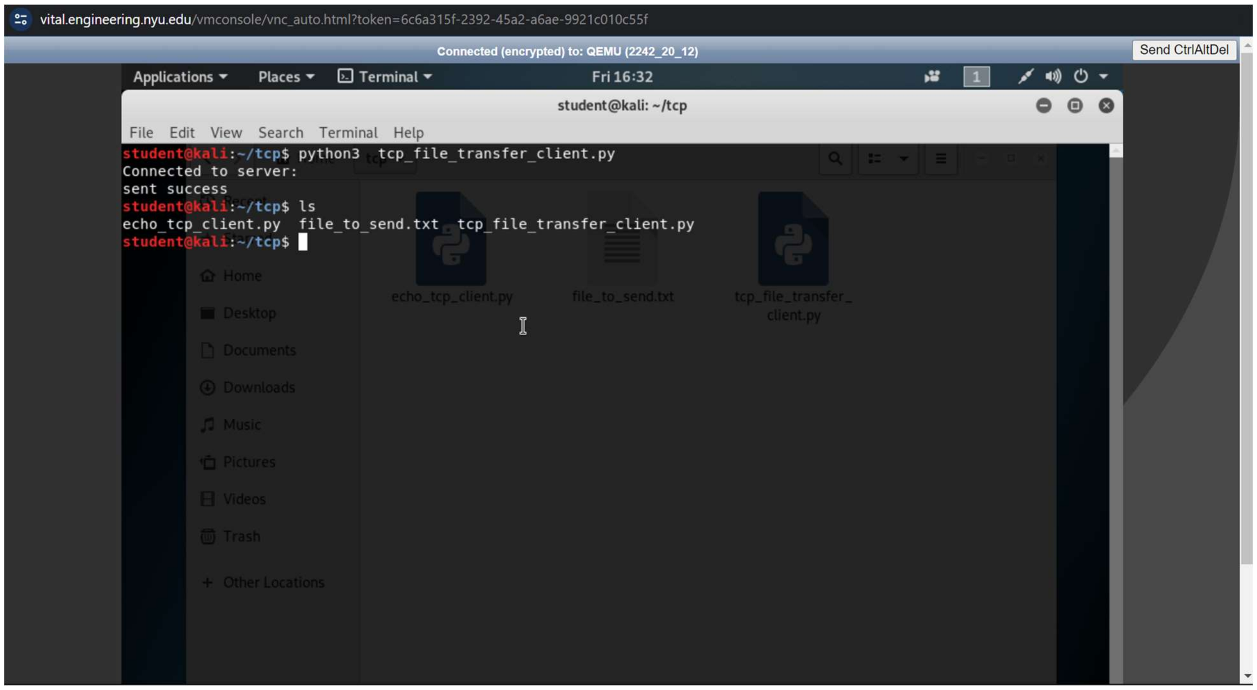
5. Screenshots showing the file transfer in Part 3: show the original file on KALI, the KALI terminal after transferring, and the transferred file on R2.

Original file on KALI (cat command of file_to_send.txt)



```
student@kali:~/tcp$ cat file_to_send.txt
trace
analyze
describe
formulate
support

hello there

summarize
compare
contrast
predict
infer

eeestudent@kali:~/tcp$ ls
echo_tcp_client.py  file_to_send.txt  tcp_file_transfer_client.py
student@kali:~/tcp$
```

