Group: Kazi Priom(ID: 113599594)

# P2P Messaging with Python

The goal of this project is to develop a secure P2P messaging application in Python, providing strong, confidential communication between two clients, Alice and Bob. In this application, these clients are able to securely exchange messages over the Internet with data confidentiality, that is, intercepted or altered during transmission is prevented. Using DES to perform both message encryption and decryption, the system ensures that the use of a 56-bit key is employed, which is a native feature of DES. The main reason behind choosing DES is that it was tested to work with the mentioned size of the key and proved to be able to provide a proper cryptographic solution within the project's requirements.

## Design

The system contains many enhanced features to promote functionality as well as security. Password-based key derivation plays an important role in the system, providing a secure method of generating encryption keys from a common passphrase with a predefined salt. This avoids the risk of using the password as the encryption key directly, which might weaken the security of the application. Using the PBKDF2HMAC function from the cryptography library, key derivation ensures that Alice and Bob get the same encryption key, given the same password and salt (Figure 1 and Figure 2). It does not only make the job of key exchange easier but also diminishes the risks associated with static or directly exchanged keys.

The system is also padded to ensure compatibility with DES and the integrity of data. To encrypt, plaintext messages need to be a multiple of the block size of DES, 8 bytes; this is achieved by using the standard PKCS#7 padding provided by the pycryptodome library. This PKCS#7 padding scheme works by appending bytes to the plaintext; the value of each appended byte represents the total number of bytes of padding added. For example, if the message requires six bytes of padding, then all six bytes will have the value 06. This way, the ciphertext will be correctly decrypted and the original plaintext re-established without loss or corruption of data (Figure 3).

The program utilizes Python's built-in socket library to establish and maintain connections between the clients. A server socket is created by calling the socket.socket(socket.AF_INET, socket.SOCK_STREAM), which initializes a TCP socket using the IPv4 address family, AF_INET, and a stream-oriented protocol, SOCK_STREAM. To prevent problems with port binding when the server restarts, the function setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) is used. This function allows the port to be used again right after the server has been closed down, which is quite handy during development. The server_socket.bind() function of the server binds to the localhost address, 127.0.0.1, and port 12345. After binding, the server listens for incoming connections via server_socket.listen(2), allowing two clients to join (Figure 7). Clients will join by instantiating the Client class via its __init__() function and calling the start() function (Figure 11).

Upon connecting two clients to the server, they can start communicating securely through the server. It serves as an intermediary, relaying encrypted messages between clients (Figure 12 - 19). However, if the clients have an incorrect key, this will lead to a decryption error which will disconnect the server (Figure 20-21). The architecture ensures a robust communication infrastructure that allows for the exchange of messages in an encrypted manner while maintaining simplicity at the client-side architecture. Threading implemented at the client side ensures encryption and decryption processes occur simultaneously to achieve delay-free and interruption-free real-time communication (Figure 11). This is achieved by the use of two classes, EncryptMessage and DecryptMessage, which can create threads for encryption and decryption, respectively (Figure 9-10). The main client class manages these threads so that messages can be passed and the users can interact with the application without interruption (Figure 11).

The method message_encrypt plays a central role in the application; it converts plaintext messages into ciphertext. It uses the generated encryption key and a randomly generated IV to initialize the DES cipher in CBC mode. The function performs PKCS#7 padding on the plaintext before encryption, just to make sure the data is aligned with the block size requirements of DES1 (Figure 3-4). After receiving a message, the decryption is done using the message_decrypt function. This function splits the received IV and ciphertext, resets the DES cipher with the derived key of the recipient and the transmitted IV, and decrypts the message. Finally, it removes the padding to get the original plaintext (Figure 5-6). Both encryption and decryption rely on the pycryptodome library, which allows the program to execute secure cryptographic operations.

The use of random initialization vectors (IVs) for each message is a crucial security feature that significantly enhances the cryptographic strength of the application. A random IV ensures that even when the same plaintext is sent multiple times, the resulting ciphertext will always be different. This attribute, called semantic security, ensures that an attacker cannot infer anything about message contents simply by observing repeated patterns in ciphertext. Without a random IV, the same key and encryption mode would produce identical ciphertexts when encrypting identical plaintexts, which renders the communication vulnerable to pattern analysis attacks.

One particular attack against which a random IV provides good protection is the known-plaintext attack. In this context, an attacker who has knowledge of at least one plaintext message and its corresponding ciphertext can analyze the encryption process to deduce either the cryptographic key or identifiable patterns in the encryption scheme. For instance, when the user keeps sending "hello" in plaintext without an IV, it would generate identical ciphertext blocks every time "hello" is encrypted, making it even easier for an attacker to recognize the encryption key or detect a message pattern. By introducing randomness through a unique IV in every encryption, the attacker cannot associate the same plaintexts with consistent ciphertexts, even if they intercept several messages.

In addition, random IVs avoid the replay attack, where an attacker intercepts a valid ciphertext and replays it to impersonate a legitimate sender or replay an old message. Since every message contains an embedded unique IV, the replayed ciphertexts yield no meaningful plaintext on the recipient's end due to the specific IV required during the decryption process. This makes sure that every ciphertext is connected to a unique encryption context; hence, any replayed messages are invalidated.

