University of Science and Technology of Southern Philippines

College of Information Technology and Computing
Department of Computer Science

Final Project Documentation: Secure Chat Application

December 16, 2024

Gefferson A. Balase
Instructor

Members:
Bañas, James Darwin
Galos, Zee
Roldan, Allen Vincent

1. Project Overview

A. Objective

The objective of this project is to design and implement a secure communication between end-users through the use of security measures such as message encryption and user authentication. The goal of this project is to develop a functional application that allows end-to-end communication and allowing messages to be secured as it goes through the network.

2. System Architecture

This project uses a client-server application where clients can interact and send messages to other end-clients through the use of server connection. The server will act as middleman to handle messages from the sender and sends to the designated recipient.

A. Client

The client side of this architecture initializes the connection. Clients can connect to the server through login credentials and a successful authentication. It can also send an encrypted messages to the server and decrypts the received messages.

B. Server

The server side acts like a middleman between the clients through handling of encrypted messages. It opens the server and listens to the connection. The server also handles authentication of clients through successful credentials, and handles messages from one client to another. It works by forwarding encrypted messages to the designated recipient.

C. Protocol Layer

1. **Key Matching** – A client must have a key that is matched in the server. This allows client to initially have a connection with the server

2. **Credentials** – Client must authenticate through log-in credentials. This allows the client to connect to the server and allow sending and receiving of messages. Unsuccessful credentials will not be able to connect to the server

3. **Client Feedback** – Used as debugging tool or a feedback system that is determined by client's actions. For example, A client tries to login, the client feedback sends an output that shows the status of the client's action.

4. **Server Feedback** – Used as debugging tool or a feedback system by the server, it shows an output regarding client authentication, forwarding and receiving messages from the clients.

5. **Connection** – Sender client sends a message to the receiver client must have that receiver client online or connected to the server

D. Protocol

This project uses a **TCP/IP** for secure, reliable, and connection-oriented data transmission. The server listens for client connections, and the client establishes a TCP connection.

