```
+-----------------------+
|        CS 433         |
| Assignment 1:Problem 1 |
|   DESIGN DOCUMENT      |
+-----------------------+
```

Name : Lipika Rajpal Roll Number : 20110102

Overview of the solution:

As a solution to problem statement 1, a full-fledged socket-client application has been built. This application is written in the python language. It uses socket programming principles and OS APIs to create a network that enables file transfer and command execution between a server and a client.

The application consists of the following components/files:

1. A python file 'server.py' works as a server on the network.
2. A python file 'client.py' works as a client on the network.
3. The TCP protocol is used at the transport layer.

The application allows the following commands:

CMD	Description	Status
CWD	Retrieve the path of the current working directory for the user	
LS	List the files/folders present in the current working directory	
CD <dir>	Change the directory to <dir> as specified by the client	OK/NOK
DWD <file>	Download the <file> specified by the user on server to client	OK/NOK
UPD <file>	Upload the <file> on client to the remote server in CWD	OK/NOK

The client can request any of the above services from the server.

MODES OF LAYERING

1. **File service Layer:**

The file service layer enables the client to request services from the server.
The server responds by executing the requested commands like some OS APIs :

- **LS:** The server procures the list of all the folders in the current working directory of the server and transfers it to the client.

```python
    if (job == "LS"):
        res = ' '.join(os.listdir())

        send_encryp(res)

        print("[SERVER] Service Provided")
```

Fig 1: Implementation of the response to the ‘LS’ command in server.py

- **CWD:** The server responds by sending the current working directory of the server to the client.

```python
    if (job == "CWD"):
        res = os.getcwd()

        send_encryp(res)

        print("[SERVER] Service Provided")
```

Fig 2: Implementation of the response to the ‘CWD’ command in server.py

- **CD <dir>:** The server changes its current directory to the directory mentioned by the client in the request(<dir>)

```python
    if (job[0:2] == "CD"):
        dir = ""
        for i in range(3, len(job)):
            dir += job[i]
        dir_exist = os.path.exists(dir)
        if (dir_exist):
            os.chdir(dir)
            confirmation = "OK"
        else:
            confirmation = "NOK"

        send_encryp(confirmation)
        # c.send(confirmation.encode())
        print(os.getcwd())
        print("[SERVER] Service Provided")
```

Fig 3: Implementation of the response to the ‘CD <dir>’ call in server.py

<u>The file transfer functionality between the server and the client:</u>

UPD <file>:

- This service enables the client to upload any file (.txt, .png, etc.) to the current directory of the server.

- If the file gets uploaded successfully, the server responds by sending **'STATUS: OK'** to the client. If, somehow, the file upload fails, '**STATUS: NOK**' is sent.

Overview of the service using an example:

For instance, the client wants to upload the file with the name *'abc.txt'* to the server.

We will go with the intuitive notion that the file should exist if the client wants to upload some file. Therefore, the client first checks if the file exists using an OS API - os.file.exists. If it returns true, the file exists and sends a positive response to the server to continue the upload process. Else, it sends a message "**FAIL**" to the server. If the server gets a 'FAIL' message, it responds to the client by a status code 'NOK' and ends this service.
However, if the file exists, the client starts reading the file's contents as bytes. It sends this content to the server in several packets of 2048 bytes. A while loop over the condition that the file is not yet fully read ensures that all the file's data is read and sent to the server.
The server receives these packets in succession and writes them in a file with the name : '*name of the original file + from_CLIENT.extension*.' In our example, the name of the file received on the server would be '*abc_from_CLIENT.txt*.'

After the server successfully receives the file, it sends the client a confirmation message **'STATUS: OK'**, and the service ends.

