Al-Imam University
Computer Science Department
College of Computer and Information Sciences
1st Semester 1443-1444 H / Fall 2021

CS 330: Computer Networks

TCP-Based Client/Server Application

(Project Report)

Section Code: 373.
Due date: 11th Dec 2021
Group number: 1

Submitted by:

Student Name	**Student ID**	**Email**
Raghad Khaled Albosais	440020209	rkaalbosais@sm.imamu.edu.sa
Ghadeer Muidh Alshalawi	440020493	gmhalshalawi@sm.imamu.edu.sa
Nouf Mohamed Alajmi	441022333	nomoalajmi@sm.imamu.edu.sa

Supervisor:

Dr. Manan Almusallam

Content table

page

Figures table

page

1. Introduction:

Nowadays, The Internet has become a significant part of the daily life of a modern person. The Transmission Control Protocol (TCP) is one of the main protocols of the Internet protocol suite.

TCP works with the Internet Protocol (IP), which defines how computers send packets of data to each other. Thus, it becomes highly important to secure these data to avoid MITM (Man in the Middle attacks).

Internet-connected applications that is using TCP, need to do its operations through the implementation of sockets in their networking code. Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket (node) listens on a particular port at particular IP address, while the other socket reaches out to the other to form a connection. The server forms the listener socket while the client reaches out to the server.

In this project, we developed a simple TCP-based client-server application that exchanges messages between device-to-device, client-to-server, and vice versa using socket programming with Python language. Where we implemented two modes of exchanging the messages, the open mode, where the client sends and receives the data as cleartext, and the secure mode, where our TCP tunnel is protected using the Advanced Encryption Standard (AES) encryption algorithm to protect the traffic in the transit path.

AES is a symmetric encryption algorithm. It uses the same key in both the client and the server for both encryption and decryption process.

Note: while working on our project, the IP addresses of the devices we used had been changing frequently, as we learned that client/host IP addresses are dynamic, to not get confused while reading the report.

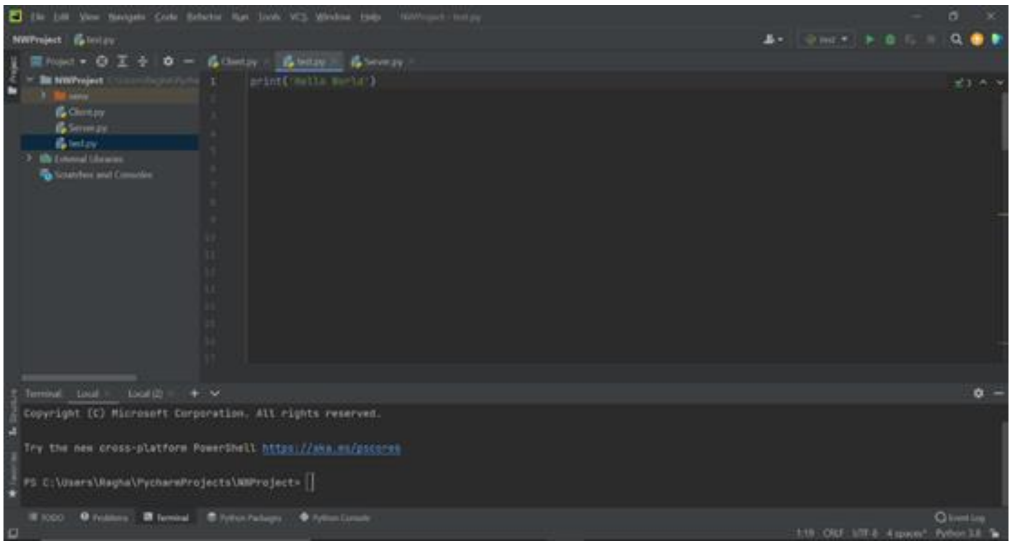
2. Setting up the Programming Environment:

2.1 Programming language:

Python language, is the programming language that we have used to create the TCP client/server program. Because, it is simple and it was easy to write network programs in Python compared to other programming languages as it plays an essential role in network programming where the Python Standard Library has full support for network protocols, data encryption and decryption and other networking concepts.

2.2 Programming environment:

We build and run this project using PyCharm IDE, the figure (1) below shows a screenshot of the environment.



```
print('Hello World')
```

```
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Ragha\PycharmProjects\NWProject>
```

Figure 1: PyCharm IDE interface

How to install:

1. Enter the PyCharm download link from the official website https://www.jetbrains.com/pycharm/download/#section=windows .
2. Click on the Download the latest available free version button .
3. Click on Save File to start the download .
4. After the download is complete, open the file to start installing the tools .
5. Right-click on the file, then click on Run as administrator.
6. Click on the Next button to start the program installation steps .
7. Click on the Next button to start the installation of the program in the specified path .
8. Check the boxes .py and choose the 32-bit launcher or 64-bit launcher then click on the Next button .
9. Click on the Next button .
10. Click on the Next button to start the actual installation of the program in the specified path and based on the options selected .
11. After the program has been successfully installed, click on the Finish button to exit.

How to compile and run:

After writing your code, you can compile and run the code at the same time either by

:

- Right-click on the file then choose 'Run file'
- Or, open the terminal (below the code space) and write 'python FinleName.py'

2.3 Libraries:

We have used some libraries provided by Python to help us in coding, listed as following:

1. The " `socket` " library, used to define the socket object. We can use several methods to manage the connections between the server and the client.
2. The " `Crypto.Cipher` " package which contains algorithms for encryption and decryption.
3. The " `AES` " (Advanced Encryption Standard) algorithm with mode CBC in client and server encryption and decryption methods.

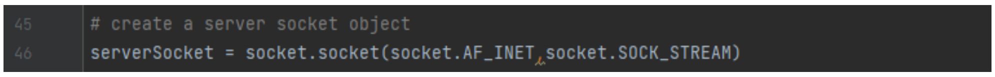
3. Steps for TCP socket programming for client-server connection:

In writing our code, especially in writing the basics of TCP client/server connection, we relied on the source code described in section 2.7.2 in "*Computer Networking A Top-Down approach*" book [1].

As for the encryption algorithm, we have used CBC mode code on the "**stackoverflow**" web site [2]. And the rest of the code is written by ourselves.

Server side:

1. Server will create a socket.

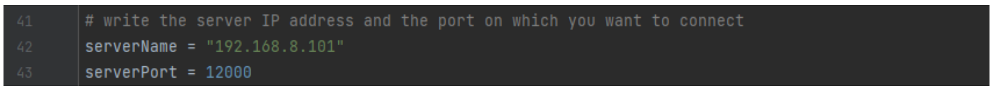


```
# create a server socket object
serverSocket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

Figure 2: server socket creation

The server creates its own socket, by using the `socket()` method with two parameters, the first parameter indicates the address domain, AF_INET which means that the network is using IPv4. The second parameter indicates to the socket type, SOCK_STREAM, which means that the type of socket we are using is the TCP socket.

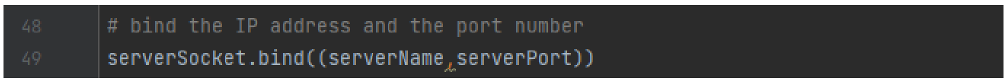
2. Binding server socket with IP address and port number.



```
# write the server IP address and the port on which you want to connect
serverName = "192.168.8.101"
serverPort = 12000
```

Figure 3:server IP address and port number

Server's socket information must be provided, the server's IP address in the `serverName` attribute, and the server port number in the `serverPort` attribute. The port number is ranged from [0 – 65,535], we used port numbers in the range [1024 – 65,535] to avoid any conflicts in the *well-known ports* from [0 – 1023].

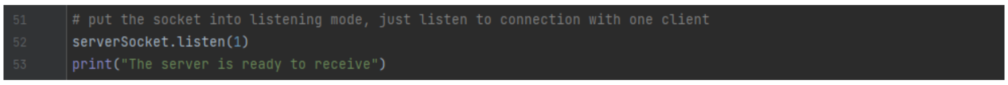


```
# bind the IP address and the port number
serverSocket.bind((serverName,serverPort))
```

Figure 4: bind server socket

Associate the server's IP address and the port number with the server's socket using the `bind()` method provided by the `socket` library. Now the `serverSocket` is ready to be our welcoming socket.

3. Server is ready to listen to client(s).



```
# put the socket into listening mode, just listen to connection with one client
serverSocket.listen(1)
print("The server is ready to receive")
```

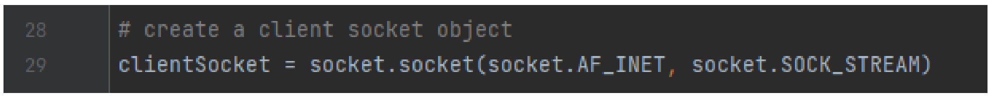
Figure 5: server's socket listeining to client.

After creating the server socket, the server uses the method `listen()` to listen to client's messages, and the number in this method's parameter refers to the maximum number of clients that the server can serve or listen to, which is one client in our case.

Client side:

After completing the previous server steps, we will switch to see the client side.

4. Client will create socket.

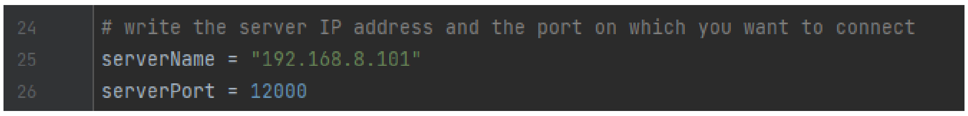


```
# create a client socket object
clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Figure 6: Client Socket Creation.

Same as in the server socket creation step, a client socket is created using the `socket()` method as well. As we mentioned earlier, AF_INET means that the network type we are using is the IPv4, and the SOCK_STREAM means that the socket type being used is the TCP socket.

