SecureP2P communication client application with dedicated central server and fully end to end encrypted communication.

Introduction

The SecureP2P client application that has been implemented in Python with GUI (Graphical User Interface) provided by Python's default Tkinter GUI framework, uses state of the art encryption algorithms and primitives for establishing secure communication between two users that are registered with the database of our central server. We are using the `cryptography` package for using the cryptographic primitives like **Elliptic Curve Cryptography (ECC)** and ciphers like **AES (Advanced Encryption Standard)**. We have used **ECC** as our public key cryptosystem**,** for establishing a shared secret which will be hashed using **SHA-256** to develop a shared secret key of a fixed size of 32 bytes (256 bits), since we are using **AES** encryption in **CTR** (counter) mode with a 256-bit key size for maximum security possible. The reason why we have chosen Elliptic Curve Cryptography as our public key cryptosystem is because of the fact that Elliptic Curve keys, both public and private, are much smaller in size compared with **RSA** and ECC offers stronger security than **RSA.** The reason why we have chosen **AES** is because **AES,** has become the 'golden standard' for symmetric encryption ever since it's standardization by the **United States (US) Government**. We have chosen the **CTR** mode, because the counter mode converts **block ciphers** to ***stream ciphers*** and we get some good features like different ciphertext for same subsequent plaintext and the length of the ciphertext will be exactly equal to the plaintext and there is no need to worry about padding.

Description of security features and demonstration

In this section the security description of the application is discussed in detail. We have developed two layers of security here. First every client app, that connects with the central server will establish their own respective **shared secret keys** for communicating with the server **alone,** via **ECDH (Elliptic Curve Diffie-Hellman)**. The client app uses its shared **secret key**, to initialize the **AES** encryption cipher to encrypt login information or registration information, that will be sent to the server, and to decrypt the response from the server. This is the first layer of security. Note that the purpose of this layer of encryption is to protect the username and password when they are sent to the server for authentication or for registration. But even after this login / registration information has been successfully sent to the server, this first layer of encryption is still used for encrypting further data transfer from the client app to the central server, irrespective of the data contents. When the user logged in with the server, via a client app, wants to communicate securely with another user logged in with the server via another client app, the communication initiating user (first user) sends their **public key** to the other user via the central server, to the second user and the second user also sends their **public key,** to the first user and thereby they both establish a shared secret key using their respective **private keys** (which they both keep as a secret, to themselves) and the shared secret key is used by the respective client apps, in encrypting messages sent and decrypting messages received. This is the second layer of encryption that provides the **end-to-end** encryption feature. The main advantage of this feature is that we have double protection while the data is in transit, and while the messages sent by a user is inside the server (so that it can be relayed to the right user), the messages *themselves* are still encrypted with military grade encryption of the AES algorithm using the shared secret key which ***the central server itself has no knowledge of***!

It is to be noted that there are two types of secret keys at play here, one is the secret key **established with the server**, via **ECDH** by the individual client applications themselves (which is different for every client) and then we have the **shared secret key** established between two client applications (also via **ECDH)**, that are interested in communicating with one another.

Now, we look at a few functions in the source code that will be used for generating public and private keys, as well as for establishing the shared secret key that will be used for symmetric encryption using the AES cipher. Most of the other codes in the

source code are concerned with the accurate transfer of public keys, encrypted messages etc. The entire working of the application is very intricate, and adequate comments have been added to the source code for intuitive understanding of the reader.

```
# Function to generate our (PEM encoded) public key & the private key (as an object) for performing ECDH
# Returns a tuple consisting of the PEM encoded public key (first element) and the private key object
(secret) , that we can use for,
# establishing the shared secret.
def generate_ecdh_keys():
    # We are using the SECP256K1 elliptic curve.
    # Because its security is widely accepted amd used for securing popular crypto currencies like
Bitcoin.
    private_key = ec.generate_private_key(ec.SECP256K1())

    # We obtain the PEM encoded public key from the private key
    public_key = private_key.public_key().public_bytes(encoding=serialization.Encoding.PEM,
format=serialization.PublicFormat.SubjectPublicKeyInfo)
    # We return the public key and private key together as a tuple.
    return (public_key, private_key)
```

Figure – 1

The function **`generate_ecdh_keys()`** generates the public key and private key for the Elliptic Curve Diffie-Hellman process. It uses the **`ec`** (Elliptic Curve) class from **`cryptography.hazmat.primitives.asymmetric`** module for generating the **private key** based on the Elliptic Curve **SECP256k1**. We have chosen this elliptic curve for performing **ECDH**, because this particular curve has a great reputation in the cryptographic community for being safe and it's used in crypto currencies like Bitcoin. The prime number as well as the order used by the curve is 256 bits in size and hence, we have '256' in the name. This curve can provide practical levels of security for our present day and age. Now, we use this private key for generating our public key, which we return as a **PEM** encoded byte string, since we want to send to the other entity, we are interested in communicating with. This same function is used without any changes by both the central server and client application in their implementations. It is to be noted that the function **`generate_ecdh_keys()`** returns a tuple, the first element being the PEM encoded byte string containing the public key and the second is the **private key** as a Python object. The **private key** is to be kept secret by the caller. But notice that we are using this function to generate public and private keys ***on the fly,*** and we are not storing any our keys (public or private) anywhere and hence there is no need for the end user, client application or central server to worry about key storage!

Next, we investigate the function that establishes the shared secret key using **ECDH**, by using the other entity's public key (server's public key or another user's public key) and our already generated private key.

```
# Function that performs ECDH, with the PEM encoded public key of the other client and our private key.
# It returns a SHA256 hash of the shared secret.
def establish_shared_secret_ecdh(public_key, private_key):
    shared_secret = b""
    # First we initialize the Elliptic Curve public key object from the PEM encoded data.
    ec_public_key = serialization.load_pem_public_key(public_key)
    # The ECDH process is finally performed.
    shared_secret = private_key.exchange(ec.ECDH(), ec_public_key)
    # Now we take the hash of the shared secret which returns a 32 byte byte-string.
    shared_secret = sha256(shared_secret)
    # The hash is returned. This ensures that the shared secret key is always 32 bytes.
    return shared_secret.digest()
```

Figure – 2

The above function loads the public key as a Python object from the first argument (the PEM encoded public key) and then uses the **`private_key.exchange(..)`** method to perform ECDH, using the other entity's public key to generate a shared secret which is a Python byte string. Since we are performing symmetric encryption using AES in CTR mode, with a key size of 256 bits we use

the SHA-256 hash function from Python's **hashlib** module to generate a 32-byte hash of the shared secret. This hash is returned by the function and will be used as the key for AES-256 in CTR mode.

Now, we will move on to the demonstration and we will investigate the code when needed. It is necessary that we make sure to install the necessary packages first before proceeding with the demonstration. Please make sure to run the command ***`pip install cryptography`*** in the terminal and make sure this requirement is satisfied. It is also recommended to use the latest version of Python as well.

To make anything work, the central server needs to be started first. For that, we have opened terminal and navigated to the directory containing our project and we type the command **`python3 central_server.py`** and hit ENTER. The server will report that it started successfully, and it is waiting for incoming connections. This server is a **multi threaded** server and it's capable of handling many client applications at the same time.

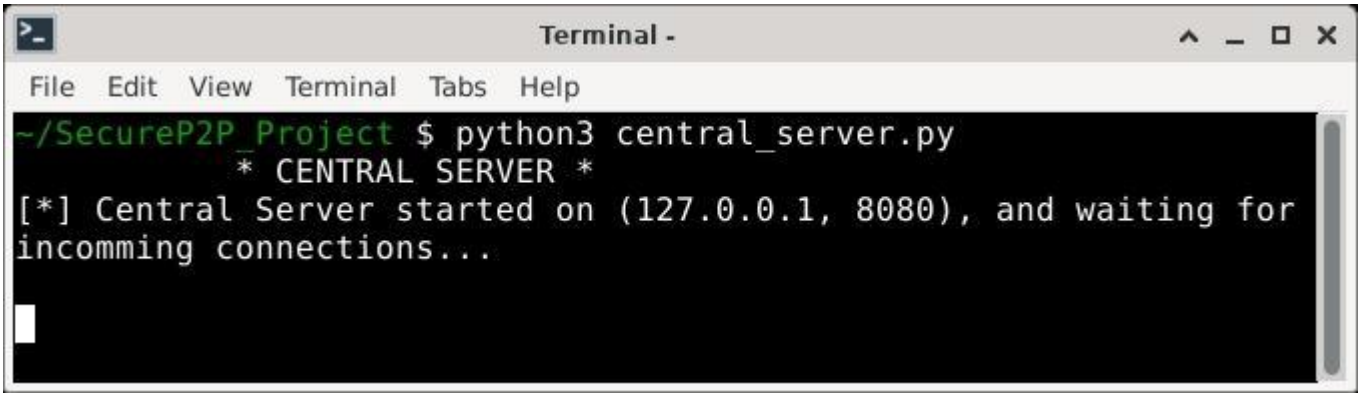


Figure – 3

The above is the screenshot of the server application (central_server.py) started up and waiting for incoming connections. Next we open up a new terminal tab and start up our client application by typing: **`python3 clientapp.py`** and pressing ENTER key. The GUI, of the client application can be seen on the screen (Figure – 4).

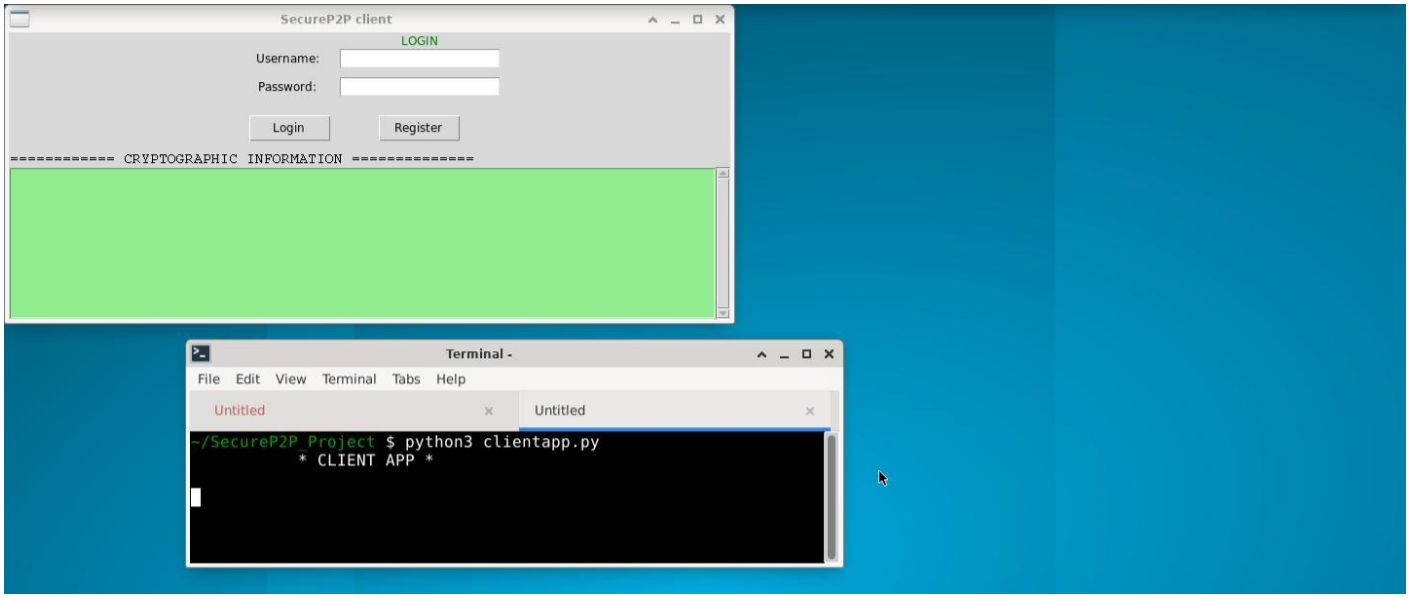


Figure – 4

The SecureP2P client application has Tkinter widgets for providing username, password and two buttons one for login and another for registration. The text widget at the bottom of the window with a green background is where all the cryptographic information like formatted plaintext, ciphertext and other information will be displayed. Now, since we are starting from a fresh slate, no new users are registered on the server's database yet. So, we first register a new user for example **alice** and provide a password **alice#111**, just as an example and press the **"Register"** button. Once registration is successfully completed as shown in the screenshot below.