Associated challenges while implementing UPD file service:

The sender has to send chunks of 2048 bytes continuously to the server in succession. Due to this, a synchronization problem arose between the sending and receiving packets. Some packets got lost during the transmission. This led to considerable data loss, and the file could not be uploaded successfully.
To ensure a lossless transmission, a feedback mechanism was implemented. After the client sends one packet, it waits for the server to send a confirmation message 'DONE.' Only after this, the client sends the next packet.

```
while rem_size>=0:
    content_h = c.recv(2054)

    mode_encryption = content_h[0] - 48
    content = content_h[1:]

    if (mode_encryption == 1):
        content = encrypt_cipher(content, -1*shift_cipher)
    if (mode_encryption == 2):
        content= transpose(content)


    file.write(content)
    send_encryp("DONE")
    rem_size -= 2048
```

Fig 4: The while loop that ensures lossless arrival of file data in server.py

DWD <file>:

- This service enables the client to download any file (.txt, .png, etc.) from the server to the current working directory of the client.
- The client can download a particular file by specifying its path.
- The server confirms if the file download is successful by sending a status code '**OK**.' Else responds with '**STATUS: NOK**.'
- The requested file gets downloaded in the client directory with the name: 'original name + _from_SERVER.extension'

Overview of DWD service with an example:

The implementation of this service is the same as that of the UPD <file> service. The only change is that now we assume that the server should have access to the file required by the client. Hence, the server checks for the existence of the file. If it exists, the process continues. Otherwise, the server responds by a status code 'NOK'and terminates the process.

The server reads and sends the file's contents in chunks of 2048 bytes. A while loop ensures that the client safely receives all the contents. A feedback mechanism similar to the UPD service is implemented here. The server only sends the next chunk of data if the client confirms that it has received the previous chunk.

The feedback mechanism was implemented to **overcome the challenge** of lossless transmission as, without it, a significant number of chunks were getting lost over the network.

```python
        if (file_exist):
            send_encryp("file is there")
            c.recv(1024)
            file= open(file_name, "rb")


            filesize = os.path.getsize(file_name)
            rem_size = filesize
            send_encryp(str(rem_size))
            c.recv(1024)
            while rem_size>=0:
                content = file.read(2048)

                send_encryp(content)
                c.recv(2048)


                rem_size -= 2048


            file.close()

            confirmation = "STATUS : OK, file download successfully"
```

Fig 5: The implementation of the download process if the requested file exists in server.py

2. Encryption Layer:

As required by the problem statement, there are three modes of encryption:

1. Plain text:
   The data is transmitted over the network without any encryption. It is represented by ‘**MODE 0**’ in the application.
2. Caesar cipher:
   Represented by ‘MODE 1’
   The data is encrypted before transmitting over the network. There is a shift factor set as an integer (ex. 2). Each alphanumeric character in the data is converted to the ASCII character, with the ASCII value having an offset equal to the shift factor.
   The sender encrypts the message, i.e., shifts the alphanumeric characters with an offset of shift factor, say N.
   The receiver decrypts the data by shifting the characters with an offset of -1*shift factor or -N.

Hence, because of the above logic, a single function can be used for both encryption and decryption of the data.

```python
def encrypt_cipher(text, shift):

    if (type(text) is bytes):
        text = list(text)
        for i in range(0, len(text)):
            text[i] = (text[i] + shift)%256
        return bytes(text)


    result = ""
    for i in text:

        if (i.isdigit()):
            result += chr((ord(i) + shift - 48)%10 + 48)

        elif (i.isupper()):
            result += chr((ord(i) + shift-65) % 26 + 65)

        elif (i.islower()):
            result += chr((ord(i) + shift - 97) % 26 + 97)
        else:
            result += i

    return result
```

Fig 6: The function responsible for the encryption and decryption in MODE 1. This is present in both server.py and client.py

3. <u>Transpose:</u>
   Represented by 'MODE 2'
   The function reverses the content of the message in a word-by-word manner. The same function can do both encryption and decryption.
   For instance, consider the encryption and decryption of the string:
   'the dog' -> 'eht god' -> 'the dog'

```python
def transpose(text):

    if (type(text) is bytes):
        text = list(text)

        text.reverse()
        return bytes(text)

    lines = text.splitlines()
    encrypted_lines = []
    for line in lines:
        result = ""
        words = line.split()
        for word in words:
            result += " "
            result += word[::-1]
        encrypted_lines.append(result[1:])

    return '\n'.join(encrypted_lines)
```