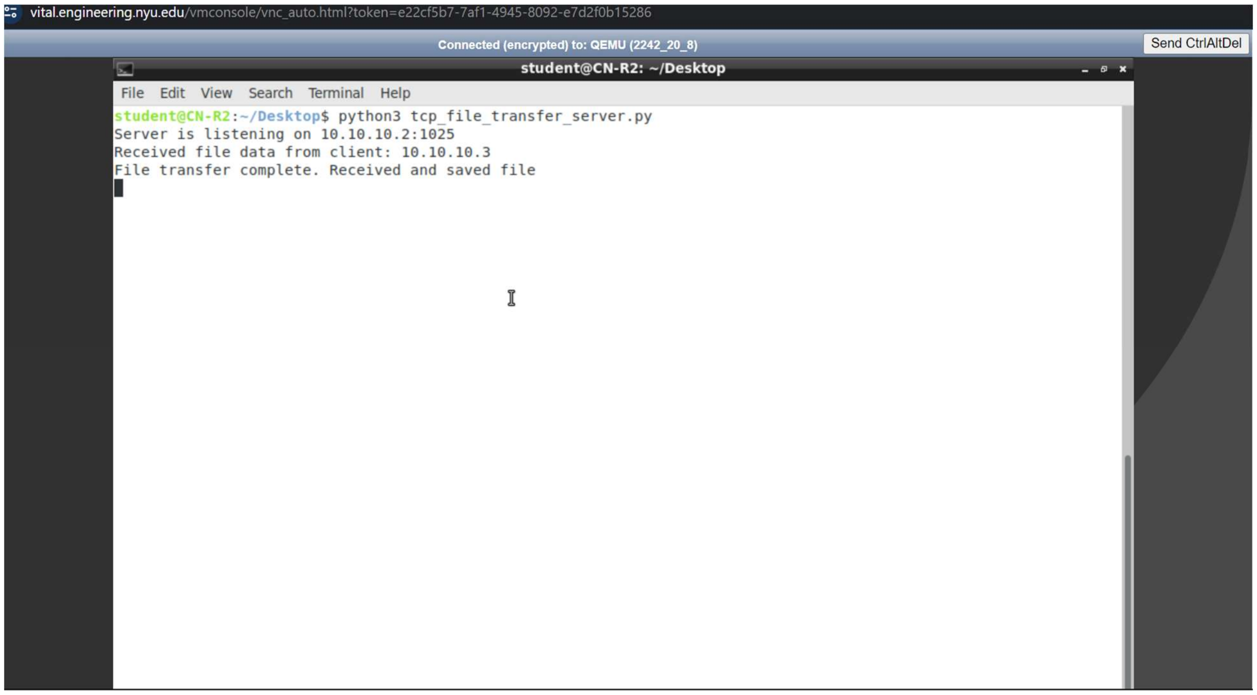
The KALI terminal after transferring (also showing ls command showing file_to_send.txt)



vital.engineering.nyu.edu/vmconsole/vnc_auto.html?token=6c6a315f-2392-45a2-a6ae-9921c010c55f
Connected (encrypted) to: QEMU (2242_20_12)
Send CtrlAltDel
Applications
Places
Terminal
Fri 16:32
student@kali: ~/tcp
File Edit View Search Terminal Help

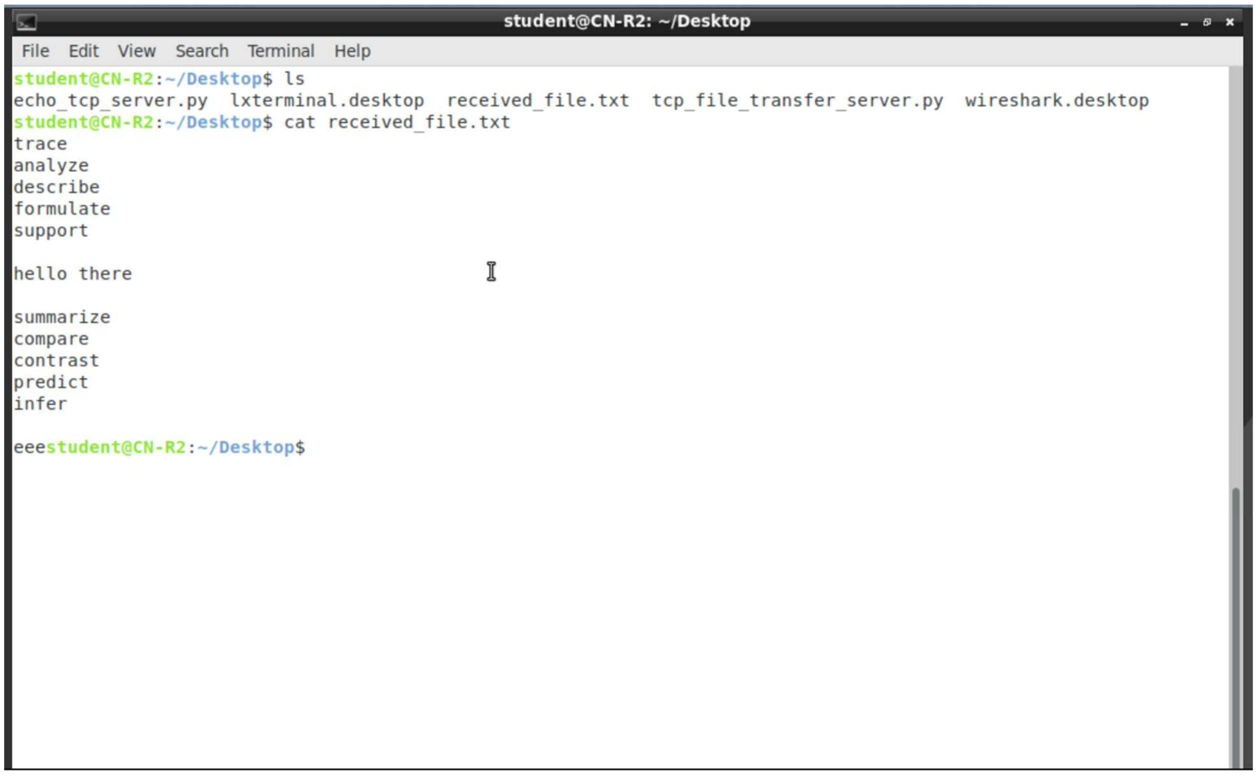
```
student@kali:~/tcp$ python3 tcp_file_transfer_client.py
Connected to server:
sent success
student@kali:~/tcp$ ls
echo_tcp_client.py  file_to_send.txt  tcp_file_transfer_client.py
student@kali:~/tcp$
```