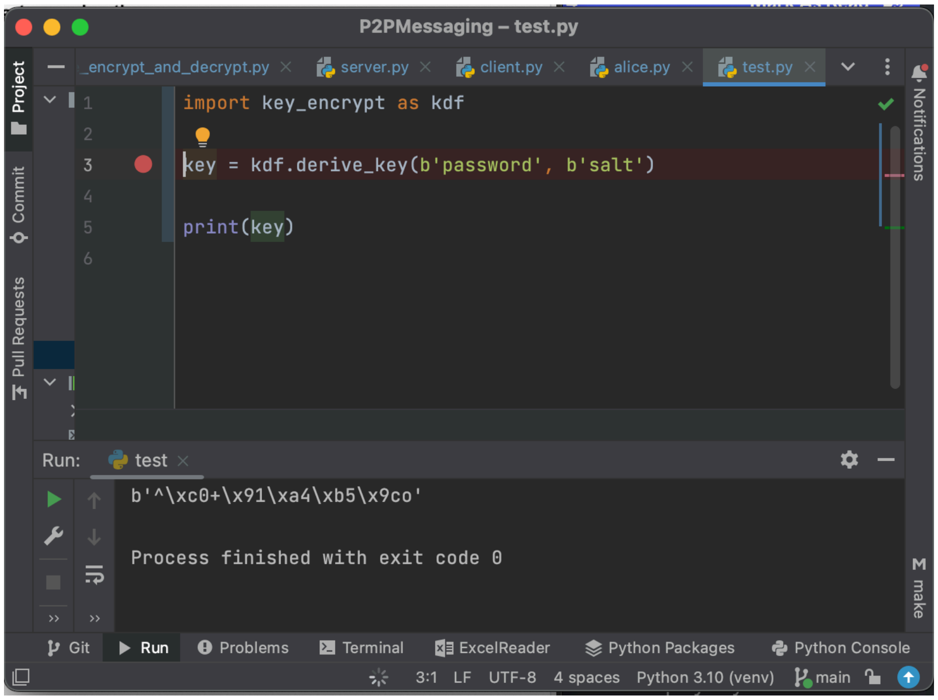
Dynamic generation of IVs used in this application enhances its use. The IV was generated using the Python's secrets.token_bytes function, highly qualitative randomness, and thus truly unpredictable to attackers. This has been done by concatenating the IV with the ciphertext and passing both into the recipient, using the IV to initialize his/her cipher object for decryption. This ensures that decryption to the receiving entity remains effortless while ensuring optimal security of data. Again, the IV need not be secret; its purpose is for introducing variance in encryption, not to provide any kind of confidentiality. The design ensures simplicity in implementation with strong cryptographic practice.

This project is an indication that it is very feasible to develop a secure P2P messaging application using Python. By overcoming challenges like secure key derivation, message encryption, and real-time communication, the design successfully implements a robust and efficient messaging platform. Figures included in the report showcase the outputs of key functions, including encryption, decryption, and message transmission, providing a clear understanding of the system's functionality.

Figures:

```
def derive_key(password, salt):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=8,
        salt=salt,
        iterations=10000,
    )

    key = kdf.derive(password)

    return key
```

Figure 1: derive_key function used to derive a key using a password and salt. PBKDF2HMAC function is taken from cryptography library



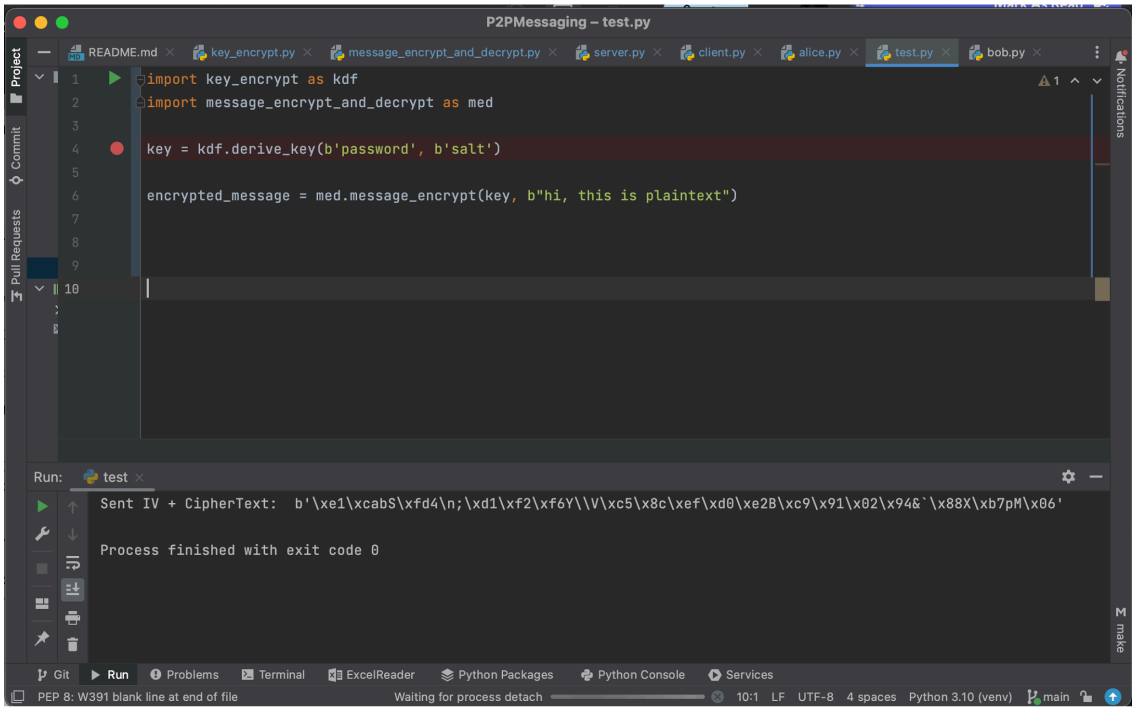
```
import key_encrypt as kdf

key = kdf.derive_key(b'password', b'salt')

print(key)
```

```
b'^\xc0+\x91\xa4\xb5\x9co'

Process finished with exit code 0
```