Fig 7: The transpose function present in both server.py and client.py

Incorporation of the encryption layer:

We have to make an encryption layer for the application. This means that each message sent over the network has to be encrypted. Along with this, a header should be attached to the data containing the information about the mode of encryption. The program/host at the receiving end will read this header, remove it and decrypt the message according to the information in the header.
This is ensured by a custom function to send the data over the network:

```python
def send_encryp(text):

    if (mode_encryption == 1):
        text= encrypt_cipher(text, shift_cipher)
    if (mode_encryption == 2):
        text= transpose(text)

    if (type(text) is not bytes):
        msg = str(mode_encryption) + text
        c.send(msg.encode())
    else:
        c.send(str(mode_encryption).encode() + text)
```

Fig 8: This function encrypts the data and attaches the suitable header, then sends the data

3. **TCP:**

The Transmission Control Protocol is at the heart of our application. It is the fourth layer in the OSI model of networking. The TCP ensures reliable end-to-end communication between the client and the server.
It is the second layer of the TCP/IP model. It acts as an intermediary between the application layer and the network layer.
The TCP uses a handshake protocol to establish a connection between two hosts.
We have used TCP as the transport layer protocol to ensure a safe/lossless transmission between the hosts.

Snapshots of the commands:

1. **LS**

```
BYE                                          python -u "c:\Users\hii
\Documents\CN_SEM5_ASS1\server.py"ASS1>
[SERVER]: Socket successfully created
[SERVER]: socket binded to 50345
[SERVER]: got connected to ('127.0.0.1', 56755)
[CLIENT]  LS
[SERVER] Service Provided
```

```
BYE
PS C:\Users\hii\Documents\CN_SEM5_ASS1> python client.py
[SERVER] : [SERVER]: HELLO! Please enter the commands below
Enter the command :LS
[SERVER] :
actual_out_from_CLIENT.txt actual_out_from_SERVER.txt client.py
 download_file.txt download_file_from_SERVER.txt ex1.png ex1_fr
om_CLIENT.png ex1_from_SERVER.png server.py tempCodeRunnerFile.
py upload_file.txt upload_file_from_CLIENT.txt
Enter the command :
```

Fig 9

2. **CWD**

```
[SERVER]: Socket successfully created
[SERVER]: socket binded to 50345
[SERVER]: got connected to ('127.0.0.1', 56767)
[CLIENT]  CWD
[SERVER] Service Provided
```

```
PS C:\Users\hii\Documents\CN_SEM5_ASS1> python client.py
[SERVER] : [SERVER]: HELLO! Please enter the commands below
Enter the command :CWD
[SERVER] :
C:\Users\hii\Documents\CN_SEM5_ASS1
```

Fig 10

3. **CD <dir>**
   Here, CD C:\Users\hii\Documents\Oj

```
[SERVER] Service Provided
[CLIENT]  CD C:\Users\hii\Documents\OJ
C:\Users\hii\Documents\OJ
[SERVER] Service Provided
[CLIENT]  LS
[SERVER] Service Provided
```

```
Enter the command :CD C:\Users\hii\Documents\OJ
[SERVER] :
OK
Enter the command :LS
[SERVER] :
actual_out.txt actual_out_from_CLIENT.txt input.txt out.txt
```

Fig 11

4. **UPD <file>**
   Here we upload a file with the name: ‘ex1.png’

Fig 12: ex1.png

```
[CLIENT]  UPD ex1.png
ex1_from_CLIENT.png
[SERVER] Service Provided
```

```
Enter the command :MODE 1
Enter the command :UPD ex1.png
[SERVER] :
STATUS : OK, file upload successful
Enter the command :
```

Fig 13: We have also changed the mode to MODE 1. And uploaded the file 'ex1.png'

Fig 14: ex1_from_CLIENT.png
The file received by the server

5. **DWD <file>**
   Here, the client wants to download a text file 'download_file.txt'

```
[SERVER] STATUS OK, Service Provided
[CLIENT]  DWD C:\Users\hii\Documents\CN
_SEM5_ASS1\download_file.txt
[SERVER] STATUS OK, Service Provided
```

```
Enter the command :DWD C:\Users\hii\Documents\CN_SEM5_ASS1\download_file.txt
download_file_from_SERVER.txt
[SERVER] :
STATUS : OK, file download successfully
```

