Sri Sivasubramaniya Nadar College of Engineering, Kalavakkam, 603110

(An Autonomous Institution, Affiliated to Anna University)

Department of Computer Science and Engineering

II Year CSE (III Semester)

Batch 2024-2028

Academic Year 2025-26

UCS3361 - Object Oriented Programming (TCP)

Mini Project

P2P Secure Messenger

Submitted by,

Shivaprakash N P (3122245001150)

Sarvesh K P (3122245001143)

Sharon Nivedha S (3122245001148)

INDEX

1. Problem Statement

Contemporary digital communication is overwhelmingly dominated by the client-server architecture, a model where centralized servers act as obligatory intermediaries for all data exchange.[1] While this model has enabled global connectivity, it introduces fundamental architectural vulnerabilities. Centralized systems represent a single point of failure; an outage or cyberattack on a server can disrupt communication for its entire user base. More critically, the concentration of user data on company-controlled servers creates significant privacy and security risks, making sensitive conversations susceptible to surveillance, unauthorized access, data breaches, and censorship. Users are forced to trust a third-party entity with the confidentiality and integrity of their communications.

The core problem addressed by this project is the inherent fragility and lack of user sovereignty in centralized communication systems. There exists a need for a robust, secure, and serverless communication paradigm that empowers users with direct control over their data and conversations, particularly within trusted local network environments.

This project proposes a solution in the form of a Peer-to-Peer (P2P) Secure Messenger. In a P2P network, the traditional distinction between client and server is eliminated. Instead, the network consists of interconnected nodes, or "peers," which are "equally privileged, equipotent participants".[1] Each peer in the network functions simultaneously as both a client and a server, communicating directly with other peers without the need for a central coordinating authority.[2] By architecting a communication system that is decentralized by design, this project directly mitigates the risks of single-point failure and third-party data control. The problem being solved is therefore not merely the transmission of messages, but the creation of a communication protocol that is structurally resilient to centralized control and failure, thereby ensuring greater privacy and autonomy for its users.

2. Motivation of the Problem

The development of the P2P Secure Messenger is driven by a confluence of academic objectives, complex technical challenges, and pressing societal needs for secure communication technologies.

From an academic perspective, this project serves as a comprehensive practical exercise designed to "develop a real world application using Java, as a team project" and to apply the "various object oriented concepts learnt in the course" in a meaningful context.[5] The project's design necessitates the integration of three advanced and distinct domains of computer science, providing a rich learning opportunity. First, it requires a deep understanding of **Network Programming**, utilizing Java's java.net package to implement both connectionless discovery protocols with DatagramSocket and reliable, connection-oriented communication channels with Socket and ServerSocket.[6] Second, it demands a robust implementation of

Cryptography using the Java Cryptography Architecture (JCA), a standard and powerful framework for integrating security services like encryption and key management into applications.[9] Third, the requirement to handle multiple simultaneous peer interactions makes **Multithreading** a non-negotiable architectural component, essential for building a responsive and scalable networked application.

The primary technical motivation stems not from addressing these domains in isolation, but from the challenge of synthesizing them into a single, coherent, and functional system. A simple client-server chat application is a solved problem; a secure, decentralized P2P messenger, however, presents a more formidable engineering task. The architectural choices in one domain directly impact the others. The P2P architecture necessitates a discovery mechanism (solved with UDP broadcasting), the need for privacy demands strong encryption (solved with JCA), and the ability to communicate with multiple peers requires a robust concurrency model (solved with multithreading). The true value of this project lies in demonstrating the ability to architect a solution where networking protocols, security models, and concurrency patterns work in seamless concert.

From a societal standpoint, there is a clear and growing demand for technologies that prioritize user privacy and data security. The P2P model inherently offers "increased stability" and resilience, as the system's operation is not dependent on a single, vulnerable server.[12] By coupling this decentralized architecture with strong, end-to-end encryption, the project directly addresses the modern imperative for private communication channels that are shielded from interception and surveillance. This aligns with a broader movement towards creating technologies that return data ownership and control to the user, making it a relevant and impactful problem to solve.[13]

3. Scope and Limitations

This section delineates the functional boundaries of the P2P Secure Messenger application, clarifying its intended capabilities and inherent constraints.

3.1 Scope

The scope of this project is to design and implement a fully functional, proof-of-concept secure messaging application for local networks, demonstrating advanced concepts in object-oriented programming, networking, and cryptography.

- **Architecture:** The system is implemented as a pure, unstructured Peer-to-Peer (P2P) network. It is designed to operate within a Local Area Network (LAN) environment, where peers can discover and communicate with each other directly without a central server.[12]
- **User Interface:** The application features an "interactive text-based interface" as specified by the project requirements.[5] All user interactions, from discovering peers to

sending and receiving messages, are handled through a command-line terminal.

