Paras Gupta
19110128

CS 433
Computer Networks
Assignment 1

Part I: Network Application using Socket Programming
Q1.

(a) Source Code: The working code implementation for the Remote File System (RFS) service has been uploaded to GitHub. One can refer to the readme file for details on code execution.

(b) Design Document: *Network Application using Socket Programming*

This design document details the following:

1. Modes of Layering
2. Protocol Design
3. Associated Challenges
4. Code Execution Results

Modes of Layering

Layering has been done following the same template provided in the question description. The layering principles for each of the above layers can be mapped to specific layers in the stack for the OSI reference model.

The OSI or the Open Systems Interconnection model provides seven layers to facilitate communication across a network. The seven layers, in a top to down order, include *Application Layer, Presentation Layer, Session Layer, Transport Layer, Network Layer, Data Link Layer, and Physical Layer*. The three layers in our RFS system, with the corresponding mappings to the OSI model, are given below:

1) File Service Layer :: Application Layer

The File Service Layer of our system forms the application layer of the network. The files **service_client.py** and **service_server.py** in the Client and Server directories respectively of the submitted code contain this layer. The files contain utilities that form the data to be transmitted across the client-server link and also allow displaying received or other meaningful information at both ends. The layer makes use of the services of the Crypto Layer, as detailed in the Protocol Design Section.

2) Crypto Layer :: Presentation Layer

The Crypto Layer of our system forms the presentation layer of the network. The files **server_crypt.py** and **client_crypt.py** in the Client and Server directories respectively of the submitted code contain this layer. The files allow for manipulation of the data before sending it over the network using encryption and decryption techniques given below:

- Plain Text - Sending the data as it is
- Substitute - Substituting alphanumeric characters with fixed offset before sending
- Transpose - Reversing the contents in a word-by-word manner before sending

The layer makes use of the services of the Network Layer, as detailed in the protocol design section.

3) Networking Layer :: Transport Layer

The Networking Layer of our system forms the transport layer of the network. The files **server.py** and **client.py** in the Client and Server directories of the submitted code contain this layer. The files contain the source code to establish a non-persistent connection between server and client using the socket package in python. The layer sends and transmits the data received from the upper layer, across the network.

Protocol Design

a) Client

Upon starting the client (see README.md on GitHub), the **File Service Layer** asks the user for the encryption mode and the service that is to be requested from the server. The input is then checked to see if the service is the upload or “upd” service, which requires reading and sending the contents of a file present in the client along with the service information. The File service layer interacts with OS to read the file contents.

The service information includes service command and other input metadata like file or directory path. The File Service Layer appends the input encryption mode as a **header** to the service information. The complete unit or protocol data unit (PDU) is then passed down to the **Crypto Layer** as its service data unit (SDU).

The Crypto Layer file contains utilities that allow the encryption and decryption of data. Since its SDU needs to be sent across the network, the Crypto Layer encrypts the SDU based on the encryption mode header present in the SDU, forms the PDU of the Crypto Layer, and then passes it down to the third layer, the **Networking Layer**.

The Networking Layer, upon receiving the data unit from the Crypto Layer, establishes a socket connection the running client and the server using **IPv4** (Internet Protocol Version 4) over **TCP** (Transmission Control Protocol), and sends the data unit to the server.

b) Server

The **Networking Layer** of the server receives the data units from the Networking Layer of the client. The data is then passed up to the **Crypto Layer** of the server where it is decrypted using the encryption mode information available in the form of a header. The decrypted data is next passed to the **File Service Layer** of the server as its SDU, where the requested service is extracted and separated from the mode header. The service request is then processed at the server, and the corresponding **service response is** gets ready to be transmitted through the network.

The service response follows a flow (top to down) across the three server-side network layers, similar to the one described above in the client protocol design. The response is transmitted to the client, and the data flows down to the top, to the File Service Layer of the Client which displays the response to the client user.

Associated Challenges

The major challenge was to structure the whole code to form separate layers that used each other's services. Challenges and difficulties faced while implementing those layers are given below:

1) File Service Layer :: Application Layer

Parsing the input data, appending header information, checking for certain services and processing them, and displaying information to the user, were some of the major tasks performed in this layer.

Initially, the header containing the encryption mode was sent through a separate send operation to the server. Although it wouldn't have caused many problems in the submitted python code, if a packet loss were to occur when the header was being sent, it would have become difficult to decrypt any data being sent next.

The download and upload services required different processing from the other services and were prone to various errors. The thought process involved for these services was longer and it was more difficult to code them.

2) Crypto Layer :: Presentation Layer

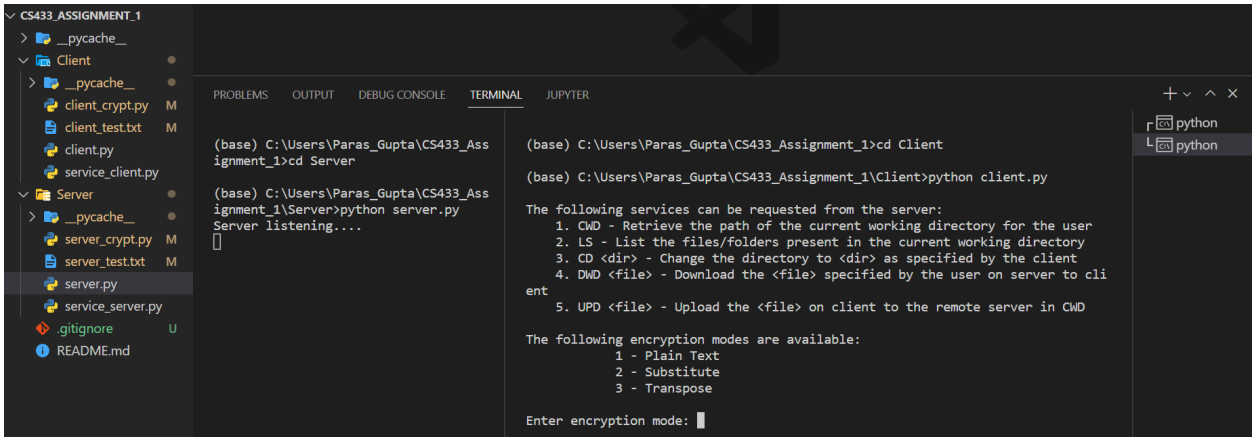
Encryption and Decryption using substitute and transpose posed some challenges. The substitute and transpose methods were initially implemented using the split operation in python that did not account for line breaks. The code was changed to account for line breaks.

3) Networking Layer :: Transport Layer

The socket package in python was used for the first time, and so it was difficult to debug whenever errors were encountered. The final submitted code establishes a non-persistent connection between the client and the server. It was difficult to think of the steps to establish a persistent connection.