Transferred file on R2:


vital.engineering.nyu.edu/vmconsole/vnc_auto.html?token=e22cf5b7-7af1-4945-8092-e7d2f0b15286
Connected (encrypted) to: QEMU (2242_20_8)
Send CtrlAltDel
student@CN-R2: ~/Desktop
File Edit View Search Terminal Help

```
student@CN-R2:~/Desktop$ python3 tcp_file_transfer_server.py
Server is listening on 10.10.10.2:1025
Received file data from client: 10.10.10.3
File transfer complete. Received and saved file
```

Ls command on R2 and Cat command on the received_file.txt on R2

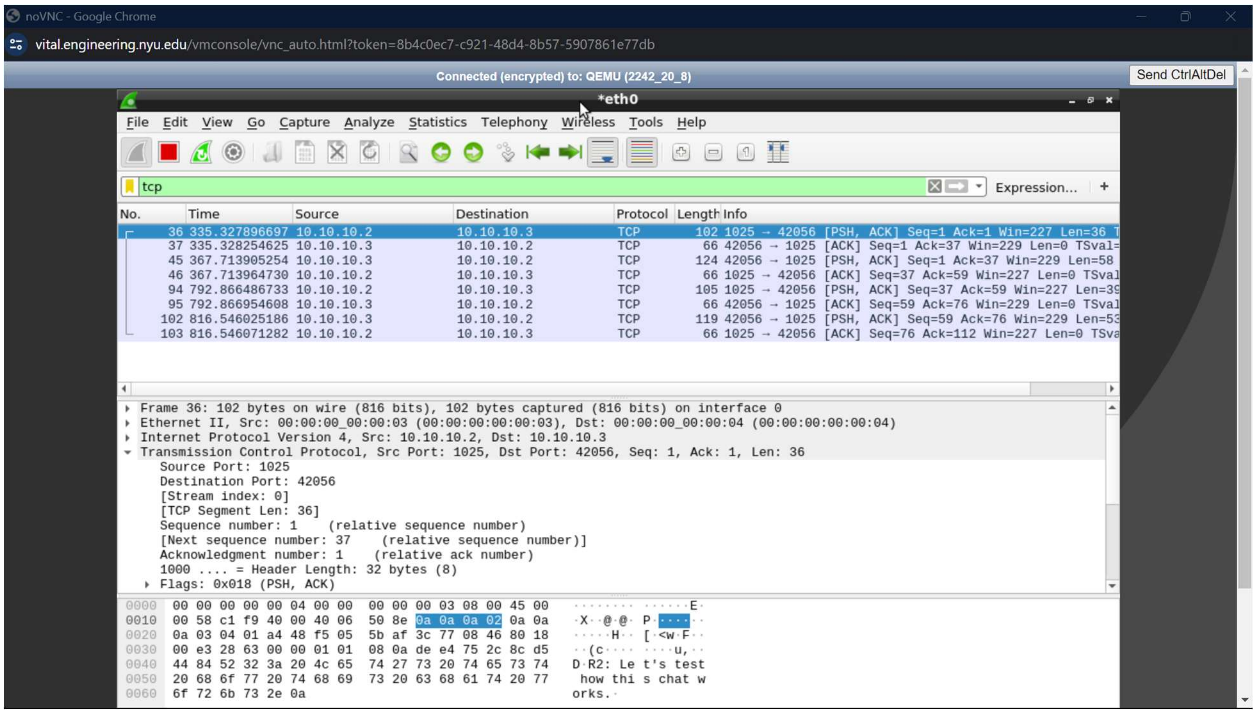


```
student@CN-R2:~/Desktop$ ls
echo_tcp_server.py  lxterminal.desktop  received_file.txt  tcp_file_transfer_server.py  wireshark.desktop
student@CN-R2:~/Desktop$ cat received_file.txt
trace
analyze
describe
formulate
support

hello there

summarize
compare
contrast
predict
infer

eeestudent@CN-R2:~/Desktop$
```