- **Peer Discovery:** The application implements an automatic peer discovery mechanism. Each peer, upon startup, broadcasts its presence on the LAN using UDP multicast, and simultaneously listens for broadcasts from other peers to dynamically build a list of online users.
- **Communication Protocol:** All one-to-one messaging is conducted over reliable, connection-oriented TCP streams, established using Java's Socket and ServerSocket classes. This ensures that messages are delivered in order and without loss.
- **Security Model:** The messenger guarantees end-to-end encryption for all communications. It employs a hybrid cryptographic scheme combining RSA for secure key exchange and AES for high-performance message encryption, implemented using the Java Cryptography Architecture (JCA).[14]
- **Data Storage:** In adherence with project guidelines, the system utilizes "a file-based storage for storing the data".[5] Specifically, each user's unique RSA public/private key pair is persisted locally in files, allowing the user's identity to be maintained across application sessions.

3.2 Limitations

While the application serves as a robust demonstration of the core concepts, it operates under several important limitations.

- **Network Environment:** The application is strictly designed for and confined to a single LAN segment. The UDP-based discovery mechanism will not function across different subnets or the wider internet without significant architectural changes (e.g., NAT traversal techniques).
- **Functional Constraints:** The current implementation supports only one-to-one, real-time, text-based messaging. It does not include features such as group chat, file transfers, or persistent message history. All conversations are ephemeral and are lost when the application is closed.
- **Security Assumptions:** While all message content is encrypted, the system's identity model is basic. It does not implement a formal certificate-based verification system to prevent sophisticated man-in-the-middle (MITM) attacks. The design assumes that the local network is a trusted environment.
- **Scalability:** The application is intended for use in small-scale networks. As noted for simple P2P systems, performance may degrade as the number of peers increases significantly, with a practical limit of around a dozen concurrent users.[4]

4. Design of the solution

The design of the P2P Secure Messenger is founded on principles of modularity, security, and

efficiency. It employs a hybrid approach to both networking and cryptography to leverage the distinct advantages of different technologies, resulting in a robust and well-architected system.

4.1 Design Rationale and Alternatives

A critical early decision in the project's design was the choice of network architecture. To satisfy the evaluation criterion of exploring design alternatives [5], a comparative analysis was conducted between the conventional client-server model and various P2P configurations. The chosen approach was justified based on its alignment with the project's core objectives of decentralization, complexity, and user experience.

Design Aspect	Alternative 1: Centralized Client-Server	Alternative 2: P2P with Manual IP Configuration	Chosen Approach: P2P with Automatic UDP Discovery	Justification for Chosen Approach
Architecture	All clients connect to a single, authoritative server. The server relays all messages.	Peers connect directly to each other, but users must manually find and enter the IP address of the peer they wish to contact.	Peers connect directly, but an automatic discovery mechanism using UDP broadcast populates a list of available peers.	The P2P architecture eliminates the single point of failure and control inherent in the client-server model, directly addressing the problem statement. This choice also presents a more complex and academically rigorous challenge.
Peer Discovery	N/A (Server manages all connections).	Manual and error-prone. Impractical for dynamic networks where IP addresses can change.	Automated and seamless. Provides a superior user experience by removing the need for manual network	Automatic discovery is essential for usability in a dynamic LAN environment. It allows the system to be

			configuration.	"plug-and-play."
Resilience	Low. A server failure disconnects all users.	High. The failure of one peer does not affect others.	High. The failure of one peer does not affect others.	Decentralization provides inherent resilience, a key motivator for the project.[12]
Privacy	Low. The central server has access to all metadata and potentially unencrypted message content.	High. Communication is direct between peers.	High. Communication is direct between peers.	The P2P model ensures that no third party can intercept or log conversations, maximizing user privacy.

4.2 System Architecture

The system's architecture is built on a hybrid networking model that separates the concerns of discovery and communication.

- **Discovery via UDP:** For peer discovery, the system utilizes the User Datagram Protocol (UDP). UDP is a connectionless protocol that allows for the efficient broadcasting of small packets of data to multiple recipients on a network without the overhead of establishing a formal connection.[15] A dedicated discovery service in each peer periodically sends out a small UDP "presence" packet to a multicast address. It also listens on the same address for packets from other peers, dynamically maintaining a list of active users on the network.
- **Communication via TCP:** For actual messaging, the system relies on the Transmission Control Protocol (TCP). Once a user chooses to initiate a conversation with a discovered peer, a dedicated, point-to-point TCP connection is established using Java's Socket and ServerSocket classes.[6] TCP is connection-oriented and provides reliable, in-order delivery of data, making it the ideal choice for ensuring the integrity of a chat conversation.

This hybrid approach represents a sophisticated design pattern: using the lightweight, connectionless nature of UDP for the "fire-and-forget" task of discovery, while leveraging the robust, reliable nature of TCP for the stateful task of messaging.