Fig 15

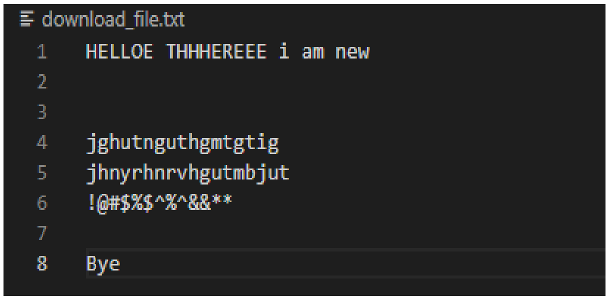

Fig 16: Original file

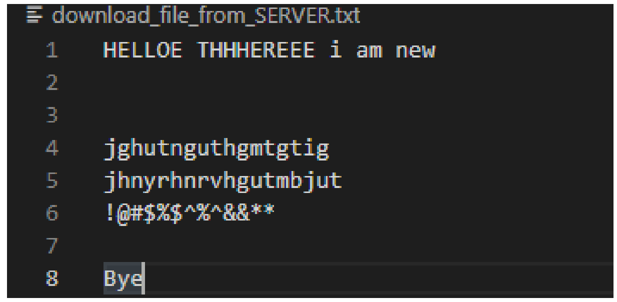

Fig 17: File received by the client

WIRESHARK analysis indicating the correct encryption by the encryption layer-

1. *The command LS in plain text/MODE 0 indicates no encryption:*

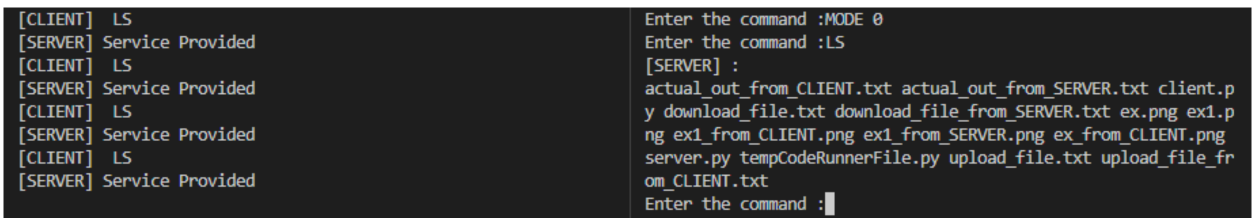

Fig: 18

Wireshark information:

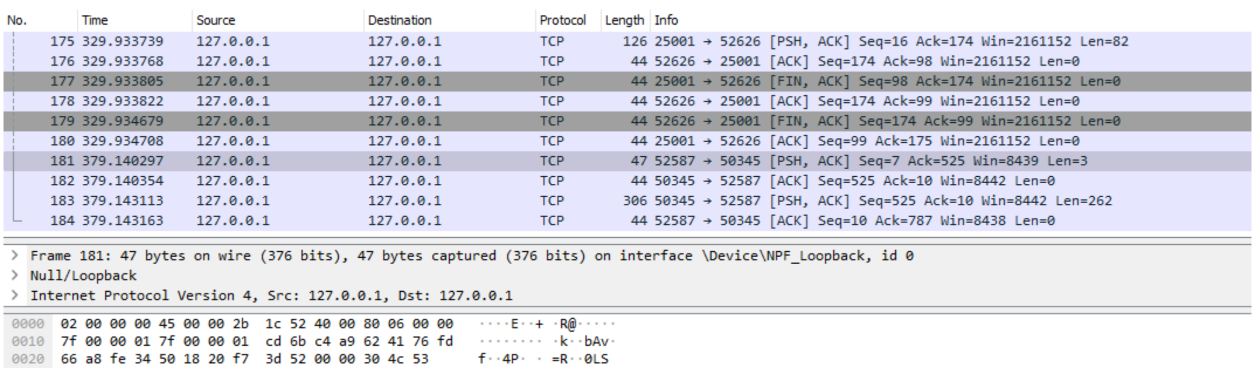

Fig 19: We can see that the command LS is sent as it is, without any encryption.