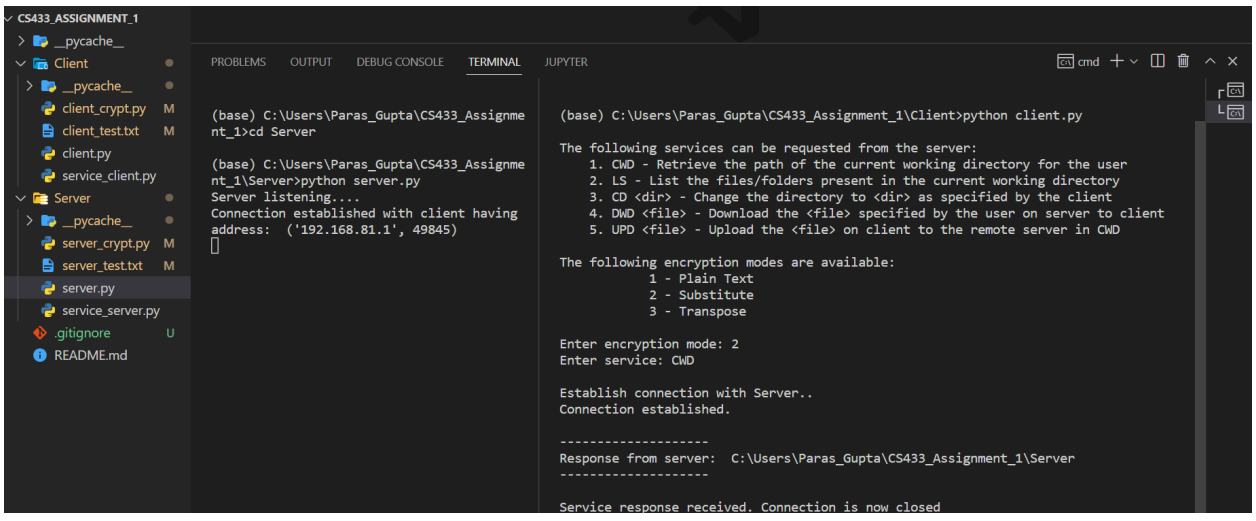
Code Execution Results

The initial directory structure and the responses from the server (left terminal) and the client (right terminal) are shown below:



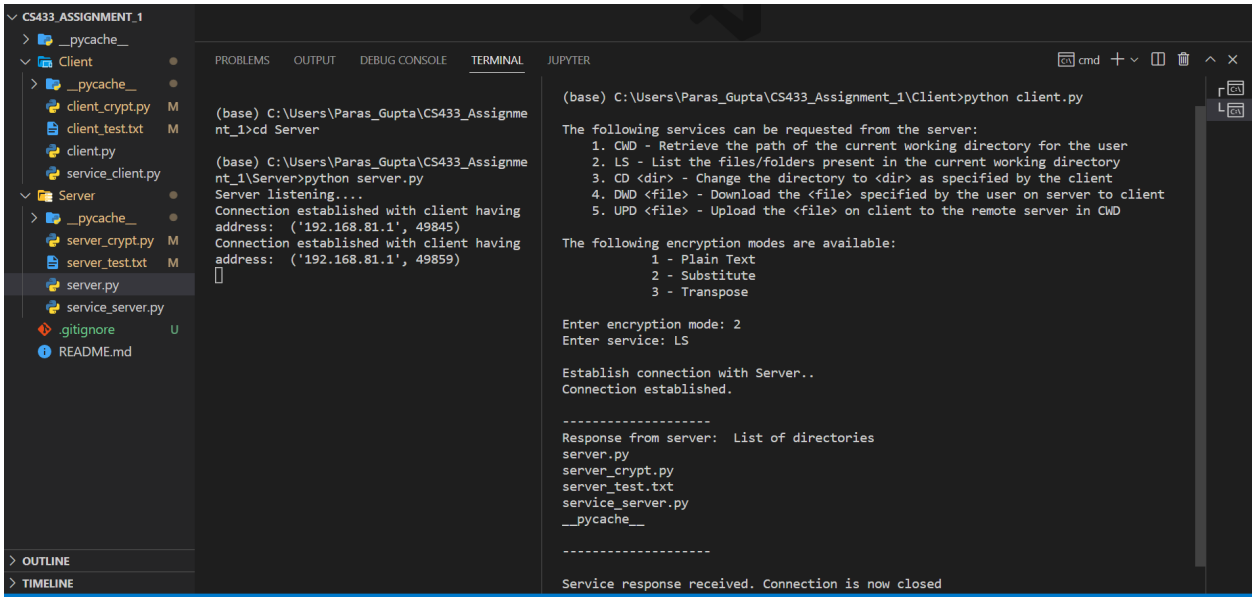
The code execution for the five service commands is as follows:

1. **CWD**



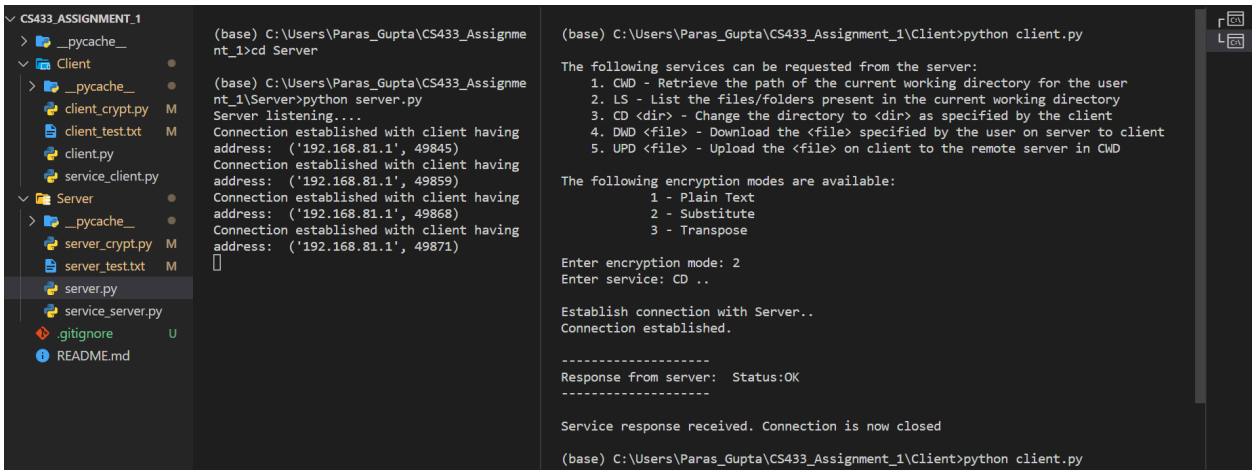
The current working directory of the server is returned as a response to the client.