Figure 2: Execution of derive_key

```python
def message_encrypt(key, plaintext):

    # print("Sent Message:")
    block_size = 8 # 56-bits for DES

    iv = secrets.token_bytes(block_size)

    # print(f"Generated IV : {iv}")

    cipher = DES.new(key, DES.MODE_CBC, iv=iv)


    #using a pkcs7 padding style
    padded_plaintext = pad(plaintext, DES.block_size)


    ciphertext = cipher.encrypt(padded_plaintext)
    print(f"Sent IV + CipherText:  {iv + ciphertext}")

    return iv + ciphertext
```

Figure 3: message_encrypt function encrypts a plaintext using a key. The function generates an IV and creates a DES using the key and IV. Then the plaintext is padded using PKCS7 style and inputted into the cipher. The ciphertext is output and this and the IV is sent to the server. The DES.new function is from pycryptodome's library.



```python
import key_encrypt as kdf
import message_encrypt_and_decrypt as med

key = kdf.derive_key(b'password', b'salt')

encrypted_message = med.message_encrypt(key, b"hi, this is plaintext")
```

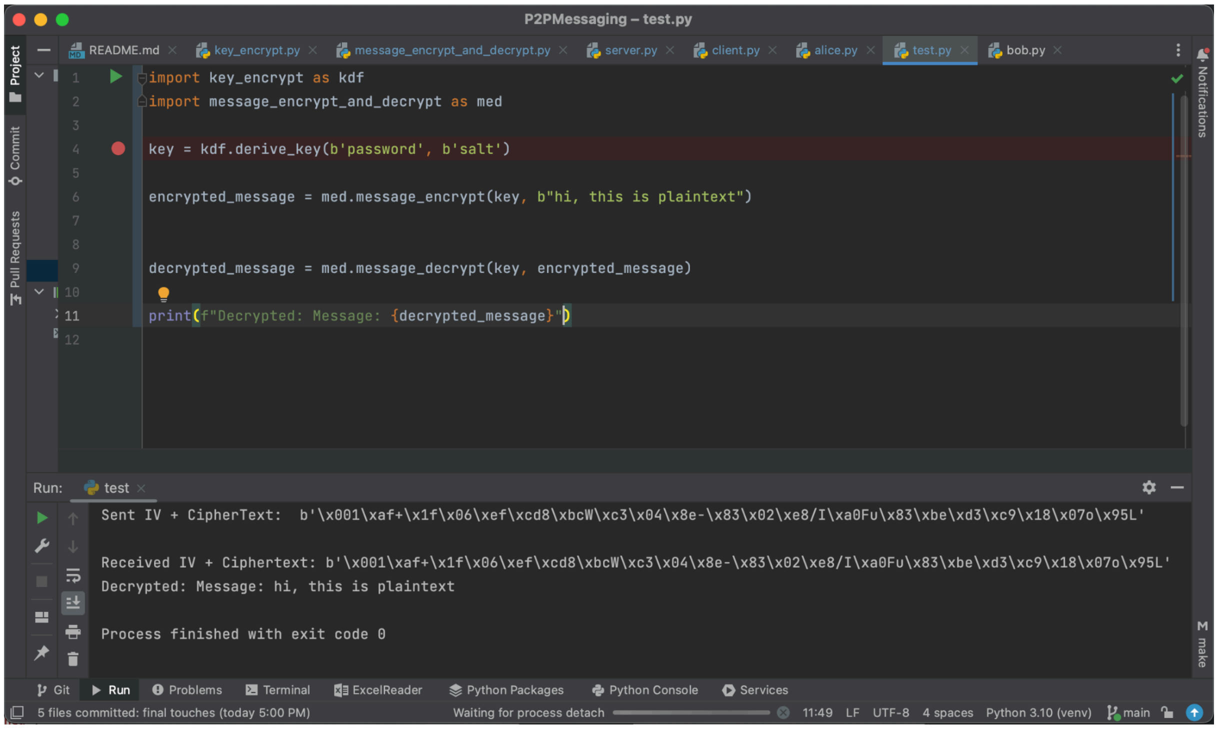
```
Sent IV + CipherText:  b'\xe1\xcabS\xfd4\n;\xd1\xf2\xf6Y\\V\xc5\x8c\xef\xd0\xe2B\xc9\x91\x02\x94&`\x88X\xb7pM\x06'

Process finished with exit code 0
```

PEP 8: W391 blank line at end of file

Figure 4: Execution of message_encrypt

```python
def message_decrypt(key, encrypted_message):
    iv_and_ciphertext = encrypted_message
    iv = iv_and_ciphertext[:8]
    ciphertext = iv_and_ciphertext[8:]

    if not ciphertext:
        raise ValueError("Ciphertext is missing")
    print(f"\nReceived IV + Ciphertext: {iv_and_ciphertext}")

    cipher = DES.new(key, DES.MODE_CBC, iv)

    try:
        padded_plaintext = cipher.decrypt(ciphertext)
        plaintext = unpad(padded_plaintext, DES.block_size)
        return plaintext.decode()
    except ValueError as e:
        print(f"Decryption error: {e}")
        raise
```

Figure 5: message_decrypt function decrypts a ciphertext using a key and the IV provided. The plaintext is also unpadded.