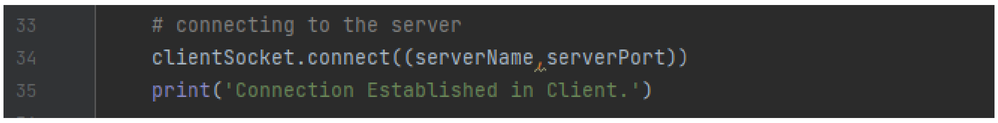
5. Client connection establishment setup.



```
# write the server IP address and the port on which you want to connect
serverName = "192.168.8.101"
serverPort = 12000
```

Figure 7: Connection establishment - part1

First, before establishing the connection, the client must know the server's IP address, we copied the server IP address from the server device, then store it in the `serverName` attribute. Also, we put a common port number between server and client, which is stored in the `serverPort` attribute.

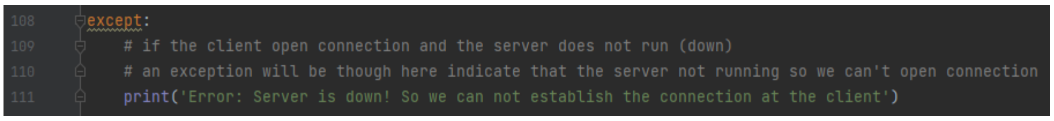


```
# connecting to the server
clientSocket.connect((serverName,serverPort))
print('Connection Established in Client.')
```

Figure 8: connection establishment - part2

Second, to establish the connection, a `connet()` method is invoked, which initiates the 3-way handshaking between the client and the server. This method has only one parameter, which holds the address of the destination server, which is basically the pair of the server's IP address and the server's port number (`serverName`, `serverPort`) specified before. After completing this step, the client will be waiting for the server to accept this connection to start exchanging messages.

6. Client will check if the server is running:



```
108 except:
109     # if the client open connection and the server does not run (down)
110     # an exception will be though here indicate that the server not running so we can't open connection
111     print('Error: Server is down! So we can not establish the connection at the client')
```

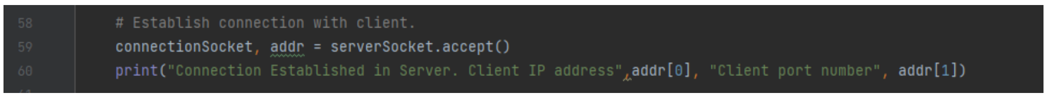
Figure 9: try-except block

While the client's socket is trying to establish the connection, there is a possibility that the server is down, in this case, it will throw an exception. So, we put our connection establishment step, and the sending and receiving process in the client side, from line '32' to '107', all in a try-except block, to avoid any crash of the program, and to show an error message to the client, as shown in line '111', if the server is not running, then stop the process.

Server side:

After the client has established the connection successfully, the client will be in the waiting state for the server to accept its connection.

7. Server will accept the client connection establishment.



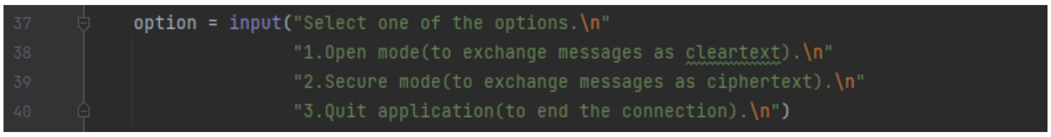
```
# Establish connection with client.
connectionSocket, addr = serverSocket.accept()
print("Connection Established in Server. Client IP address",addr[0], "Client port number", addr[1])
```

Figure 10: server accepts client's connection establishment

When the client knocks the server's door, the method `accept()` is called, which will create a new socket specially for that client, which is the `connectionSocket`. At this point, the 3-way handshaking is done successfully, so the client and server will start exchanging messages.

Client side:

8. A list of options will be displayed for the client to be chosen.



```
option = input("Select one of the options.\n"
               "1.Open mode(to exchange messages as cleartext).\n"
               "2.Secure mode(to exchange messages as ciphertext).\n"
               "3.Quit application(to end the connection).\n")
```

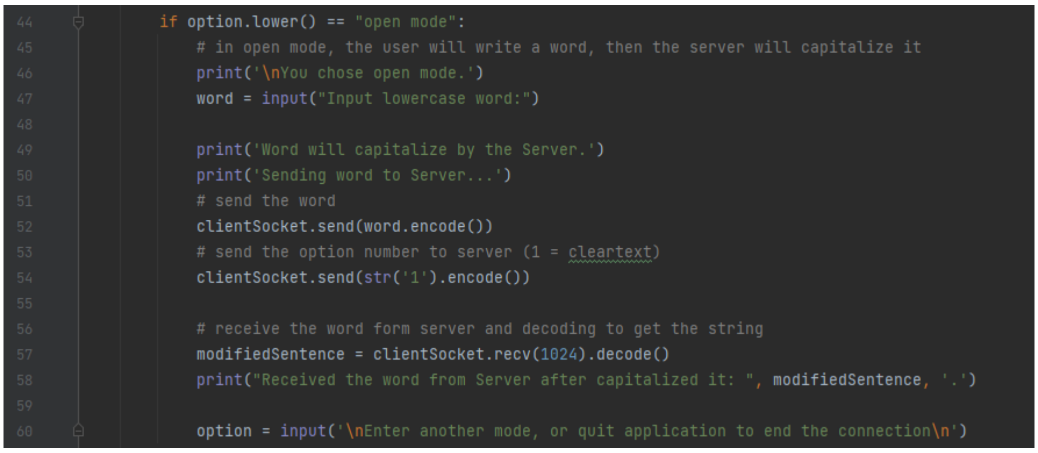
Figure 11: printing the option list to the client

After successfully completing the 3-way handshaking, a window including the three options, open mode, secure mode and quit application, will be printed to the client, and the client is expected to enter the option they desire.

We put the options in a while loop, so the client can choose options until the 'quit application' option is chosen to close the connection with the server.

Note: the next steps are divided into three scenarios, each scenario refers to one of the options mentioned earlier. (8.1) is the open mode scenario, (8.2) is the secure mode scenario, lastly, (8.3) is the quit application scenario.

8.1 Option (1): open mode.

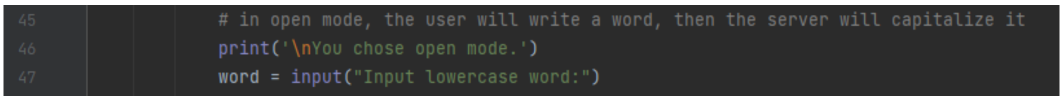


```
if option.lower() == "open mode":
    # in open mode, the user will write a word, then the server will capitalize it
    print('\nYou chose open mode.')
    word = input("Input lowercase word:")

    print('Word will capitalize by the Server.')
    print('Sending word to Server...')
    # send the word
    clientSocket.send(word.encode())
    # send the option number to server (1 = cleartext)
    clientSocket.send(str('1').encode())

    # receive the word form server and decoding to get the string
    modifiedSentence = clientSocket.recv(1024).decode()
    print("Received the word from Server after capitalized it: ", modifiedSentence, '.')

    option = input('\nEnter another mode, or quit application to end the connection\n')
```

Figure 12: option (1) open mode

Open mode, where the client is sending a cleartext to the server, then the server is receiving the message and converting it to its capitalized form, then sending it back to the client as a cleartext.

Figure (12) is just an overview of the open mode block of code, now we will explain each statement in details.

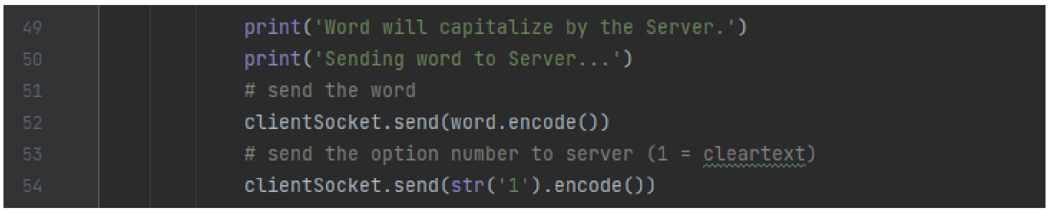


```
    # in open mode, the user will write a word, then the server will capitalize it
    print('\nYou chose open mode.')
    word = input("Input lowercase word:")
```

Figure 13: request to input the cleartext from client

A message is printed to indicate that the client has chosen the 'open mode', then requesting to write a text to be sent to the server.

8.1.1. Client will send a cleartext to the server.



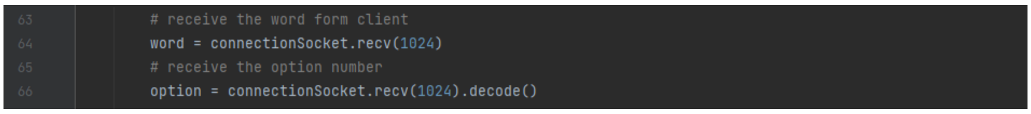
```
49    print('Word will capitalize by the Server.')
50    print('Sending word to Server...')
51    # send the word
52    clientSocket.send(word.encode())
53    # send the option number to server (1 = cleartext)
54    clientSocket.send(str('1').encode())
```

Figure 14: send cleartext message from client to server

The client socket will use the method `send()` ,as shown in line '52', to send the cleartext message, along with the option number, in line '54'. Sending the option number is important for the server side to recognize the type of the message. And all of them are sent as a stream of bytes using the `.encode()` method.