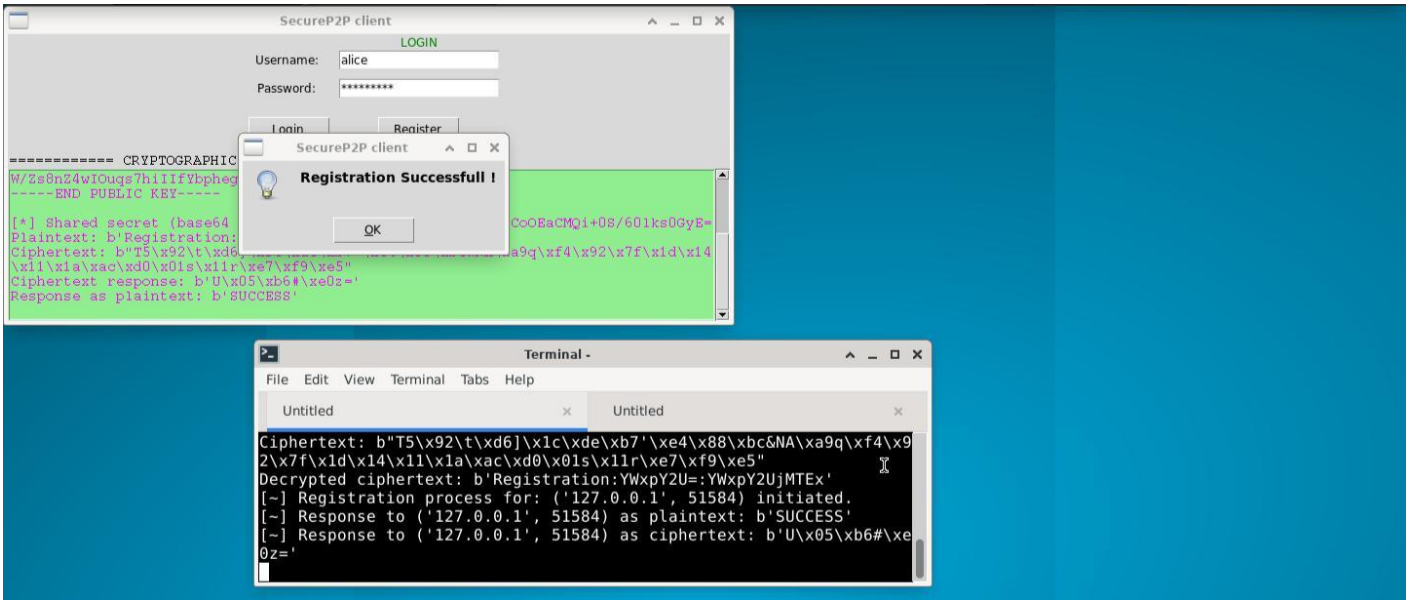


Figure – 5

If we scroll through the output of the central server on the terminal we can find that the server has started a thread for serving the SecureP2P client application and we can find all the details of the cryptographic process like the public key sent by the client application, the public key of the server sent to the client, establishment of the shared secret key as well as the cipher texts sent by the client app and their decryptions as well as the server's responses in plaintext as well as ciphertext generated by the server which are sent to the client application. If we scroll up and then slowly down in the green text widget of the client application we can find similar information and on comparing both the outputs we get a clear picture of how secure communication takes place. Since the **private keys** generated using the cryptography module are based on its internal pseudo random number generator, the keys generated (both public & private, and hence the shared secret and ciphertext) will be different for every trial run. The following is the output extracted from the client application as well as the central server terminal, at the time of writing this report.

Output from the client application extracted from the green text widget:

```
============ CRYPTOGRAPHIC INFORMATION ==============
===== Registration Process =====
[*] Our public key: -----BEGIN PUBLIC KEY-----
MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEmpjZohJsAPL7UhSCgcIyc1a3PqbFv1hx
yW6cou/MqoyaobG3z475epH2k1ZsFg6iH6s6cjPZAnaUNMde5JnJdA==
-----END PUBLIC KEY-----

[*] Central server's public key: -----BEGIN PUBLIC KEY-----
MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEo1pxxYqKZ24f4fcEcHSRUu4Bkq5uUbk2
W/Zs8nZ4wIOuqs7hiIIfYbphegnWjlH/we5J9YuLWs8ytYcwTYx7Kg==
-----END PUBLIC KEY-----
```

Figure – 6.0

```
============ CRYPTOGRAPHIC INFORMATION ==============
W/Zs8nZ4wIOuqs7hiIIfYbphegnWjlH/we5J9YuLWs8ytYcwTYx7Kg==
-----END PUBLIC KEY-----

[*] Shared secret (base64 encoded): asxjKIsFFk6etB2oILCVZCoOEaCMQi+0S/60lks0GyE=
Plaintext: b'Registration:YWxpY2U=:YWxpY2UjMTEx'
Ciphertext: b"T5\x92\t\xd6]\x1c\xde\xb7'\xe4\x88\xbc&NA\xa9q\xf4\x92\x7f\x1d\x14
\x11\x1a\xac\xd0\x01s\x11r\xe7\xf9\xe5"
Ciphertext response: b'U\x05\xb6#\xe0z='
Response as plaintext: b'SUCCESS'
```

Figure – 6.1

Output from the server application extracted from the first tab of the terminal shown in the screenshot:

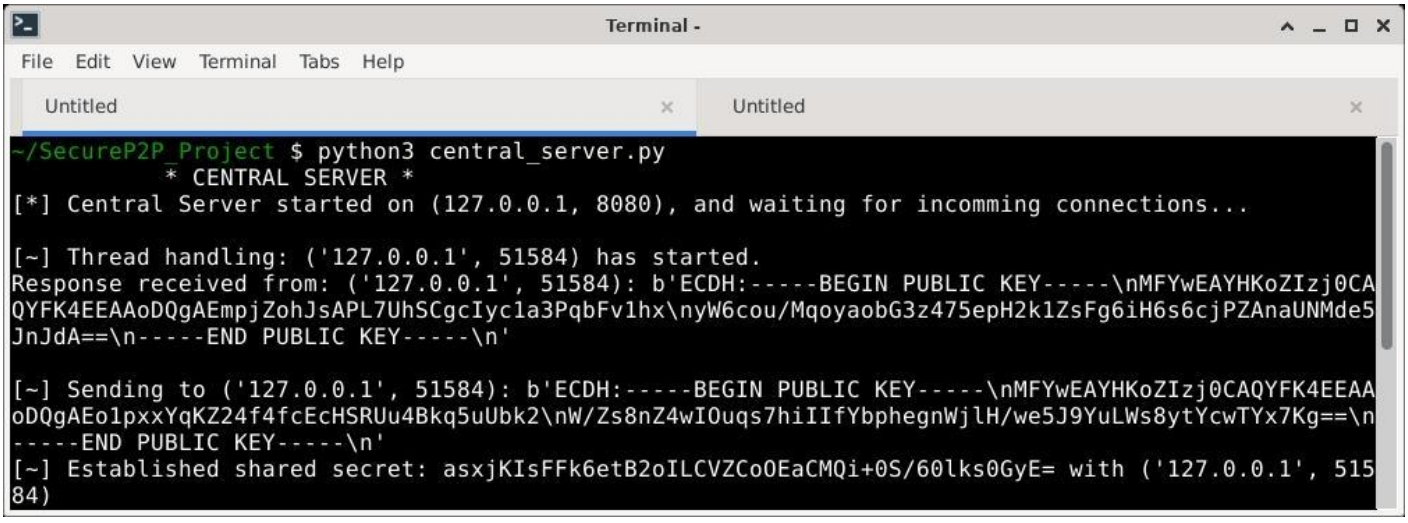


Figure – 6.2

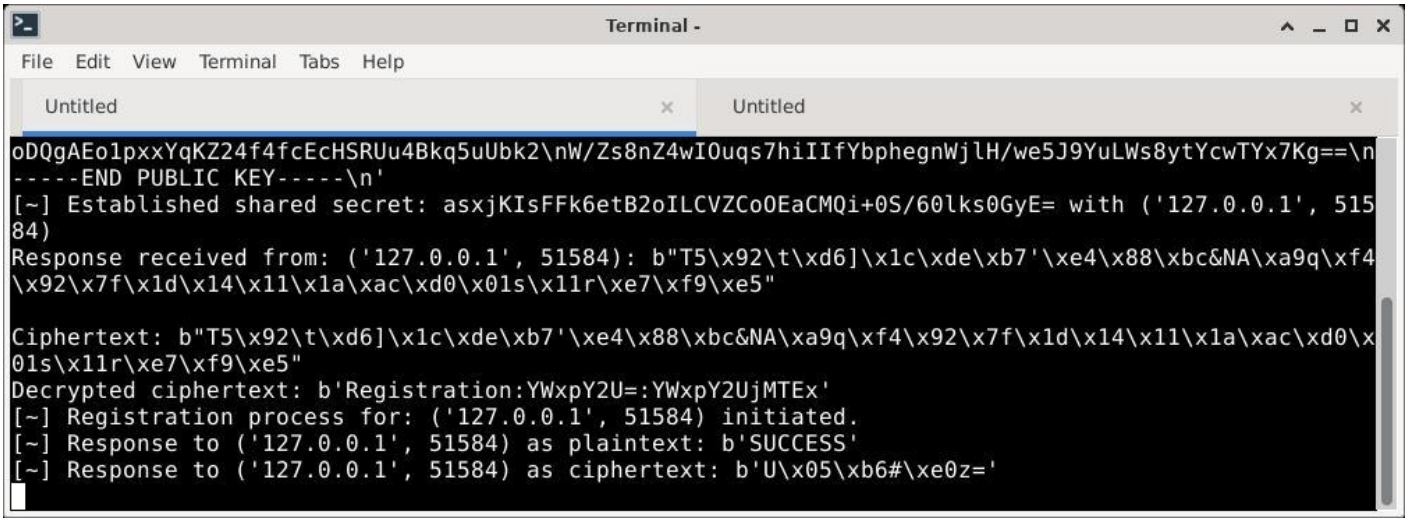
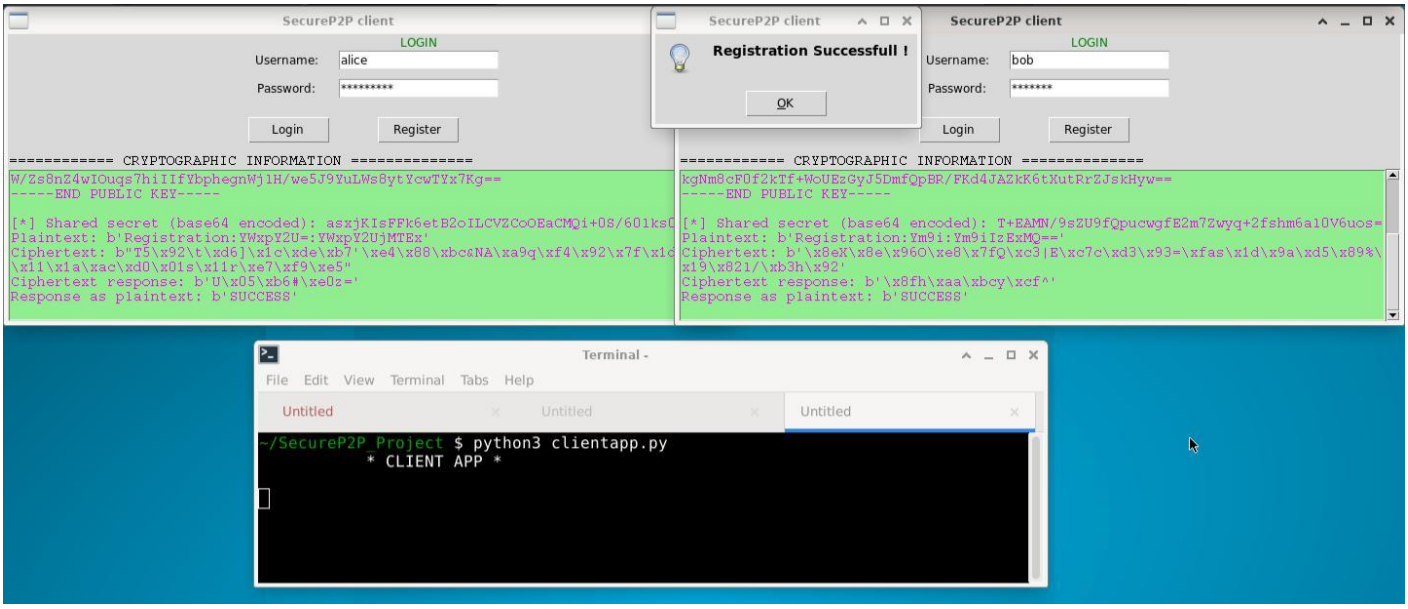


Figure – 6.3

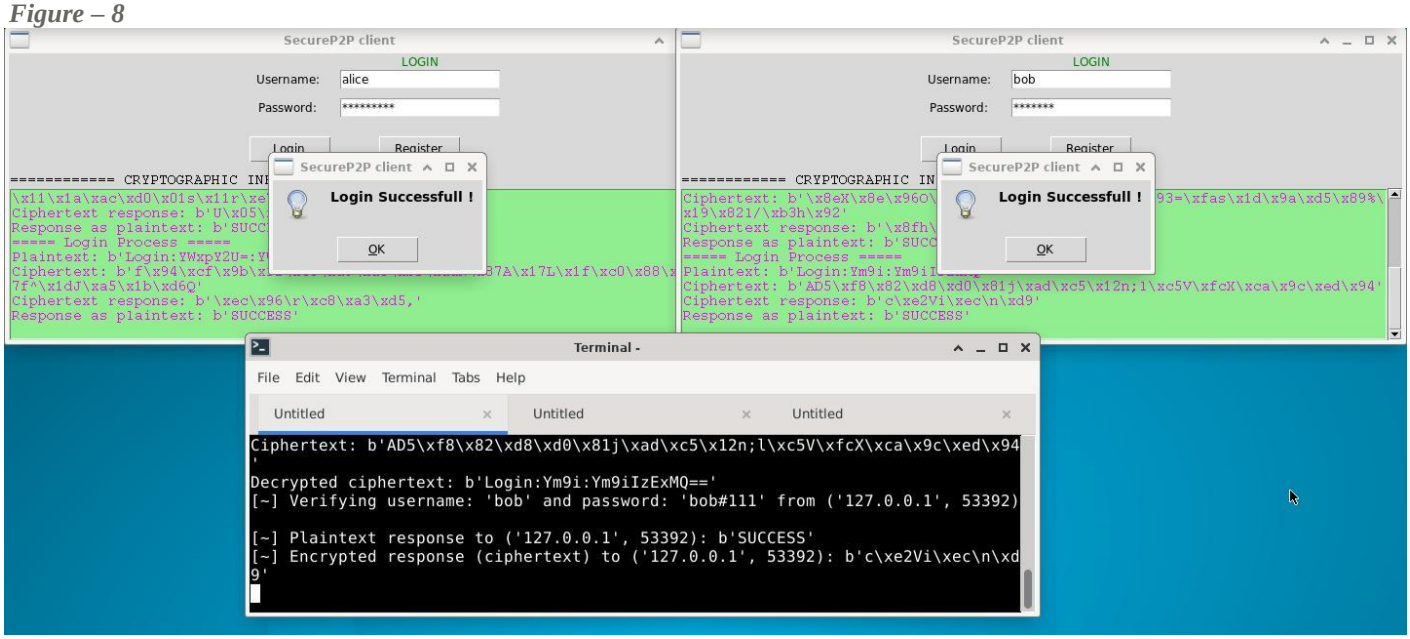
Next, we start up another client application by opening a third tab on the terminal emulator and running the command: python3 clientapp.py as before. Like before, we register a new user having username **bob** and we give this user a password **bob#111** , just for the sake of this example and press the **"Register"** button. If we check the outputs like before, we can see that the client application sends the public key, that it generated on the fly, the server's public key which it also created on the fly, is received by the client application and a new shared secret key is established for encrypted communication between our server and the new client application and the registration process is successfully completed, because **bob** is a new user in the server's database as shown in the attached screenshot below.