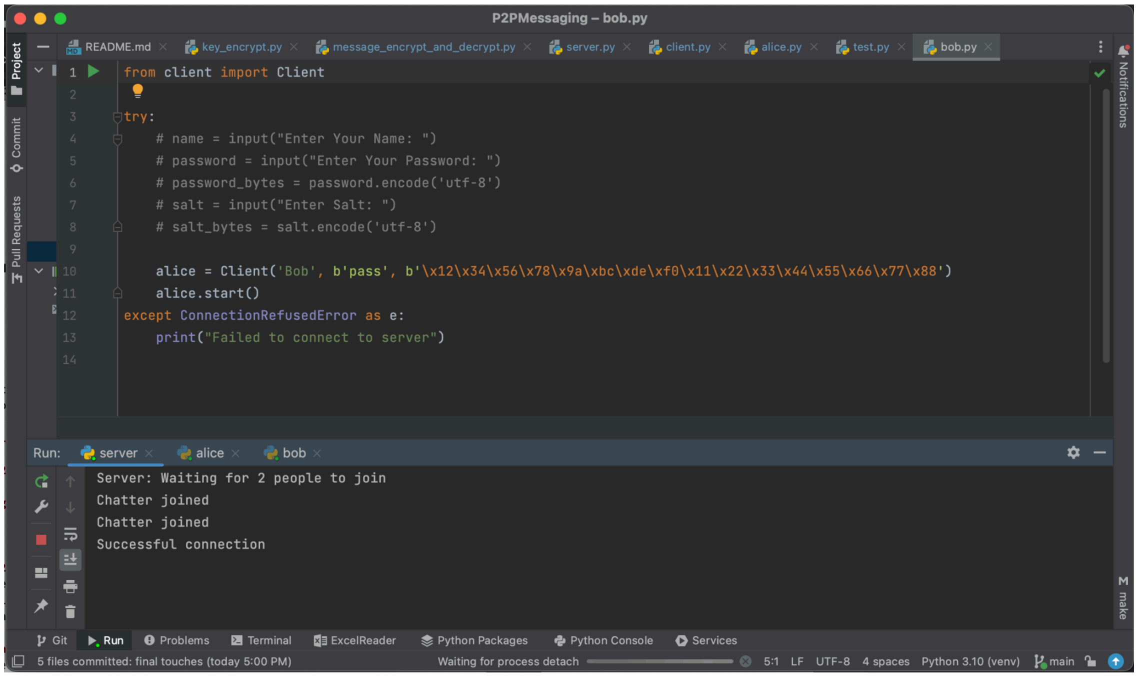
```python
import key_encrypt as kdf
import message_encrypt_and_decrypt as med

key = kdf.derive_key(b'password', b'salt')

encrypted_message = med.message_encrypt(key, b"hi, this is plaintext")


decrypted_message = med.message_decrypt(key, encrypted_message)

print(f"Decrypted: Message: {decrypted_message}")
```

```
Sent IV + CipherText:  b'\x001\xaf+\x1f\x06\xef\xcd8\xbcW\xc3\x04\x8e-\x83\x02\xe8/I\xa0Fu\x83\xbe\xd3\xc9\x18\x07o\x95L'

Received IV + Ciphertext: b'\x001\xaf+\x1f\x06\xef\xcd8\xbcW\xc3\x04\x8e-\x83\x02\xe8/I\xa0Fu\x83\xbe\xd3\xc9\x18\x07o\x95L'
Decrypted: Message: hi, this is plaintext

Process finished with exit code 0
```

Figure 6 : Execution of message_decrypt

```python
try:
   server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
   server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)  #
for port reuse
   server_socket.bind(('127.0.0.1', 12345))

   server_socket.listen(2)

   print("Server: Waiting for 2 people to join")

   clients = []

   while len(clients) < 2:
       conn, addr = server_socket.accept()
       print("Chatter joined")
       clients.append(conn)

   print('Successful connection')
   threading.Thread(target=handle_clients, args=(clients[0],
clients[1])).start()

except OSError as e:
   print(f"OS error: {e}")
```

Figure 7 : Creation of socket server. This class leverage's the library 'socket' found in Python's standard library



```python
from client import Client

try:
    # name = input("Enter Your Name: ")
    # password = input("Enter Your Password: ")
    # password_bytes = password.encode('utf-8')
    # salt = input("Enter Salt: ")
    # salt_bytes = salt.encode('utf-8')

    alice = Client('Bob', b'pass', b'\x12\x34\x56\x78\x9a\xbc\xde\xf0\x11\x22\x33\x44\x55\x66\x77\x88')
    alice.start()
except ConnectionRefusedError as e:
    print("Failed to connect to server")
```

```
Server: Waiting for 2 people to join
Chatter joined
Chatter joined
Successful connection
```

Figure 8: Server creation and successful connection

```python
class EncryptMessage:
    def __init__(self, socket, key):
        self.socket = socket
        self.key = key

    def run(self):
        while True:
            message = input()
            print()
            if message.lower == "exit":
                print("Exiting...")
                break
            encrypted_message = med.message_encrypt(self.key,
message.encode())
            self.socket.send(encrypted_message)
```

Figure 9: EncryptMessage class that manages the encryption of client’s information

```python
class DecryptMessage:
   def __init__(self, socket, key):
       self.socket = socket
       self.key = key

   def run(self):
       while True:
           response = self.socket.recv(1024)
           try:
               decrypted_message = med.message_decrypt(self.key, response)
               print(f"Plaintext from Client: {decrypted_message}")
               print("")
           except ValueError as e:
               print(f"\ndecryption error:  {e}")
               print(f"\nWill disconnect user")
               break
```