Server side:

8.1.2. Server will receive the message from client.



```
# receive the word form client
word = connectionSocket.recv(1024)
# receive the option number
option = connectionSocket.recv(1024).decode()
```

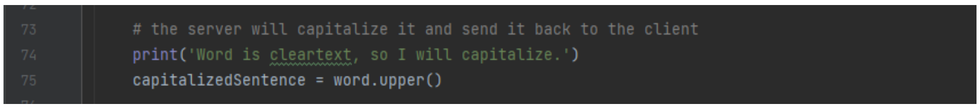
Figure 15: receive message from client to server

In general, in receiving messages, the `recv()` method is invoked, with the parameter indicating the maximum number of bytes to be received, which is 1024 bytes in our case.

`Word`, is the attribute which stores the message received from the client, as a cleartext or encrypted message. `Option`, in the attribute which stores the option number received form the client. Option '1' refers to the open mode, option '2' refers to the secure mode, and option '3' refers to quit application.

In the open mode scenario, the server will receive option number '1'.

8.1.3. Server will capitalize the message.

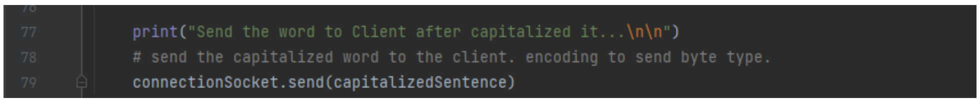


```
# the server will capitalize it and send it back to the client
print('Word is cleartext, so I will capitalize.')
capitalizedSentence = word.upper()
```

Figure 16: option (1) open mode

The server will capitalize the message using `.upper()` method provided by Python. And store the capitalized text in `capitalizedSentence` variable.

8.1.4. Server will send the capitalized message back to client.

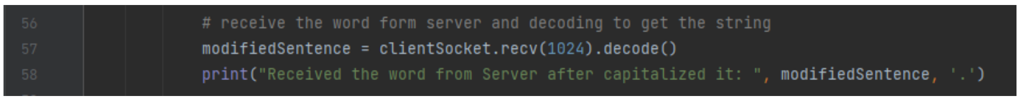


```
print("Send the word to Client after capitalized it...\n\n")
# send the capitalized word to the client. encoding to send byte type.
connectionSocket.send(capitalizedSentence)
```

Using the `send()` method to send the capitalized message from the server connection socket to the client socket, where the parameter of this method holds the `capitalizedSentence` to be sent.

Client side:

8.1.5. Client will receive the message from the server.



```
        # receive the word form server and decoding to get the string
        modifiedSentence = clientSocket.recv(1024).decode()
        print("Received the word from Server after capitalized it: ", modifiedSentence, '.')
```

Figure 17: receive cleartext message from server to client.

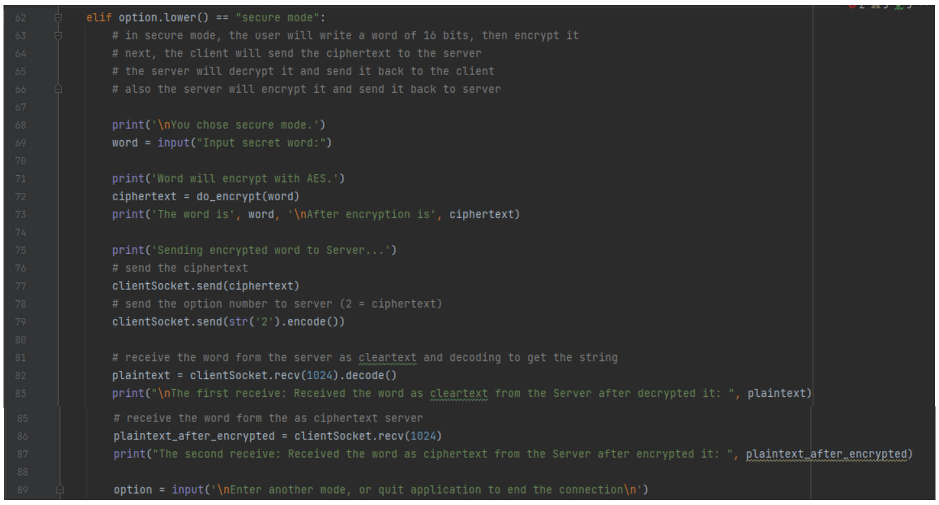
Using the method `.recv()`,to receive the returned message in the capitalized form from the server, as a stream of bytes, and using the `.decode()` method to convert the message into a string. Then printing it.

The open mode scenario is finished, and the options list will appear to the client again to enter any other option until it chooses the quit application option.

In the next page we will explain the secure mode scenario.

Client side:

8.2. Option (2): secure mode.



```python
elif option.lower() == "secure mode":
    # in secure mode, the user will write a word of 16 bits, then encrypt it
    # next, the client will send the ciphertext to the server
    # the server will decrypt it and send it back to the client
    # also the server will encrypt it and send it back to server

    print('\nYou chose secure mode.')
    word = input("Input secret word:")

    print('Word will encrypt with AES.')
    ciphertext = do_encrypt(word)
    print('The word is', word, '\nAfter encryption is', ciphertext)

    print('Sending encrypted word to Server...')
    # send the ciphertext
    clientSocket.send(ciphertext)
    # send the option number to server (2 = ciphertext)
    clientSocket.send(str('2').encode())

    # receive the word form the server as cleartext and decoding to get the string
    plaintext = clientSocket.recv(1024).decode()
    print("\nThe first receive: Received the word as cleartext from the Server after decrypted it: ", plaintext)
    # receive the word form the as ciphertext server
    plaintext_after_encrypted = clientSocket.recv(1024)
    print("The second receive: Received the word as ciphertext from the Server after encrypted it: ", plaintext_after_encrypted)

    option = input('\nEnter another mode, or quit application to end the connection\n')
```

Figure 18: option (2): secure mode.

Secure mode, where the client is sending an encrypted message to the server. The cryptography algorithm we used to encrypt the message is the AES, the Advanced Encryption Standard algorithm [3].

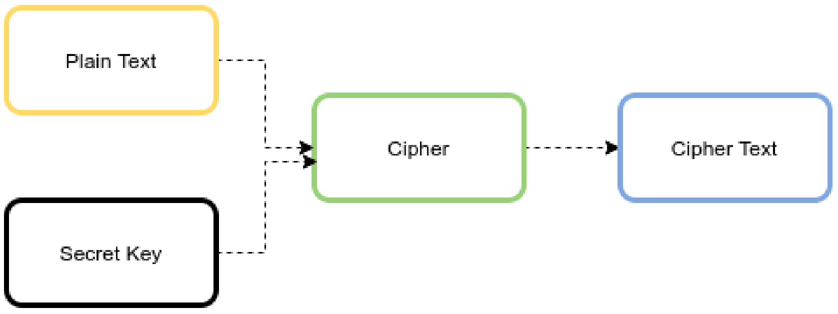
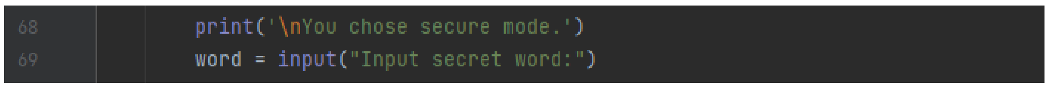


Figure 19: High-level description of AES

The figure (19) above is showing a high-level description of the process of the AES algorithm. This algorithm has six modes of operation, one of these modes is the CBC, Cipher Block Chaining mode, which we have implemented in our code.

CBC mode: in this mode there are two important factors, which are the key and the initialization vector (IV).

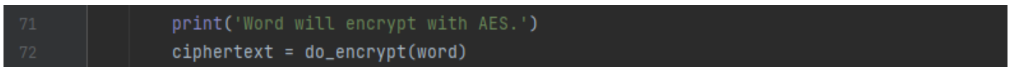


```python
print('\nYou chose secure mode.')
word = input("Input secret word:")
```

Figure 20: request to input the text from client to be sent in the secure mode.

Initially, same as in the open mode, here a message is printed, as shown in figure (20), to indicate that the client had chosen the 'secure mode', then it will request the client to enter a text, which will be encrypted and sent to server in next steps.

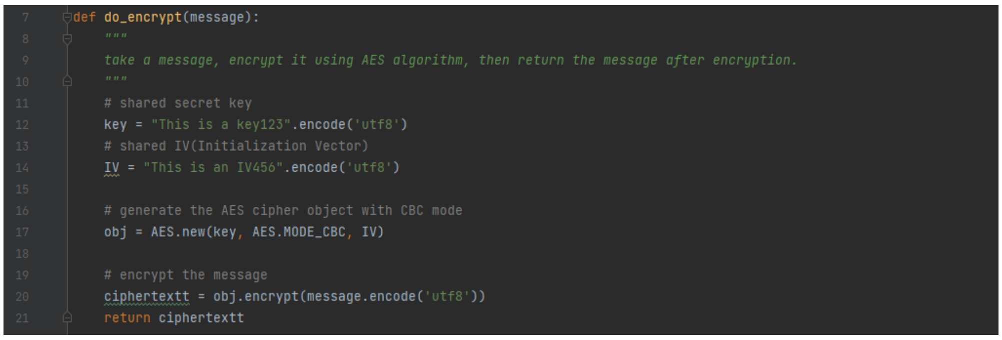
8.2.1 client will encrypt the message.



```
print('Word will encrypt with AES.')
ciphertext = do_encrypt(word)
```

Figure 21: calling the encryption function

The encryption function `do_encrypt()` is called, with the parameter including the word or the text we want to encrypt.



```
7   def do_encrypt(message):
8       """
9       take a message, encrypt it using AES algorithm, then return the message after encryption.
10      """
11      # shared secret key
12      key = "This is a key123".encode('utf8')
13      # shared IV(Initialization Vector)
14      IV = "This is an IV456".encode('utf8')
15
16      # generate the AES cipher object with CBC mode
17      obj = AES.new(key, AES.MODE_CBC, IV)
18
19      # encrypt the message
20      ciphertextt = obj.encrypt(message.encode('utf8'))
21      return ciphertextt
```

Figure 22: do_encryption() function

In the block of encryption method `do_encrypt()`, it receives the cleartext in the variable `message`, that was entered by the client.

