**Introduction:**

In this project We will Develop a real-time secure chat application from scratch like WhatsApp & Signal with end-to-end encryption. Our project will cover:

* Socket Programming
* Real-Time System
* End-to-end Encryption

Our secure chat will provide the following:

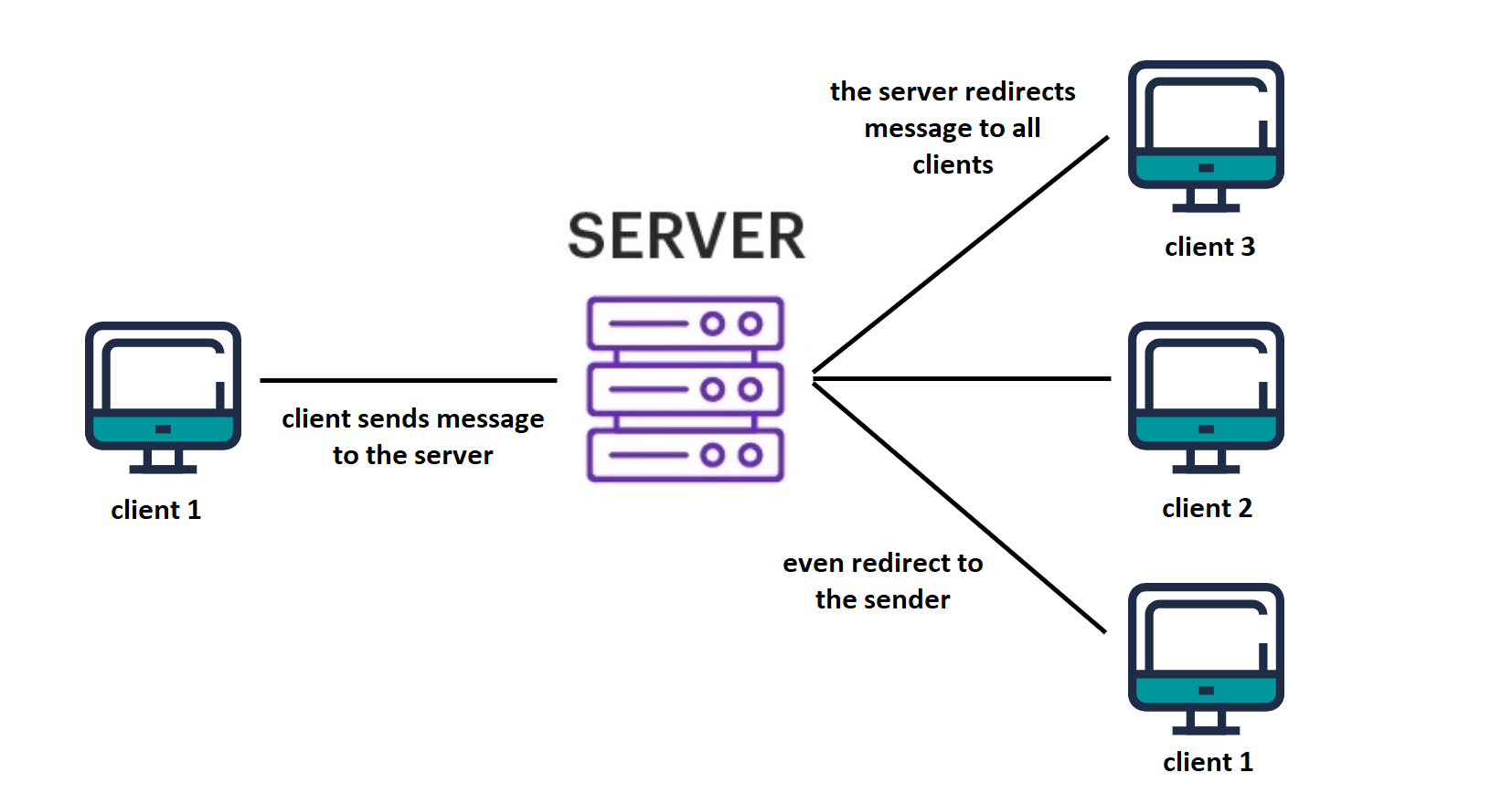
* Multi-User group chat application
* Users can choose between encryption modes:
* Single key (DES)
* Two keys (RSA, EL GAMAL)
* TTP (Third Trusted Party) which facilitates interactions between two parties who both trust the third party

Technology Stack:

* Python as the server-side language
* Tkinter library for UI
* Socket library for real-time message exchange

**System Design:**

It was challenging to make the app real-time because the user needs to be kept updated with any activity happening as soon as possible.