Figure – 7



Now we can press the **"Login"** button on both the clients, and we can see that the we get confirmation message boxes for the successful login operations as shown in Figure – 8.

Figure – 8



After pressing the “OK” button on the respective message boxes we can see clearly that the GUI of the SecureP2P client application transforms right before our eyes like ***origami*** to evolve to a GUI capable of sending & receiving messages. Please refer to the screenshot attached below.

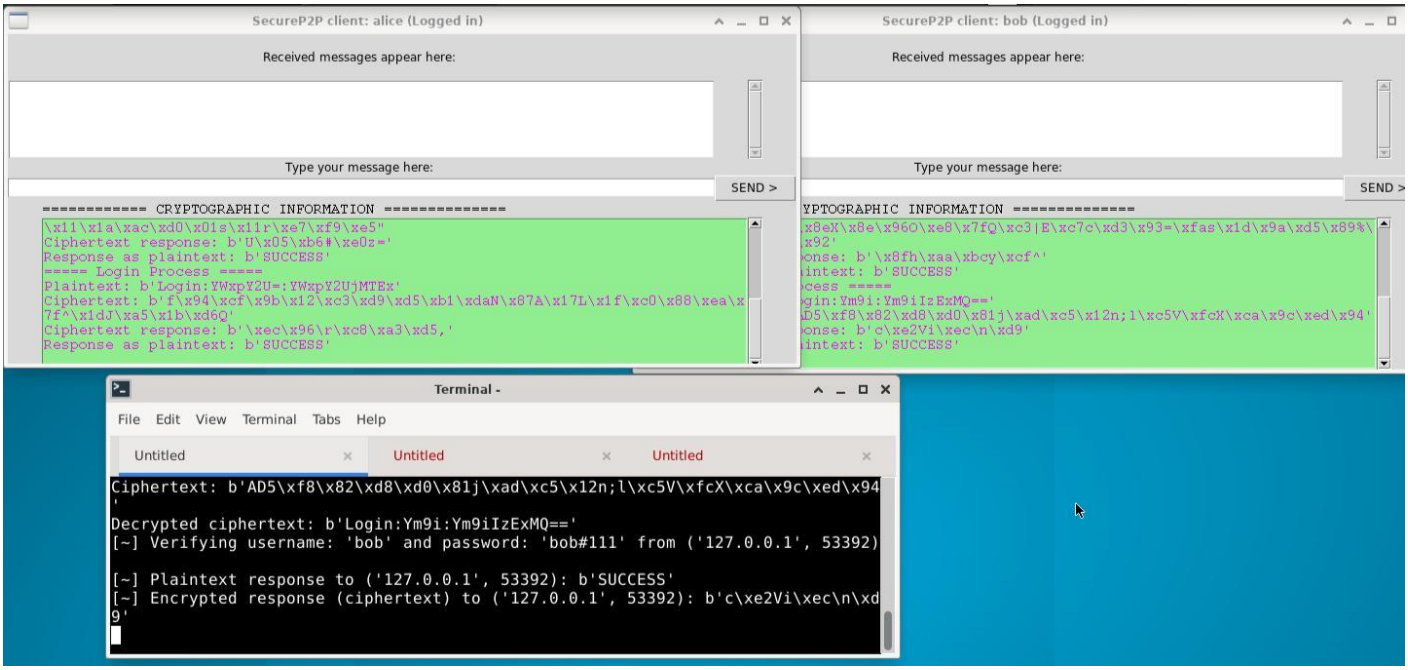


Figure – 9

Now, we demonstrate the sending of end-to-end encrypted messages back and forth between the two logged in users. For example, if **alice** wants to send a message to **bob** , alice would have to type in the following format in the entry widget under "Type your message here:" like this: **@bob: … message alice wants to send to bob …** and if **bob** wants to send a message to **alice,** he would have to type in a similar format: **@alice: … message bob wants to send to alice …** and then they would have to press the **SEND >** button. Initially, there is no shared secret key established between **alice** and **bob**, but at the very instant one of them presses **SEND >** , the public key of that user is sent to the destination user via the central server and the other client application of the destination user sends their public key back and finally a shared secret key is established between **alice** and **bob** and this key will be used for encrypting communication between **alice** and **bob**. After the shared secret key has been established, the user who wanted to send the message initially (the one who initiated ECDH) must press the **SEND >** button again to send the pending message. Please refer to screenshots below for full clarity.

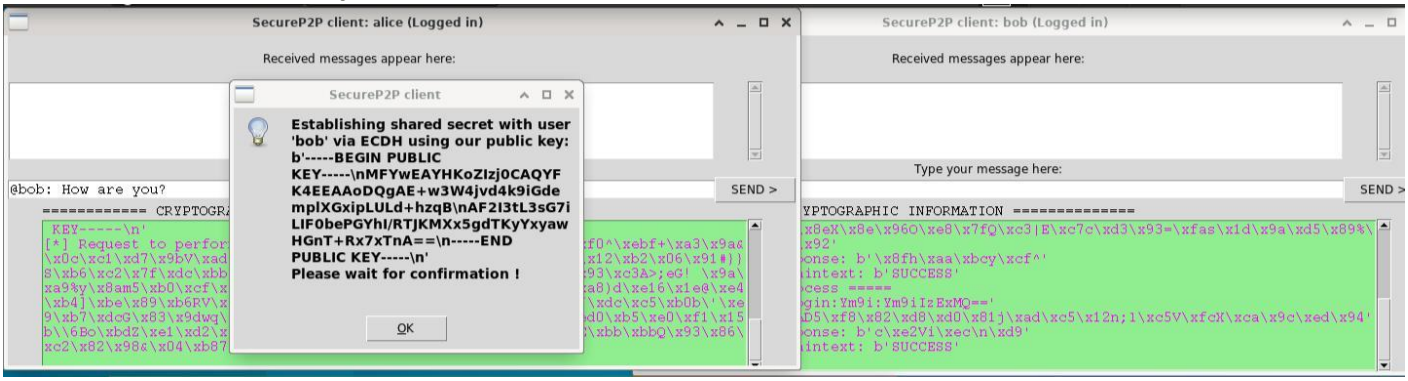


Figure – 10

In the above screenshot we can clearly see that **alice**, is trying to send **bob** the message **How are you?**. When she clicks the "SEND >" button, a message box displays the public key, **alice**, is using for establishing a shared secret key via ECDH with **bob.** Now when we press **OK,** we can clearly see in the next screenshot (Figure – 10.1) that the client application for **bob** receives this public key internally and notifies **bob** using a message box displaying the public key, **bob**, will be using for establishing a shared secret key with **alice.**

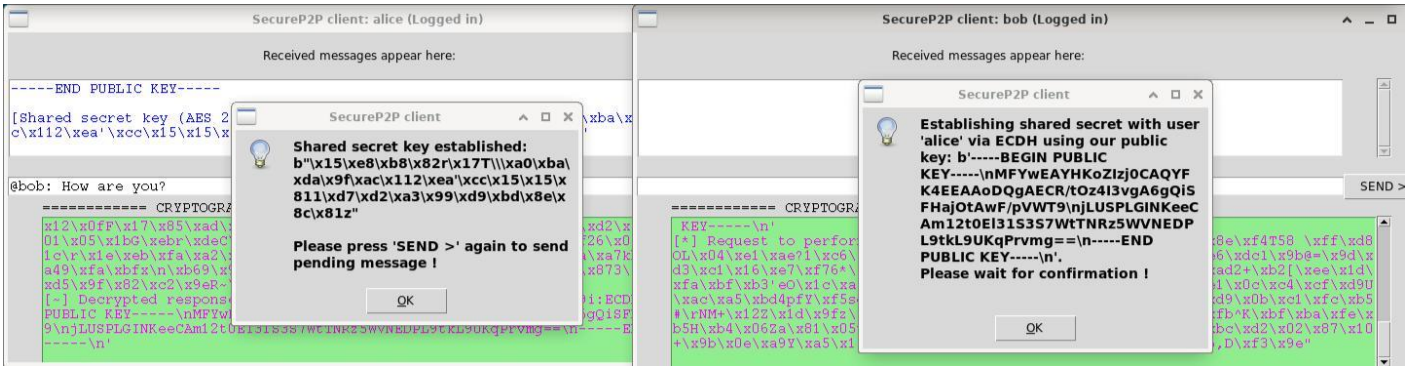


Figure – 10.1

Since, the client application for **bob** has already sent the **public key,** before displaying the message box about the public key we can clearly see that the client application being used by **alice**, has calculated the shared secret key that will be used for symmetric encryption. Now when **bob** presses OK, he will also be able to see the same shared secret key.

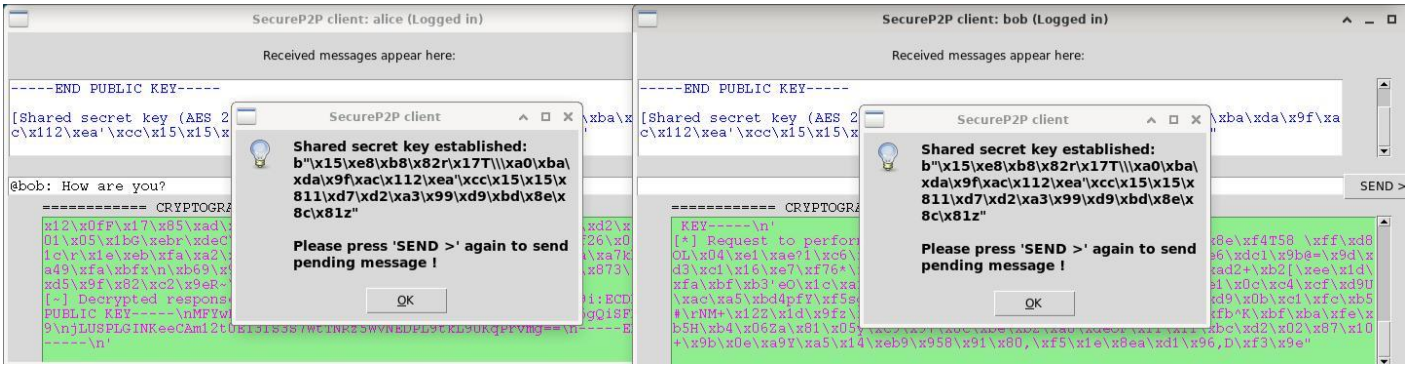


Figure – 10.2

If we look closely, we get to see some output in the upper text window as well. Notice that the output is in blue color. This text window will display the public key sent by the other user using the client application, the shared secret key established with the other user as well as history of messages sent by the other user, both in ciphertext as well as in plaintext.

If we look at the output shown in the client application used by **alice** we can clearly see the received public key sent by **bob.**

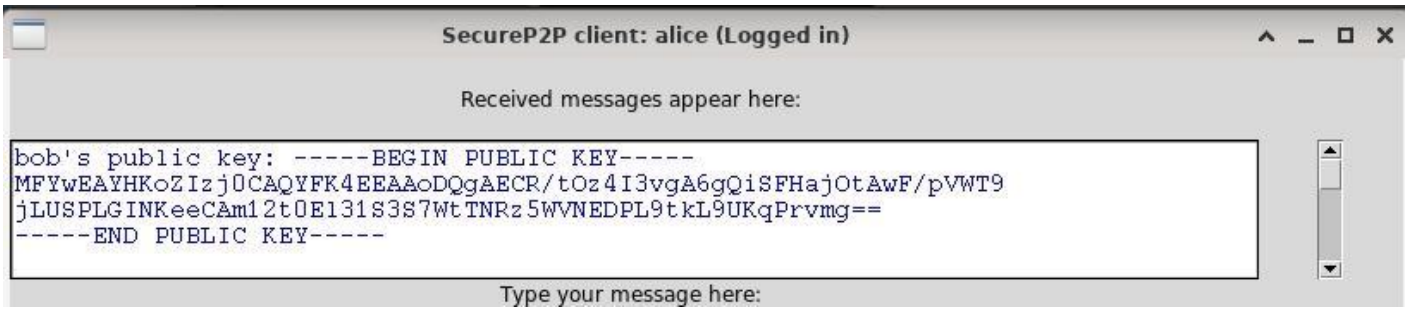


Figure – 10.3

Similarly, if we look into the same section in the client application used by **bob** we can see the received public key sent by **alice**.