Line 12 and 14: In order to do the encryption, or even the decryption, in the CBC mode, there must be a **shared key** and **initialization vector (IV)** in both client and server sides. Both those attributes must be in the form of bytes, so we convert them from String to Bytes using the `encode()` method.

Line 17: create an AES cipher block in the CBC mode, by using the `new()` provided by the `Crypto.Cipher` package method in the form [4]:

```
Crypto.Cipher.<algorithm>.new(key, mode, iv=None)
```

`<algorithm>` indicates to the AES algorithm in our case.

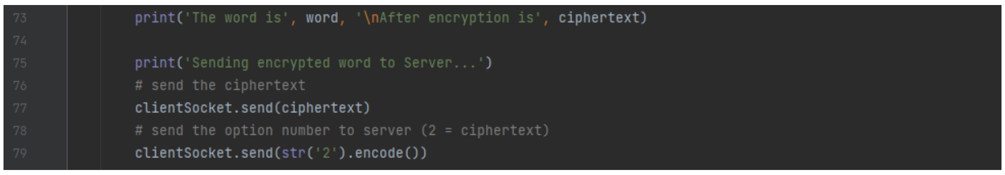
Parameters:

- `key` (bytes): the cryptographic key.
- `mode`: the constant AES.MODE_CBC.
- `iv` (bytes): initialization vector.

Line 20: using the AES cipher block we created earlier to encrypt the required message, as a stream of bytes, by sending the message as a parameter of the encrypt() method.

Line 21: return the encrypted message to the client be sent to the server.

8.2.2. Client will send the encrypted message to server.



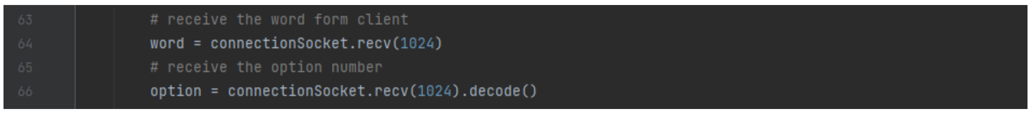
```
73          print('The word is', word, '\nAfter encryption is', ciphertext)
74
75          print('Sending encrypted word to Server...')
76          # send the ciphertext
77          clientSocket.send(ciphertext)
78          # send the option number to server (2 = ciphertext)
79          clientSocket.send(str('2').encode())
```

Figure 23: send encrypted message from client to server

In line '73', the cleartext is printed along with the encrypted message to ensure the encryption is processed successfully. Then, the client sends the encrypted message in line '77', along with the option number in line '79' to the server.

Server side:

8.2.3. Server will receive the encrypted message from client.

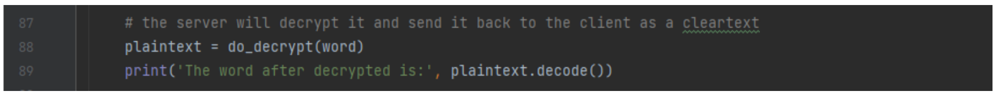


```
        # receive the word form client
        word = connectionSocket.recv(1024)
        # receive the option number
        option = connectionSocket.recv(1024).decode()
```

Figure 24: receive message from client to server

This piece of code is explained in details in scenario "8.1.2." page (p.10). In the secure mode, the word received is an encrypted stream of bytes and the option number will be '2', to notify the server that this is an encrypted message.

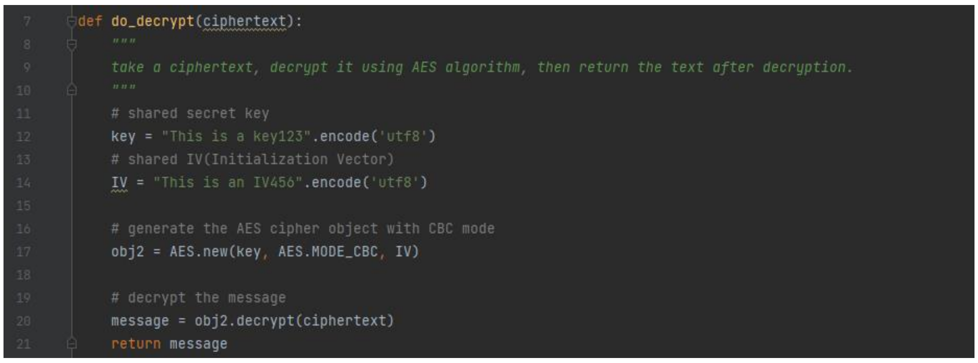
8.2.4. Server will decrypt the message.



```
        # the server will decrypt it and send it back to the client as a cleartext
        plaintext = do_decrypt(word)
        print('The word after decrypted is:', plaintext.decode())
```

Figure 25: call the decryption function.

The decryption function `do_decrypt()` is called, with the parameter including the word or the text we want to be decrypted.

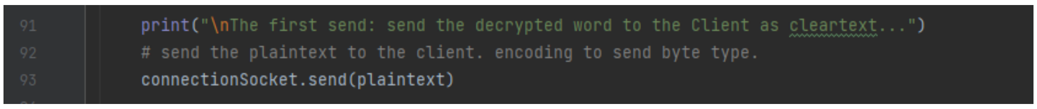


```
def do_decrypt(ciphertext):
    """
    take a ciphertext, decrypt it using AES algorithm, then return the text after decryption.
    """
    # shared secret key
    key = "This is a key123".encode('utf8')
    # shared IV(Initialization Vector)
    IV = "This is an IV456".encode('utf8')

    # generate the AES cipher object with CBC mode
    obj2 = AES.new(key, AES.MODE_CBC, IV)

    # decrypt the message
    message = obj2.decrypt(ciphertext)
    return message
```

Figure 26: do_decrypt() function.

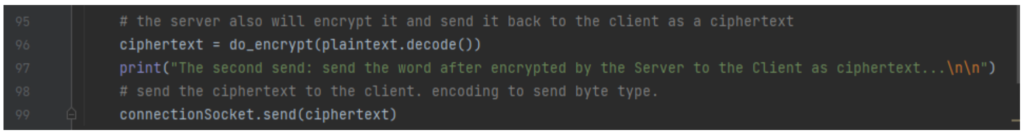
The `do_decrypt()` function has the same structure as the `do_encrypt()` function, which is explained in page (p.14), but the main difference between them is that it uses the `decrypt()` method, as shown in figure (26), line '20'. And it will return the message after it has been decrypted in the `message` variable.

8.2.5. Server will send back the encrypted message to client.



```
        print("\nThe first send: send the decrypted word to the Client as cleartext...")
        # send the plaintext to the client. encoding to send byte type.
        connectionSocket.send(plaintext)
```

Figure 27: send back the message as a cleartext from server to client.



```
        # the server also will encrypt it and send it back to the client as a ciphertext
        ciphertext = do_encrypt(plaintext.decode())
        print("The second send: send the word after encrypted by the Server to the Client as ciphertext...\n\n")
        # send the ciphertext to the client. encoding to send byte type.
        connectionSocket.send(ciphertext)
```

Figure 28: send back the message after encryption from server to client

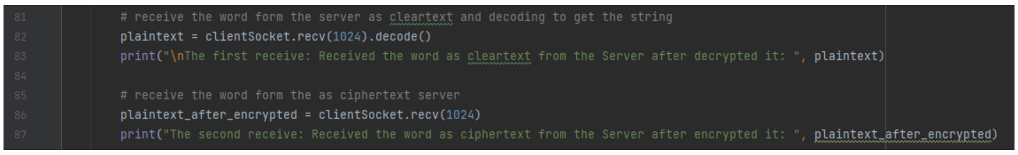
In the process of sending the message back to client, we decided to make the server sends two copies of the message, one is the cleartext, which is the encrypted message that has been decrypted, stored in the `plaintext` variable.

And the other copy which is the decrypted message after being encrypted again using the `do_encrypt()` function (this method is explained in details in p.14).

Both messages will be sent to client through the connection socket using the `.send()` method.

Client side:

8.2.6 Client will receive two messages, encrypted message and cleartext message from server.



```
81          # receive the word form the server as cleartext and decoding to get the string
82          plaintext = clientSocket.recv(1024).decode()
83          print("\nThe first receive: Received the word as cleartext from the Server after decrypted it: ", plaintext)
84
85          # receive the word form the as ciphertext server
86          plaintext_after_encrypted = clientSocket.recv(1024)
87          print("The second receive: Received the word as ciphertext from the Server after encrypted it: ", plaintext_after_encrypted)
```

Figure 29: receive encrypted message and cleartext message from server to client

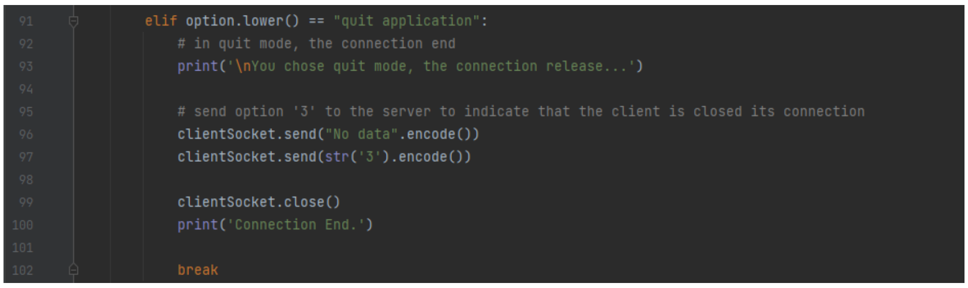
The client will receive the cleartext message as bytes from the server, then convert it to string to print it. And in line '86' it receives the encrypted message as bytes, then print it.