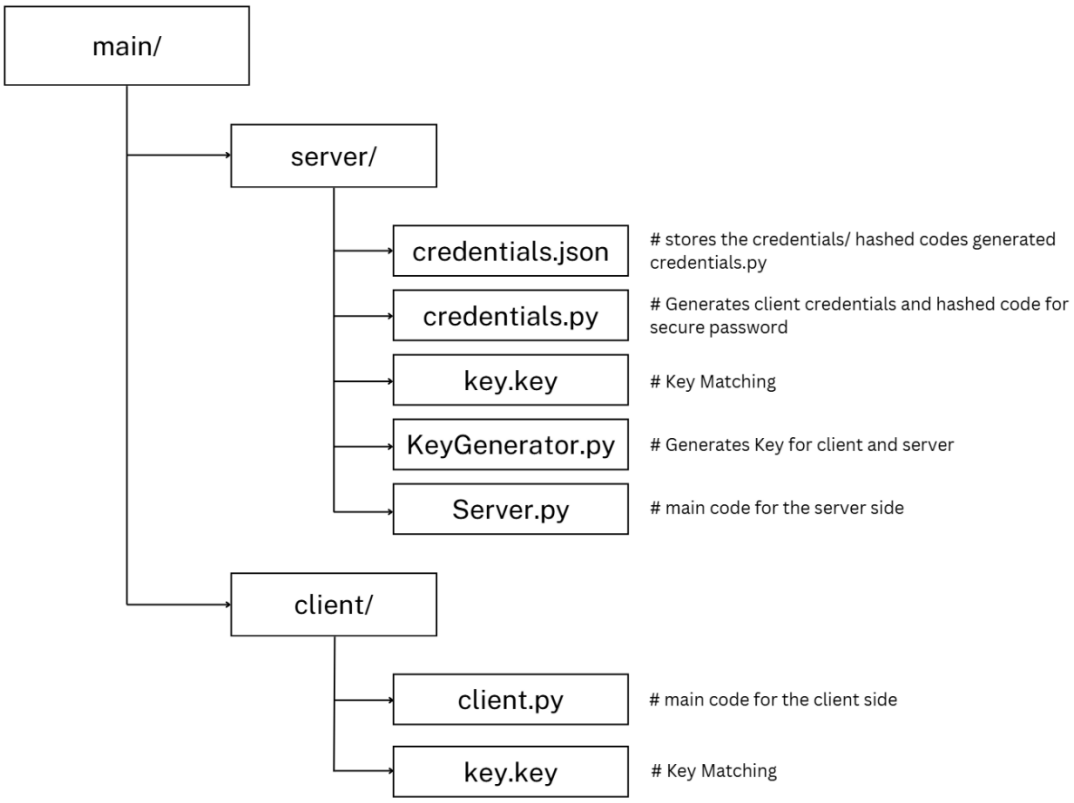
E. Technologies Used

This project uses a **python programming language** along with its libraries to perform a secure chat application with encrypted messages. This project also uses **JSON format** to store client credentials and hashed codes.

F. Libraries

import socket	Provides a networking interface to create and manage connections between the server and clients.
import threading	Allows concurrent handling of multiple client connections using threads. For client, it allows background message reception without blocking the GUI.
import json	Handles encoding and decoding of data in JSON format for user credentials.
import bcrypt	Implements secure password hashing and verification for authentication
from cryptography.fernet import Fernet	Provides secure encryption and decryption for the messages sent and received, ensuring security.
import tkinter as tk	Used to create the graphical user interface (GUI) for the chat application
from tkinter import scrolledtext	Extends tkinter by providing a scrollable text box for displaying chat messages.

3. **Code Structure**



4. **Client Implementation**

The client-side is responsible for initializing the connection, connects to the server via credentials and authentication. Client can send encrypted messages to the server; client can also receive encrypted messages and decrypt it. The messages are being displayed at the client's chat box. Lastly, client can close the connection.

The code below is the implementation of the client-side in python:

```
import socket
from cryptography.fernet import Fernet
import tkinter as tk
from tkinter import scrolledtext
import threading  # For receiving messages in a separate thread

# Load the encryption key
def load_key():
    with open("key.key", "rb") as key_file:
        return key_file.read()
```

```python
# Encrypt a message
def encrypt_message(message, key):
    fernet = Fernet(key)
    return fernet.encrypt(message.encode('utf-8'))

# Decrypt a message
def decrypt_message(encrypted_message, key):
    fernet = Fernet(key)
    return fernet.decrypt(encrypted_message).decode('utf-8')

# Connect to the server and authenticate
def connect_to_server(username, password):
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect(("127.0.0.1", 12345))

    # Send authentication data (username and password)
    auth_data = f"{username},{password}"
    client.send(auth_data.encode('utf-8'))

    # Receive authentication response
    response = client.recv(1024).decode('utf-8')
    if response == "AUTH_SUCCESS":
        print(f"{username} authenticated successfully!")
        return client
    else:
        print(f"{username} authentication failed!")
        client.close()
        return None

# Send encrypted message to the server
def send_message(client, message, recipient, key, username):
    # Encrypt the message
    encrypted_message = encrypt_message(message, key)

    # Format the data for the server (include recipient only for routing purposes)
    data = f"{recipient}||from {username}||{encrypted_message.decode('utf-8')}"

    print(f"Sending to server: {data}")  # Debug for what is sent
    client.send(data.encode('utf-8'))

# Function to receive messages from the server
def receive_message(client, key, chat_box):
    while True:
        try:
            # Receive message from the server
            data = client.recv(4096).decode('utf-8')
            if data:
                # Debug: Print the received raw message
                print(f"Received from server: {data}")

                # Parse the message (sender||<encrypted_message>)
                parts = data.split("||", 2)  # Split into at most 3 parts
                if len(parts) == 3:
                    sender = parts[0]  # First part is the sender
                    encrypted_message = parts[2]  # Third part is the encrypted message

                    # Decrypt the message
                    decrypted_message = decrypt_message(encrypted_message.encode('utf-8'), key)
                    print(f"Decrypted message from {sender}: {decrypted_message}")

                    # Display the message in the chat box with the sender's name
                    chat_box.insert(tk.END, f"{sender}: {decrypted_message}\n")
                else:
                    print("Error: Received message in incorrect format.")
            else:
                print("No data received or connection lost.")
        except Exception as e:
            print(f"Error receiving message: {e}")
            break
```

```
# Create the chat GUI
def chat_ui(client, key):
    def send_message_gui():
        message = message_input.get()
        recipient = recipient_input.get()
        if message and recipient:
            if recipient == username:
                chat_box.insert(tk.END, "Error: You cannot message yourself.\n")
                message_input.delete(0, tk.END)
                return
            send_message(client, message, recipient, key, username)
            chat_box.insert(tk.END, f"You: {message}\n")
            message_input.delete(0, tk.END)

    root = tk.Tk()
    root.title(username)

    chat_box = scrolledtext.ScrolledText(root, width=50, height=20)
    chat_box.pack(padx=10, pady=10)

    recipient_input = tk.Entry(root, width=40)
    recipient_input.pack(side=tk.LEFT, padx=10, pady=10)
    recipient_input.insert(0, "doggy")

    message_input = tk.Entry(root, width=40)
    message_input.pack(side=tk.LEFT, padx=10, pady=10)
    message_input.insert(0, "Meow Meow")

    send_button = tk.Button(root, text="Send", command=send_message_gui)
    send_button.pack(side=tk.RIGHT, padx=10, pady=10)

    # Start a thread to receive messages
    threading.Thread(target=receive_message, args=(client, key, chat_box), daemon=True).start()

    root.mainloop()

# Example usage
if __name__ == "__main__":
    username = input("Username: ")
    password = input("Password: ")

    # Connect to the server and authenticate
    client = connect_to_server(username, password)
    if client:
        # Load encryption key
        key = load_key()

        # Run chat UI
        chat_ui(client, key)
```