2. LS

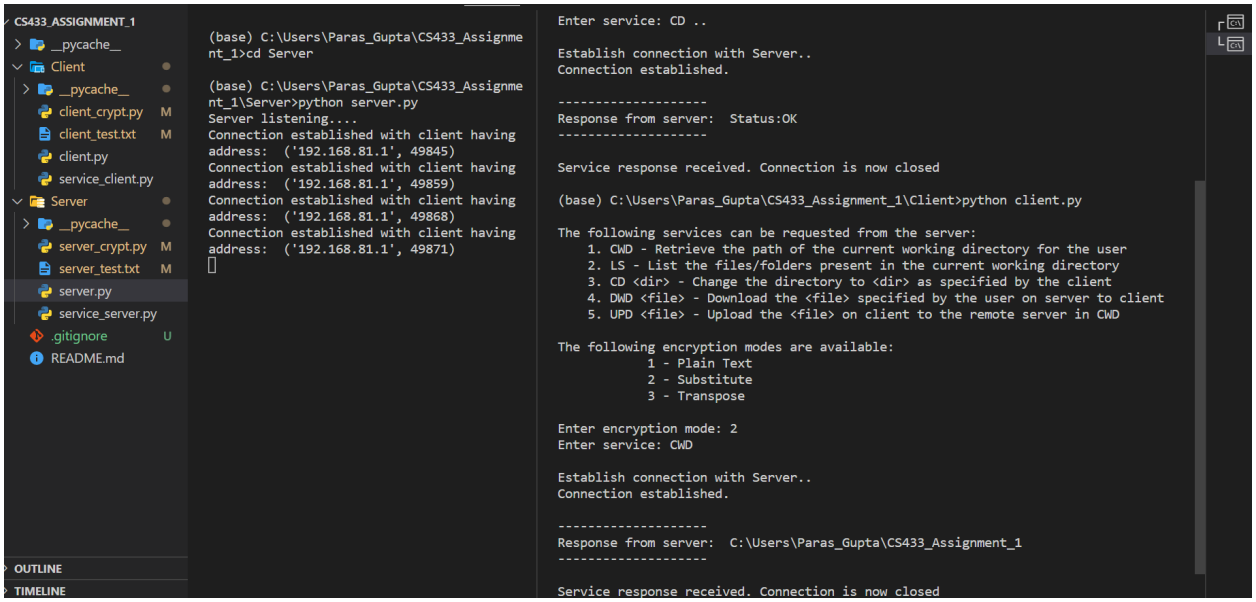


The list of files and folders present in the server are listed out.

3. CD <dir>

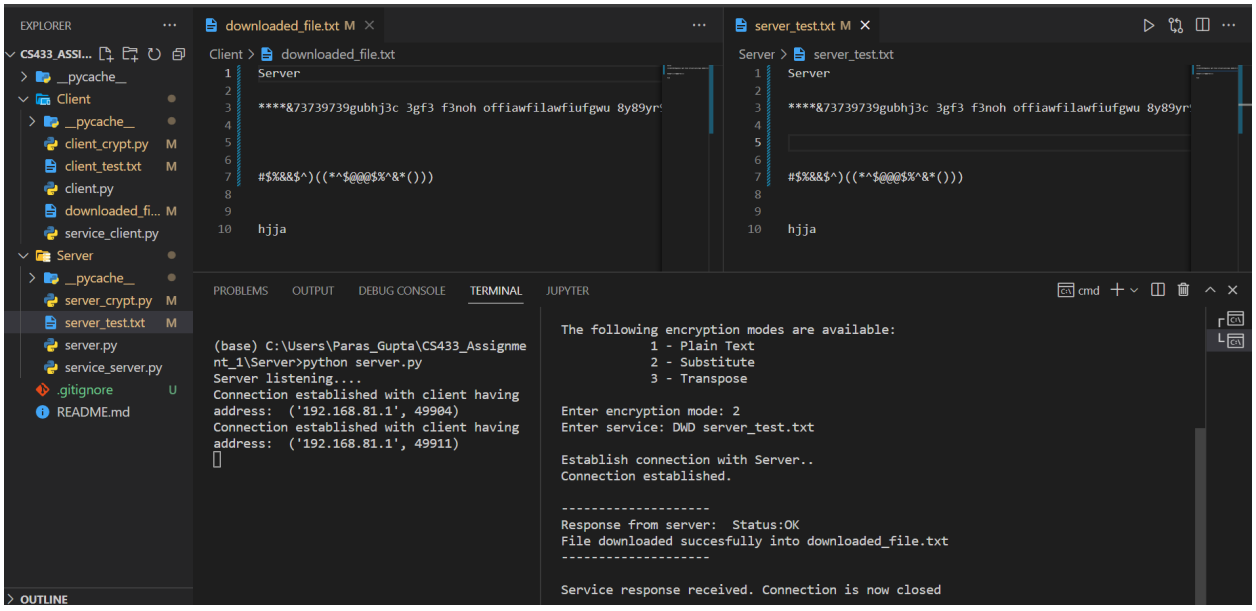


The current working directory of the server is requested to be changed. Response status is shown to be OK which means that the task has been completed successfully. This can be verified by the following CWD service request as shown below.



We can see the current working directory has indeed been changed.

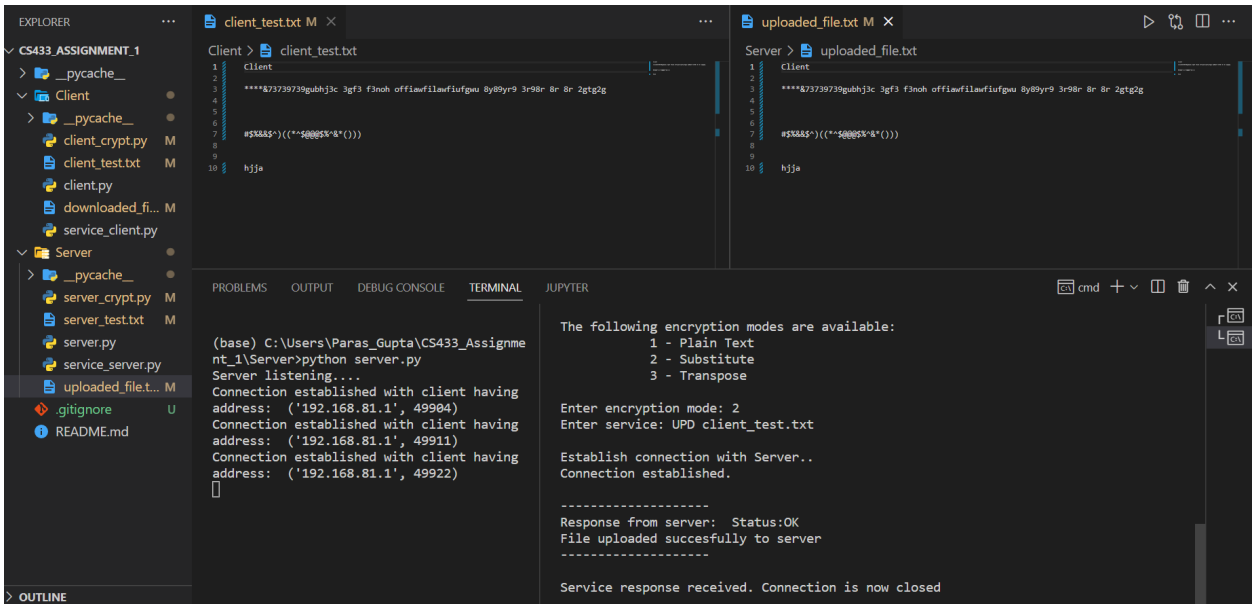
4. DWD <file>



A file present in the server directory is requested for download. Response status is shown to be OK which means that the task has been completed successfully.