We can describe what is happening in the system in simple steps:

- server will start at the HOST IPv4 at the chosen PORT

- client tries to connect to the server

- server accept the client connection

- server generates session key for the accepted client

- client sends username to the server

- client and server start listener thread

- user sends a message (after encryption using one of the encryption modes)

- server sends the cipher to all clients even the sender

- client receives the cipher (then start decryption)

- server still receives message from another client

- server keep listening for any new client connections

- server responsible to send and receive messages contain information like

(username, message, required keys)

**DES – (Data Encryption Standard):**

DES is a block cipher and encrypts data in blocks of size of 64 bits each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption.

In case of messages longer than 64 bits we divide the massage into separated blocks. Let’s discuss the algorithm steps:

1. The 64-bit plain text go into an initial Permutation (IP) function.
2. The first permutation is done on the plain text.
3. Following that, the initial permutation (IP) creates two halves of the permuted block, Left Plain Text and Right Plain Text.
4. The encryption process now goes through 16 rounds for each Right and Left.
5. Finally, Left and Right are rejoined, and the combined block is goes into a Final Permutation (FP).
6. This process produces 64-bit ciphertext as a result, then repeat the steps again for the rest of message blocks.

**DES Algorithm Run:**

Graphical user interface

Description automatically generated

**RSA – (Rivest-Shamir-Adleman):**

RSA algorithm is asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e., Public Key and Private Key. As the name describes that the Public Key is given to everyone and Private key is kept private.

Let’s discuss the algorithm steps:

1. **Key generation:**
   1. Choose two prime numbers p and q
   2. Calculate n=p\*q
   3. Calculate
   4. Select an integer where 1<e<and GCD (e, )=1 and they co primes (public key with n)
   5. d= where d is modular multiplicative inverse of e mod (private key with n)
2. **Encryption:**

Turn the message into ciphertext by using modular exponentiation

C=

1. **Decryption:**

Turn the ciphertext back into plaintext by using private key exponent

m=

**El Gamal Cryptography:**

ElGamal encryption system is a public-key cryptography asymmetric key encryption scheme based on the Diffie–Hellman key exchange. Taher Elgamal first described it in 1985. The free GNU Privacy Guard software, latest versions of PGP, and other cryptosystems employ ElGamal encryption.

Let’s discuss the algorithm steps:

1. **Key generation:**
   1. Choose one prime number q
   2. Finding primitive root of q🡪 a
   3. Generating randomly private key🡪XA must be less than q-1
   4. Generating key YA 🡪 YA=aXA mod q
   5. Public key= [q, a, YA]
2. **Encryption:**
   * Turn the message into two different ciphertext (C1,C2) by using M as a pair of integers where
     + C1 = ak mod q ; C2 = KM mod q
3. **Decryption:**

**Changing the cipher key(C1, C2) to plain text by finding the recovering key where  
recovering key K as K = C1xA mod q**