No.	Time	Source	Destination	Protocol	Length	Info
175	329.933739	127.0.0.1	127.0.0.1	TCP	126	25001 → 52626 [PSH, ACK] Seq=16 Ack=174 Win=2161152 Len=82
176	329.933768	127.0.0.1	127.0.0.1	TCP	44	52626 → 25001 [ACK] Seq=174 Ack=98 Win=2161152 Len=0
177	329.933805	127.0.0.1	127.0.0.1	TCP	44	25001 → 52626 [FIN, ACK] Seq=98 Ack=174 Win=2161152 Len=0
178	329.933822	127.0.0.1	127.0.0.1	TCP	44	52626 → 25001 [ACK] Seq=174 Ack=99 Win=2161152 Len=0
179	329.934679	127.0.0.1	127.0.0.1	TCP	44	52626 → 25001 [FIN, ACK] Seq=174 Ack=99 Win=2161152 Len=0
180	329.934708	127.0.0.1	127.0.0.1	TCP	44	25001 → 52626 [ACK] Seq=99 Ack=175 Win=2161152 Len=0
181	379.140297	127.0.0.1	127.0.0.1	TCP	47	52587 → 50345 [PSH, ACK] Seq=7 Ack=525 Win=8439 Len=3
182	379.140354	127.0.0.1	127.0.0.1	TCP	44	50345 → 52587 [ACK] Seq=525 Ack=10 Win=8442 Len=0
183	379.143113	127.0.0.1	127.0.0.1	TCP	306	50345 → 52587 [PSH, ACK] Seq=525 Ack=10 Win=8442 Len=262
184	379.143163	127.0.0.1	127.0.0.1	TCP	44	52587 → 50345 [ACK] Seq=10 Ack=787 Win=8438 Len=0

```
> Frame 183: 306 bytes on wire (2448 bits), 306 bytes captured (2448 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

0000  02 00 00 00 45 00 01 2e  1c 54 40 00 80 06 00 00   ····E··. ·T@·····
0010  7f 00 00 01 7f 00 00 01  c4 a9 cd 6b 66 a8 fe 34   ········ ···kf··4
0020  62 41 77 00 50 18 20 fa  c1 ae 00 00 30 61 63 74   bAw·P· · ····0act
0030  75 61 6c 5f 6f 75 74 5f  66 72 6f 6d 5f 43 4c 49   ual_out_ from_CLI
0040  45 4e 54 2e 74 78 74 20  61 63 74 75 61 6c 5f 6f   ENT.txt  actual_o
0050  75 74 5f 66 72 6f 6d 5f  53 45 52 56 45 52 2e 74   ut_from_ SERVER.t
0060  78 74 20 63 6c 69 65 6e  74 2e 70 79 20 64 6f 77   xt clien t.py dow
0070  6e 6c 6f 61 64 5f 66 69  6c 65 2e 74 78 74 20 64   nload_fi le.txt d
0080  6f 77 6e 6c 6f 61 64 5f  66 69 6c 65 5f 66 72 6f   ownload_ file_fro
0090  6d 5f 53 45 52 56 45 52  2e 74 78 74 20 65 78 2e   m_SERVER .txt ex.
00a0  70 6e 67 20 65 78 31 2e  70 6e 67 20 65 78 31 5f   png ex1. png ex1_
00b0  66 72 6f 6d 5f 43 4c 49  45 4e 54 2e 70 6e 67 20   from_CLI ENT.png
00c0  65 78 31 5f 66 72 6f 6d  5f 53 45 52 56 45 52 2e   ex1_from _SERVER.
00d0  70 6e 67 20 65 78 5f 66  72 6f 6d 5f 43 4c 49 45   png ex_f rom_CLIE
00e0  4e 54 2e 70 6e 67 20 73  65 72 76 65 72 2e 70 79   NT.png s erver.py
00f0  20 74 65 6d 70 43 6f 64  65 52 75 6e 6e 65 72 46    tempCod eRunnerF
0100  69 6c 65 2e 70 79 20 75  70 6c 6f 61 64 5f 66 69   ile.py u pload_fi
0110  6c 65 2e 74 78 74 20 75  70 6c 6f 61 64 5f 66 69   le.txt u pload_fi
0120  6c 65 5f 66 72 6f 6d 5f  43 4c 49 45 4e 54 2e 74   le_from_ CLIENT.t
0130  78 74                                              xt
```

