Project Report: Secure Messaging Application Overview

Encho Genchev

Dept. of Computer Science

University of Oklahoma

[encho.s.genchev-1@ou.edu](mailto:encho.s.genchev-1@ou.edu)

**Abstract**  
This project for CS 4173 involved designing and implementing a secure point-to-point (P2P) messaging application. The task was to create an application that allows two users, Alice and Bob, to exchange encrypted messages, including text, images, voice, and any other files. For encryption and decryption, they will rely on a shared password and symmetric key cryptography. This report outlines the design architecture, encryption for 56-bit and 128-bit keys respectively, socket-based communication, GUI implementation, and overall system security. The design ensures confidentiality via keys derived from Alice and Bob’s shared password and includes a full experimental demonstration with successful message transfers.

**1. Introduction**

Secure communication is an essential requirement in modern messaging systems. For this project, I developed a secure P2P instant messaging application that encrypts messages between two users, Alice and Bob, based on a pre-shared password. The system allows users to send and receive not only plaintext, but also images, audio messages, and files. A GUI was created to improve usability and display messages based on successful or unsuccessful encryption/decryption.

This project challenged me to practice system design, cryptography, and consider user experience. By combining socket programming with cryptographic libraries, I created a system that can be extended to secure real-world communication. As it stands now, the system is only supported locally but can be easily scaled to be hosted on a remote server.

**2. System Design**

**2.1 Architecture Overview**

The system is implemented in Python and uses sockets to achieve two-way communication between a client and a locally hosted server. Both Alice and Bob connect to the server through their respective client interfaces. Threading is used so that each client and the server can constantly listen for incoming or outgoing messages or files.

* 1. **User Workflow**

At startup, each user provides:

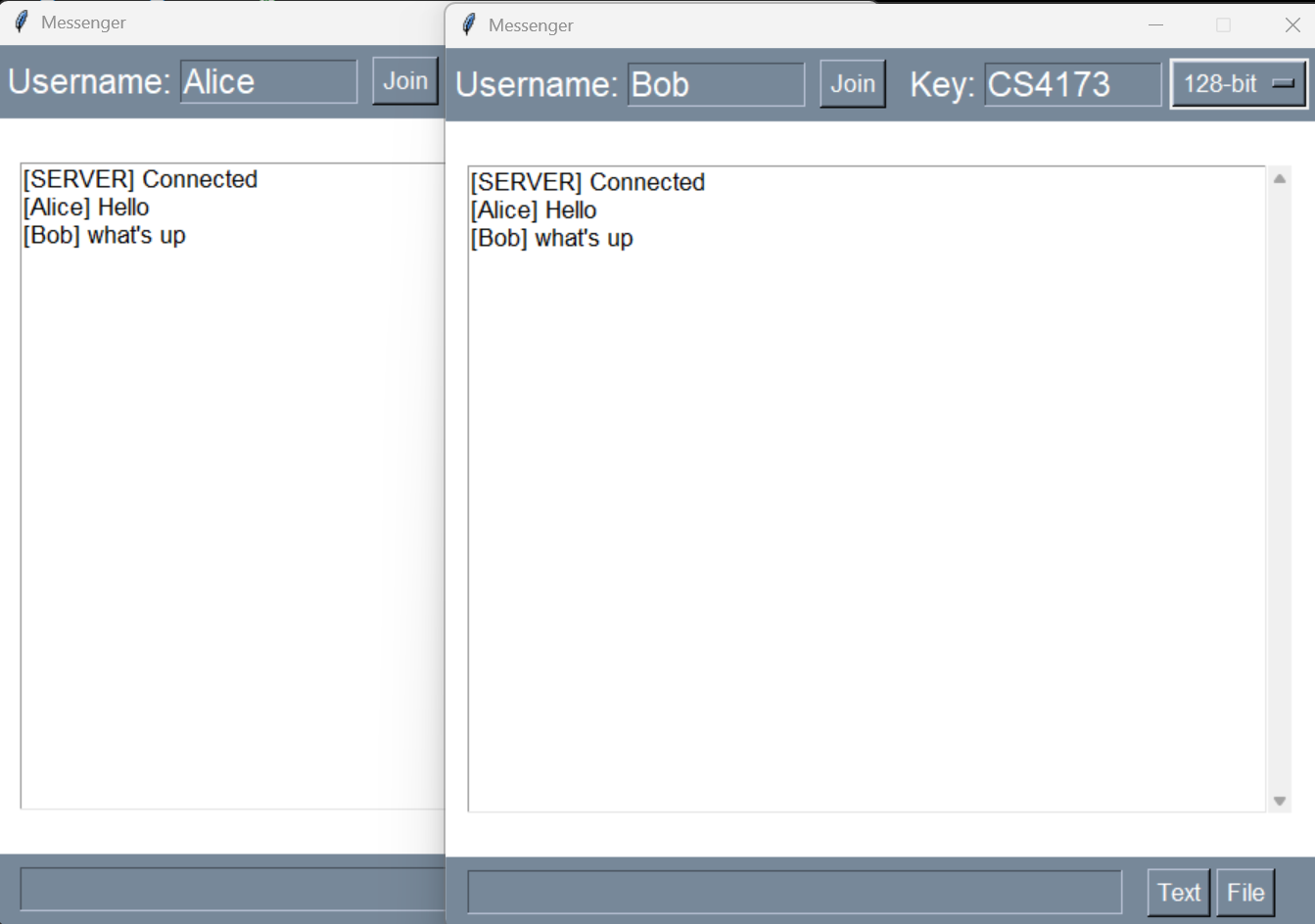
* username
* shared password
* key size selection: either 56-bit or 128-bit