We can see in the editor files open that the file has indeed been downloaded to a file named downloaded_file.txt. Note that the file was not present earlier.

5. **UPD <file>**



A file in the client directory is requested to be uploaded to the server. Response status is shown to be OK which means that the task has been completed successfully.

We can see in the editor files open that the file has indeed been uploaded to a file named uploaded_file.txt. Note that the file was not present earlier.

(c) Wireshark dump and Analysis

The Wireshark dump and analysis indicating that data was correctly encrypted before requesting a service (client) and before sending the response (server) has been done for all the three encryption modes as below:

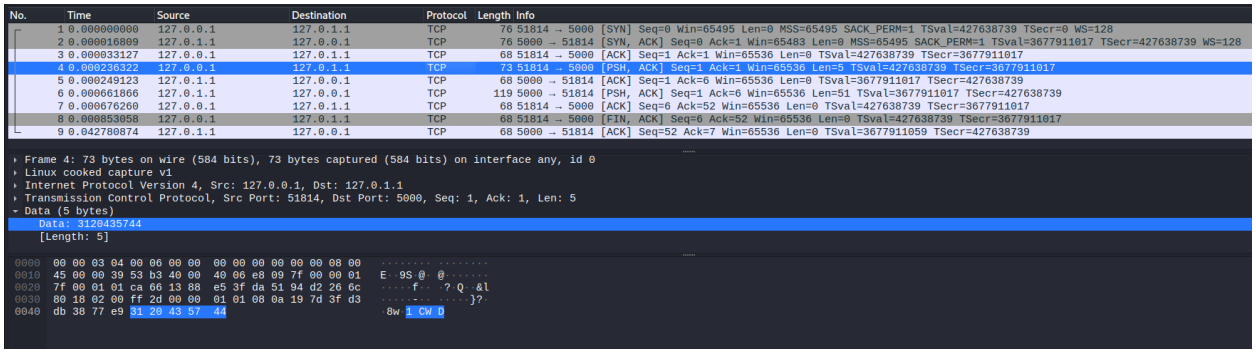
1. **Plain Text**

```
┌──(kali㉿kali)-[~/Downloads/CS433_Assignment_1-main/Server]
└─$ python server.py
Server listening....
Connection established with client having address:  ('127.0.0.1', 51814)

Encrypted Service Response
Response:  /home/kali/Downloads/CS433_Assignment_1-main/Server
Hex Response  2f686f6d652f6b616c692f446f776e6c6f6164732f43533433335f41737369676e6d656e745f312d6d61696e2f536572766572
```

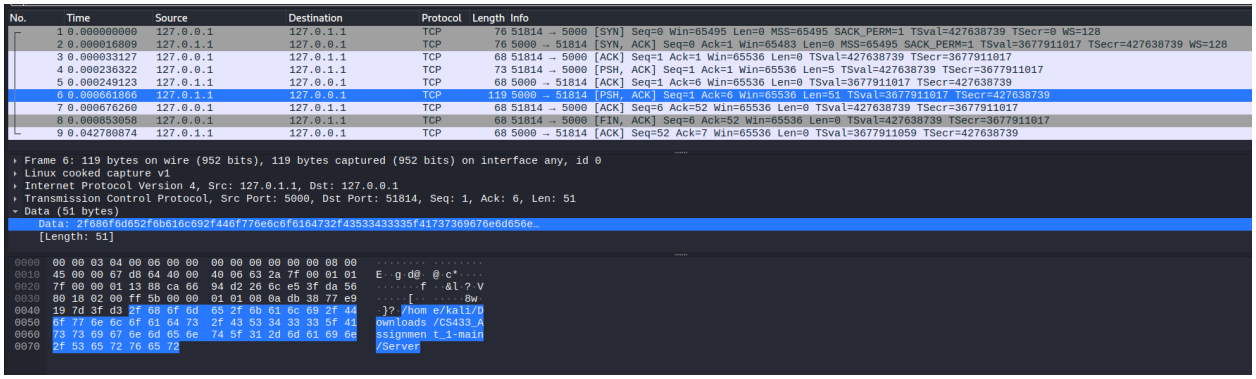
(1a) Server Response Encryption

```
┌──(kali㉿kali)-[~/Downloads/CS433_Assignment_1-main/Client]
└─$ python client.py

The following services can be requested from the server:
    1. CWD - Retrieve the path of the current working directory for the user
    2. LS - List the files/folders present in the current working directory
    3. CD <dir> - Change the directory to <dir> as specified by the client
    4. DWD <file> - Download the <file> specified by the user on server to client
    5. UPD <file> - Upload the <file> on client to the remote server in CWD

The following encryption modes are available:
        1 - Plain Text
        2 - Substitute
        3 - Transpose

Enter encryption mode: 1
Enter service: CWD

Encrypted Service Request
Request:  1 CWD
Hex Request:  3120435744


Establish connection with Server..
Connection established.

____________________
Response from server:  /home/kali/Downloads/CS433_Assignment_1-main/Server
____________________

Service response received. Connection is now closed
```

(1b) Client Response Encryption



(1c) Wireshark Request Packet Capture



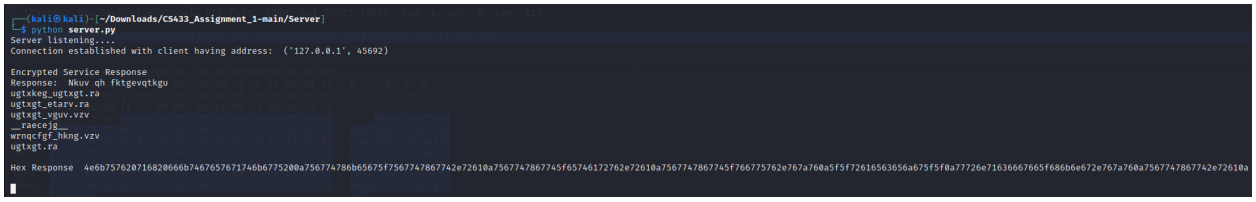
(1d) Wireshark Response Packet Capture

From the **(1a)** and **(1b)** screenshots above, we can see how the request and response messages were encrypted using **Plain Text encryption mode,** that is data sent as it is, before sending and receiving them across the network.

From the **(1c)** and **(1d)** screenshots above, we can see how the request and response messages were transmitted in the network, as captured by Wireshark.

It can be observed that encrypted request and response messages are the same when transmitted in the network as packets, indicating that the data was correctly encrypted.