6. Answers to questions 4a-4g

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.

Source port : 1025
Destination port: 42056


noVNC - Google Chrome
vital.engineering.nyu.edu/vmconsole/vnc_auto.html?token=8b4c0ec7-c921-48d4-8b57-5907861e77db
Connected (encrypted) to: QEMU (2242_20_8)
Send CtrlAltDel
*eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp
Expression...

```
No.  Time           Source      Destination Protocol Length Info
 36 335.327896697 10.10.10.2  10.10.10.3  TCP   102 1025 → 42056 [PSH, ACK] Seq=1 Ack=1 Win=227 Len=36 T
 37 335.328254625 10.10.10.3  10.10.10.2  TCP    66 42056 → 1025 [ACK] Seq=1 Ack=37 Win=229 Len=0 TSval=
 45 367.713905254 10.10.10.3  10.10.10.2  TCP   124 42056 → 1025 [PSH, ACK] Seq=1 Ack=37 Win=229 Len=58
 46 367.713964730 10.10.10.2  10.10.10.3  TCP    66 1025 → 42056 [ACK] Seq=37 Ack=59 Win=227 Len=0 TSval
 94 792.866486733 10.10.10.2  10.10.10.3  TCP   105 1025 → 42056 [PSH, ACK] Seq=37 Ack=59 Win=227 Len=39
 95 792.866954608 10.10.10.3  10.10.10.2  TCP    66 42056 → 1025 [ACK] Seq=59 Ack=76 Win=229 Len=0 TSval
102 816.546025186 10.10.10.3  10.10.10.2  TCP   119 42056 → 1025 [PSH, ACK] Seq=59 Ack=76 Win=229 Len=53
103 816.546071282 10.10.10.2  10.10.10.3  TCP    66 1025 → 42056 [ACK] Seq=76 Ack=112 Win=227 Len=0 TSva
```

```
▸ Frame 36: 102 bytes on wire (816 bits), 102 bytes captured (816 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: 1025, Dst Port: 42056, Seq: 1, Ack: 1, Len: 36
    Source Port: 1025
    Destination Port: 42056
    [Stream index: 0]
    [TCP Segment Len: 36]
    Sequence number: 1    (relative sequence number)
    [Next sequence number: 37    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    1000 .... = Header Length: 32 bytes (8)
  ▸ Flags: 0x018 (PSH, ACK)
```

```
0000  00 00 00 00 00 04 00 00  00 00 00 03 08 00 45 00   ········ ······E·
0010  00 58 c1 f9 40 00 40 06  50 8e 0a 0a 0a 02 0a 0a   ·X··@·@· P·······
0020  0a 03 04 01 a4 48 f5 05  5b af 3c 77 08 46 80 18   ·····H·· [·<w·F··
0030  00 e3 28 63 00 00 01 01  08 0a de e4 75 2c 8c d5   ··(c···· ····u,··
0040  44 84 52 32 3a 20 4c 65  74 27 73 20 74 65 73 74   D·R2: Le t's test
0050  20 68 6f 77 20 74 68 69  73 20 63 68 61 74 20 77    how thi s chat w
0060  6f 72 6b 73 2e 0a                                  orks.·
```

Wireshark screenshot of R2 showing the client port 42056 that the server uses to send messages to the client.