5. Server Implementation

The server side is responsible for opening the server and listens to the connection, it also manages client authentication through successful credentials. Server receives an encrypted message from client and forwards it to the designated recipient.

The code below is the implementation of the server-side in python:

```python
# server.py

import socket
import threading
import json
import bcrypt  # For secure password hashing

clients = {}  # Store connected clients by username

# Load credentials
def load_credentials():
    with open("C:/Users/Blanc/Desktop/Server/credentials.json", "r") as file:
        return json.load(file)

# Authenticate user
def authenticate(username, password):
    credentials = load_credentials()
    print(f"Authenticating user: {username} with password: {password}")

    if username in credentials:
        stored_hash = credentials[username]
        print(f"Stored hash for {username}: {stored_hash}")  # Debugging line

        if bcrypt.checkpw(password.encode('utf-8'), stored_hash.encode('utf-8')):  # Ensure both are bytes
            print("Authentication successful!")
            return True
        else:
            print("Authentication failed: Incorrect password")
            return False
    print(f"Authentication failed: {username} not found")
    return False


# Handle client connections
def handle_client(client_socket):
    try:
        # Receive authentication data
        auth_data = client_socket.recv(1024).decode('utf-8')
        username, password = auth_data.split(',')

        if authenticate(username, password):
            client_socket.send("AUTH_SUCCESS".encode('utf-8'))
            print(f"User {username} authenticated")
            clients[username] = client_socket
            relay_messages(client_socket, username)
        else:
            client_socket.send("AUTH_FAIL".encode('utf-8'))
            client_socket.close()
    except Exception as e:
        print(f"Error handling client: {e}")
        client_socket.close()

# Relay messages between clients
def relay_messages(client_socket, username):
    try:
        while True:
            # Receive encrypted message
            data = client_socket.recv(4096)
            if not data:
                break

            # Decode recipient and encrypted message
            recipient, encrypted_message = data.decode('utf-8').split("||", 1)

            print(f"Received encrypted message for {recipient}: {encrypted_message}")

            # Forward the encrypted message to the recipient if they are online
            if recipient in clients:
                # Forward only sender information and encrypted message
                clients[recipient].send(f"{username}||{encrypted_message}".encode('utf-8'))
            else:
                client_socket.send("Recipient not online".encode('utf-8'))
    except Exception as e:
        print(f"Error relaying messages: {e}")
    finally:
        print(f"User {username} disconnected")
        del clients[username]
        client_socket.close()

# Main server function
def start_server():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(("0.0.0.0", 12345))
    server.listen(5)
    print("Server is running...")
    while True:
        client_socket, _ = server.accept()
        threading.Thread(target=handle_client, args=(client_socket,)).start()

# Run server
if __name__ == "__main__":
    start_server()
```