Figure – 10.4

Now **alice** presses the **SEND >** button to send the **How are you?** message and we can see that **bob** receives the message and the client application used by **bob** is able to decrypt the message. The decrypted message is shown inside a message box by the SecureP2P client application being used by **bob** as shown in the attached screenshot below.

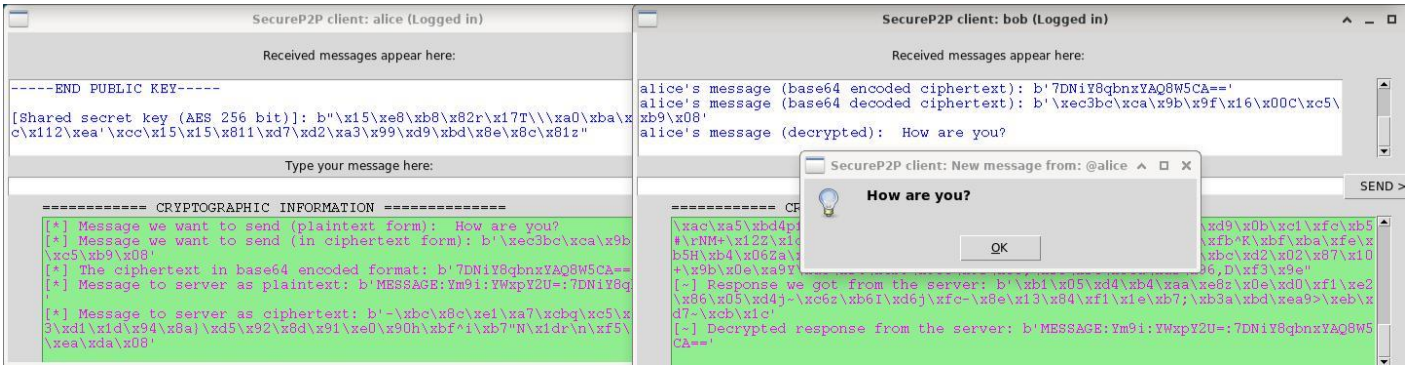


Figure – 10.5

Now, **bob** tries to send a message to **alice**, but since there was a delay in communication of more than **two minutes** (120 seconds) the shared secret already established with **alice** is considered expired and a fresh new shared secret key is established with **alice**, by **bob** by performing ECDH again with a new set of public and private keys. We can see in the screenshot below that when **bob** attempts to send a message to **alice**, the client application used by **bob** provides a warning stating that the established shared secret key has expired and **ECDH** needs to be performed again.

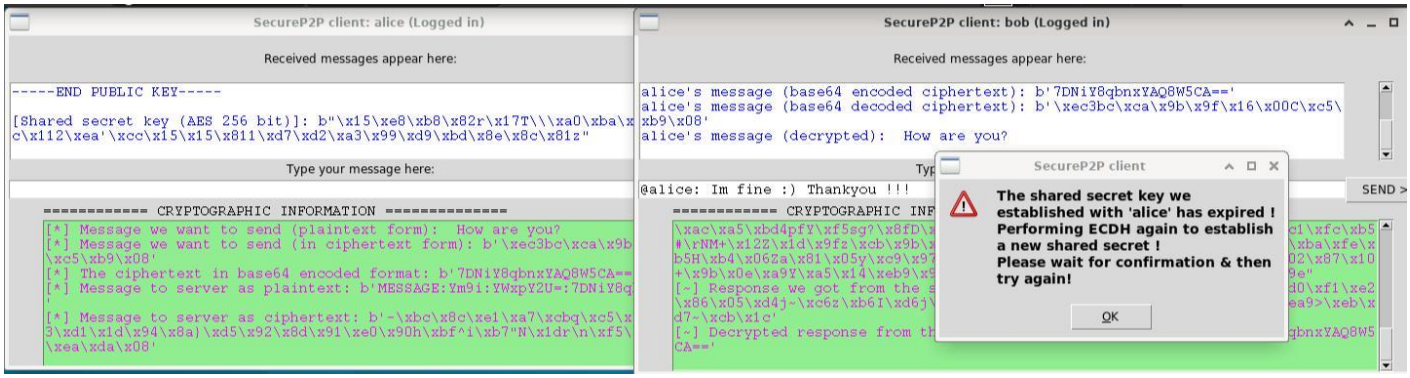


Figure – 10.6

We can clearly see that the client application used by **bob** sends a new public key, so does the application used by **alice,** and a new shared secret key is established before **bob** is allowed to send its message. Please refer screenshots from Figure – 10.7 to Figure – 10.9

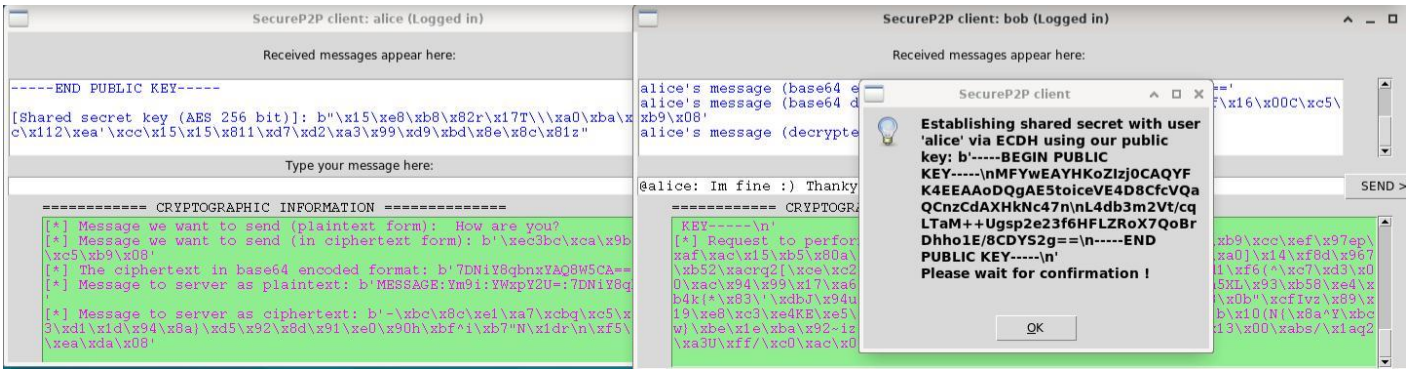


Figure – 10.7

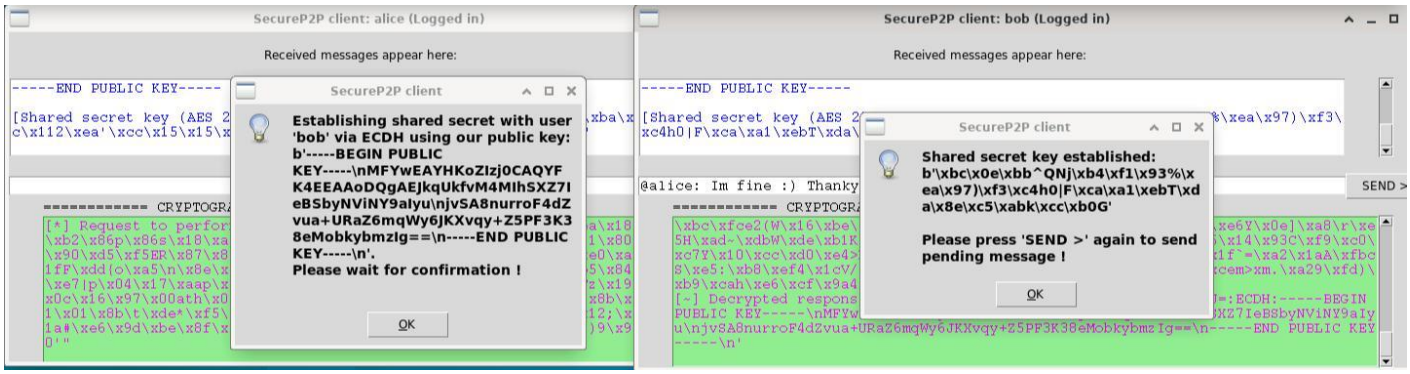


Figure – 10.8

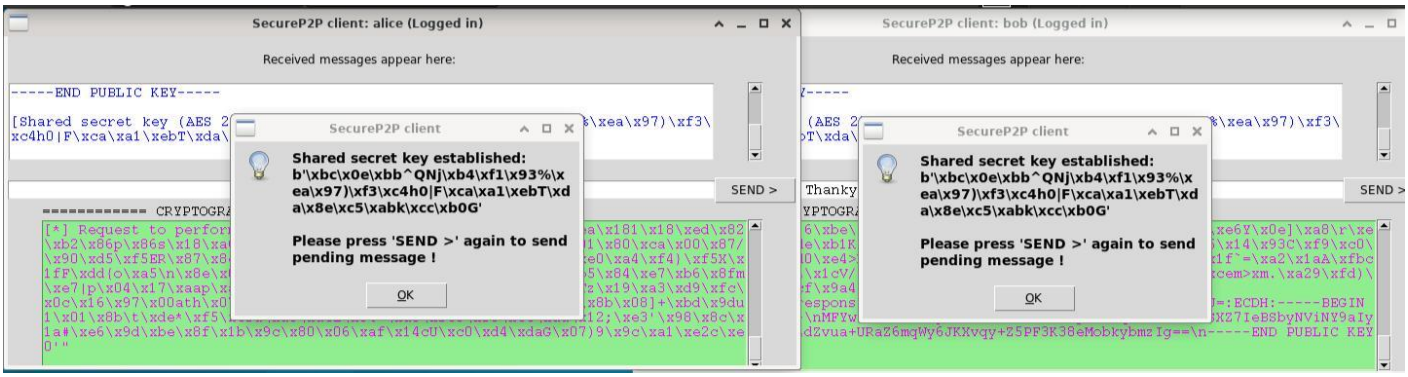


Figure – 10.9

After the shared secret key has been re-established then **bob** can send the message he wanted to send and **alice** successfully receives it in the same way **bob** received her messages earlier.

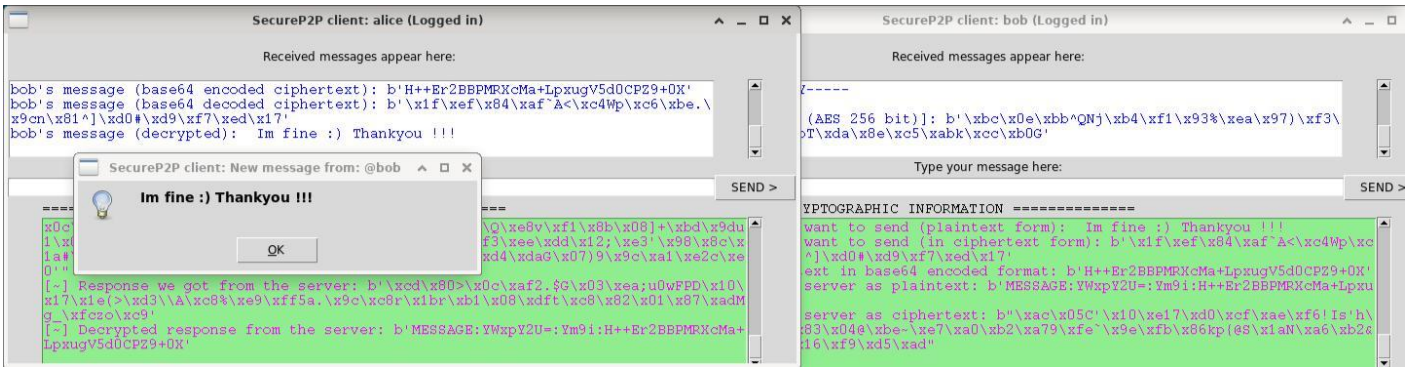


Figure – 10.10

We have demonstrated the core security features of our SecureP2P client application and at this point it is worth noticing that the central server is the application responsible for relaying the public keys generated by the individual client applications and for relaying the encrypted messages from one logged in user to another logged in user.

The following is a screenshot of the output produced on the server's terminal while relaying public key of **alice** to **bob.**

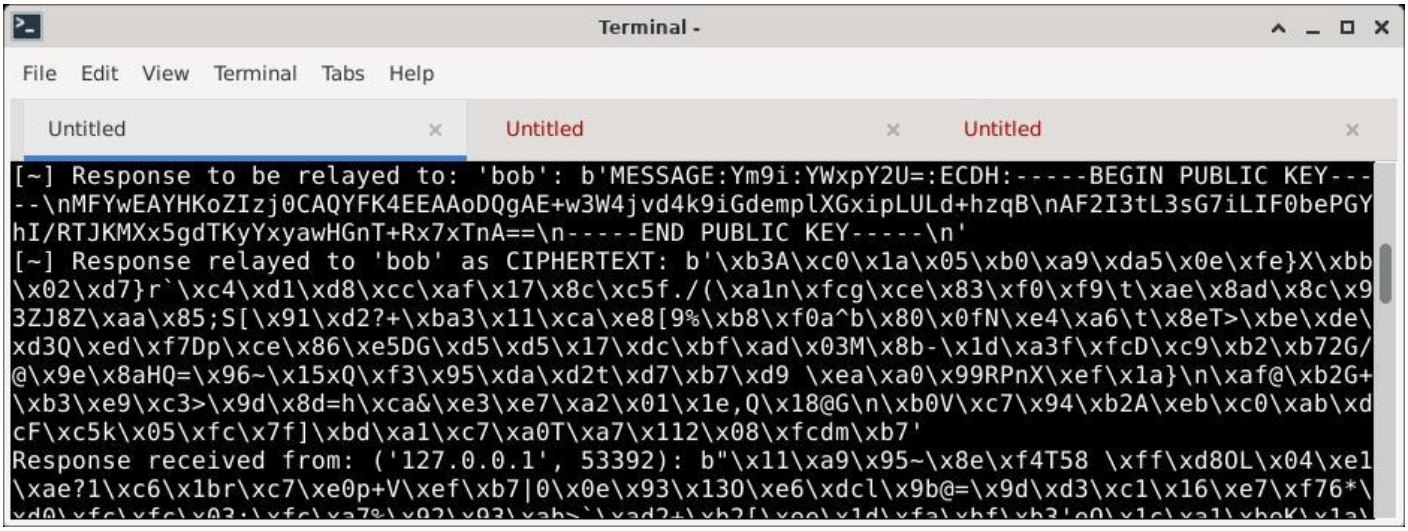


Figure – 11

If the server's output terminal is examined, we can clearly see that the server transfers the messages correctly to the right logged in user. It is to be noted that the messages sent by one user to another are encrypted using the shared secret key established between the users concerned with the communication and therefore the server cannot read the contents of the message sent by **alice** to **bob** and **bob** to **alice.** However, we have added some meta data in the responses from the client application to the server and this meta data is only encrypted using the shared secret established by the individual client applications with the server and hence the server can decrypt such information and understand which user to send the encrypted messages (or public keys) to. Before relaying messages to a particular user, the server checks if the destination user is logged in with the server.