A close-up of a box

AI-generated content may be incorrect.

*Fig. 1: username and password input interface*

The GUI displays all received messages in plaintext and reports any decryption errors. Simple messages can just be typed in the bottom textbox. After that the user clicks the ‘Text’ button and the message should appear for both users connected to the server with the correct key and bit selection.



*Fig. 2: successful transmission of messages from both Alice and Bob*

* 1. **Message Types Supported**
* Text
* Image files
* Voice/audio files
* Any other file

Files are sent by typing their filename in the GUI and clicking the ‘File’ button. Since the system is locally hosted, sent and received files appear in the same directory but with updated filenames to distinguish them. This would not happen on a remote server, but it works for the sake of the project

A screenshot of a computer

AI-generated content may be incorrect.

*Fig. 3: directory contents before any file transfer occurs*

*A screenshot of a computer

AI-generated content may be incorrect.*

*Fig. 4: contents of sample file*

*A screenshot of a computer

AI-generated content may be incorrect.*

*Fig. 5: successful transmission of sample file from Bob to Alice, along with previously sent messages*

A screenshot of a computer

AI-generated content may be incorrect.

*Fig. 6: content of directory after file transmission and contents of the file Alice received*

**3. Cryptography Design**

* 1. **Password-Derived Key Generation**

Instead of using the shared password directly as a cryptographic key (which is insecure) the password is hashed using SHA-256, and the first *n* bytes of the hash are used:

* 8 bytes (64 bits) for DES
* 16 bytes (128 bits) for AES

This allows the program to use that same hashing algorithm for both encryption options, only using the number of bits needed for each one. This is not only good for security, but also simplicity.

* 1. **DES vs. AES**

The application supports two key lengths:

* 56-bit DES: this is an older encryption standard that is easily overcome by bruteforce attacks by powerful, modern computers. The option is still here though to meet project requirements.
* 128-bit AES: This is a more modern encryption standard that is still widely used and much more secure than DES.

AES is much more secure since DES's shorter key size is no longer considered secure against modern, powerful computers.

* 1. **Encryption and Padding**

All messages are encrypted using cipher block chaining (CBC) mode with a randomly generated IV for each message. This ensures confidentiality of the ciphertext and makes it so all ciphertext is unique, even if the same message is being encrypted.