The secure mode scenario is finished, and the options list will appear to the client again to enter any other option until it chooses the quit application option.

In the next page we will explain the quit application scenario.

Client side:

8.3 Option (3): quit application.

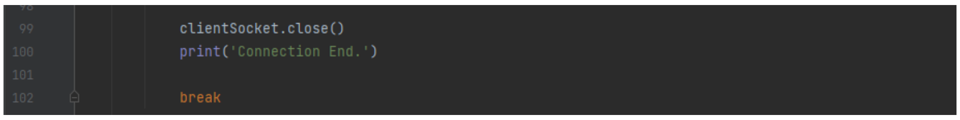


```
        elif option.lower() == "quit application":
            # in quit mode, the connection end
            print('\nYou chose quit mode, the connection release...')

            # send option '3' to the server to indicate that the client is closed its connection
            clientSocket.send("No data".encode())
            clientSocket.send(str('3').encode())

            clientSocket.close()
            print('Connection End.')

            break
```

Figure 30: option (3) quit application

Quit application, where the client wants to close the connection with the server and stop the process of exchanging messages.

8.3.1 Client will close the connection.



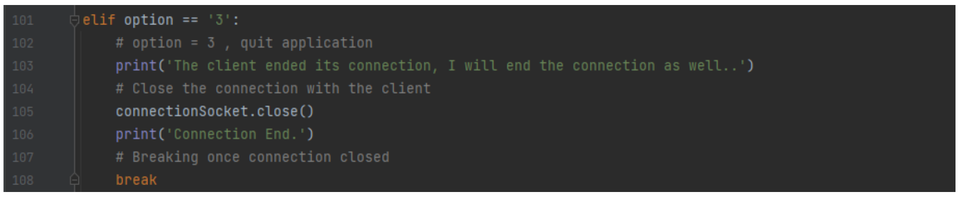
```
            clientSocket.close()
            print('Connection End.')

            break
```

Figure 31: client will close the connection.

With the help of `.close()` method provided by `Socket` library, the client can close the connection with the server.

Server side:

8.3.2 Server will close the connection.



```
elif option == '3':
    # option = 3 , quit application
    print('The client ended its connection, I will end the connection as well..')
    # Close the connection with the client
    connectionSocket.close()
    print('Connection End.')
    # Breaking once connection closed
    break
```

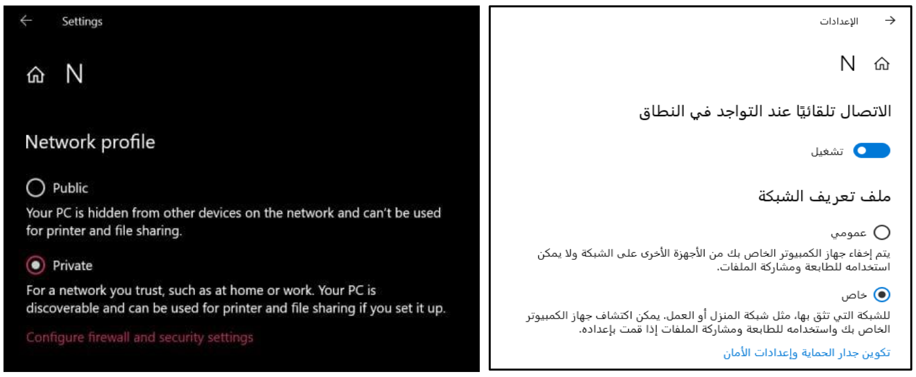
Figure 32: option (3) quit application.

The server will receive the closing message, then it will close the connection with the help of `close()` method.

4. Steps for setting up the network:

To start the connection through the network, we have applicated the following steps:

1. First, connect the client device and the server device with the same network, we used a wireless network (Wi-Fi).
2. Change the properties of the Wi-Fi for the two devices from public mode (which hides the device from the network for the purpose of protection) to private (which allows the device to appear on the network), as shown in figure(33).


← Settings
N
Network profile

Public [✓ Private]

Public: Your PC is hidden from other devices on the network and can't be used for printer and file sharing.

Private: For a network you trust, such as at home or work. Your PC is discoverable and can be used for printer and file sharing if you set it up.

Configure firewall and security settings
الإعدادات
N
الاتصال تلقائيًا عند التواجد في النطاق
تشغيل
ملف تعريف الشبكة

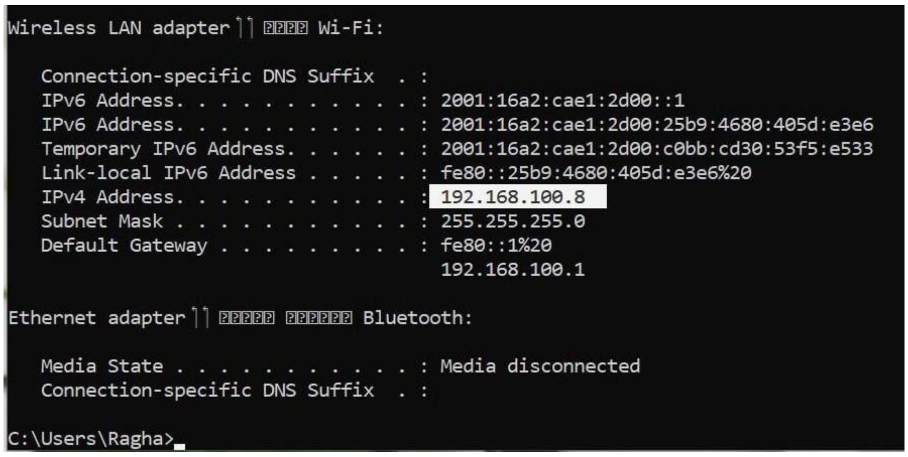
عمومي [✓ خاص]

عمومي: يتم إخفاء جهاز الكمبيوتر الخاص بك من الأجهزة الأخرى على الشبكة ولا يمكن استخدامه للطابعة ومشاركة الملفات.

خاص: للشبكة التي تثق بها، مثل شبكة المنزل أو العمل. يمكن اكتشاف جهاز الكمبيوتر الخاص بك واستخدامه للطابعة ومشاركة الملفات إذا قمت بإعداده.

Figure 33: Connecting both the server and the client to the same Wi-Fi network with changing Wi-Fi settings.

3. We used the command "`ipconfig`" on the server device's command line to get the server IP address, as shown in figure (34) bellow.



```
Wireless LAN adapter ▯▯▯▯ Wi-Fi:

   Connection-specific DNS Suffix  . :
   IPv6 Address. . . . . . . . . . . : 2001:16a2:cae1:2d00::1
   IPv6 Address. . . . . . . . . . . : 2001:16a2:cae1:2d00:25b9:4680:405d:e3e6
   Temporary IPv6 Address. . . . . . : 2001:16a2:cae1:2d00:c0bb:cd30:53f5:e533
   Link-local IPv6 Address . . . . . : fe80::25b9:4680:405d:e3e6%20
   IPv4 Address. . . . . . . . . . . : 192.168.100.8
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : fe80::1%20
                                       192.168.100.1

Ethernet adapter ▯▯▯▯▯ ▯▯▯▯▯▯ Bluetooth:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

C:\Users\Ragha>_
```

Figure 34: "ipconfig" command to get server IP address.

4. Include the server's IP address in both client code and server code.
5. Put the same port number in both client code and server code.

After completing these steps, the two devices are ready to communicate through the network to start exchanging messages.

5. TCP-based client/server application:

5.1 Application assumptions:

We have assumed that:

- There is only one client with one server.
- In open mode, client can enter the text with any size, it will be accepted.
- In secure mode, the client can enter the word with only 16-bit length only.
- In open mode, the client will send the word to the server, and the server will send back a copy of the word after capitalizing it.
- In secure mode, the server will receive the encrypted message from the client, then it will send back two messages, one is the cleartext message after it has been decrypted, and the other message is the message after it has been encrypted again by the server.

5.2 Application flow:

To facilitate the understanding of the code and to keep track of our program flow, we have created a Flow Diagram [5], as shown in figure (35) bellow.

Which describes, in high-level, the connection process and exchanging messages between client and server.