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.

Explanation of code from Part 2

Here the HOST is 10.10.10.2 i.e., IP address of R2 and the PORT chosen is 1025.

Server side code:

The server code defines the MyTCPHandler class, which derives from socketserver.BaseRequestHandler. This class represents the server's request handler. It is constructed once per server connection and overrides the handle() method to implement client interactions.
The server gets data from the client using self.request.recv(1024) within the handle() method. The client socket can send up to 1024 bytes of data to this procedure.
The received data is then printed, which includes the client's IP address (self.client_address[0]) as well as the data itself.
The server checks for the string "SECRET" in the received data. If it does, it extracts all of the digits from the data and creates a response that includes the digits and their count. Self.request.sendall() is used to send this response back to the client.
If the received data does not contain the string "SECRET," the server responds that the “secret code was not found”.
Using socketserver, the server listens for incoming connections on the HOST(10.10.10.2) and PORT.TCPServer((HOST, PORT), MyTCPHandler) is a TCP server.
The server enters the main loop and invokes server.serve_forever(), which causes the server to continue indefinitely and handle client connections and requests.

Client side code:

The client code begins by specifying the HOST and PORT to which it will connect. The client takes any command-line inputs and unites them into a single string that will be delivered as data to the server.

The client uses socket.socket(socket.AF_INET, socket.SOCK_STREAM) to create a socket, specifying the address family (socket.AF_INET) and socket type (socket.SOCK_STREAM for TCP). The client establishes a TCP connection with the server by calling sock.connect((HOST, PORT)).

The data is sent to the server by the client using sock.sendall(bytes(data + "n", "utf-8")), where data is the string to be delivered. The data is encoded as bytes and transferred across the established TCP connection using the UTF-8 encoding.

The client then uses sock.recv(1024) to wait for a response from the server, which receives up to 1024 bytes of data from the server.
Finally, the client prints the data that was sent as well as the response that was received.

Explanation of code from Part 3

tcp_file_transfer_server.py (server-side):

MyTCPHandler is a class defined in the server code that derives from socketserver.StreamRequestHandler. This class represents the server's request handler. It overrides the handle() method to implement client communication.

The server gets file data from the client using self.rfile.read() within the handle() method. In a single call, this function reads all of the data sent by the client. The data from the received file is then written to a file named "received_file.txt" using open('received_file.txt,'wb') as file.

The server sends a message indicating that the file transfer is complete, and the received file is saved as "received_file.txt".
Using socketserver, the server listens for incoming connections on the provided HOST and PORT.TCPServer((HOST, PORT), MyTCPHandler) is a TCP server.

The server enters the main loop and executes the command server.serve_forever(), which causes the server to run indefinitely and handle client connections and file transfers.

Client-side (tcp_file_transfer_client.py):

The client takes the file path as mentioned in the tcp folder. To handle the file transfer procedure, the client defines the method send_file(). The client gets file data from the supplied file path using open(file_path, 'rb') as file inside send_file().

The client uses socket.socket(socket.AF_INET, socket.SOCK_STREAM) to create a socket, specifying the address family (socket.AF_INET) and socket type (socket.SOCK_STREAM for TCP). The client establishes a TCP connection with the server by calling sock.connect((HOST, PORT)).

The client uses sock.sendall(file_data) to send the file data to the server, where file_data is the content of the file read previously. The client prints a message indicating that the file was successfully transmitted. To start the script, the send_file() function is invoked at the end.

c) What does the socket system call return?

In most operating systems, the socket system call returns a socket descriptor, which is an integer value that represents the newly established socket. This socket descriptor is used in subsequent socket-related function calls to identify and reference the socket.

The socket descriptor is a handle or reference to the socket that allows the operating system and application to interact with it. It acts as a unique identification for the socket within the operating system's socket table.

If the socket creation fails, the socket descriptor is normally a non-negative integer, and a value of -1 is returned. In these circumstances, an error code can be checked to establish the exact cause of the failure.

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

The bind system call is used to associate a socket with a specified network address (IP address and port number) on the local machine. It gives the socket a local address, allowing it to listen for incoming connections or transmit data to a specific location.

The bind system call returns an integer value indicating whether the binding procedure was successful or unsuccessful. A return value of 0 normally signifies success, but a negative value (usually -1) usually indicates an error. When an error occurs, a suitable error code is assigned, and the precise error can be found by inspecting the error code.