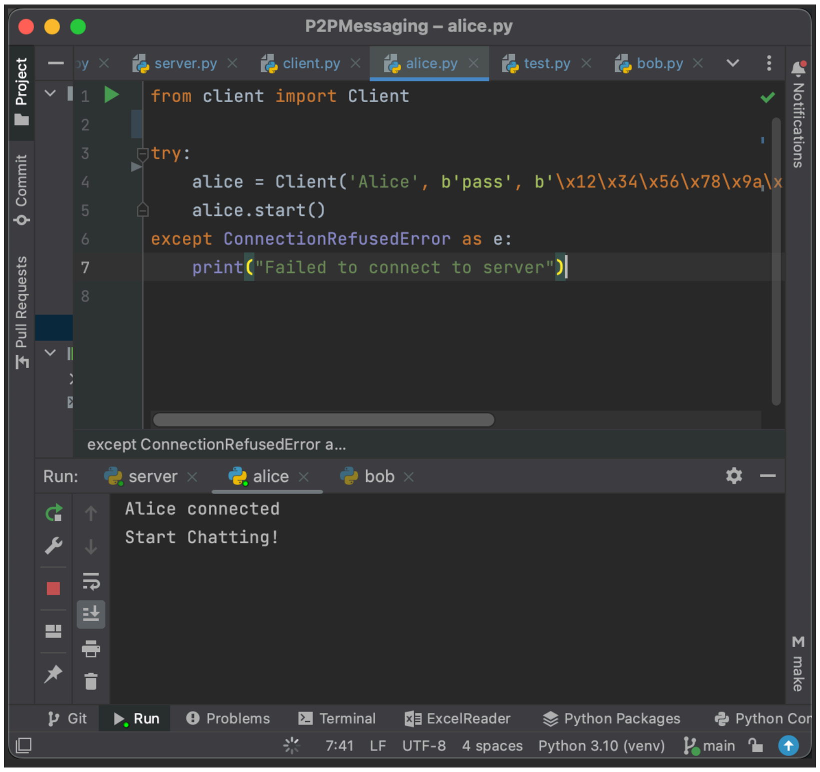
Figure 10: EncryptMessage class that manages the decryption of client's information. Important to note is if the client can't properly decrypt a ciphertext, this will lead to incorrect padding and thus a Decryption error.

```
class Client:

   def __init__(self, name, password, salt):
       self.salt = salt
       self.name = name
       self.password = password

       self.c_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
       self.c_socket.connect(('127.0.0.1', 12345))

       print(f"{self.name} connected")
       self.key = kdf.derive_key(password, self.salt)

   def start(self):
       print("Start Chatting!")
       decrypt_thread =
threading.Thread(target=DecryptMessage(self.c_socket, self.key).run)
       encrypt_thread =
threading.Thread(target=EncryptMessage(self.c_socket, self.key).run)

       encrypt_thread.start()
       decrypt_thread.start()

       encrypt_thread.join()
       decrypt_thread.join()
       self.c_socket.close()
```

Figure 11: Client class uses password and salt inorder to generate aa key. Then, the client connects to the server. After which one thread is created for encrypting messages and one for decrypting messages.

```
from client import Client

try:
   alice = Client('Alice', b'pass',
b'\x12\x34\x56\x78\x9a\xbc\xde\xf0\x11\x22\x33\x44\x55\x66\x77\x88')
   alice.start()
except ConnectionRefusedError as e:
   print("Failed to connect to server")
```

Figure 12: Alice joins the server by calling the Client class and providing her password and salt



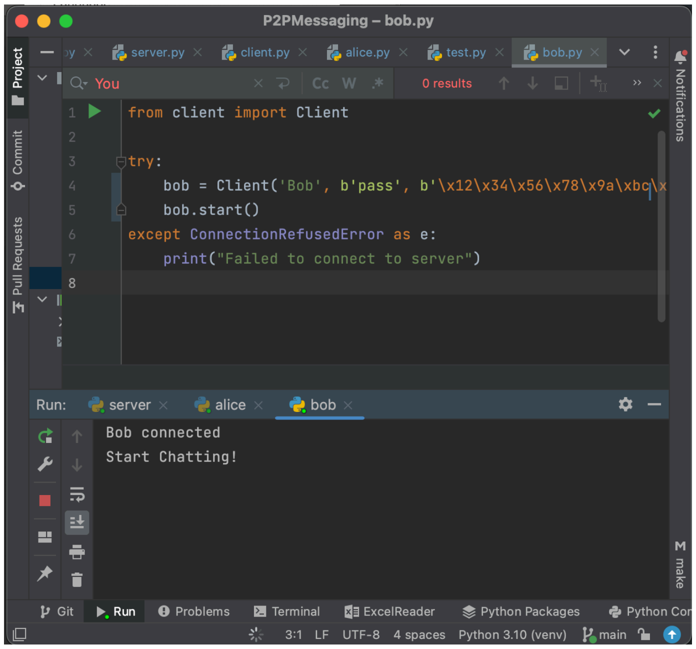
```
from client import Client

try:
    alice = Client('Alice', b'pass', b'\x12\x34\x56\x78\x9a\x
    alice.start()
except ConnectionRefusedError as e:
    print("Failed to connect to server")
```

```
Alice connected
Start Chatting!
```