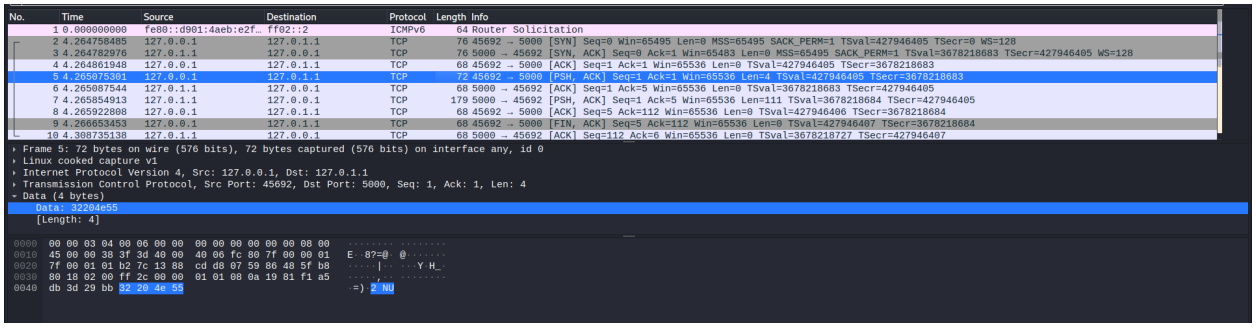
2. Substitute



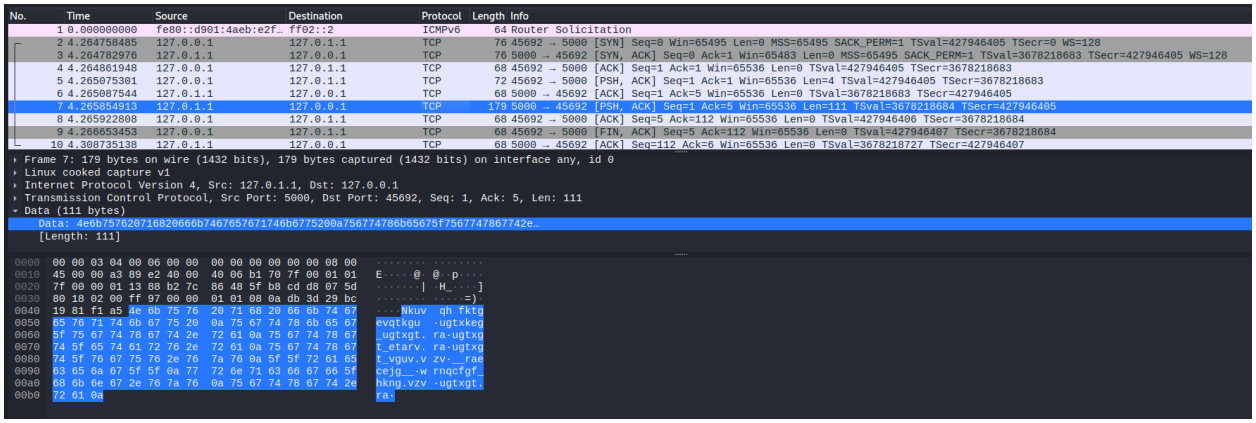
(2a) Server Response Encryption

```
┌──(kali㉿kali)-[~/Downloads/CS433_Assignment_1-main/Client]
└─$ python client.py

The following services can be requested from the server:
    1. CWD - Retrieve the path of the current working directory for the user
    2. LS - List the files/folders present in the current working directory
    3. CD <dir> - Change the directory to <dir> as specified by the client
    4. DWD <file> - Download the <file> specified by the user on server to client
    5. UPD <file> - Upload the <file> on client to the remote server in CWD

The following encryption modes are available:
            1 - Plain Text
            2 - Substitute
            3 - Transpose

Enter encryption mode: 2
Enter service: LS

Encrypted Service Request
Request:  2 NU
Hex Request:  32204e55


Establish connection with Server..
Connection established.

____________________
Response from server:  List of directories
service_server.py
server_crypt.py
server_test.txt
__pycache__
uploaded_file.txt
server.py

____________________

Service response received. Connection is now closed
```

(2b) Client Response Encryption



(2c) Wireshark Request Packet Capture



(2d) Wireshark Response Packet Capture

From the **(2a)** and **(2b)** screenshots above, we can see how the request and response messages were encrypted using **Substitute encryption mode**, that is data sent with alphanumeric characters substituted with a fixed offset, before sending and receiving them across the network.

From the **(2c)** and **(2d)** screenshots above, we can see how the request and response messages were transmitted in the network, as captured by Wireshark.

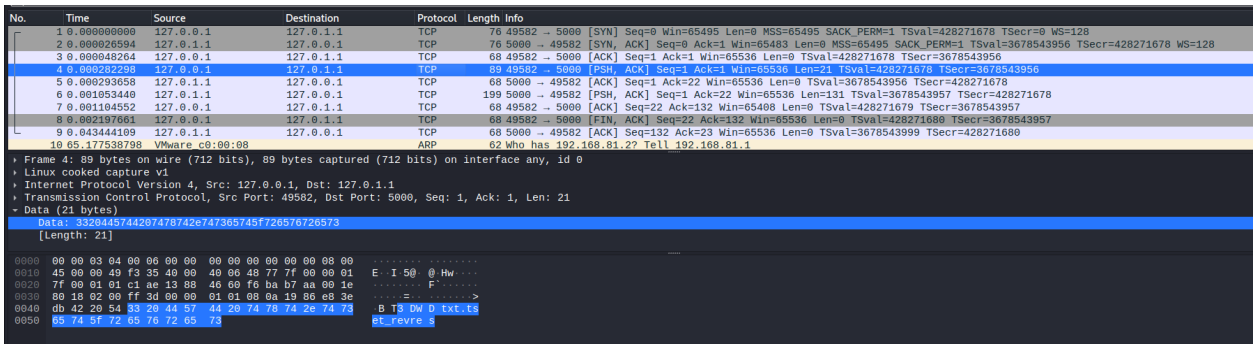
It can be observed that encrypted request and response messages are the same when transmitted in the network as packets, indicating that the data was correctly encrypted.