The bind system call can be used by both the client and the server in the context of client-server communication.
Both the client and the server can use the bind system call in client-server communication, however their usage and purpose differ:

Server: The bind system call is often used by the server to bind a specific network address (IP address and port) to the server socket. This instructs the server to monitor that IP for inbound connections. Before entering the listening state, the server binds the socket.

Client: Typically, the client does not explicitly invoke the bind system call. Instead, when a client socket is formed, the operating system immediately assigns it a local address. This address is usually ephemeral and is determined by the operating system. The client sends data to the server using the provided local address.

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.)

The User Datagram Protocol (UDP) would be the superior choice over the Transmission Control Protocol (TCP) in the context of sending an urgent message from a remote client to a server as rapidly as feasible.

TCP is a connection-oriented protocol, which means it creates a connection before beginning data transfer and assures reliable packet delivery. It handles packet loss with a method called Retransmission Timeout (RTO) and retransmits missing packets, which might create a delay in the transmission of urgent messages.

Based on Reduced RTT (Round-Trip Time): TCP requires the client and server to establish a connection, which involves a handshake process (TCP three-way handshake). This handshake adds more time (RTT) before any data can be transferred. Because UDP is not connected, it does not require this handshake and can begin transferring data instantly. This lower RTT in UDP speeds up message delivery.

UDP, on the other hand, is a protocol that does not require a connection. It does not establish a link or guarantee packet delivery. It merely sends packets to their destination and does not acknowledge receipt. As a result, UDP becomes faster and more suitable for transmission of urgent messages.

f) What is Nagle's algorithm? What problem does it aim to solve and how?

Nagle's algorithm is a TCP (Transmission Control Protocol) mechanism for optimizing network traffic by minimizing the amount of tiny packets delivered across the network. It seeks to address the issue of tiny packets, which cause network inefficiencies and costs.

The problem addressed by Nagle's technique is known as the "small packet problem" or "silly window syndrome." Each data segment delivered across the network in TCP often carries some overhead, such as the TCP header. When a sender application sends a small quantity of data as a discrete TCP segment, such as a single character, the overhead of the TCP header becomes rather substantial in comparison to the actual data being delivered. This might lead to wasteful network consumption and higher latency.

Nagle's method works by buffering small pieces of incoming data at the sender's end and merging it into bigger segments before sending it over the network. Instead of instantly transmitting small packets, it waits for further data to come or for an acknowledgement of previously delivered data. Nagle's technique reduces the overhead associated with each individual packet by consolidating numerous little writes into a bigger chunk.

Another aspect of Nagle's technique addresses the issue of a network system becoming clogged when a large datagram is transmitted repeatedly because it takes too long to fill a sending window (the receiving buffer) with data from the datagram.

The algorithm has two main components:

Delayed Acknowledgement: Nagle's algorithm delays delivering acknowledgment (ACK) messages for received segments. This enables the sender to deliver more data in a single segment before sending an ACK, lowering the number of ACK packets sent.

TCP_NODELAY Option: The TCP_NODELAY option can be used to disable Nagle's algorithm. When this option is enabled, short segments are transmitted without waiting for additional data or ACKs. This can be advantageous for real-time communication or interactive applications that require low latency over efficient network consumption.

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

Delayed Acknowledgement (ACK) is a mechanism used by some Transmission Control Protocol (TCP) implementations to increase network speed. It reduces protocol overhead by merging numerous ACK responses into a single response.

However, in some cases, Delayed ACK can be problematic. One example is when the sender is waiting for the recipient to acknowledge before sending more data. If the receiver uses Delayed ACK and fails to deliver an acknowledgment within the given delay, the sender may presume the acknowledgment was lost and retransmit the data. This can result in redundant retransmissions and waste of network resources.
Furthermore, Delayed ACK might present problems in applications that require minimal latency. In a real-time gaming environment, for example, where each frame of the game is sent as a distinct packet, the Delayed ACK delay might cause perceptible lag. The sender may wait for the acknowledgment before delivering the following frame, causing a perceptible delay to the user.

In such circumstances, modifying or removing the Delayed ACK timer may be required to reduce latency and improve application performance. However, these adjustments should be made with caution because they can have an impact on overall network performance.