Figure 13: Execution of Alice's script

```
from client import Client

try:
    bob= Client('Bob', b'pass',
b'\x12\x34\x56\x78\x9a\xbc\xde\xf0\x11\x22\x33\x44\x55\x66\x77\x88')
   bob.start()
except ConnectionRefusedError as e:
    print("Failed to connect to server")
```

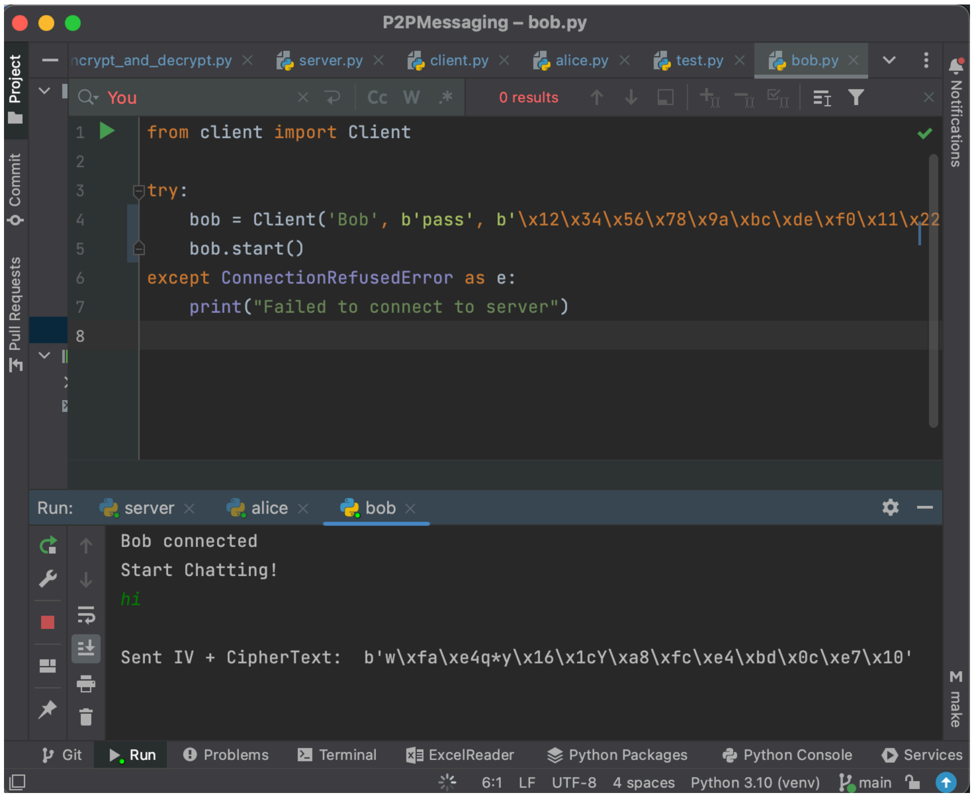
Figure 14: Bob joins the server by calling the Client class and providing his password and salt



```
from client import Client

try:
    bob = Client('Bob', b'pass', b'\x12\x34\x56\x78\x9a\xbc\x
    bob.start()
except ConnectionRefusedError as e:
    print("Failed to connect to server")
```

```
Bob connected
Start Chatting!
```

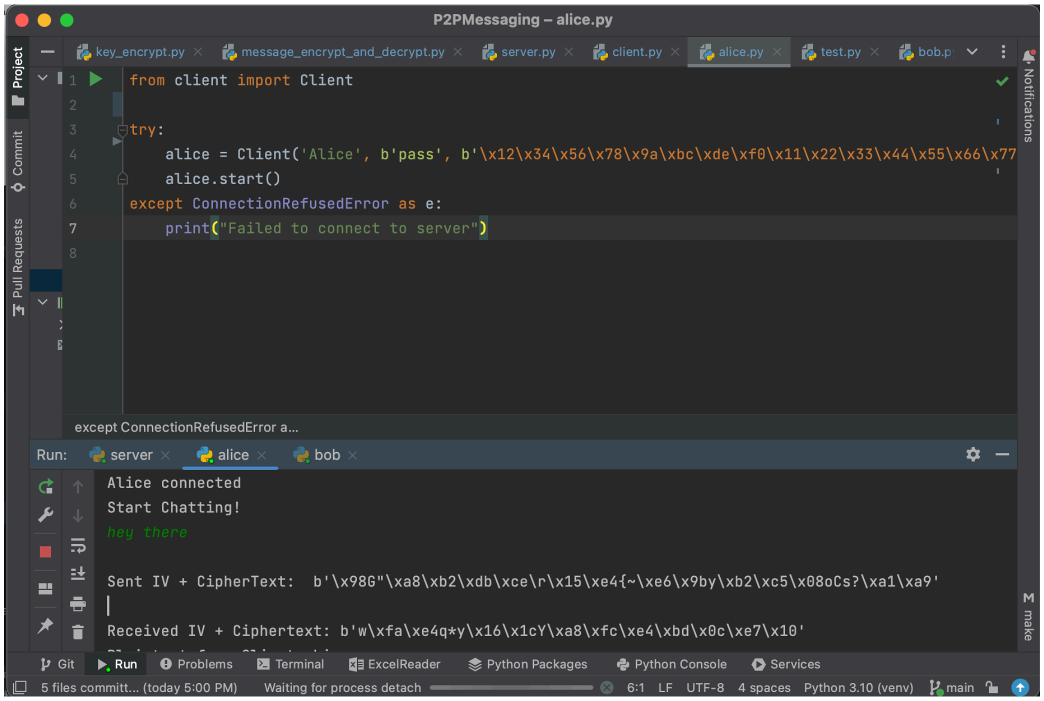
Figure 15: Execution of Bob’s script



```
from client import Client

try:
    bob = Client('Bob', b'pass', b'\x12\x34\x56\x78\x9a\xbc\xde\xf0\x11\x22
    bob.start()
except ConnectionRefusedError as e:
    print("Failed to connect to server")
```

```
Bob connected
Start Chatting!
hi

Sent IV + CipherText:  b'w\xfa\xe4q*y\x16\x1cY\xa8\xfc\xe4\xbd\x0c\xe7\x10'
```