Fig 20: The response by the server is also not encrypted

Now, MODE is changed to 1 from 0. It ensures that the data will be encrypted according to the Caesar cipher with a shift factor of 2.
NOTE: the shift factor is hard-coded as 2 in server.py and client.py.
The same request 'LS' is made by the client (refer to Fig 21)

```
[CLIENT]  LS
[SERVER] Service Provided
[CLIENT]  LS
[SERVER] Service Provided
[CLIENT]  LS
[SERVER] Service Provided
[CLIENT]  LS
[SERVER] Service Provided
```

```
Enter the command :MODE 1
Enter the command :LS
[SERVER] :
actual_out_from_CLIENT.txt actual_out_from_SERVER.txt client.p
y download_file.txt download_file_from_SERVER.txt ex.png ex1.p
ng ex1_from_CLIENT.png ex1_from_SERVER.png ex_from_CLIENT.png
server.py tempCodeRunnerFile.py upload_file.txt upload_file_fr
om_CLIENT.txt
Enter the command :
```

Fig 21

Its Wireshark analysis-

No.	Time	Source	Destination	Protocol	Length	Info
219	660.129035	127.0.0.1	127.0.0.1	TCP	77	14517 → 49671 [PSH, ACK] Seq=647 Ack=287 Win=8239 Len=33
220	660.129066	127.0.0.1	127.0.0.1	TCP	44	49671 → 14517 [ACK] Seq=287 Ack=680 Win=1269 Len=0
221	660.129166	127.0.0.1	127.0.0.1	TCP	48	49671 → 14517 [PSH, ACK] Seq=287 Ack=680 Win=1269 Len=4
222	660.129187	127.0.0.1	127.0.0.1	TCP	44	14517 → 49671 [ACK] Seq=680 Ack=291 Win=8239 Len=0
223	660.129347	127.0.0.1	127.0.0.1	TCP	77	29844 → 49672 [PSH, ACK] Seq=439 Ack=764 Win=8375 Len=33
224	660.129392	127.0.0.1	127.0.0.1	TCP	44	49672 → 29844 [ACK] Seq=764 Ack=472 Win=8406 Len=0
241	673.302889	127.0.0.1	127.0.0.1	TCP	47	52587 → 50345 [PSH, ACK] Seq=10 Ack=787 Win=8438 Len=3
242	673.302935	127.0.0.1	127.0.0.1	TCP	44	50345 → 52587 [ACK] Seq=787 Ack=13 Win=8442 Len=0
243	673.304450	127.0.0.1	127.0.0.1	TCP	306	50345 → 52587 [PSH, ACK] Seq=787 Ack=13 Win=8442 Len=262
244	673.304481	127.0.0.1	127.0.0.1	TCP	44	52587 → 50345 [ACK] Seq=13 Ack=1049 Win=8437 Len=0

```
> Frame 241: 47 bytes on wire (376 bits), 47 bytes captured (376 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

0000  02 00 00 00 45 00 00 2b  1c 5e 40 00 80 06 00 00   ····E··+ ·^@·····
0010  7f 00 00 01 7f 00 00 01  cd 6b c4 a9 62 41 77 00   ········ ·k··bAw·
0020  66 a8 ff 3a 50 18 20 f6  39 48 00 00 31 4e 55      f··:P· · 9H··1NU
```

Fig 22

In Fig 22, we can see that the command 'LS' is encrypted to 'NU' over the network. This is because the shift factor is 2. Therefore, L was changed to N, and S was changed to U. Hence, our data is correctly encrypted.