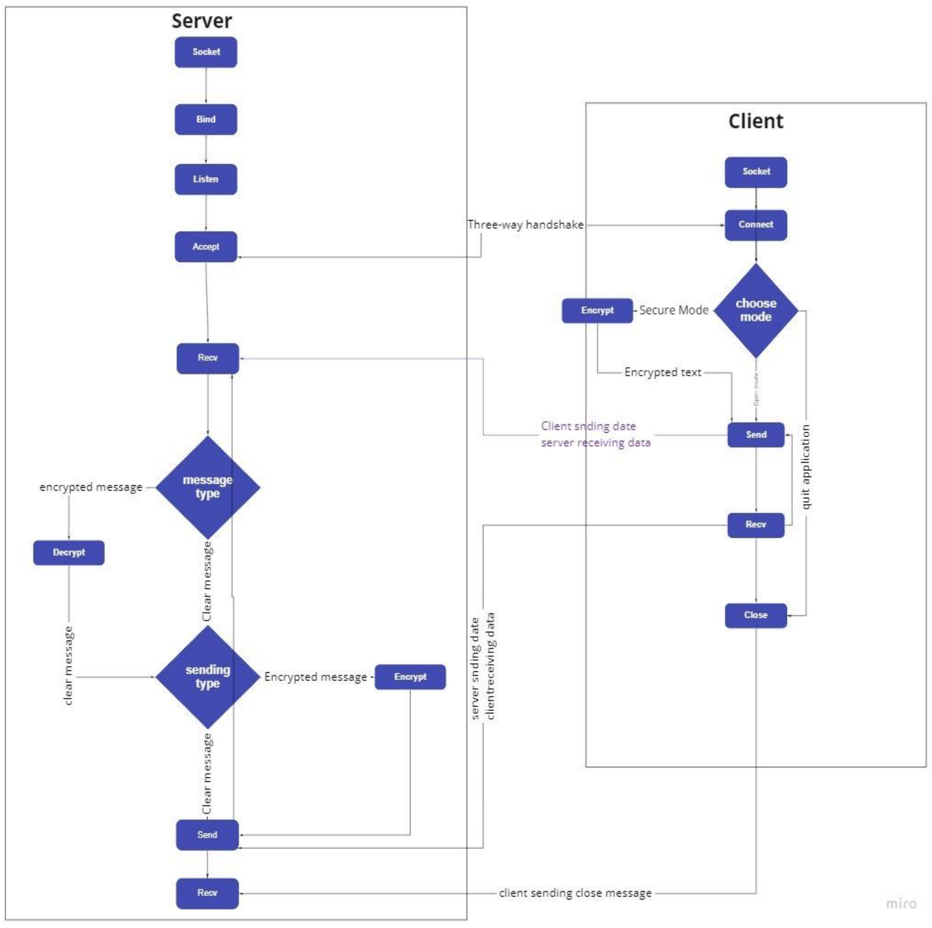


Figure 35: Application Flow Diagram

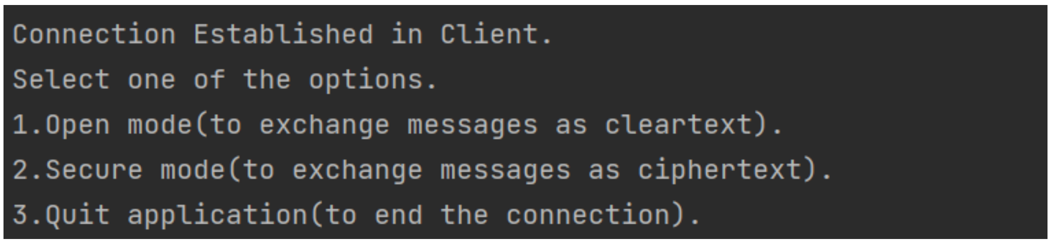
5.3 Code and comments:

For better quality, we have attached two captures in the related .zip file, one for the code of the server side and the other one for the code of the client side.

6. Snapshot of the application outputs:

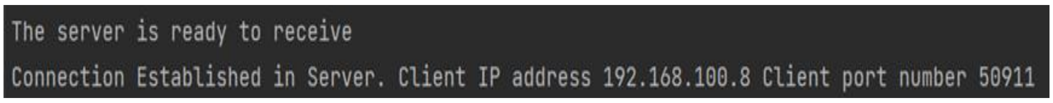
In this section, we provided snapshots of the TCP application outcomes, in both client and server sides.

Figure (36) and figure (37) show the connection establishment result in client and server, respectively.



```
Connection Established in Client.
Select one of the options.
1.Open mode(to exchange messages as cleartext).
2.Secure mode(to exchange messages as ciphertext).
3.Quit application(to end the connection).
```

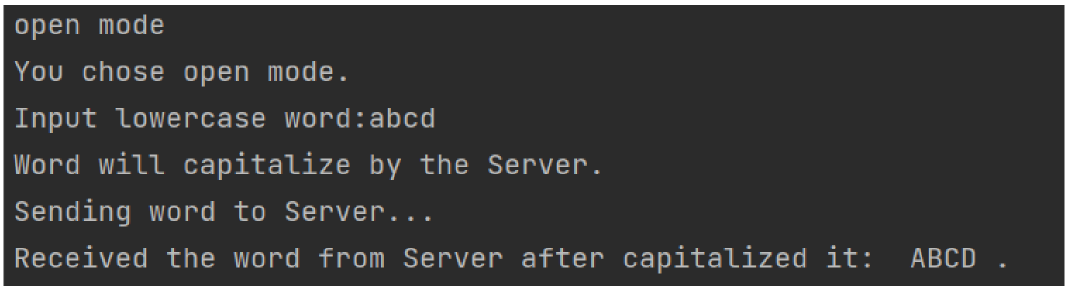
Figure 36: client side



```
The server is ready to receive
Connection Established in Server. Client IP address 192.168.100.8 Client port number 50911
```

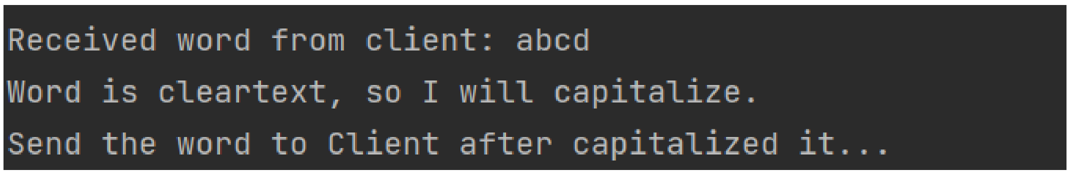
Figure 37: server side

When the user chooses the open mode, the result of this process in both client and server are shown in figure (38) and figure (39) bellow, respectively.



```
open mode
You chose open mode.
Input lowercase word:abcd
Word will capitalize by the Server.
Sending word to Server...
Received the word from Server after capitalized it:  ABCD .
```

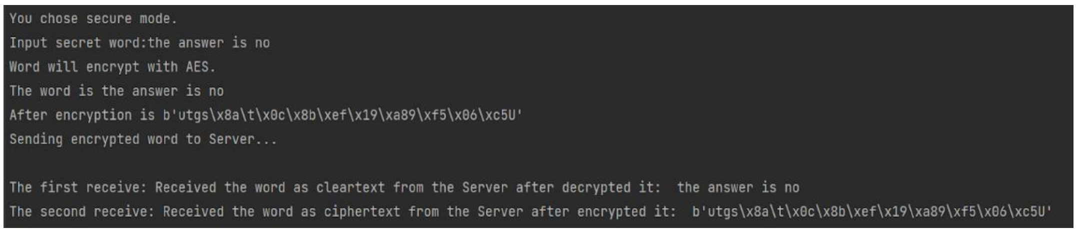
Figure 38: client side - open mode



```
Received word from client: abcd
Word is cleartext, so I will capitalize.
Send the word to Client after capitalized it...
```

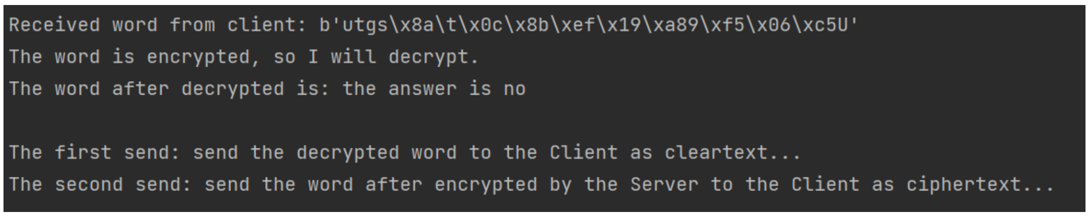
Figure 39: server side - open mode

When the user chooses secure mode, the result of the process for secure mode in client and server are shown in figure (40) and figure (41), respectively.



```
You chose secure mode.
Input secret word:the answer is no
Word will encrypt with AES.
The word is the answer is no
After encryption is b'utgs\x8a\t\x0c\x8b\xef\x19\xa89\xf5\x06\xc5U'
Sending encrypted word to Server...

The first receive: Received the word as cleartext from the Server after decrypted it:  the answer is no
The second receive: Received the word as ciphertext from the Server after encrypted it:  b'utgs\x8a\t\x0c\x8b\xef\x19\xa89\xf5\x06\xc5U'
```

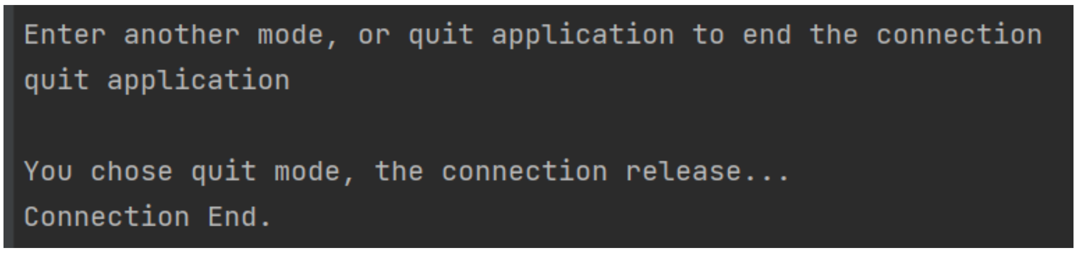
Figure 40: client side - secure mode



```
Received word from client: b'utgs\x8a\t\x0c\x8b\xef\x19\xa89\xf5\x06\xc5U'
The word is encrypted, so I will decrypt.
The word after decrypted is: the answer is no

The first send: send the decrypted word to the Client as cleartext...
The second send: send the word after encrypted by the Server to the Client as ciphertext...
```

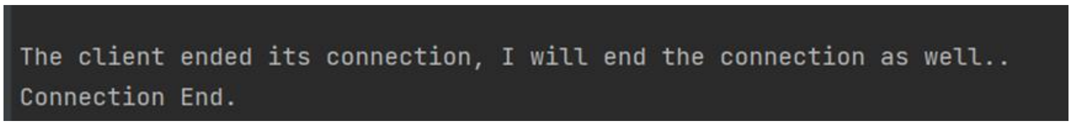
Figure 41: server side - secure mode

When the user chooses the quit application option, the connection will be ended on both sides, figure (42) and figure (43) show the connection end process in client and server.



```
Enter another mode, or quit application to end the connection
quit application

You chose quit mode, the connection release...
Connection End.
```

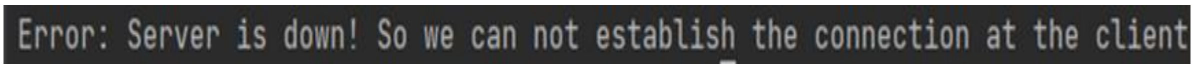
Figure 42: client side - quit application



```

The client ended its connection, I will end the connection as well..
Connection End.
```

Figure 43: server side - quit application

When the client tries to establish the connection but there is no server running, it can't establish any connection while there is no server. Figure (44) shows the result.