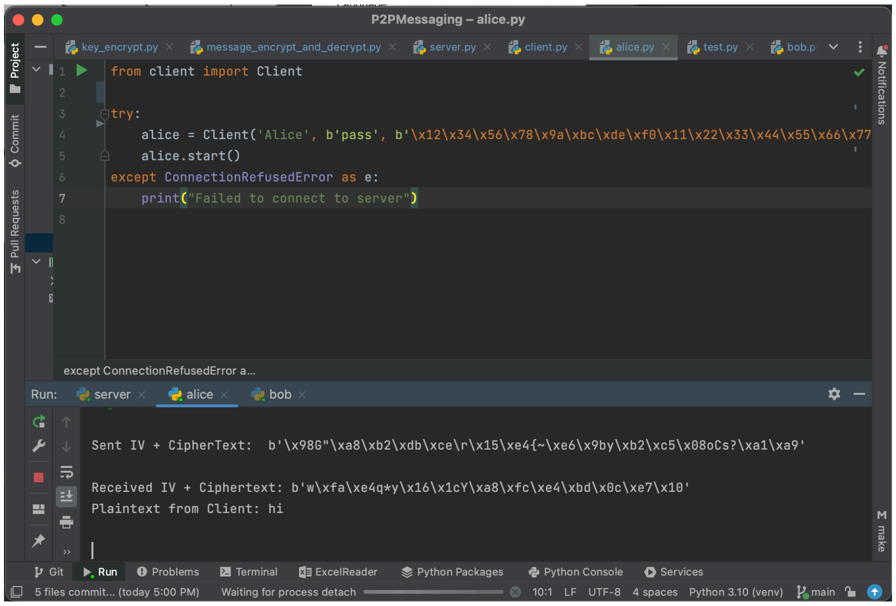
Figure 16: When Bob sends a message it displays the concatenation of IV and ciphertext



```
from client import Client

try:
    alice = Client('Alice', b'pass', b'\x12\x34\x56\x78\x9a\xbc\xde\xf0\x11\x22\x33\x44\x55\x66\x77
    alice.start()
except ConnectionRefusedError as e:
    print("Failed to connect to server")
```

```
Alice connected
Start Chatting!
hey there

Sent IV + CipherText:  b'\x98G"\xa8\xb2\xdb\xce\r\x15\xe4{~\xe6\x9by\xb2\xc5\x08oCs?\xa1\xa9'

Received IV + Ciphertext: b'w\xfa\xe4q*y\x16\x1cY\xa8\xfc\xe4\xbd\x0c\xe7\x10'
```

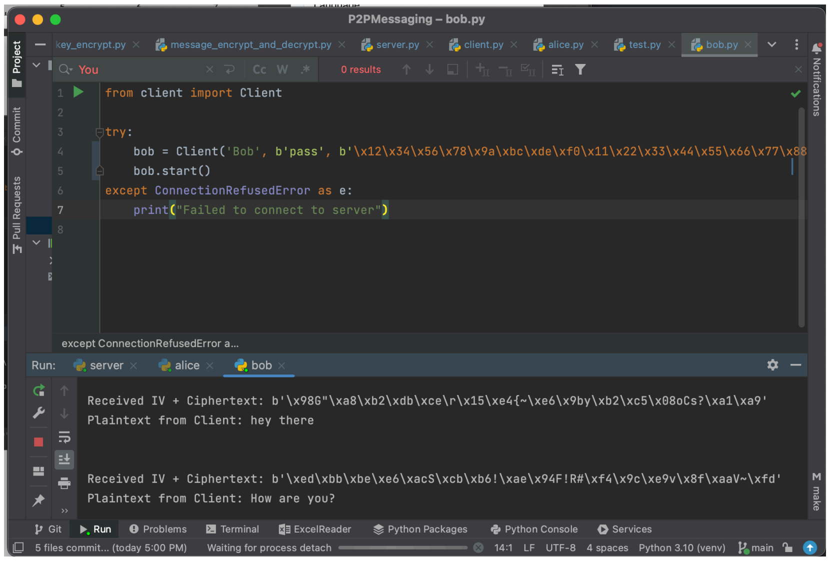
Figure 17: When Alice sends a message it displays its concatenation of IV and ciphertext



```
from client import Client

try:
    alice = Client('Alice', b'pass', b'\x12\x34\x56\x78\x9a\xbc\xde\xf0\x11\x22\x33\x44\x55\x66\x77
    alice.start()
except ConnectionRefusedError as e:
    print("Failed to connect to server")
```

```
Sent IV + CipherText:  b'\x98G"\xa8\xb2\xdb\xce\r\x15\xe4{~\xe6\x9by\xb2\xc5\x08oCs?\xa1\xa9'

Received IV + Ciphertext: b'w\xfa\xe4q*y\x16\x1cY\xa8\xfc\xe4\xbd\x0c\xe7\x10'
Plaintext from Client: hi
```