3. Transpose

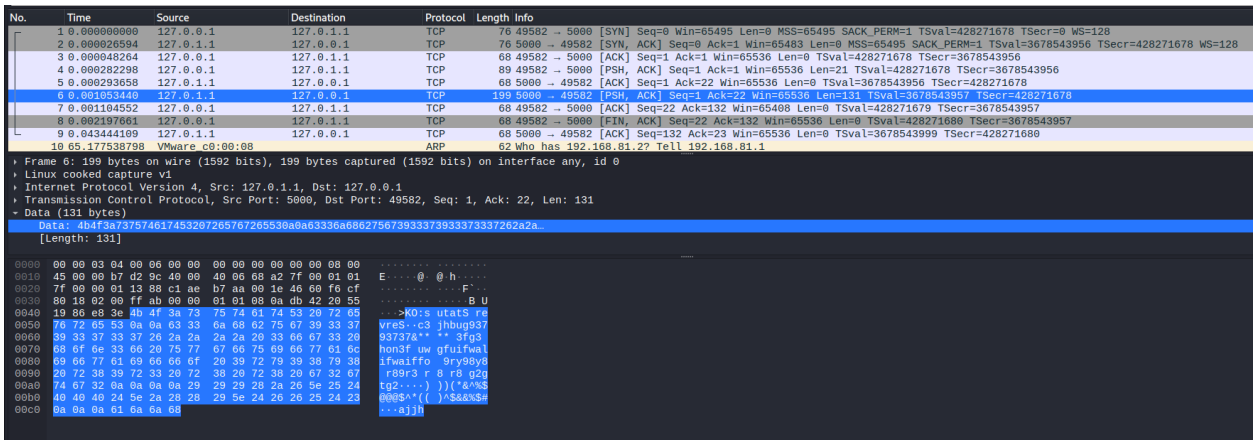
(3a) Server Response Encryption

```
┌──(kali㉿kali)-[~/Downloads/CS433_Assignment_1-main/Client]
└─$ python client.py

The following services can be requested from the server:
    1. CWD - Retrieve the path of the current working directory for the user
    2. LS - List the files/folders present in the current working directory
    3. CD <dir> - Change the directory to <dir> as specified by the client
    4. DWD <file> - Download the <file> specified by the user on server to client
    5. UPD <file> - Upload the <file> on client to the remote server in CWD

The following encryption modes are available:
            1 - Plain Text
            2 - Substitute
            3 - Transpose

Enter encryption mode: 3
Enter service: DWD server_test.txt

Encrypted Service Request
Request:  3 DWD txt.tset_revres
Hex Request:  33204457442074787 42e747365745f726576726573


Establish connection with Server..
Connection established.

____________________
Response from server:  Status:OK
File downloaded succesfully into downloaded_file.txt
____________________

Service response received. Connection is now closed
```

(3b) Client Response Encryption



(3c) Wireshark Request Packet Capture



(3d) Wireshark Response Packet Capture

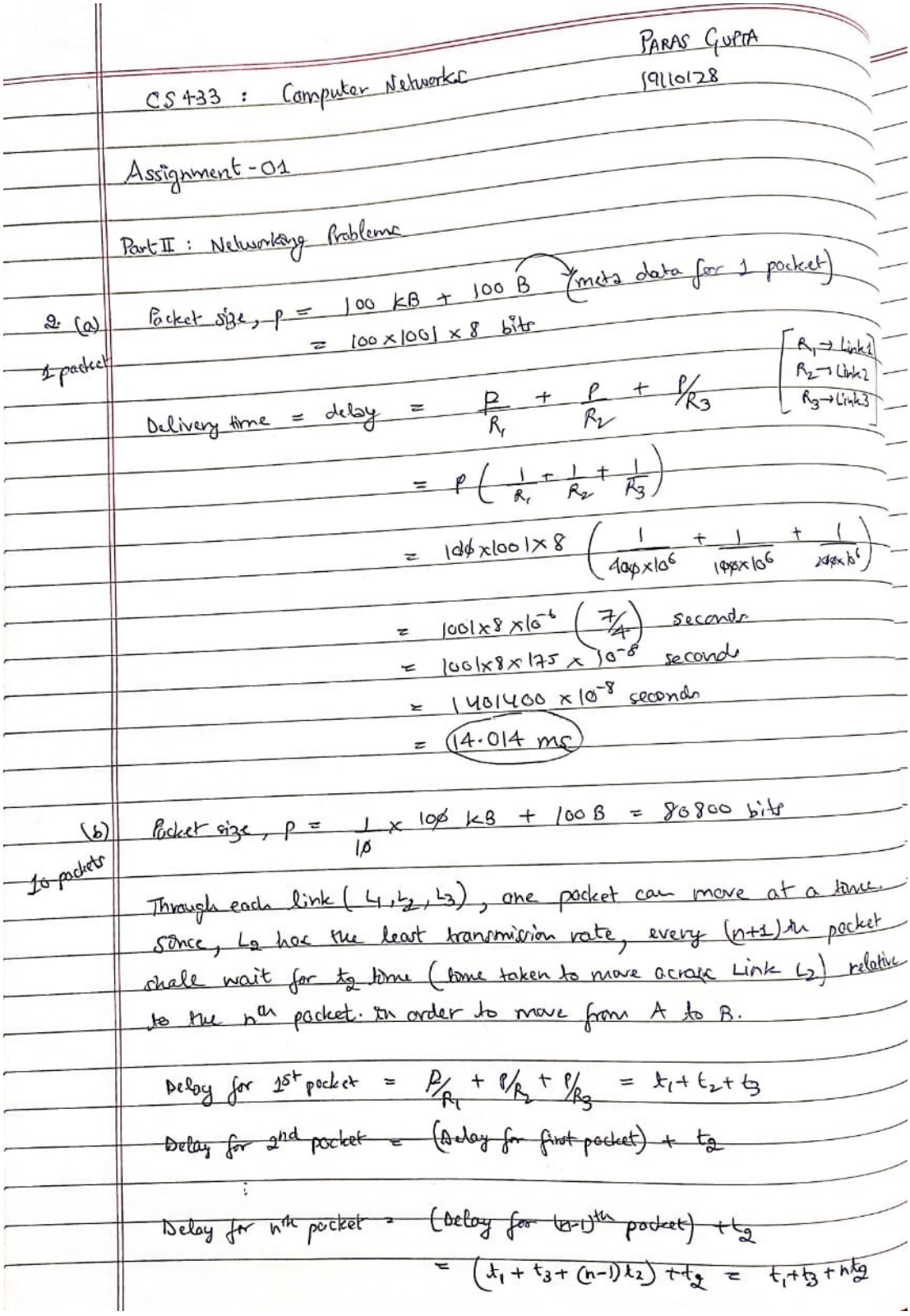
From the **(3a)** and **(3b)** screenshots above, we can see how the request and response messages were encrypted using **Transpose encryption mode**, that is reversing the contents in a word-by-word manner, before sending and receiving them across the network.

From the **(3c)** and **(3d)** screenshots above, we can see how the request and response messages were transmitted in the network, as captured by Wireshark.

It can be observed that encrypted request and response messages are the same when transmitted in the network as packets, indicating that the data was correctly encrypted.

Part II: Networking Problems

Q2.