All data is padded before encryption to meet the block size requirements. This is handled automatically by Python’s pad() function.

**4. Security Model and Threat Analysis**

This section outlines the assumptions about the scenario, different threats, and how the system responds to attacks.

* 1. **Assumptions**
* Alice and Bob have securely exchanged a password ahead of time.
* The server may be monitored by an eavesdropper
* The server hosting the application is trusted but currently local.
  1. **Threats Addressed**
* Eavesdropping: since encryption occurs on the client side, intercepted messages would be ciphertext.
* Replay Attacks: Since every message uses a different IV, even identical messages result in different ciphertexts, which would make it impossible for an attacker to find the pattern
* Incorrect keys: error messages are displayed if the incorrect key or bit length is used by either user.

Limitations: If an attacker somehow finds out the key and bit length, they can impersonate the user. There is no authentication like a generated signature to authenticate users.

**5. GUI and User Interaction**

* 1. **Design Features**

The GUI was built using Tkinter. The GUI allows for:

* Entering username, password, and key length
* Sending text messages and files
* Viewing the messages of the server and all users in a text stream

The text stream was unable to display incorrectly decrypted ciphertext because the encoding and decoding functions used did not support many of the symbols; however, ciphertext was still displayed in the encrypted files sent back and forth. In a future update I could focus more on displaying the ciphertext in the text stream, but I had limited time.

* 1. **Usability Decisions**
* GUI elements are clearly labeled to avoid clutter.
* Constants are used to define fonts and colors for easy modification
* Sending files based on filename, simplifying system design
* Error messages prompt user to modify their actions accordingly

**A computer screen shot of a message

AI-generated content may be incorrect.**

*Fig. 7: error message signaling missing encryption information*

*A screenshot of a computer

AI-generated content may be incorrect.*

*Fig. 8: error message signaling incorrect encryption/decryption information*

**6. Implementation Challenges**

Implementing encrypted P2P messaging presented several challenges:

* Threading: it is confusing but necessary to make clients and servers constantly listen for new messages and files
* File parsing: files had to be correctly read, reconstructed, encrypted, sent, and decrypted
* IV management: the random IV’s had to be sent correctly with the ciphertext otherwise decryption wouldn’t work.
* Displaying ciphertext: Attempting to show raw ciphertext causes issues due to non-printable characters.

Despite these challenges, the final application successfully meets all functional and security requirements.

**7. Experimental Results**

**7.1 Text Messaging**

As seen in Figure 2, plaintext messages between Alice and Bob were successfully encrypted, transmitted, decrypted, and displayed in real time.

* 1. **File Transfers**

Multple file types were successfully sent during development:

* jpg images
* m4a audio files
* txt files

In figures 4 and 5, we see the receiver was able to decrypt these files with the correct key and bit length otherwise just received the ciphertext.

A screenshot of a computer

AI-generated content may be incorrect.

*Fig. 9: contents of received txt file if the incorrect key or bit length was inputted by either user*

**7.3 Error Handling**

Using the wrong password or mismatched key lengths triggered visible error messages in the GUI, preventing incorrect data display or crashes as seen in figures 7 and 8.

**8. Limitations and Future Work**

* 1. **No Key Rotation**

Currently, the same password-derived key is used for the entire session. Future iterations could implement:

* Key rotation after N messages or time
* Key rotation after a certain amount of time has
  1. **Authentication and Identity**

There is no mechanism to verify the identity of the sender. In the future:

* Message signing with MAC’s or digital signatures
* Secure login and two factor authentication could be added

**8.3 Remote Hosting**

Currently, the application is only hosted locally. Future updates could involve hosting the server remotely.

**9. Conclusion**

This secure messaging system demonstrated the application of cryptographic principles in a practical, usable application. From socket-based communication to password-based encryption with DES and AES, the system met the project goals and provided a functional secure communication platform. It serves as a decent prototype for more advanced, secure chat systems and reinforces the importance of computer security practices in real-world applications.