6. Protocol Design

The protocol ensures secure communication by verifying users with a username and password, allowing access only if correct. Messages are protected with encryption so only the intended receiver can decrypt and read them. When sending a message, the server makes sure it reaches the right recipient if they're online; if not, the sender is notified. The system also handles disconnections by keeping track of who is online to ensure smooth messaging.

A. Protocol Outline

1. **Handshake**: The client connects to the server and sends a username and password for verification. The server checks the credentials and responds with either "Authentication Successful" or "Authentication Failed." If successful, the user can start messaging.

2. **Message Transfer**: Messages are encrypted to ensure privacy before being sent. The client includes the recipient's name along with the encrypted message, which the server forwards to the correct user. If the recipient is offline, the server notifies the sender.

3. **End of Transfer**: When a user disconnects, the server updates its list of active users, ensuring no further messages are sent to users who are offline.

4. **Encryption**: All messages are securely encrypted so only the intended recipient can read them, maintaining privacy during transmission.

5. **Error Handling**: If a connection is lost or issues arise, the server keeps the system stable and notifies users of any problems to ensure smooth communication.

B. Protocol Flow

1. **Client Connection**: The client establishes a connection with the server.

2. **Handshake (Authentication)**: The client sends the username and password to the server. The server verifies the credentials and responds with either "Authentication Successful" or "Authentication Failed." If authentication fails, the connection is closed.

3. **Message Encryption**: Before sending a message, the client encrypts it using a secure encryption key to ensure privacy.

4. **Message Transfer**: The encrypted message, along with the recipient's name, is sent to the server. The server identifies the recipient and forwards the message if they are online. If the recipient is offline, the sender is notified.

5. **Message Decryption**: The recipient receives the encrypted message and decrypts it using the shared encryption key to read its content.

6. **Disconnection**: When a user disconnects, the server removes them from the active user list and stops routing messages to them.

7. Error Handling and Logging

A. Error Handling

The system ensures smooth operation by managing unexpected issues during communication. If a user provides incorrect credentials during authentication, the server denies access and closes the connection. When a message cannot be delivered because the recipient is offline, the sender is notified immediately. If the server detects a lost connection or abrupt disconnection, it removes the user from the active list to prevent further issues. These measures ensure that the system remains stable even when errors occur.

Here's a sample of how the application notifies user in case of errors:

Event	Message Output
Incorrect Password	Authentication failed: Incorrect Password
Incorrect Username	Authentication failed: User not found
Unsuccessful Logins	Authentication failed
Offline Recipient	Recipient not online
Message Receive Error	No data received or connection lost/ Error: Received message in incorrect format/ Error receiving message
Server Forwarding Error	Error relaying messages
Client Disconnection	User {username} disconnected

B. Logging

To maintain system reliability, the server logs important events such as user logins, authentication attempts, message deliveries, and disconnections. Errors, like failed authentication or message routing issues, are also recorded for troubleshooting purposes. These logs help monitor the system's activity, identify recurring problems, and improve overall performance.