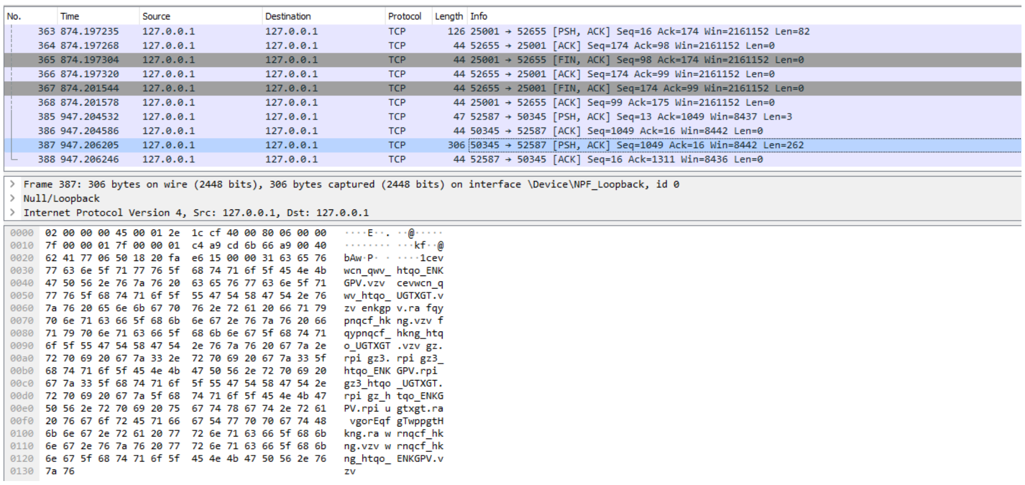

Fig 23: Encrypted response by the server.

Now, the mode is changed to 2, i.e., the data will be reversed in a word-by-word manner(refer to Fig 24).

Fig 24: The mode is changed, and the client wants to download the file 'download_file.txt'

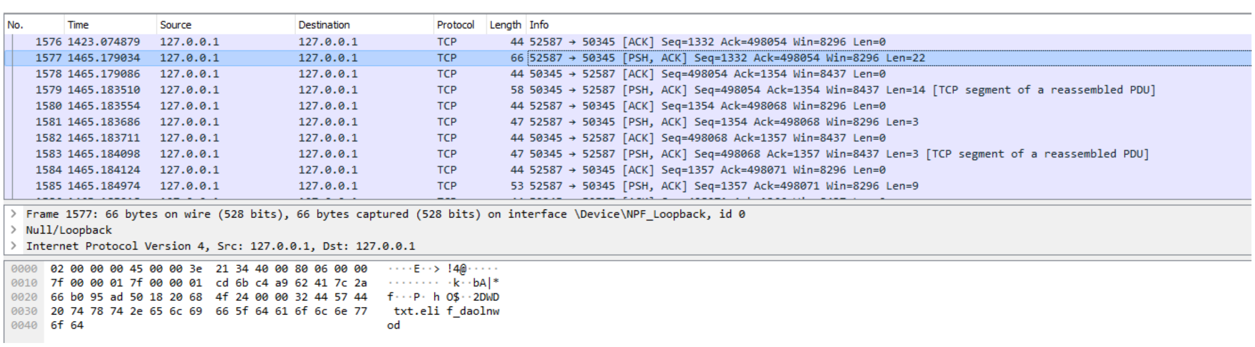

Fig 25

Fig 25 shows that the command was 'DWD download_file.txt.' But on the network, we can see that it got reversed word-by-word as 'DWD txt.elif_doadnwod'
Hence, our data is correctly encrypted.

NOTE: The wireshark dump of the above commands was very huge as I also did some other commands without saving the previous dump.
Hence, the dump provided has the information of the following commands:

Comm 1: LS (MODE = 0)
Comm 2: MODE 1
Comm 3: LS
Comm 4: MODE 2
Comm 5: DWD download_file.txt
Comm 6: UPD ex1.png
Comm 7: BYE (the application closes)

Therefore, the Wireshark dump verifies the correctness of the encryption in all three modes.

***