CS433 : Computer Networks — PARAS GUPTA 19110128

Assignment - 01

Part II : Networking Problems

2. (a) 1 packet

Packet size, $P = 100\ kB + 100\ B$ (meta data for 1 packet)
$= 100 \times 1001 \times 8$ bits

Delivery time = delay $= \frac{P}{R_1} + \frac{P}{R_2} + \frac{P}{R_3}$

[$R_1 \rightarrow$ Link1, $R_2 \rightarrow$ Link2, $R_3 \rightarrow$ Link3]

$= P\left(\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}\right)$

$= 100 \times 1001 \times 8 \left(\frac{1}{100 \times 10^6} + \frac{1}{100 \times 10^6} + \frac{1}{200 \times 10^6}\right)$

$= 1001 \times 8 \times 10^{-6} \left(\frac{7}{4}\right)$ seconds

$= 1001 \times 8 \times 175 \times 10^{-8}$ seconds

$= 1401400 \times 10^{-8}$ seconds

$= 14.014$ ms

(b) 10 packets

Packet size, $P = \frac{1}{10} \times 100\ kB + 100\ B = 80800$ bits

Through each link (L_1, L_2, L_3), one packet can move at a time. Since, L_2 has the least transmission rate, every $(n+1)^{th}$ packet shall wait for t_2 time (time taken to move across Link L_2) relative to the n^{th} packet. In order to move from A to B.

Delay for 1st packet $= \frac{P}{R_1} + \frac{P}{R_2} + \frac{P}{R_3} = t_1 + t_2 + t_3$

Delay for 2nd packet = (Delay for first packet) + t_2

⋮

Delay for n^{th} packet = (Delay for $(n-1)^{th}$ packet) + t_2

$= (t_1 + t_3 + (n-1)t_2) + t_2 = t_1 + t_3 + nt_2$

Delivery time = Delay for last packet

$= t_1 + t_3 + n t_2$

Here, $n = 10$

$\therefore$ Delivery time $= P/R_1 + P/R_3 + 10 \times P/R_2$

$= 8080 \left(\frac{1}{100 \times 10^6} + \frac{1}{200 \times 10^6} + 10 \times \frac{1}{400 \times 10^6} \right)$

$= 808 \times 10^{-6} \times \left(\frac{1}{4} + \frac{1}{2} + 10 \right)$

$= \frac{202}{808} \times \frac{43}{4} \times 10^{-6}$

$= 8686 \times 10^{-6} = 8.686$ ms

(c) 50 packets

Packet size, $P = \frac{1}{50} \times 100$ kB + 100 B = 16800 bits

Delivery time $= t_1 + t_3 + 50 \times t_2$

$= P/R_1 + P/R_2 + P/R_3 \times 50$

$= 16800 \left(\frac{1}{4} + \frac{1}{2} + 50 \right) \times 10^{-6} \times \frac{1}{100}$

$= \frac{4.2}{16.8} \times \frac{203}{4} \times 10^{-3} = 8.526$ ms

(d) 100 packets

Packet size, $P = \frac{1}{100} \times 100$ kB + 100B = 8800 bits

Delivery time $= \frac{P}{R_1} + \frac{P}{R_2} + 100 \times P/R_3$

$= 8800 \times 10^{-6} \times \frac{1}{100} \left(\frac{1}{4} + \frac{1}{2} + 100 \right)$

$= \frac{22}{88} \times 10^{-6} \times \frac{403}{4}$

$= 8.866$ ms

$\Rightarrow$ Lowest Delivery time is for **50 packets** $\rightarrow$ answer

Q3. & Q4.

3.

(a) Propagation delay = $\frac{\text{length}}{\text{speed}} = \frac{10^5 \times 1000 \text{ m}}{\frac{1}{5} \times 3 \times 10^8 \text{ m/s}} = 50 \times 10^{-6} \text{ s} = 50 \mu s$

(b) Maximum number of bits that can be sent between R_1 and R_2 in the time taken by the first bit to reach R_2 (or 50 us)

= (Bandwidth) × (time taken by first bit)

= $(100 \times 10^9 \text{ bits/s}) \times (50 \times 10^{-6} \text{ s})$

= 5000×10^3 bits

= 5×10^6 bits = 5 Mb

(c) Bit width = (length) / (maximum number of bits)

= $\frac{10^2 \times 1000 \text{ m}}{5 \times 10^6 \text{ bits}} = 2 \times 10^{-3} \text{ m/bits}$

= 2 mm / bit

4.

RTT = 10 ms (round trip)

(a) Non-persistent

Let the file transmission time be 't' seconds for each KB of data.

Page load time = [Request time] + [Response time]

= [(#requests) × (RTT) + (# requests) × (RTT)] + [t + 10 × 100 × t]

- (#requests) × (RTT): non-persistence requiring connection to be set for each request
- (# requests) × (RTT): time to send each request
- t: Response time for webpage
- 10 × 100 × t: Response time for all objects

$= (10+1) \times 50 \times 10^{-3} + (10+1) \times 10 \times 10^{-3} + (t + 1000t)$

↳ (# requests = 1 webpage + 10 objects = 11)

$= \frac{11}{100} + \frac{11}{100} + 1001t$

$= 0.22 + 1001t$ s

(b) Persistent

Again, let file transmission time = t seconds for 1 kB

Page load time = [Request time] + [Response time]

$= [(\#\text{requests}) \times (RTT) + RTT] + [t + 10 \times 100t]$

(# requests = 11; RTT: Persistent connection requiring connection to be set only once)

$= [2 \times 10 \times 10^{-3}] + [1001t]$

$= 0.19 + 1001t$ s

(c)

Let file transmission time = t seconds for 1 kB of data

Page load time = [Initial connection time] + [Request + Response time for web-page] + [Request + Response time for objects]

$= [RTT] + [RTT + t] + [RTT + 10 \times 100 \times t]$

$= 3 \times RTT + t + 1000t$

$= 3 \times 10 \times 10^{-3} + 1001t$

$= 0.02 + 1001t$ s

Part III: Network Tools
Q5.

(a)

1. **ICMPv6** - It is the implementation of the Internet Control Manager Protocol for IPv6. It performs several functions when data is transmitted from one end to the other including router discovery, error reporting, diagnostics, etc. It is used by routers, hosts, and other intermediary devices to other devices.

   Associated RFC Number: **4443**

2. **mDNS** - mDNS or Multicast DNS is an alternative to the DNS protocol for resolving hostnames to IP addresses in smaller networks. It uses the same interfaces, formats, and semantics as DNS.

   Associated RFC Number: **6762**

3. **OCSP** - OCSP or Online Certificate Status Protocol is used to determine the revocation status of an identified X.509 digital certificate in real time. It is useful in time-sensitive requests such as bank transactions.

   Associated RFC Number: **6960**

4. **SSDP** - SSDP or Simple Service Discovery Protocol is used in small networks such as home networks to advertise and discover network services without configuration mechanisms such as DNS. It is a simple text-based protocol that uses UDP as the transport protocol.

   Associated RFC Number: **NO RFC**

5. **TLSv1.2** - TLS or Transport Layer Security is a protocol used to provide secure communications across a network. It is majorly used to secure HTTPS, but also has applications in services such as email, voice over IP, and messaging.