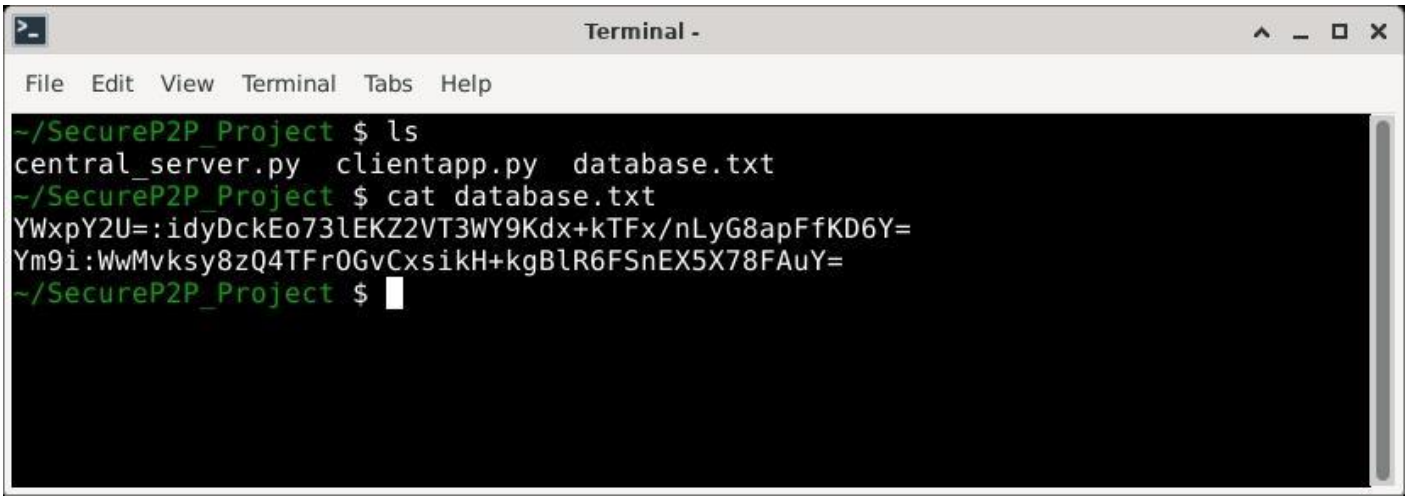
How is the entire system secure?

For an adversary intercepting the communications they will have no information of the user that another user is talking to because such information is encrypted using the symmetric cipher that is instantiated with the shared secret key established between a particular client application and server. The messages themselves are encrypted using the shared secret key established between the two users communicating. So, there is double protection to the messages while data is in transit. Also, even if the adversary gains access to the server via a side channel attack, or via hacking they will still not be able to read the contents of the message sent by **alice** to **bob** or vice-versa, because the adversary cannot find the shared secret key established between **alice** and **bob** since the ***server does not have knowledge of it***!

Implementation details

Now we focus on some very important implementation details and some challenges that were needed to overcome during the development process. First, we need to understand the protocol of communication between the client application (SecureP2P) and the central server and how the messages are relayed by the server to the right client applications referenced by their destination usernames. The **central server** is implemented in central_server.py and it's a **multi threaded** server, which means it can handle client applications (SecureP2P client applications) concurrently. When the server first starts up it checks if the **`DATABASE_FILE`** (hard coded as **`"./database.txt"`**) exists or not. This file contains the ***base64*** encoded forms of the registered usernames and the ***base64*** encoded forms of the SHA-256 hashes of the passwords of the registered users separated by ":". Every line in the database corresponds to the login credentials for a unique registered user. Since we have already registered two usernames **alice** and **bob** as part of our demonstration we can check the "database.txt" file in the current working directory of the server. We can use the **cat** command to view the contents of this file in terminal and it's easy to understand the above explanation now.

Figure – 12



We can clearly see the base64 encoded usernames and base64 encoded SHA-256 password hashes separated by ":" in Figure – 12

If this file DATABASE_FILE exists, then the server application loads the usernames and password hashes into a dictionary data structure for easy lookup by username later when needed. The access to this dictionary is protected using mutex locks because it can be accessed by other threads concurrently and we need to prevent race conditions. But now, since the server has just started and there

are no other threads running now (at server startup) you can see that no mutex locks are used while loading the dictionary at server startup. The following screenshot (Figure – 13) is the code snippets that does database loading at the time of server startup.

```
    # We need to load the database containing the user names and passwords.
    if os.path.exists(DATABASE_FILE):
        with open(DATABASE_FILE, "r") as fObj:
            # We read the file line by line.
            for line in fObj:
                # We remove leading & trailing white spaces.
                line = line.strip()

                if len(line) != 0:
                    # Every line contains the username & password SHA256 hash (all base64 encoded)
separated by ":"
                    tokens = line.split(":")

                    # The username is mapped to the password hash and this mapping is made effective
using a dictionary
                    password_database[tokens[0]] = tokens[1]
```

Figure – 13

The server then creates a TCP socket and binds to a hard-coded SERVER_IP and PORT to listen for incoming connections. In our implementation **SERVER_IP = "127.0.0.1"** and **PORT = 8080** . This can be changed if required but the only thing to keep in mind is that these same variables should be updated in the corresponding source code of **clientapp.py** and everything will be okay. In the central server's implementation, we also register a signal handler for intercepting Ctrl + C (SIGINT) signal so that we can gracefully shutdown the multi threaded server when we are done with it. The thread functions that handle a new connection internally run a loop that checks a global flag (in a thread safe manner) before entering a blocking **recv(..)** that waits on the socket for response from the connected client. This flag can be toggled to **False** by the signal handler and hence prevents the threads from entering another blocking state. Figure – 14 is the implementation of the signal handler. Please refer to the comments for more information.

```
# We implement the signal handler for handling SIGINT to stop our server.
def signal_handler(signum, frame):
    # In the signal handler we make the "run" variable global and set it to False, so the server can stop
gracefully.
    global run
    run = False

    global server_sock
    server_sock.close()
```

Figure – 14

```
    while run:
        # We wait to accept(..) a new connection.

        try:
            cli_sock, cli_address = server_sock.accept()
        except:
            # If we are unable to accept(..) a connection we simply continue.
            continue


        # We append the socket descriptor to the list of all client sockets ever created.
        # Since this list is a critical section we use mutex locks to protect the access.
        mutex_lock.acquire()
        list_of_cli_socks.append(cli_sock)
        mutex_lock.release()

        t = Thread(target=handle_connection, args=(cli_address, cli_sock))
        # We start the thread!
        t.start()
```

Figure – 15

Whenever a new connection is **accepted** by the server via the **`accept()`** call of the socket library a new thread is spawned by the main thread of the server as shown in the code snippet above (Figure – 15).

The function **`handle_connection(..)`** (the function that runs as a thread) is the most important function for us since it contains all the logic of handling the communication with the client applications, establishing shared secret keys with them, registering and authenticating users and relaying of messages from one client application to another based on the destination username. The following code snippet shows the beginning few lines of the implementation of **`handle_connection(..)`**

```
# Function that will run as a separate thread for handling every newly accepted connection.
def handle_connection(cli_address, cli_socket):

    mutex_lock.acquire()
    print(f"[~] Thread handling: {cli_address} has started. ")
    mutex_lock.release()


    # The shared secret key for AES CTR encryption / decryption established after ECDH with client app.
    shared_secret_key = b""

    # The cipher we use for encrypting / decrypting data using AES CTR
    cipher = None
```

Figure – 16

To better understand how the **`handle_connection(..)`** works internally we are now taking a break from looking in the server's code and now we look into the implementation of the client application **`clientapp.py`** as described below. The core functionality of the client application is implemented in a class called **ClientAPP.** Like in the server's code the IP of the server and port are hard coded here too, so client application can successfully connect to the server before doing anything.

```
# Main entry point of our application.
def main():
    print("\t   * CLIENT APP *\n")

    # We start our client application.
    app = ClientAPP(SERVER_IP, PORT)

    return 0




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

Figure – 17

```
# The main application class, for our client!
class ClientAPP:
    # The constructor of the class that takes the address of the central server's IP and port number.

    def __init__(self, server_ip, port):
        # The 32 byte shared secret key used for AES CTR  encryption / decryption with the central server
        self.shared_secret_server = b""

        # The 32 byte shared secret key used for AES CTR encryption / decryption of the messages with the
other user we are interested in communicating.
        self.shared_secret_user = b""
        # The time in which the shared secret key has been established with the other client
        self.creation_time_ssu = 0
        # The cipher class we use for both encryption and decryption.
        self.cipher = None
```

Figure – 18

The reason for encapsulating the code in a class is because there are a lot of objects to keep track of and keeping them inside a class is the best way to manage it. If we look through the code of the class, we can see that there are several widgets that are being created and placed on the frame. The most important widgets at this stage are the **"Register"** and **"Login"** buttons. In Tkinter, the buttons can be bound to a callback function to be called when pressed. This can be clearly understood from the following screenshot of the code:

```
# Next we create our text entry widget for username & password.
self.uname_text = ttk.Entry(self.frm)
self.passwd_text = ttk.Entry(self.frm, show="*")

# We create our login button & registration button.
self.login_button = ttk.Button(self.frm, text="Login", command=self.login)
self.registration_button = ttk.Button(self.frm, text="Register", command=self.register)

# We create our string var object for storing username & password.
self.username = tk.StringVar()
self.password = tk.StringVar()
```

Figure – 19

Now for code reusability, both the functions **self.login()** and **self.register()** call a more general function **self. _login_or_register** that takes an argument **action**, which could either be "Login" or "Registration" respectively, that tells the function what to do. If a shared secret is not yet established with the server, the function **self. _login_or_register** will establish a shared secret key with the server irrespective of the **action** argument. The function first checks if a shared secret key is established, otherwise it generates the client application's elliptic curve public and private keys and sends a message to the server in the following format: **ECDH:… client_application_PEM_encoded_public_key …**

```
        if len(self.shared_secret_server)==0:
            # We generate our Elliptic Curve (SECP256k1) public & private key pairs.
            # Our private key must be kept secret.
            # Our public key must be send to the server so it responds with its own public key.
            # The public key returned from generate_ecdh_keys(), is a PEM encoded byte string.
            public_key, private_key = generate_ecdh_keys()
            # We send our public key to the central server to establish a shared secret that can be used
for creating a symmetric key.
            self.insert_to_text_widget(self.text_widget, tk.END, ("[*] Our public key: %s\n") %
(public_key.decode("utf-8"),))

            self.serv_sock.send("ECDH:".encode("utf-8") + public_key)
            # We finally receive the public key of the central server which we can use to establish the
shared secret using our private key!
            public_key_server = self.serv_sock.recv(MAX_SIZE).split(b":")[1]
            self.insert_to_text_widget(self.text_widget, tk.END, ("[*] Central server's public key:
```

Figure – 20

From the above we can see that that the client application also receives the public key of the server and using that it immediately generates a shared secret via **ECDH**, using which an **AES** cipher in **CTR** mode is instantiated so that now login / registration

```
253             self.insert_to_text_widget(self.text_widget, tk.END, ("[*] Central server's public key:
    %s\n") % (public_key_server.decode("utf-8"),))
254
255
256             # Finally we can use the public key of the server and our private key to establish our shared
    secret.
257             self.shared_secret_server = establish_shared_secret_ecdh(public_key_server, private_key)
258             self.insert_to_text_widget(self.text_widget, tk.END, "[*] Shared secret (base64 encoded):
    {0}\n".format(base64.b64encode(self.shared_secret_server).decode("ascii")))
259
260
261             # We instantiate a new cipher object for AES CTR encryption / decryption.
262             self.cipher = Cipher(algorithms.AES(self.shared_secret_server), modes.CTR(b"\x00"*16))
263             self.enc = self.cipher.encryptor()
264             self.dec = self.cipher.decryptor()
```

credentials can be send to the server in encrypted form. **Figure - 21**

Now, we look into the **handle_connection** function in our server's code to see how the server handles different messages from the client application.