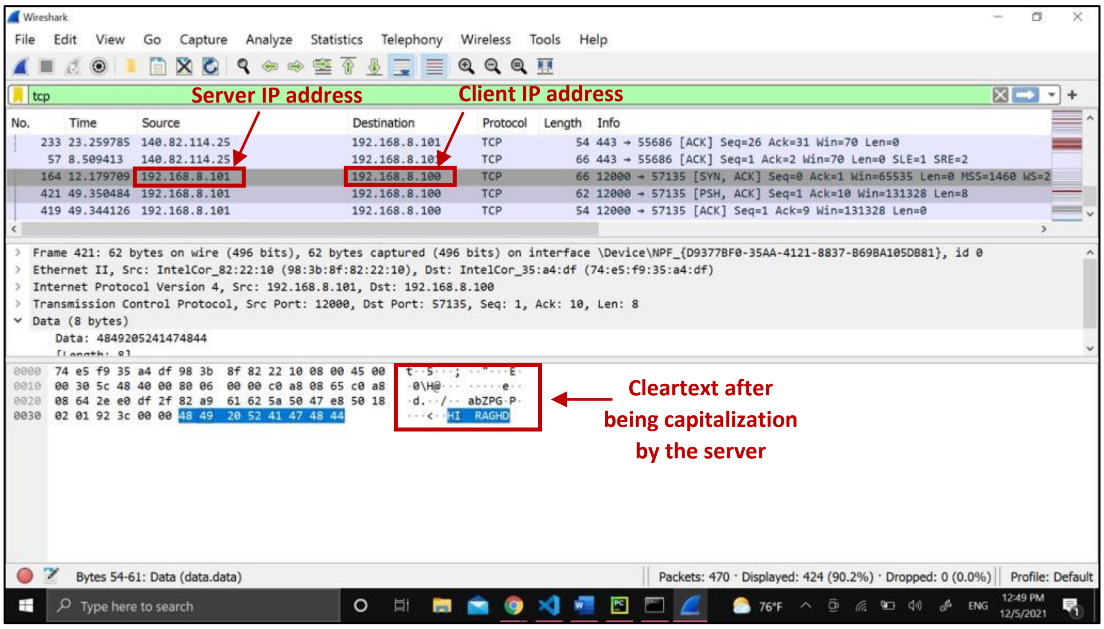
```
Error: Server is down! So we can not establish the connection at the client
```

Figure 44: client side - server is down (error message)

7. Analysis of communication using Wireshark:

To make sure our messages are sent through the network connection correctly, we have used the Wireshark to capture them. The idea of using the Wireshark analysis in TCP Client/Server application is inspired from the "Cyber Wilk"[6] web site.

In the open mode, the client sends a cleartext to the server, the server will capitalize it and send it back to the client. Figure (45) shows the capitalized cleartext sent by the server.

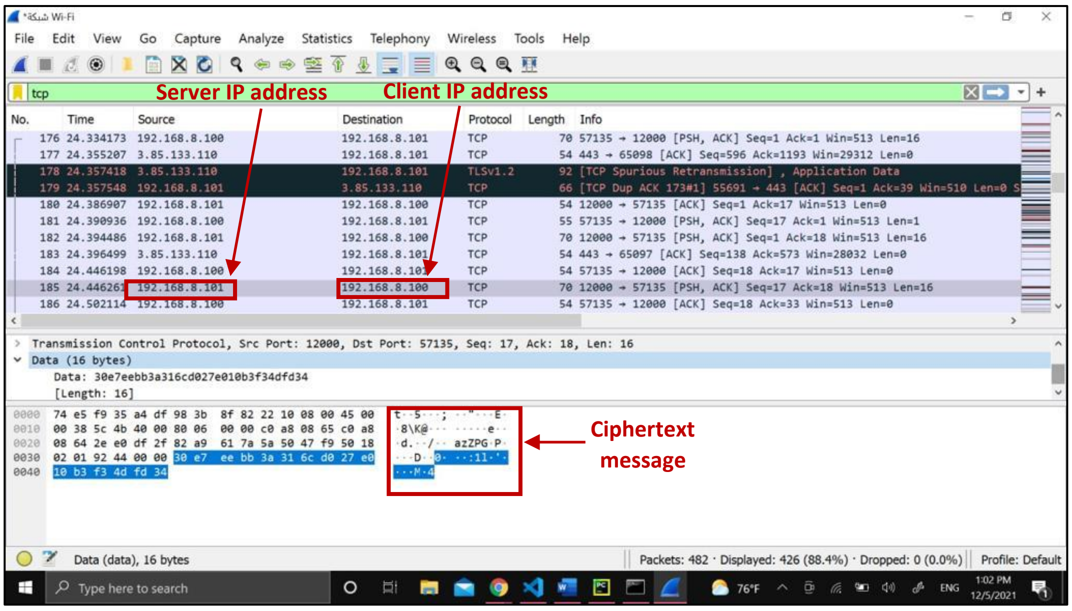

Wireshark
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp
Server IP address
Client IP address
Cleartext after being capitalization by the server

```
No.  Time        Source          Destination     Protocol  Length  Info
233  23.259785   140.82.114.25   192.168.8.101   TCP       54  443 → 55686 [ACK] Seq=26 Ack=31 Win=70 Len=0
 57  8.509413    140.82.114.25   192.168.8.101   TCP       66  443 → 55686 [ACK] Seq=1 Ack=2 Win=70 Len=0 SLE=1 SRE=2
164  12.179709   192.168.8.101   192.168.8.100   TCP       66  12000 → 57135 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=2
421  49.350484   192.168.8.101   192.168.8.100   TCP       62  12000 → 57135 [PSH, ACK] Seq=1 Ack=10 Win=131328 Len=8
419  49.344126   192.168.8.101   192.168.8.100   TCP       54  12000 → 57135 [ACK] Seq=1 Ack=9 Win=131328 Len=0

> Frame 421: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface \Device\NPF_{D9377BF0-35AA-4121-8837-B69BA105DB81}, id 0
> Ethernet II, Src: IntelCor_82:22:10 (98:3b:8f:82:22:10), Dst: IntelCor_35:a4:df (74:e5:f9:35:a4:df)
> Internet Protocol Version 4, Src: 192.168.8.101, Dst: 192.168.8.100
> Transmission Control Protocol, Src Port: 12000, Dst Port: 57135, Seq: 1, Ack: 10, Len: 8
v Data (8 bytes)
    Data: 4849205241474844

0000  74 e5 f9 35 a4 df 98 3b  8f 82 22 10 08 00 45 00   t··5···;  ··"···E·
0010  00 30 5c 48 40 00 80 06  00 00 c0 a8 08 65 c0 a8   ·0\H@···  ·····e··
0020  08 64 2e e0 df 2f 82 a9  61 62 5a 50 47 e8 50 18   ·d.··/··  abZPG·P·
0030  02 01 92 3c 00 00 48 49  20 52 41 47 48 44         ···<··HI  RAGHD

Bytes 54-61: Data (data.data)    Packets: 470 · Displayed: 424 (90.2%) · Dropped: 0 (0.0%)    Profile: Default
```

Figure 45: Wireshark analysis, sending back capitalized cleartext word from server to client.

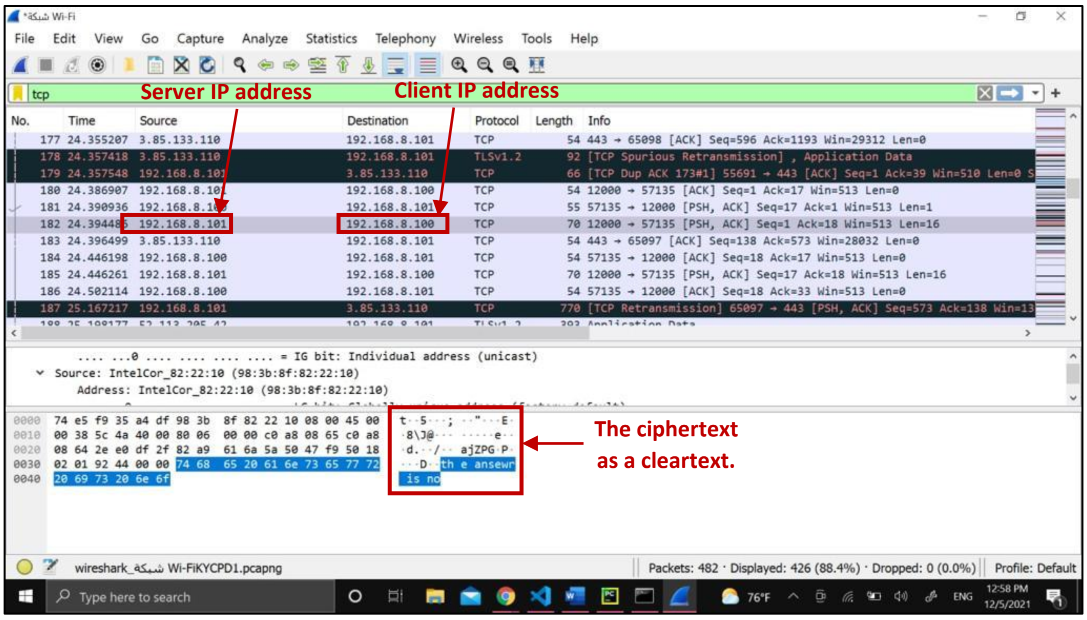
In the secure mode, the client sends an encrypted message to server. The server will then decrypt it, and send it back to client as a cleartext, as well as an encrypted message.