Here's a sample of how the application notifies user during logging:

Event	Type	Message Output
Server Started	Server	Server is running…
Successful Authentication	Server	User [Username] authenticated successfully
Successful Authentication	Client	Authenticated successfully!
Send Messages	Client	Sending to server [encrypted message]
Received Messages	Server	Received Message from [user], [encrypted messages]
Received Messages	Client	Received message from server [from user], [encrypted messages]

8. Documentation

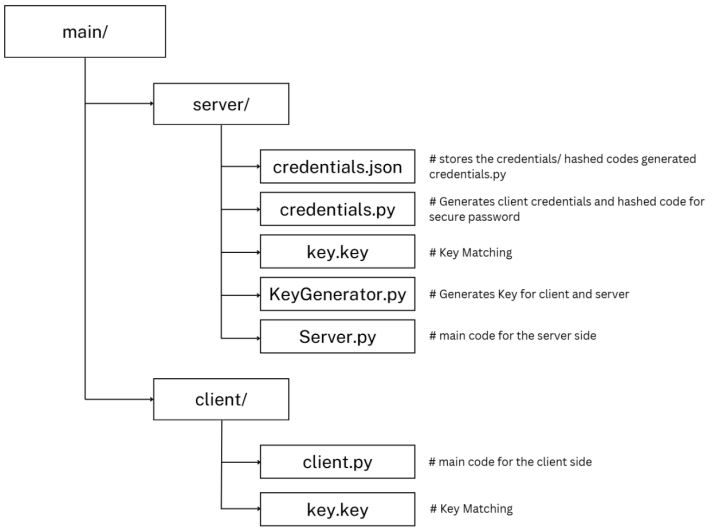
A. Setup

In order to run this program, user must setup the client and server. First, users should make folders where the client and server located. Users can choose to separate the client and server in different folders but ensure to update the path in the code later on.

In the server side, it must have 'credentials.json', 'credentials.py', KeyGenerator.py', and 'Server.py'. First, update the folder directory of the code in KeyGenerator.py to where the path folder of the client and server located. Next, run the code to generate a 'key.key' for client and server. For the credentials.py, you can add users, run the program to generate a hashed password and save it in 'credentials.json'.

Once the initial setup is complete, users can now run the server. For client side, users can run the code and enter user credentials to connect to the server.

Here's a sample structure of how the setup would look like:



B. **Configuration Settings**

This program uses python libraries such as '**bcrypt**' and '**fernet**'. In order to install these libraries, users should download a python package dedicated to each of them.

In windows, users may open their cmd and type '**pip install cryptography**' & '**pip install bcrypt**'. Ensure that a python is already installed in the device.

C. API Reference

In server side, the code uses different functions for user authentication, receiving messages, forwarding message, and other protocols.

The table below defines the function codes of the server:

Code	Description
def load_credentials()	Load credentials of the users
def authenticate(username, password)	Handling user authentication
def handle_client(client_socket)	Responsible for handling client connections
def relay_messages(client_socket, username):	Responsible for forwarding messages between clients
def start_server()	Main server function and runs the server
if __name__ == "__main__": start_server()	Runs the server code

In client side, the code uses functions that can connect to the server through authentication, can send and encrypt messages and receive and decrypt.

The table below defines the function codes of the client:

Code	Description
def load_key()	Load the encryption key
def encrypt_message(message, key):	Encrypts messages before it was being sent
def decrypt_message(encrypted_message, key)	Decrypt received messages
def connect_to_server(username, password)	Connect to the server and authenticate
def send_message(client, message, recipient, key, username)	Send encrypted message to the server
def receive_message(client, key, chat_box)	Function to receive messages from the server
def chat_ui(client, key)	Create the chat GUI
if __name__ == "__main__"	Runs the client code

D. Troubleshooting

Troubleshooting is essential for identifying and resolving common issues that may occur during system operation. Below are some common problems, their causes, and suggested solutions.