```
159         print(f"Response received from: {cli_address}: {response}\n")
160         if len(shared_secret_key) != 0:
161             print(f"Ciphertext: {response}")
162             # We decrypt the response with the previously established shared secret key
163             response = dec.update(response)
164             print(f"Decrypted ciphertext: {response}")
165
166         # Since our message has fields separated by ':' we tokenize based on the character ':'
167         tokens = response.decode("utf-8").split(":")
168
169         if tokens[0] == "ECDH":
170             # This is our signal to perform ECDH with the client and generate a shared secret
171             # that will be hashed using SHA256 to generate a symmetric key for AES encryption/decryption
    in counter mode.
172             # The second token is the public key of the client application in PEM encoded format.
```

Figure – 22

The server always checks if the shared secret key has been established with the client application. If so it decrypts the response from the client application using the **decrypting context object** called **`dec`** in the code, and using it's **`update()`** method. Once the secret key with the server has been established, then the server must **decrypt** every message from the client application before it can tokenize on the basis of the ":" symbol. If the first token, **tokens[0]** is **`"ECDH"`** then the server understands that it needs to store the public key of the client application in a variable, generate a new pair of keys, send it's public key to the client application in **PEM** encoded format and establish a shared secret key that will be used for instantiation an **AES** cipher in **CTR** mode or in the **new design** our custom cipher class (will be discussed in detail later).

If **tokens[0]** is **`"Login"`** or **`"Registration"`,** the server performs login or registration respectively. In the case of the login process the server checks if the username in bsse64 encoded format is into loaded database in memory in a thread safe manner using mutex locks, and if found it checks to see if the base64 encoded SHA-256 hash of the received password matches with the one in the database and if so, the server responds with a **`"SUCCESS"`** message. Also, the response from the server to the client application is encrypted using the symmetric cipher already instantiated with the established shared secret key.

Figure – 23

```
            if username_exists:
                # The username exists, now we need to check if the hash of the password matches.
                # First we need to calculate the hash of the provided password.
                # Decoding the base64 encoded string.
                decoded = base64.b64decode(tokens[2].encode("utf-8"))

                # Next we compare the calculated hash with the hash stored in the database.
                if password_hash.encode("utf-8") == base64.b64encode(sha256(decoded).digest()):
                    # Login is successfull!
                    is_login_success = True
                    client_username = tokens[1]
                    our_response = "SUCCESS".encode("utf-8")
```

Figure – 24

```
                    # We map the username to the socket file descriptor as well as with the encryptor and
decryptor objects.
                    # This helps us to reference this client when other clients wants to send it messages.

                    mutex_lock.acquire()
                    users_logged_in[tokens[1]] = (cli_socket, enc, dec)
                    mutex_lock.release()
                else:
                    # The client has provided wrong password !
                    our_response = "Wrong password !".encode("utf-8")

            else:
                # The username does not exist in our database!
                our_response = "Username not found in database of server! please register first !".
encode("utf-8")
```

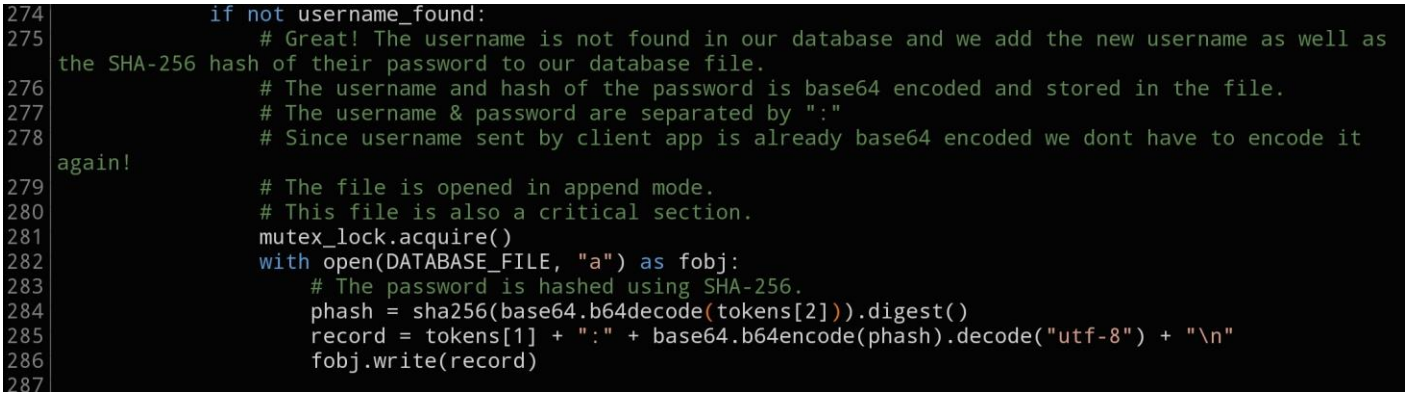
If we closely look at the code snippet in ***Figure – 24*** we can see that the client socket object, and the **encryptor** and **decryptor** context objects are bundled in a three-element tuple object are stored in a dictionary mapped using the username provided by the client application. This is necessary, because we want to use the encryption context while encrypting relayed messages sent to this username from another user using another client application. If we closely look into ***Figure – 25*** below we can clearly see the response from our server gets encrypted before sent to the client application.

Figure - 25

```
249             print(f"[~] Plaintext response to {cli_address}: {our_response}")
250             # We use our encryptor to encrypt the above response before sending to the client app.
251             our_response = enc.update(our_response)
252             print(f"[~] Encrypted response (ciphertext) to {cli_address}: {our_response}")
253             # Finally we send our encrypted response to the server.
254             cli_socket.send(our_response)
255
```

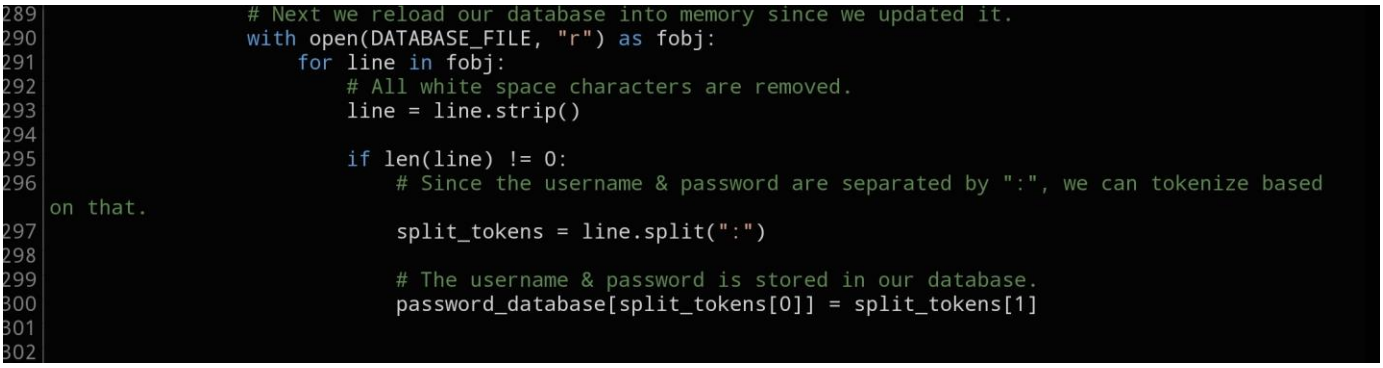
Now, if the client application wanted to perform the registration process, the server checks if the username exists in the loaded database of the server, if so the registration process is rejected with an error response (also encrypted). If the username **does not** exist in the loaded database of the server, then the server will append the new base64 encoded username and base64 encoded SHA-256 hash of the password to the database file of the server after which the server reloads the entire new database into memory (see Figure – 26 to 27)

Figure - 26



```
            if not username_found:
                # Great! The username is not found in our database and we add the new username as well as the SHA-256 hash of their password to our database file.
                # The username and hash of the password is base64 encoded and stored in the file.
                # The username & password are separated by ":"
                # Since username sent by client app is already base64 encoded we dont have to encode it again!
                # The file is opened in append mode.
                # This file is also a critical section.
                mutex_lock.acquire()
                with open(DATABASE_FILE, "a") as fobj:
                    # The password is hashed using SHA-256.
                    phash = sha256(base64.b64decode(tokens[2])).digest()
                    record = tokens[1] + ":" + base64.b64encode(phash).decode("utf-8") + "\n"
                    fobj.write(record)
```

Figure – 27



```
                # Next we reload our database into memory since we updated it.
                with open(DATABASE_FILE, "r") as fobj:
                    for line in fobj:
                        # All white space characters are removed.
                        line = line.strip()

                        if len(line) != 0:
                            # Since the username & password are separated by ":", we can tokenize based on that.
                            split_tokens = line.split(":")

                            # The username & password is stored in our database.
                            password_database[split_tokens[0]] = split_tokens[1]
```

If the registration process is successfully completed, then the server responds with a **"SUCCESS"** or a proper error response. It is to be understood that no matter what the response they are encrypted before sending to the client application.

Now, we move back to the client application and understand more about its internal workings. Before we moved on to discussing more about how the server application processes messages from the client application, we were discussing about how the **_login_or_register(..)** function works inside the client application. The following screenshot clearly shows how the messages with login or registration information are formatted before sending to the server. The message that will be sent to the server, takes the form (syntax): ***action:base64_encoded_username:base64_encoded_password*** , where action could be ***Login*** or ***Registration*** .

Figure – 28

```
270         # We prepare to send the username and password to the server.
271         # NOTE: We base64 encode the username and password.
272         message = action.encode("utf-8") + ":".encode("utf-8") + base64.b64encode(self.username.get().
    encode("utf-8")) + ":".encode("utf-8") + base64.b64encode(self.password.get().encode("utf-8"))
273
274         # The plaintext & ciphertext is printed.
275         self.insert_to_text_widget(self.text_widget, tk.END,f"Plaintext: {message}\n")
276
277
278         # We encrypt the above login / registration information with AES CTR encryption using the shared
    symmetric key
279         message_encrypted = self.enc.update(message)
280
281         # The ciphertext is printed to the output window.
282         self.insert_to_text_widget(self.text_widget, tk.END,f"Ciphertext: {message_encrypted}\n")
```

Now, once registration is successfully completed, the user would have to **Login** again. After the ***login process*** is successfully completed then the client application transforms the GUI, to make it suitable for messaging. The GUI, will have a new text widget at the top, that will show the received messages from the other user, including their public key as well as the shared secret key established with the other user. The new transformed GUI, will also have an entry widget so that the logged in user can type in messages as well as a "**SEND >"** button for sending encrypted messages to the other user via the central server.

Figure – 29

```
328         ret = self._login_or_register("Login")
329
330         if not ret:
331             # If the login process failed, then we simply return without updating the UI, because the
    user needs to login successfully for the next step.
332             return
333
334         # The login process is successful & we can update the user interface for messaging!
335         self.clear(self.frm)
336
337         # We update the UI, so that the user can send and receive messages.
338         self.update_ui_for_messaging()
339
340         # We also make the root Tkinter widget call our special function every 250 milliseconds to check
    for messages from the socket.
341         self.root.after(250, self._receive_messages)
```

The above screenshot, Figure – 29 is a code snippet from the **login()** function that gets called when the **Login** button is pressed. We can clearly see, that the function for updating the UI, is called after which we see something very important. We make the **root** of our Tkinter application schedule, using **`root.after(..),`** the running of a function called **_receive_messages()** after a 250 milli second delay in the internal main loop (that updates the GUI) of the Tkinter application. Once this function gets called by Tkinter internally the function reschedule Tkinter to make it run again after a delay of 250 milli seconds. The reason why we are doing this is because **Tkinter widgets** are not designed to work in a multi threaded environment, so for **thread safety** any code that modifies the widgets' and their properties must happen from the main thread. That's why we make Tkinter itself run the function that receives messages from the central server, which contains encrypted messages sent by the other user logged in with the central server. We will look into the function **_receive_messages()** later, but for the time being let's focus what happens when the user types a message to be sent to the other user and presses the **"SEND >"** button. The send button of the client application's **GUI,** is connected to the **send_message()** function inside the **ClientAPP** class. This function gets called when the button labelled "SEND >" is pressed by the user. Here is a code snippet showing the first few lines of code of this function:

Figure-30

```
409     # Function that deals with sending of message to the other users.
410     def send_message(self):
411         # The user types the message into the testbox in the format: @username: Message format.
412         # First we get the contents from the text field.
413         message = self.message_var.get().strip('@ ')
414 
415         # Since the user submits the message in the format
416         # @user: My Message!
417         # We need to extract the username and message from that.
418         tokens = message.split(":")
419 
420         # If we dont have the sufficient number of tokens the message is ignored and not sent.
421         if len(tokens) < 2:
422             messagebox.showinfo(title="SecureP2P client", message="Please type messages to send in the
    format\n as shown in this example: \n\n@example_other_username: My message to sent! \n\nPress the 'SEND
    >' button to send the message to the client ! ")
```

This function first extracts the contents of the **entry** widget (contained in **`message_var`**) and extracts the username of the destination user and the message to be sent to that destination user referenced by their username. It displays an information message box if the user makes a mistake in the input format, and also shows an example. Now, if the user provided the input in the expected form the function first checks if a **shared secret key** is established with the user we are going to communicate with. If not this function makes the client application generate a new public and private key and a message is sent to the central server in the following format:
`MESSAGE:base64_encoded_destination_user_name:base64_encoded_logged_in_username:ECDH:public_key_generated_for_logged_in_user_in_PEM_format` .This message in the above format is fully encrypted using the symmetric cipher instantiated with the shared secret key established with the central server and it is send to the central server. An example of this in action can be seen in the output obtained from the green text widget (**Figure - 31**) of the client application in which **alice** is logged into in the demonstrations that was illustrated at the beginning of our document.

Figure - 31