The messages received from the server in the secure mode is shown in both figures (46) and (47) bellow.



```
No.  Time      Source          Destination     Protocol Length Info
176 24.334173 192.168.8.100   192.168.8.101   TCP       70 57135 → 12000 [PSH, ACK] Seq=1 Ack=1 Win=513 Len=16
177 24.355207 3.85.133.110    192.168.8.101   TCP       54 443 → 65098 [ACK] Seq=596 Ack=1193 Win=29312 Len=0
178 24.357418 3.85.133.110    192.168.8.101   TLSv1.2   92 [TCP Spurious Retransmission] , Application Data
179 24.357548 192.168.8.101   3.85.133.110    TCP       66 [TCP Dup ACK 173#1] 55691 → 443 [ACK] Seq=1 Ack=39 Win=510 Len=0 S
180 24.386907 192.168.8.101   192.168.8.100   TCP       54 12000 → 57135 [ACK] Seq=1 Ack=17 Win=513 Len=0
181 24.390936 192.168.8.100   192.168.8.101   TCP       55 57135 → 12000 [PSH, ACK] Seq=17 Ack=1 Win=513 Len=1
182 24.394486 192.168.8.101   192.168.8.100   TCP       70 12000 → 57135 [PSH, ACK] Seq=1 Ack=18 Win=513 Len=16
183 24.396499 3.85.133.110    192.168.8.101   TCP       54 443 → 65097 [ACK] Seq=138 Ack=573 Win=28032 Len=0
184 24.446198 192.168.8.100   192.168.8.101   TCP       54 57135 → 12000 [ACK] Seq=18 Ack=17 Win=513 Len=0
185 24.446261 192.168.8.101   192.168.8.100   TCP       70 12000 → 57135 [PSH, ACK] Seq=17 Ack=18 Win=513 Len=16
186 24.502114 192.168.8.100   192.168.8.101   TCP       54 57135 → 12000 [ACK] Seq=18 Ack=33 Win=513 Len=0

> Transmission Control Protocol, Src Port: 12000, Dst Port: 57135, Seq: 17, Ack: 18, Len: 16
v Data (16 bytes)
    Data: 30e7eebb3a316cd027e010b3f34dfd34
    [Length: 16]

0000  74 e5 f9 35 a4 df 98 3b  8f 82 22 10 08 00 45 00   t··5···;  ··"···E·
0010  00 38 5c 4b 40 00 80 06  00 00 c0 a8 08 65 c0 a8   ·8\K@···  ·····e··
0020  08 64 2e e0 df 2f 82 a9  61 7a 5a 50 47 f9 50 18   ·d.··/··  azZPG·P·
0030  02 01 92 44 00 00 30 e7  ee bb 3a 31 6c d0 27 e0   ···D··0·  ··:1l·'·
0040  10 b3 f3 4d fd 34                                  ···M·4

Data (data), 16 bytes    Packets: 482 · Displayed: 426 (88.4%) · Dropped: 0 (0.0%)    Profile: Default
```

Figure 46: Wireshark analysis, sending encrypted message from server to client.



```
No.  Time      Source          Destination     Protocol Length Info
177 24.355207 3.85.133.110    192.168.8.101   TCP       54 443 → 65098 [ACK] Seq=596 Ack=1193 Win=29312 Len=0
178 24.357418 3.85.133.110    192.168.8.101   TLSv1.2   92 [TCP Spurious Retransmission] , Application Data
179 24.357548 192.168.8.101   3.85.133.110    TCP       66 [TCP Dup ACK 173#1] 55691 → 443 [ACK] Seq=1 Ack=39 Win=510 Len=0 S
180 24.386907 192.168.8.101   192.168.8.100   TCP       54 12000 → 57135 [ACK] Seq=1 Ack=17 Win=513 Len=0
181 24.390936 192.168.8.100   192.168.8.101   TCP       55 57135 → 12000 [PSH, ACK] Seq=17 Ack=1 Win=513 Len=1
182 24.394486 192.168.8.101   192.168.8.100   TCP       70 12000 → 57135 [PSH, ACK] Seq=1 Ack=18 Win=513 Len=16
183 24.396499 3.85.133.110    192.168.8.101   TCP       54 443 → 65097 [ACK] Seq=138 Ack=573 Win=28032 Len=0
184 24.446198 192.168.8.100   192.168.8.101   TCP       54 57135 → 12000 [ACK] Seq=18 Ack=17 Win=513 Len=0
185 24.446261 192.168.8.101   192.168.8.100   TCP       70 12000 → 57135 [PSH, ACK] Seq=17 Ack=18 Win=513 Len=16
186 24.502114 192.168.8.100   192.168.8.101   TCP       54 57135 → 12000 [ACK] Seq=18 Ack=33 Win=513 Len=0
187 25.167217 192.168.8.101   3.85.133.110    TCP      770 [TCP Retransmission] 65097 → 443 [PSH, ACK] Seq=573 Ack=138 Win=13

    .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  v Source: IntelCor_82:22:10 (98:3b:8f:82:22:10)
      Address: IntelCor_82:22:10 (98:3b:8f:82:22:10)

0000  74 e5 f9 35 a4 df 98 3b  8f 82 22 10 08 00 45 00   t··5···;  ··"···E·
0010  00 38 5c 4a 40 00 80 06  00 00 c0 a8 08 65 c0 a8   ·8\J@···  ·····e··
0020  08 64 2e e0 df 2f 82 a9  61 6a 5a 50 47 f9 50 18   ·d.··/··  ajZPG·P·
0030  02 01 92 44 00 00 74 68  65 20 61 6e 73 65 77 72   ···D··th  e answr
0040  20 69 73 20 6e 6f                                   is no

wireshark_شبكة Wi-FiKYCPD1.pcapng    Packets: 482 · Displayed: 426 (88.4%) · Dropped: 0 (0.0%)    Profile: Default
```

Figure 47: Wireshark analysis, sending decrypted message from server to client.

8. Problems with solutions:

Problem (1): Installing PyCrypto library, we tried different ways such as the command "`pip install pycrypto- upgrade`" and "`pip install crypto`" but all of them generate error occurs while importing this library, the editor didn't recognize it .

Solution (1): install it using this command at the terminal of the programming environment "`pip install pycryptodome`".

Problem (2): When creating AES object or calling encrypt or decrypt method, we faced this error "Object type> class 'str <'cannot be passed to C code".

Solution (2): we encoded the strings of AES object and encrypt or decrypt method parameter to bytes type using `.encode('utf8')` method .

Problem (3): After decrypting the ciphertext, there is `b'` before the text .

Solution (3): to remove `b'` prefix from a byte string, decode the string using the `.decode()` method .

Problem (4): When we type a word with a length less than or greater than 16 bit and send it to encrypt or decrypt method, we faced this problem "Data must be padded to 16-byte boundary in CBC mode".

Solution (4): we try different methods to solve this problem by padding the word, it works in encrypt method but doesn’t work in the decrypt method. So, we assume that our code just accepts a word of 16-bit, in the future, we can generalize our code to accept any length of the word .

Problem (5): how to distinguish at the server side if the word is ciphertext or plaintext, to perform the operation accordingly .

Solution (5): we searched google, there is no definite way to know if the string is encrypted or not. One of the ways is to check the string manually if it is like a ciphertext or not. Another way is to prefix the encrypted string with a special string like `"!=!enc!=!"` then you can determine at the server when you receive the word whether a piece of string is encrypted or not by checking it is start with this special string, we did this method but it generates a lot of problems. After that, we suggested sending an option number to indicate the type of word or type of operation to be performed after sending the word .

Problem (6): Setting up the network, we try to connect each device with the same Wi-Fi connection and run the project ,we encountered a connection problem, as the client could not find the server .

Solution (6): the solution was to change the Wi-Fi settings used for both devices, it was changed from public mode (which hid the device in the network) to private mode .

9. Conclusion:

In the end, the socket is the basis for all network applications, which may vary with different additions to it, and depending on the purpose of the application, such as choosing the type of connection, in our case, we chose connection-based transmission by TCP packets, or it can be programmed using connectionless, represented by UDP packets.

We have learned a lot of things during this project. In the future, we could program the socket using different encryption algorithms, and allow more than one connection from more than one client at the same time.

10. References:

[1] Kurose, J. and Ross, K., n.d. *Computer networking Top-down approach.* 8th ed. pp.159 - 165.

[2] Sending Encrypted Strings Using Socket in Python. [online] Available at: < https://stackoverflow.com/questions/27287306/sending-encrypted-strings-using-socket-in-python> [Accessed 20 November. 2021].

[3] Baeldung n.d. [online] Available at: <https://www.baeldung.com/java-aes-encryption-decryption> [Accessed 10 Dec. 2021].

[4] Pycryptodome.readthedocs.io. Classic modes of operation for symmetric block ciphers — PyCryptodome 3.11.0 documentation. [online] Available at: <https://pycryptodome.readthedocs.io/en/latest/src/cipher/classic.html#cbc-mode> [Accessed 27 November 2021].

[5] Python, R., Socket Programming in Python (Guide) – Real Python. [online] Realpython.com. Available at: <https://realpython.com/python-sockets/> [Accessed 14 November 2021].

[6] *Python Coding- Encrypted Client/Server TCP Communication*. [online] Available at: <https://tech.awilk54.com/2019/02/python-coding-encrypted-clientserver.html?m=1> [Accessed 10 Dec. 2021].

11. Appendixes:

Along with this report document, in the same .zip folder, there are the TCP client – server source code as a python file, the captures of the client and the server code and comments, and a video of the demo representation of our project.