   Associated RFC Number: **5246**

(b)

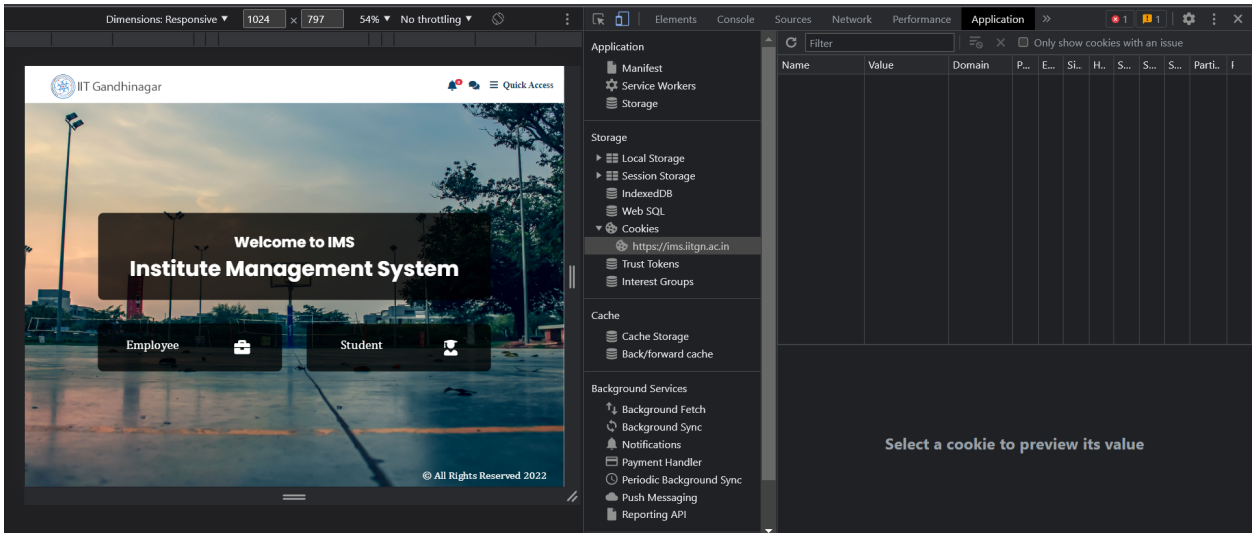
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.81.129	192.168.81.2	DNS	71	Standard query 0xf348 A yahoo.com
2	0.000182561	192.168.81.129	192.168.81.2	DNS	71	Standard query 0xba56 AAAA yahoo.com
3	0.005065982	192.168.81.2	192.168.81.129	DNS	449	Standard query response 0xf348 A yahoo.com A 74.6.231.20 A 74.6.231.2
4	0.005066461	192.168.81.2	192.168.81.129	DNS	521	Standard query response 0xba56 AAAA yahoo.com AAAA 2001:4998:44:3507:
5	0.005989090	192.168.81.129	74.6.231.20	ICMP	100	Echo (ping) request id=0xc3d3, seq=1/256, ttl=64 (reply in 6)
6	0.360639046	74.6.231.20	192.168.81.129	ICMP	100	Echo (ping) reply id=0xc3d3, seq=1/256, ttl=128 (request in 5)
7	0.361472761	192.168.81.129	192.168.81.2	DNS	86	Standard query 0x18b2 PTR 20.231.6.74.in-addr.arpa
8	0.454753312	192.168.81.2	192.168.81.129	DNS	428	Standard query response 0x18b2 PTR 20.231.6.74.in-addr.arpa PTR media

Let us estimate the RTT for the connection between request and reply packets with numbers **5** and **6** respectively in the above screenshot.

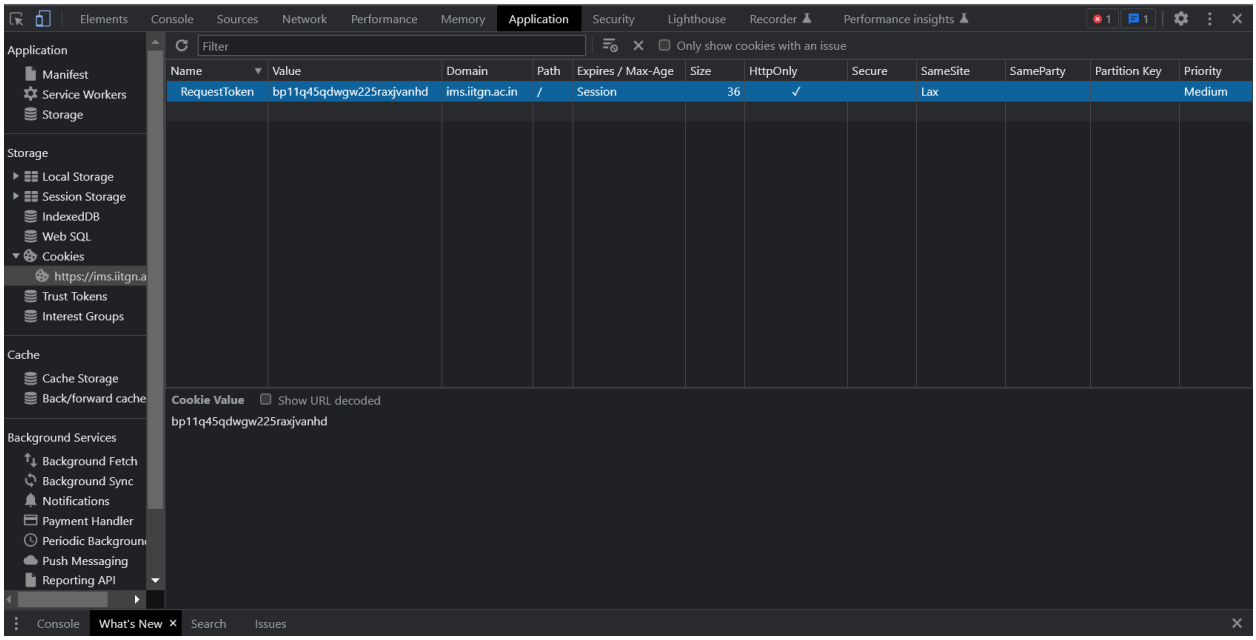
*RTT = (Time for **6**) - (Time for **5**) = (0.361 seconds) - (0.006 seconds)*
= 0.355 seconds
*= **355 milliseconds***

(c)

No cookies are found when we only visit the webpage https://ims.iitgn.ac.in/.



The following cookie “RequestToken” is found when I logged in to the IMS Student Portal.



The following are the characteristics of the “RequestToken” cookie:

- The cookie is created as soon as the person logs in and expires when we log out. It is a **Session** cookie.
- The cookie's path is ‘\’ which makes it available to the entire web pages.
- The size of the cookie is 36 bytes.
- The cookie is HttpOnly, which does not allow client-side scripts to access the cookie’s data.
- The SameSite attribute does not allow the browser to send cookies' data to cross-site requests. The lax value of the attribute allows a balance of security and usability.