```
============ CRYPTOGRAPHIC INFORMATION ==============
===== ESTABLISHING SHARED SECRET WITH OTHER CLIENT APPS VIA ECDH =====
[*] Request to perform ECDH (as plaintext): b'MESSAGE:Ym9i:YWxpY2U=:ECDH:-----BE
GIN PUBLIC KEY-----\nMFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAE+w3W4jvd4k9iGdemp1XGxipLULd
+hzqB\nAF2I3tL3sG7iLIF0bePGYhI/RTJKMXx5gdTKyYxyawHGnT+Rx7xTnA==\n-----END PUBLIC
 KEY-----\n'
[*] Request to perform ECDH (as ciphertext): b'^\xf9\x16\x8c\xf0^\xebf+\xa3\x9a&
\x0c\xc1\xd7\x9bV\xad\xb2\x86E\x05@\x15\x90\xbce\xef\xb7\xccv\x12\xb2\x06\x91#}}
S\xb6\xc2\x7f\xdc\xbb\x14~\tq\x1c\xa1\x1e\xc5\t\xcf\x07\xdd4\x93\xc3A>;eG!  \x9a\
xa9%y\x8am5\xb0\xcf\x03\x94j\x0b\x16\xca\xf5\x96\nL\xe6\x02{\xa8)d\xe16\x1e@\xe4
\xb4]\xbe\x89\xb6RV\x9f\x88V\x10\x9b\x07\xf3\xaed"\xa4%\x8cdh/\xdc\xc5\xb0b\'\xe
```

This is the code snippet that generates the output in **Figure - 31**

Figure-32

```
437             self.insert_to_text_widget(self.text_widget, tk.END, "\n===== ESTABLISHING SHARED SECRET WITH
    OTHER CLIENT APPS VIA ECDH ===== \n")
438
439             # We have not established the key yet!
440             # We generate the public key & private key
441
442             self.our_public_key, self.our_private_key = generate_ecdh_keys()
443             # We tell the other client that we want to perform ECDH
444             response = "MESSAGE:".encode("utf-8") + base64.b64encode(tokens[0].encode("utf-8")) + b":" +
    base64.b64encode(self.username.get().encode("utf-8")) + b":" + b"ECDH:" + self.our_public_key
445
446             # We print the response as plaintext.
447             self.insert_to_text_widget(self.text_widget, tk.END, f"[*] Request to perform ECDH (as
    plaintext): {response}\n")
```

When the central server receives this message, it's able to decrypt it because it was encrypted with the symmetric cipher that uses the shared secret key established with the server, and the server tokenizes it based on the symbol ":". On reading the first token, the server understands that it's a message that needs to be relayed. The server checks if the username stored in the second token is logged in and if so, fetches it's socket descriptor and encryptor context and then encrypts the decrypted message using the encryptor context of the destination client application and sends the ciphertext to it. We look into the central server's code to understand this better.

Figure - 33

```
357                 # We relay the message to the user.
358                 # But that message is encrypted with the encryptor / decryptor objects specific to that
    destination client app.
359                 other_cli_sock, other_cli_enc, other_cli_dec = users_logged_in[tokens[1]]
360                 print("[~] Response to be relayed to: '{0}': {1}".format(base64.b64decode(tokens[1].
    encode("utf-8")).decode("utf-8"), response))
361                 our_response = other_cli_enc.update(response)
362                 print("[~] Response relayed to '{0}' as CIPHERTEXT: {1}".format(base64.
    b64decode(tokens[1].encode("utf-8")).decode("utf-8"), our_response))
363                 # The above response is sent to destination client!
364                 other_cli_sock.send(our_response)
```

Now, we are assuming that the destination user is logged in and they will receive this relayed message from the server because of the ***`_receive_messages()`*** function which we discussed earlier in brief. Now we look at this function in more detail, so we move onto the concerned section in the source code of the client application.

Figure – 34

```
536     # Function to receive messages from server socket.
537     # This function is special because it gets called by Tkinter every 250 milli seconds.
538     def _receive_messages(self):
539         print("[~] Function to receive messages from server socket called. ")
540
541         # We don't want the socket to be blocking anymore.
542         self.serv_sock.settimeout(0.1)
543
544         try:
545             # We receive the response from the central server.
546             response = self.serv_sock.recv(MAX_SIZE)
547         except:
548             print("[~] Returning because of no response. ")
549             self.root.after(250, self._receive_messages)
550             return
```

Since the ***`_receive_messages()`*** is called by **Tkinter** internally every **250 milli seconds** we don't want this function to block on the socket to receive messages as this will ***freeze*** the GUI, and it's undesirable. So to prevent that from happening we have used

settimeout(..) function of the **socket** class to make ***recv()*** block for just 0.1 seconds. This function decrypts the messages from the central server using the symmetric cipher instantiated with the shared secret key established with the central server, tokenizes the decrypted message and on realizing that the other user intending to communicate, has sent their public key, immediately generates public and private keys for the logged in user of the client application and immediately formats a message buffer in the same format we discussed in page 20, and this contains the public key it just generated. This message buffer is encrypted using the symmetric cipher instantiated with the shared secret key established with the central server and is send to the server so that it can relay it to the user who wants to communicate with the destination user. Now, the ***_receive_messages()*** function of the user that started the communication will pick up the relayed message from the central server and it can establish a shared secret key as it now has the public key of the destination user and the destination user has also established the shared secret earlier. We take a look into the snippets of code that does this to get a better understanding.

Figure - 34

```
569         if tokens[0] == "MESSAGE":
570             # This is good, because we got a message from another user through our server !
571             # We decode the username of the interested client.
572             decoded_username = base64.b64decode(tokens[2].encode("utf-8")).decode("utf-8")
573 
574             # Next we examine tokens[3].
575             if len(tokens) == 5 and tokens[3] == "ECDH":
576                 # if tokens[3] is exactly "ECDH", then that means another client has sent us their public
    key (in tokens[4]) for establishing a shared secret.
577                 # First we need to verify if we have sent them our public key.
578                 flag = (self.our_public_key != b"")
579 
580                 if not flag:
581                     # We have NOT sent the other user our public key yet!
582                     # OR the public key we had expired.
```

Figure - 35

```
605                 # We can establish the shared secret now.
606                 # We have sent them a public key & we can establish shared secret key with our private key
    we already generated.
607 
608 
609                 # The public key of the client interested in communicating is the 5th token (index = 4)
610                 client_public_key = tokens[4]
611 
612                 # We print the client's public key in our text box.
613                 self.insert_to_text_widget(self.mtext_widget, tk.END, f"{decoded_username}'s public key:
    {client_public_key}\n")
614 
615                 self.shared_secret_user = establish_shared_secret_ecdh(client_public_key.encode("utf-8"),
    self.our_private_key)
616                 # We create the AES CTR cipher!
617                 self.cipher_ssu = Cipher(algorithms.AES(self.shared_secret_user), modes.CTR(b"\x00"*16))
```

Figure - 36

```
616                 # We create the AES CTR cipher!
617                 self.cipher_ssu = Cipher(algorithms.AES(self.shared_secret_user), modes.CTR(b"\x00"*16))
618                 # We instantiate the encryptor & decryptor objects using the cipher object.
619                 self.enc_ssu = self.cipher_ssu.encryptor()
620                 self.dec_ssu = self.cipher_ssu.decryptor()
621                 self.creation_time_ssu = time.time()
622 
```

Once, the shared secret key has been established between the users intending to communicate with each other, encrypted communication can take place. If we look back to the demonstration at the beginning of our document ***Figure – 10*** to ***Figure – 10.2*** we can clearly see this in action. Let's take the demonstration as an example to understand the communication after the shared secret key has been established. Now, when **alice** presses the **"SEND >"** button the ***send_message()*** function gets called again and this time it realises that the shared secret with the other user ***(bob)***. Now, if we look into the source code of the ***send_message()*** function, we can realize that the function also performs another crucial check before sending the message **alice** wants to send to **bob.** The function checks if the established shared secret key has expired or not. If we take a closer look into ***Figure – 36*** which is a code snippet from the ***_receive_messages()*** function we can see that this function takes a note of the creation time of the shared secret key with the other user. Now this saved time stamp is used by ***send_message()*** to verify if a shared secret key that was previously established expired or not. Now, if it expired then the function resets the shared secret key established with the other user to be an empty buffer and recursively calls the ***send_message()*** function to establish the shared secret key with the other user again. This is possible because the recursive call of the ***send_message()*** function detects that the shared secret key established with **bob** has expired and it does the proceedings to establish a new shared secret key with **bob.** Please refer to screenshots in **Figure – 37** to **Figure – 38** for better understanding.

Figure - 37

```
471         # We check to see if the shared secret we established with the other user (using the client app)
    expired or not.
472         #####
473         if int(time.time() - self.creation_time_ssu) > EXPIRY_TIME:
474             # We notify the user that the shared secret we established with the other user has expired
    and we need to perform ECDH again!
475             messagebox.showwarning(title="SecureP2P client", message=f"The shared secret key we
    established with '{tokens[0]}' has expired !\nPerforming ECDH again to establish a new shared secret
    !\nPlease wait for confirmation & then try again! ")
476
477             # We print similar info into our output text widget as well.
478             self.insert_to_text_widget(self.text_widget, tk.END, "===== SHARED SECRET KEY EXPIRED, NEED
    TO PERFORM ECDH AGAIN =====\n")
479
```

Figure – 38

```
478             self.insert_to_text_widget(self.text_widget, tk.END, "===== SHARED SECRET KEY EXPIRED, NEED
    TO PERFORM ECDH AGAIN =====\n")
479
480
481             # We reset all the values.
482             self.shared_secret_user = b""
483             self.our_public_key = b""
484             self.our_private_key = None
485
486
487             #####
488
489             # We call the function recursively to achieve the desired effect.
490             self.send_message()
491             return
```

If the shared secret key between **alice** and **bob** has been established and it's not expired yet, the ***send_message()*** function will proceed to encrypt the message **alice** wants to send using the symmetric cipher instantiated with the shared secret key established between **alice** and **bob,** and then use base64 encoding to encode the cipher text and the base64 encoded form of the ciphertext will be used in formatting a message buffer that will be sent to the central server encrypted using the symmetric cipher instantiated with the shared secret established by **alice** and the **central server.**

The format of the message buffer containing the message as base64 encoded ciphertext that will be sent to the central server to be relayed to the destination user, will look like:

MESSAGE:base64_encoded_destination_user_name:base64_encoded_logged_in_username:base64_encoded_message_ciphertext

The following screenshots (Figure – 39 and Figure - 40) shows the code that does exactly this and (Figure – 41) is some sample output captured from the client application used by **alice** during the demonstration.

Figure - 39

```
500         #####
501         # We use that shared secret key `self.shared_secret_user` to encrypt the message we got from the
    entry widget.
502         self.insert_to_text_widget(self.text_widget, tk.END, f"[*] Message we want to send (plaintext
    form): {tokens[1]}\n")
503
504         message_encrypted = self.enc_ssu.update(tokens[1].encode("utf-8"))
505         self.insert_to_text_widget(self.text_widget, tk.END, f"[*] Message we want to send (in ciphertext
    form): {message_encrypted}\n")
506
507         # The above encrypted message is base64 encoded.
508         message_encrypted = base64.b64encode(message_encrypted)
509
510         self.insert_to_text_widget(self.text_widget, tk.END, f"[*] The ciphertext in base64 encoded
    format: {message_encrypted}\n")
```

Figure - 40

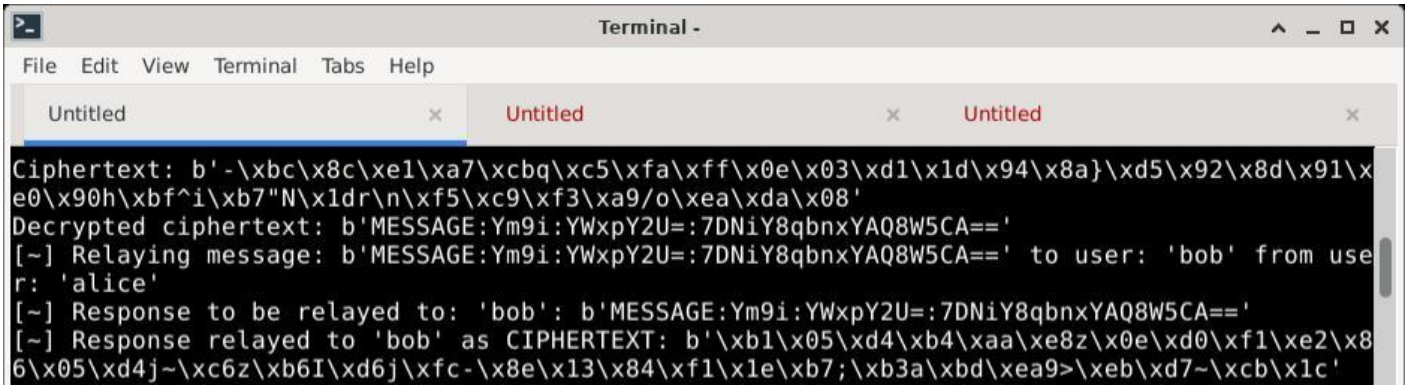
```
513         # Now we format the response to the server!
514         response = "MESSAGE:".encode("utf-8") + base64.b64encode(tokens[0].encode("utf-8")) + b":" +
    base64.b64encode(self.username.get().encode("utf-8")) +  b":" + message_encrypted
515
516         # The above formatted response is also printed to the output window.
517         self.insert_to_text_widget(self.text_widget, tk.END, f"[*] Message to server as plaintext:
    {response}\n")
518
519
520         # Since we have already established shared secret with the server the above response needs to be
    encrypted with the encryption cipher using that shared secret key
521         response = self.enc.update(response)
522
523         # The resulting ciphertext is printed to the output window.
524         self.insert_to_text_widget(self.text_widget, tk.END, f"[*] Message to server as ciphertext:
    {response}\n")
```

Figure - 41