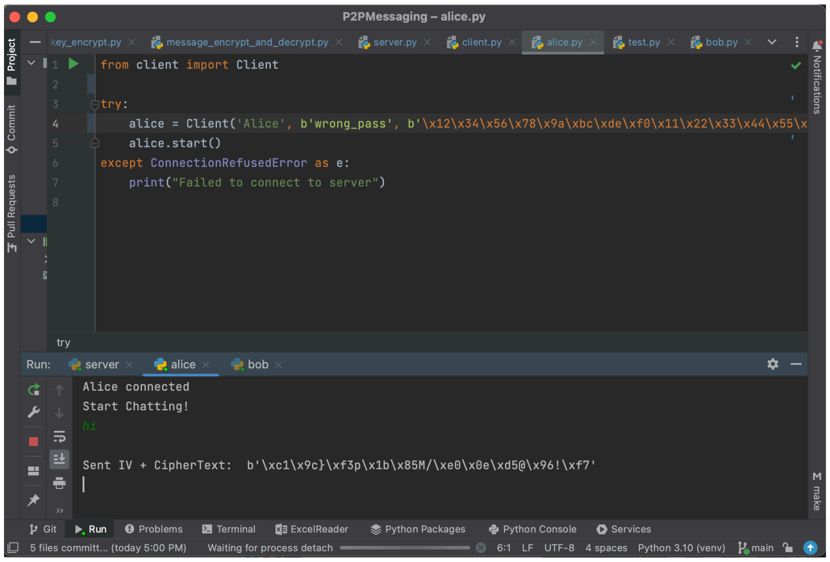
Figure 18: When Alice receives a message it displays its concatenation of IV and ciphertext. It also displayed the encrypted plaintext



```python
from client import Client

try:
    bob = Client('Bob', b'pass', b'\x12\x34\x56\x78\x9a\xbc\xde\xf0\x11\x22\x33\x44\x55\x66\x77\x88
    bob.start()
except ConnectionRefusedError as e:
    print("Failed to connect to server")
```

```
Received IV + Ciphertext: b'\x98G"\xa8\xb2\xdb\xce\r\x15\xe4{~\xe6\x9by\xb2\xc5\x08oCs?\xa1\xa9'
Plaintext from Client: hey there


Received IV + Ciphertext: b'\xed\xbb\xbe\xe6\xacS\xcb\xb6!\xae\x94F!R#\xf4\x9c\xe9v\x8f\xaaV~\xfd'
Plaintext from Client: How are you?
```

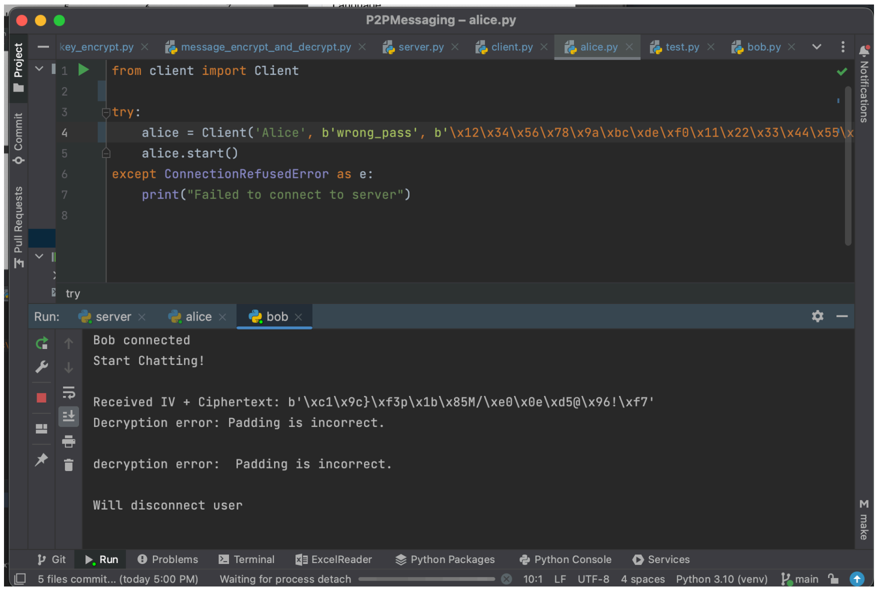
Figure 19: When Bob receives a message it displays its concatenation of IV and ciphertext. It also displayed the encrypted plaintext



```python
from client import Client

try:
    alice = Client('Alice', b'wrong_pass', b'\x12\x34\x56\x78\x9a\xbc\xde\xf0\x11\x22\x33\x44\x55\x
    alice.start()
except ConnectionRefusedError as e:
    print("Failed to connect to server")
```

```
Alice connected
Start Chatting!
hi


Sent IV + CipherText:  b'\xc1\x9c}\xf3p\x1b\x85M/\xe0\x0e\xd5@\x96!\xf7'
```

Figure 20: When Alice has an incorrect password she connects to the server and sends a message. Her ciphertext is shown



```python
from client import Client

try:
    alice = Client('Alice', b'wrong_pass', b'\x12\x34\x56\x78\x9a\xbc\xde\xf0\x11\x22\x33\x44\x55\x
    alice.start()
except ConnectionRefusedError as e:
    print("Failed to connect to server")

```

```
Bob connected
Start Chatting!

Received IV + Ciphertext: b'\xc1\x9c}\xf3p\x1b\x85M/\xe0\x0e\xd5@\x96!\xf7'
Decryption error: Padding is incorrect.

decryption error:  Padding is incorrect.

Will disconnect user
```

Figure 21: Bob gets the ciphertext, but is unable to decrypt it. So he is disconnected from the server.