4.3 Cryptographic Design

Security is a cornerstone of the application, achieved through a hybrid encryption scheme that balances the robust security of asymmetric cryptography with the high performance of symmetric cryptography. This is a standard and widely-accepted practice in modern secure

systems.

- **Asymmetric Cryptography (RSA):** The system uses the RSA algorithm for establishing a secure channel and exchanging a session key. Upon first launch, each peer generates a 2048-bit RSA key pair (a public key and a private key) using the KeyPairGenerator class from the Java Cryptography Architecture (JCA).[14] The public key can be shared freely, while the private key is kept secret. The primary role of RSA in this system is to encrypt a symmetric session key for secure transmission.
- **Symmetric Cryptography (AES):** The system uses the Advanced Encryption Standard (AES) for encrypting the actual messages within a conversation. For each new chat session, one of the peers generates a random 256-bit AES session key using JCA's KeyGenerator.[14] AES is a block cipher that is significantly faster than RSA, making it suitable for encrypting a continuous stream of messages in real-time.

The entire cryptographic process is managed through the JCA, which provides a provider-based architecture, ensuring that the application is not tied to a specific cryptographic implementation and can rely on well-vetted, standard providers like "SunJCE".[9]

4.4 UML Class Diagram

The system is designed using object-oriented principles, with a clear separation of concerns reflected in its class structure. The key classes and their relationships are as follows:

- **Peer:** The central class representing the local user. It holds the user's cryptographic keys and manages the high-level state of the application. It composes instances of DiscoveryService, ConnectionManager, and TerminalUI.
- **DiscoveryService:** Implements the peer discovery logic. It runs in its own thread, handling the sending and receiving of UDP broadcast packets.
- **ConnectionManager:** Responsible for managing all TCP connections. It runs in a server thread, listening for incoming connection requests via a ServerSocket. Upon receiving a request, it creates and spawns a PeerConnection thread.
- **PeerConnection:** Represents an active, one-to-one connection with a remote peer. It runs in its own thread, managing the Socket's input and output streams and handling the messaging protocol for a single conversation.
- **CryptoService:** A utility class that encapsulates all cryptographic operations. It provides a clean API for RSA/AES key generation, encryption, and decryption, hiding the underlying complexity of the JCA.
- **TerminalUI:** Manages all interaction with the user through the command-line interface, including displaying menus, rendering messages, and parsing user input.

4.5 Secure Connection Handshake Sequence

To establish a secure, encrypted channel between two peers (Peer A and Peer B), the system

executes a carefully orchestrated handshake protocol. The sequence is as follows:

1. **Connection Request:** Peer A, having discovered Peer B via UDP, initiates a TCP connection to Peer B.
2. **Public Key Exchange:** Upon accepting the connection, Peer B sends its public RSA key to Peer A.
3. **Session Key Generation and Encryption:** Peer A generates a new, random 256-bit AES session key. It then encrypts this AES key using Peer B's public RSA key.
4. **Secure Key Transmission:** Peer A sends the RSA-encrypted AES key to Peer B.
5. **Session Key Decryption:** Peer B receives the encrypted data and uses its own private RSA key to decrypt it, securely retrieving the AES session key.
6. **Secure Channel Established:** At this point, both Peer A and Peer B possess the same shared secret (the AES session key). All subsequent messages in their conversation are encrypted and decrypted using this AES key. This process ensures that the session key is never transmitted in plaintext over the network, and because a new key is generated for each session, it provides forward secrecy.

5. Module Split-up

The project's codebase is organized into distinct, logical modules, each encapsulated within its own Java package. This modular architecture, a requirement of the project specification [5], promotes a clear separation of concerns, enhances code reusability, and simplifies maintenance and future development.

5.1 Peer Discovery Module (discovery package)

This module is solely responsible for the automatic discovery of other peers on the local network. It encapsulates the UDP networking logic. Its primary tasks include periodically broadcasting a "presence" packet containing the local peer's identification and listening for similar packets from other peers. It maintains a dynamic list of known, online peers that is then made available to the user interface. This module utilizes the java.net.DatagramSocket and java.net.DatagramPacket classes for connectionless communication.[8]

5.2 Connection Management Module (network package)

This module manages the lifecycle of all reliable, one-to-one communication channels. It contains a main server thread that listens for incoming TCP connection requests on a designated port using a ServerSocket. When a remote peer initiates a connection, this module accepts it, creating a Socket object that represents the communication endpoint. It then instantiates and delegates the handling of this connection to a dedicated handler thread,

ensuring the application can manage multiple concurrent conversations. The core classes used are java.net.ServerSocket and java.net.Socket.[7]