**Then using the key to get the plain text M**

**computing M as M = C2 K-1 mod q**

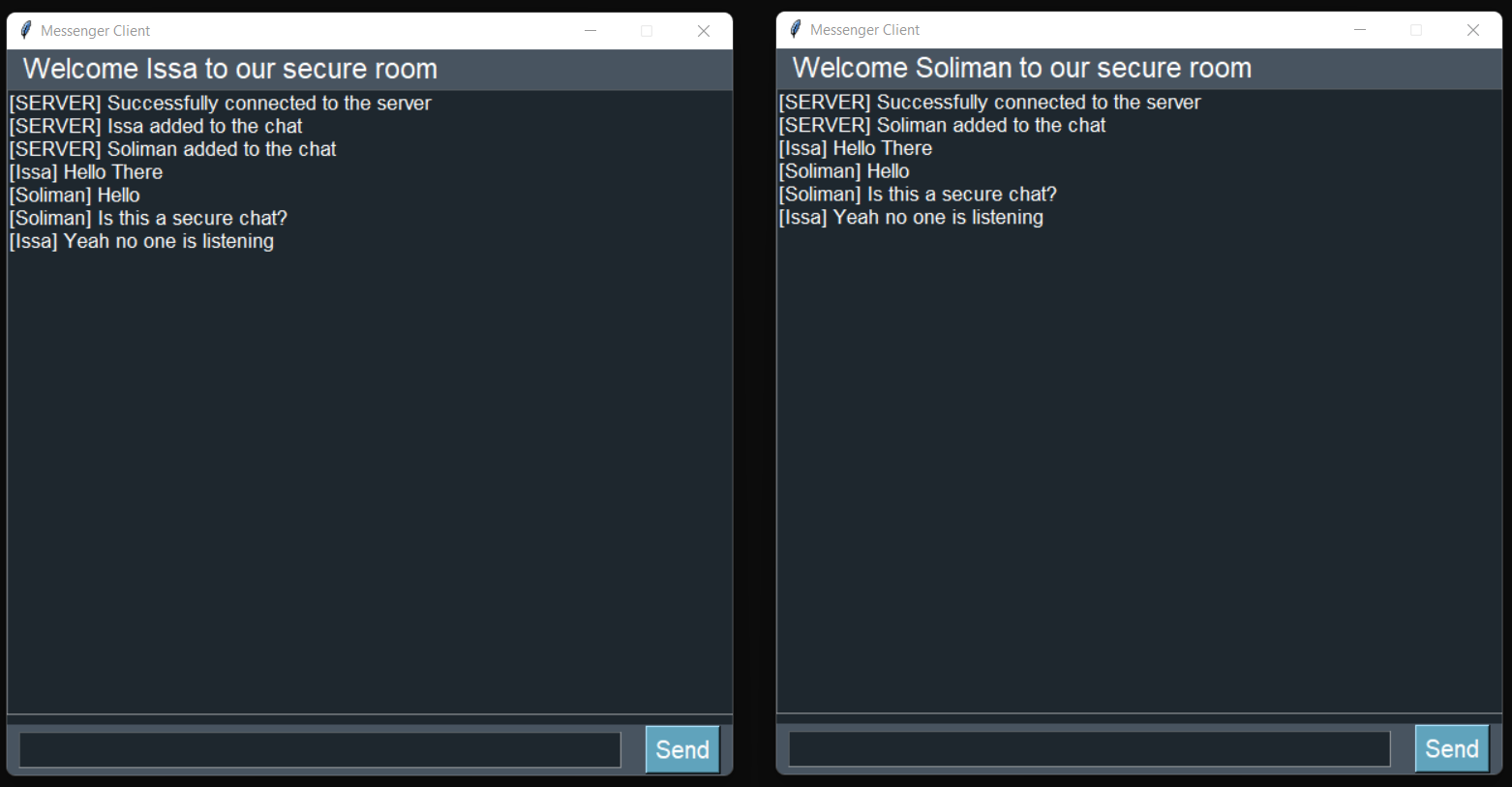
**ElGamal Cryptography run:**

A screenshot of a computer

Description automatically generated

**The Chat Application Demo:**

**GUI ( 2 Clients Chatting )**

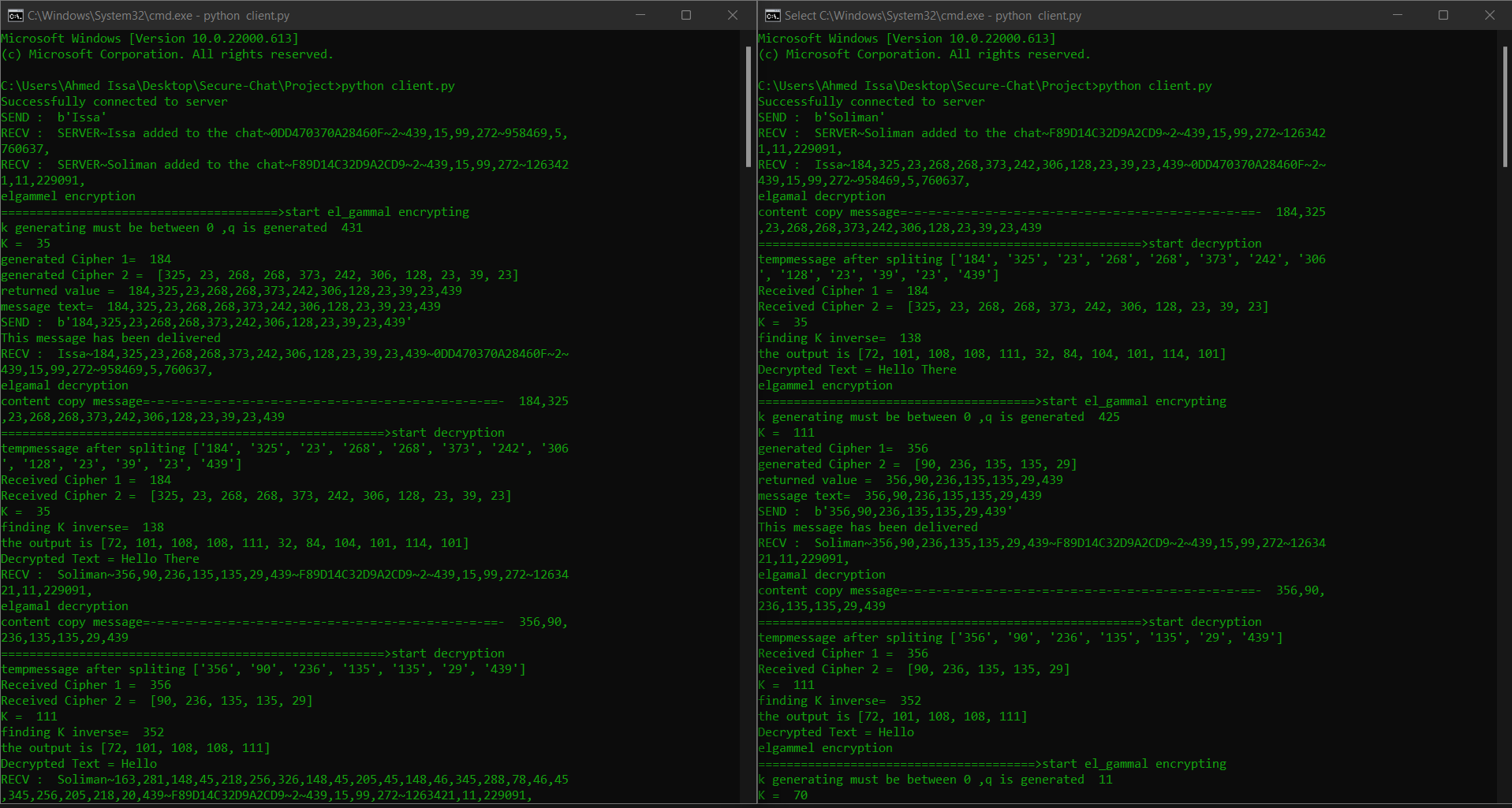


**Server Prompt ( SEND , RECV Messages)**

Text

Description automatically generated

**Clients Prompt ( SEND , RECV, ENCRYPT, DECRYPT )**

****

* The user wants to add multiple post-quantum cryptography (PQC) algorithms to the server.py so that the server can choose among them.
* The user wants these algorithms implemented in Python and saved in different files, similar to how RSA and ElGamal are implemented in RSA.py and el\_gamal.py.
* I have reviewed server.py, which currently supports DES, ElGamal, and RSA.
* I have reviewed RSA.py and el\_gamal.py, which implement RSA and ElGamal respectively.
* The user wants to add many PQC algorithms such as CRYSTALS-Kyber, CRYSTALS-Dilithium, NTRUEncrypt, FrodoKEM, NewHope, SABER, LAC, BLISS, GPV, Ring-LWE schemes, Classic McEliece, Niederreiter, BIKE, HQC, Rainbow, UOV, QUARTZ, GeMSS, SPHINCS+, XMSS, LMS, CSIDH, Picnic, Falcon, and others.
* Implementing all these algorithms from scratch is a huge task and not feasible here.
* However, there is a Python library called liboqs-python that provides bindings to the Open Quantum Safe (OQS) project, which implements many PQC algorithms including Kyber, Dilithium, Falcon, SPHINCS+, etc.
* I have installed liboqs-python in the environment.
* The best approach is to create separate Python files for each PQC algorithm or group of algorithms, using liboqs-python where possible.
* Then update server.py to include these new algorithms in the selection menu and handle their usage.
* I will first create a plan for adding these PQC algorithms modularly, integrating liboqs-python for supported algorithms, and fallback or placeholders for others.
* I will also plan to update server.py to allow choosing these new algorithms.
* I will ask the user to confirm the plan before proceeding.

Plan:

* Create a new directory pqc\_algorithms/ to hold the new PQC algorithm implementations.
* For lattice-based KEMs and signatures (Kyber, Dilithium, Falcon, SABER, NewHope, FrodoKEM, etc.), create Python wrappers using liboqs-python.
* For code-based, multivariate, hash-based, isogeny-based, and other algorithms, create placeholder Python files or minimal implementations or wrappers if available.
* Update server.py:
  + Add the new algorithms to the encryption/signature selection menu.
  + Add logic to import and use the new PQC algorithms from pqc\_algorithms/.
* Ensure the new files follow the pattern of RSA.py and el\_gamal.py for consistency.
* Test the integration by running the server and selecting a PQC algorithm.