Events	Cause	Troubleshoot
Authentication Failure	Incorrect username / password	Ensure the username and password are entered correctly. Verify that the credentials match the data stored on the server.
Connection Errors	Network instability, server downtime, or firewall blocking the connection.	Check network stability and ensure the server is running. Verify that the server is reachable and that no firewall settings are blocking the communication port
Message Delivery Failure	Recipient is offline or unavailable.	Ensure the recipient is logged in. If they are offline, wait until they reconnect. Notify users if they attempt to send messages while the recipient is not online.
Encryption / Decryption Errors	Invalid encryption key or data corruption	Verify the integrity of the encryption key (key.key) and confirm that it is being loaded correctly. Ensure both sender and receiver use the same encryption key.
Disconnections	Network issues, unstable connections, or abrupt user termination.	Implement error handling to detect and gracefully handle disconnections. Ensure that clients reconnect if they lose connection.

Invalid Message Format	Message formatting issues during encryption or sending.	Review the message formatting logic to ensure the correct message structure is maintained during encryption and sending.

E. Test Instruction

Before running and testing the application, ensure all the necessary requirements are met and the system is set up correctly. Below is the pre-requisite checklist and test instructions.

Requirements:

1. **Python Installation** – A device must have a compatible python version installed to run the application

2. **Server Configuration** – Ensure that the server is running before attempting to connect

3. **Client Configuration** – Ensure that client has been authenticated successfully

4. **Key Matching** – The ‘key.key’ file must exist in the same directory as the client and the server respectively.

5. **Libraries / Dependences** – Libraries such as ‘bcrypt’ and ‘cryphtography’ must be installed before running the program.

Run Program:

1. Ensure that the setup is correctly configured

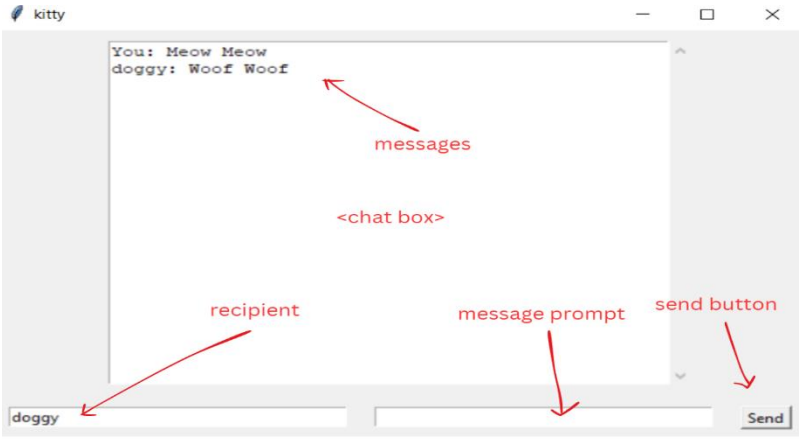
2. Run the server program

3. Run the client program (recommend to have at least 2 clients running to the server)

4. Client authenticates through successful login to connect to the server (Chat box will appear if authenticated successfully)

5. In the bottom side of the chat box, the left side is intended for the recipient, and the right side is for the message users want to send

6. Press the send button to send messages.



9. Testing

A. Connection Test & Edge Cases

We tested connection from the client and server to ensure that it is working as intended with the use of feedback. We also tested invalid client authentication to verify that the connection should only be made if clients are successfully authenticated. Purposely disconnect a client during the connection to further notify client connectivity. Lastly, we tested client sending and receiving messages.

Unsuccessful client authentication:

```
Server is running...
Authenticating user: kitty with password: 12
Stored hash for kitty: $2b$12$tgaTAkjitZUr0uo1Kk6H7uEqzwxNd5q7kQHwPMnoHc5EfS2cLF
C6y
Authentication failed: Incorrect password
```

Successful client authentication:

```
Authenticating user: kitty with password: 1234
Stored hash for kitty: $2b$12$tgaTAkjitZUr0uo1Kk6H7uEqzwxNd5q7kQHwPMnoHc5EfS2cLF
C6y
Authentication successful!
User kitty authenticated
```

Disconnected Client:

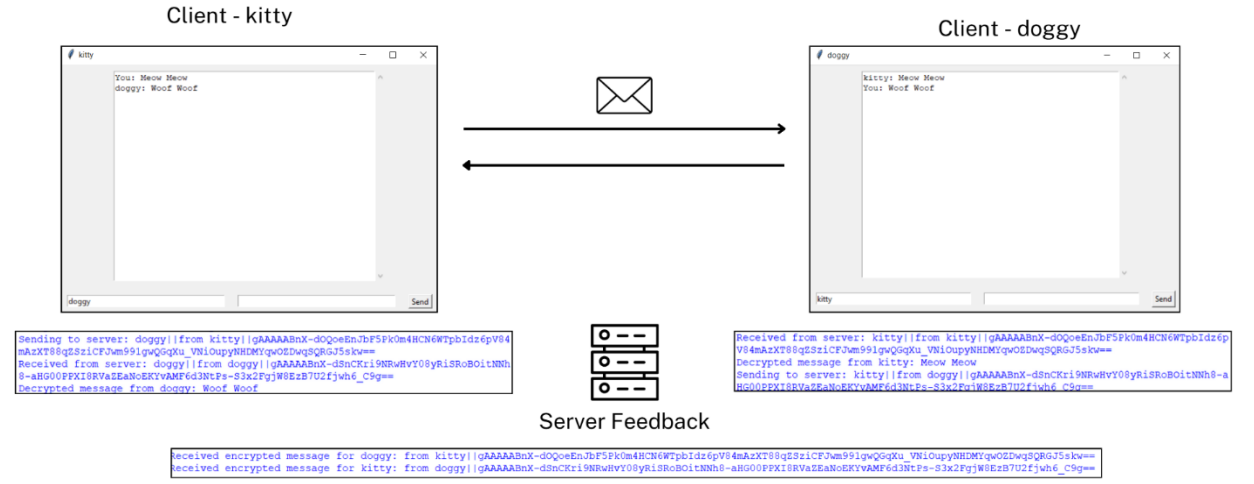
```
Error relaying messages: [WinError 10054] An existing connection was forcibly
closed by the remote host
User kitty disconnected
```

Sending / Receiving Messages:

```
Sending to server: doggy||from kitty||gAAAAABnYRVsODzzJmGHnGKR5s7RUQnCvNhWU8C-jw
eI6TTcuKCtmlc3r-JlaaQfhG0d2qdwRdjSHL6kGTfZbWh3P4P6tUICcw==
Received from server: doggy||from doggy||gAAAAABnYRVyIydcbw4YM8CYQ-Biiog1EmLMw7y
hEqu4RVBX2-gw42yDN_VnBAFbQP7hMgalM8j803Dposvy0BHmadaJbJZG-Q==
Decrypted message from doggy: Woof Woof
```

B. File / Message Transfer

We tested our program to send and receive messages through the server and check if the messages have reached to the recipient. The figure below shows a successful transfer of messages using our program:



C. Edge Cases

We tested different scenarios and purposely made errors to determine how would it works and how our program handles errors.

No server running:

```
Username: kitty
Password: 1234
Traceback (most recent call last):
  File "C:\Users\Blanc\Desktop\Client\Client.py", line 125, in <module>
    client = connect_to_server(username, password)
  File "C:\Users\Blanc\Desktop\Client\Client.py", line 25, in connect_to_server
    client.connect(("127.0.0.1", 12345))
ConnectionRefusedError: [WinError 10061] No connection could be made because the
 target machine actively refused it
```

Login with incorrect credentials:

```
Authenticating user: kitty with password: d
Authentication failed: Incorrect password

Authenticating user: meow with password: 1234
Authentication failed: meow not found
```

Connection Lost:

```
Error receiving message: [WinError 10054] An existing connection was forcibly
closed by the remote host
```

Recipient not online:

```
Received from server: Recipient not online
```

10. Conclusion

In conclusion, this application is a simple yet secure messaging system that allows users to authenticate, send, and receive encrypted messages through a client-server architecture. It combines encryption, authentication, and reliable communication to provide a smooth and secure messaging.