```
============ CRYPTOGRAPHIC INFORMATION ==============
[*] Message we want to send (plaintext form):  How are you?
[*] Message we want to send (in ciphertext form): b'\xec3bc\xca\x9b\x9f\x16\x00C
\xc5\xb9\x08'
[*] The ciphertext in base64 encoded format: b'7DNiY8qbnxYAQ8W5CA=='
[*] Message to server as plaintext: b'MESSAGE:Ym9i:YWxpY2U=:7DNiY8qbnxYAQ8W5CA==
'
[*] Message to server as ciphertext: b'-\xbc\x8c\xe1\xa7\xcbq\xc5\xfa\xff\x0e\x0
3\xd1\x1d\x94\x8a}\xd5\x92\x8d\x91\xe0\x90h\xbf^i\xb7"N\x1dr\n\xf5\xc9\xf3\xa9/o
\xea\xda\x08'
[~] Response we got from the server: b"i\x81P\xca\xea\xd3\xd7Z\xbev\x01n\xcd\x10
```

This is the central server relaying the above message to the other user (***bob***) :

Figure – 42

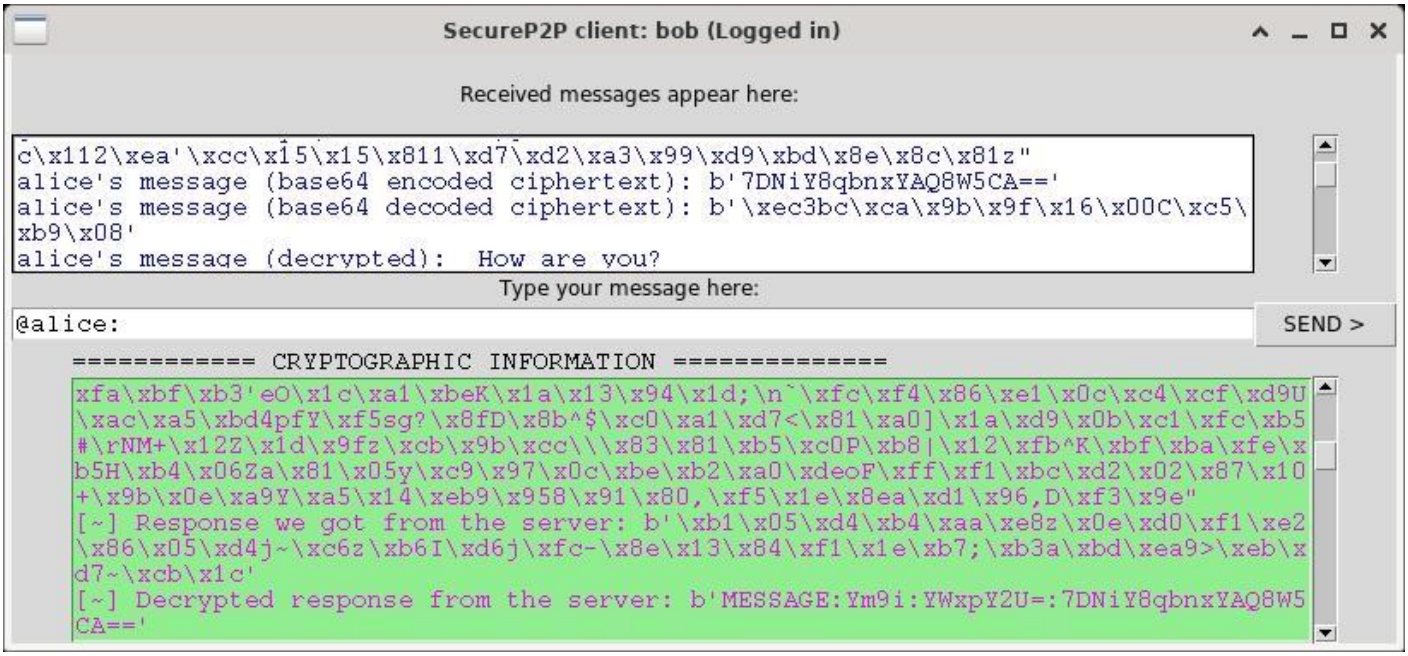


Now, ***_receive_messages()*** function of the client application used by ***bob,*** will receive the relayed message from the central server, which it decrypts using the symmetric cipher that uses the shared secret key that ***bob***, has established with the central server and finally tokenizes the decrypted message, based on the ":" symbol. The decrypted ciphertext has only **four** tokens and hence is not the **public key** from another user, and is a message that contains ciphertext to be decrypted. Because of our communication protocol, the fourth token is the base64 encoded ciphertext that can only be decrypted using the symmetric cipher that was instantiated with the shared secret key established between ***alice*** and ***bob*** . The following screenshot (Figure – 43) shows the code that does exactly what was just explained.

Figure - 43

```
629             else:
630                 # tokens[3] is the base64 encoded encrypted message that is encrypted using the shared
    secret key established with the client app.
631                 ciphertext = tokens[3].encode("utf-8")
632
633
634                 self.insert_to_text_widget(self.mtext_widget, tk.END, f"{decoded_username}'s message
    (base64 encoded ciphertext): {ciphertext}\n")
635                 ciphertext = base64.b64decode(ciphertext)
636                 self.insert_to_text_widget(self.mtext_widget, tk.END, f"{decoded_username}'s message
    (base64 decoded ciphertext): {ciphertext}\n")
637                 # Finally we decrypt the message.
638                 decrypted = self.dec_ssu.update(ciphertext)
639                 self.insert_to_text_widget(self.mtext_widget, tk.END, ("%s's message (decrypted): %s\n") %
    (decoded_username, decrypted.decode("utf-8")))
```

Figure – 43



The above screenshot (Figure – 43) of the client application used by ***bob*** shows the application receiving and decrypting the relayed message from the central server as shown in the text widget with green background, and the screenshot also shows the application being able to parse out the base64 encoded ciphertext, decoding it and finally decrypting it and showing it's output in the upper text widget with white background and blue text.

How the entire project confirms with the requirements ?

- With a key at least 56 bits, what cipher you should use?

  The 56 bit cipher that comes to my mind is the **DES**, cipher which unfortunately does not provide any security and is considered deprecated in our present day and age. In our original implementation we are using **AES** in counter mode, using **256** bit keys generated via **Elliptic Curve Diffie-Hellman (ECDH).**

- DO NOT directly use the password as the key, how can you generate the same key between **Alice** and **Bob** to encrypt messages?

  The password of either Alice or Bob is only used for granting access of communication to the other logged in user, by the **central server.** The client application for each user generates a public and private key pair, for communicating with the other user. The public key of one user is relayed to the destination user and the public key of the destination user is relayed to the initial user via the central server and a shared secret key, which is unknown to even the server is generated via **ECDH**, and the shared secret key is used for encrypted communication between the users using a symmetric cipher.

- What will be used for padding?

  We are using **stream** ciphers throughout. AES in CTR mode is a stream cipher. Counter mode converts a block cipher like **AES** into a stream cipher and hence we don't have to worry about padding.

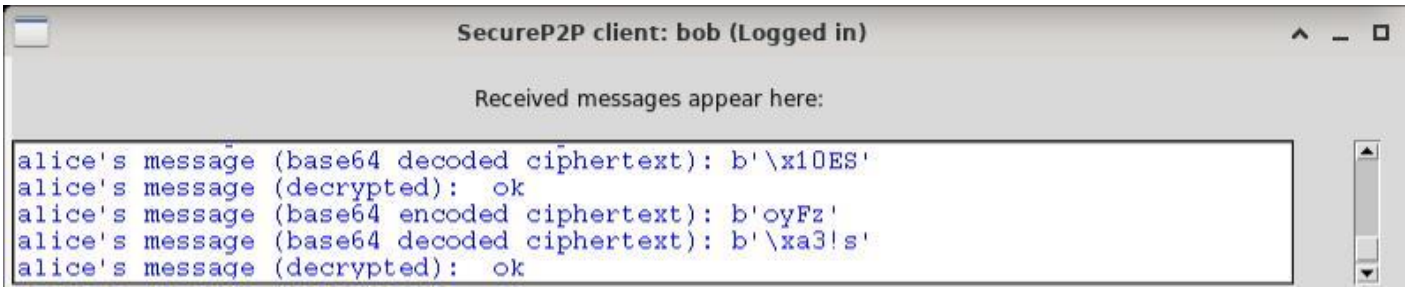
- We are using **Tkinter** for the GUI, and we have provided scrollable text widgets on the client application that shows the ciphertext, plaintext etc. The central server does not have a GUI, and it shows all it's output including plaintext & ciphertext in its terminal window.

- If Alice or Bob sends the same message multiple times (e.g., they may say "ok" many times), it is desirable to generate a different ciphertext each time. How to implement this?

  Yes, its really important that two ciphertexts need to be different for message of same contents. For this reason, our implementation uses stream ciphers (AES in counter mode) that work by generating a pseudo random key stream that gets XOR-ed with the plaintext to produce ciphertext. So if **Alice,** sends "ok" first, only two bytes of the key stream will be used for encrypting that message. Next, if **Alice,** sends "ok" again the ***next two bytes*** of the pseudo random key stream (generated internally, and different from the first two) will be used for encrypting the second "ok" message and hence the ciphertexts produced in both cases will be different. The actual process of encryption using the stream cipher is taken care of by the ***`cryptography`*** module.

  This can be illustrated very clearly, as shown in the screenshot of the client application used by **bob** as he sees different ciphertext (base64 decoded ciphertext) for two "ok" messages sent by **alice:**

  Figure - 44



- Design a key management mechanism, to periodically update the key used between ***Alice*** and ***Bob.*** Justify, why the design can enhance security?

  As already demonstrated in the screenshots from ***Figure – 10.6*** to ***Figure – 10.9*** we can clearly see that the client application detects the expiry of the shared secret established between ***alice*** and ***bob*** and re-establishes a new shared secret key via another **Elliptic Curve Diffie-Hellman** process. We have also discussed deeply of how this is implemented in the ***"Implementation details"*** section of this document. After ***`EXPIRY_TIME`*** seconds if the user of a client application attempts to send a message to another user with which it has already established a shared secret key, then the client application of the user that attempted to send a message will detect the expiry of the key, and trigger the other user to establish a new shared secret key via ECDH, by sending its public key, meanwhile the client application that attempted to send the message (after key expiry) also prepares to receive the public key send by the destination user to establish a new shared secret key with the other user. The variable ***`EXPIRY_TIME`*** is declared in the source code of the client application to be equal to ***120*** seconds (see Figure – 45). The justification for using this system is that it prevents attacks via cryptanalysis of ciphertext. Changing the keys will provide double assurance to the fact, that ciphertexts generated will be indistinguishable from a random stream of bytes.

  Figure – 45

```
48 # The expiry time in seconds after which we considered a shared secret key to be expired.
49 EXPIRY_TIME = 120
50
```

- If you can hide the detailed procedure of your encryption algorithm, how would you improve the security by designing a new algorithm?

  If we hide the details of our encryption algorithm, then the attacker will have no idea on how to approach decrypting the ciphertext they intercept. They will have a very difficult time doing cryptanalysis because most cryptanalytic techniques only work with certain ciphers only, or by exploiting some particular hyper patameters used in the process. In our case we are using very strong Elliptic Curve ***public key*** cryptography for establishing the shared secret key with the server as well as the user's ***alice*** and ***bob*** and we are using ***AES*** cipher in counter mode with ***256*** bit keys for the symmetric encryption. All these systems are tried and tested and in our present day and age offer the best security possible. In general most proprietary security software including secure chat applications keep their source code closed, to hide the exact details of how the security is maintained under the hood.

- For example, you may do two encryptions using different standard ciphers, then XOR the two outputs together. Please give your new design and justify its security and efficiency?

  This design is unfortunately not possible to implement, because if we **XOR** the outputs generated by two symmetric ciphers on the plaintext the ciphertext so produced is such that the plaintext becomes unrecoverable. The only possible way to achieve ***double encryption*** is by using two stream ciphers of different nature and first using one stream cipher to encrypt the plaintext and then the ciphertext from the application of the first stream cipher must be encrypted again via application of the second stream cipher that is completely different in nature from the first stream cipher. But however, the suggestion is to **XOR** the output ciphertexts generated by applying two symmetric ciphers on the same plaintext, which unfortunately make the plaintext ***unrecoverable*** even for the intended recipient who has knowledge of the key!

Conclusion

Finally, we have reached the conclusion of our report, and we have successfully developed our end-to-end encrypted chat application that provides authenticated and fully encrypted communication between two users registered on the central server's database. Since the usernames are what that distinguishes the individual users from each other, to communicate with another user, a person only needs to know the other user's username, and there is no need to know their IP or port, as these are internally managed by the ***central server***. The client application can generate public and private keys for establishing end to end encrypted communication between two users like **alice** and **bob**.

Bibliography

- Python ***cryptography*** module documentation for primitives: https://cryptography.io/en/latest/hazmat/primitives/
- Python's official documentation for ***Tkinter*** : https://docs.python.org/3/library/tk.html
- Python's official documentation for the ***socket*** module: https://docs.python.org/3/library/socket.html
- Wikipedia article on key exchange using Elliptic Curve Diffie-Hellman (***ECDH***): https://en.m.wikipedia.org/wiki/Elliptic-curve_Diffie%E2%80%93Hellman
- **AES (FIPS – 197):** http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf
- Wikipedia article on "Block cipher modes of operation" (for understanding about how counter mode works) : https://en.m.wikipedia.org/wiki/Block_cipher_mode_of_operation