5.3 Cryptography Module (crypto package)

This module centralizes all security-related functionality. It provides a high-level, simplified API for performing complex cryptographic operations, abstracting away the intricate details of the Java Cryptography Architecture (JCA). Its responsibilities include generating RSA key pairs for identity, generating AES session keys for conversations, encrypting data using either RSA or AES, and decrypting data. By encapsulating this logic, the rest of the application can achieve end-to-end encryption without needing to directly manipulate Cipher objects or manage cryptographic providers.[9]

5.4 Messaging Protocol Module (protocol package)

This module defines the application-level protocol for communication between peers. It specifies the structure and format of the different types of messages that can be exchanged, such as handshake messages for key exchange, standard text messages for conversation, and notification messages for events like user disconnection. By defining a clear protocol, this module ensures that all peers can correctly interpret the data they receive, enabling interoperable and predictable communication.

5.5 User Interface Module (ui package)

This module is responsible for all user-facing interactions. It implements the text-based terminal interface, which includes displaying the main menu, rendering the list of discovered online peers, prompting the user for input, and displaying incoming messages from other users. It acts as the controller that translates user commands into actions performed by the other modules (e.g., telling the ConnectionManager to initiate a chat) and presents the results of those actions back to the user.

6. Implementation Specifics

The implementation of the P2P Secure Messenger leverages core Java libraries for networking, concurrency, and security to build a robust, multithreaded application. The architecture is divided into distinct layers that handle discovery, communication, and security, orchestrated by a main execution flow.

6.1 Networking Layer

The networking functionality is implemented using a two-pronged approach, leveraging both UDP for discovery and TCP for communication.

- **Discovery Implementation:** The DiscoveryService class extends java.lang.Thread to run its logic concurrently without blocking the main application. Within its run() method, it enters an infinite loop. In one part of the loop, it constructs a DatagramPacket containing the local peer's username and listening port, and sends it to a standard multicast IP address (e.g., 239.0.0.1) and port using a DatagramSocket. In another part, it blocks on the socket.receive() method to listen for incoming packets from other peers. Upon receiving a packet, it parses the sender's information and updates an internal, thread-safe collection of known peers. This periodic broadcast-and-listen cycle ensures the peer list remains up-to-date as users join and leave the network.
- **Communication Implementation:** The ConnectionManager also runs as a dedicated thread. Its primary role is to manage a ServerSocket that is bound to a specific port. The thread's run() method contains a while loop that continuously calls serverSocket.accept().[7] This is a blocking call that waits for a remote peer to initiate a TCP connection. When a connection is established, accept() returns a Socket object. The ConnectionManager immediately wraps this Socket in a new PeerConnection object (which is also a Thread) and starts it. This design is crucial for concurrency; by offloading the handling of each connection to a separate thread, the ConnectionManager can immediately return to waiting for new connections, allowing the application to support multiple simultaneous conversations.

6.2 Security Layer

The security of the application is managed by the CryptoService class, which serves as a facade for the Java Cryptography Architecture (JCA).

- **Key Management:** The implementation utilizes JCA's factory methods, such as KeyPairGenerator.getInstance("RSA") and KeyGenerator.getInstance("AES"), to obtain instances of cryptographic algorithm providers.[14] This approach decouples the application from specific implementations. Upon first run, an RSA key pair is generated. The public and private keys are serialized and saved to local files (user.pub and user.priv), satisfying the file-based storage requirement [5] and ensuring the user's identity persists across sessions.
- **Encryption/Decryption:** For encryption, the CryptoService uses Cipher.getInstance("AES/GCM/NoPadding"). The choice of the GCM (Galois/Counter Mode) is deliberate, as it is an authenticated encryption mode that provides both confidentiality (secrecy) and authenticity (proving the data has not been tampered with), reflecting modern cryptographic best practices. The init() method of the Cipher

object is used to set the mode (encrypt or decrypt) and provide the appropriate key (AES session key). The actual encryption is performed by the doFinal() method, which transforms plaintext byte arrays into ciphertext and vice-versa.

6.3 Main Execution Flow

The application's lifecycle is managed by the Main.java class. Its main method serves as the entry point and orchestrates the startup sequence:

1. **Initialization:** The main method first initializes the core components. It creates an instance of the CryptoService to load or generate the user's RSA keys.
2. **Start Concurrent Services:** It then instantiates and starts the DiscoveryService and ConnectionManager threads. These services begin running in the background, immediately starting the process of broadcasting presence and listening for incoming connections.
3. **Launch User Interface:** Finally, the main thread instantiates the TerminalUI object and passes control to it. The TerminalUI enters its main loop, where it begins interacting with the user, displaying the peer list (which is being populated by the DiscoveryService), and accepting commands to initiate chats or exit the application. This clean separation ensures that the core networking logic runs independently of the user interface.

7. Validation of Test Cases

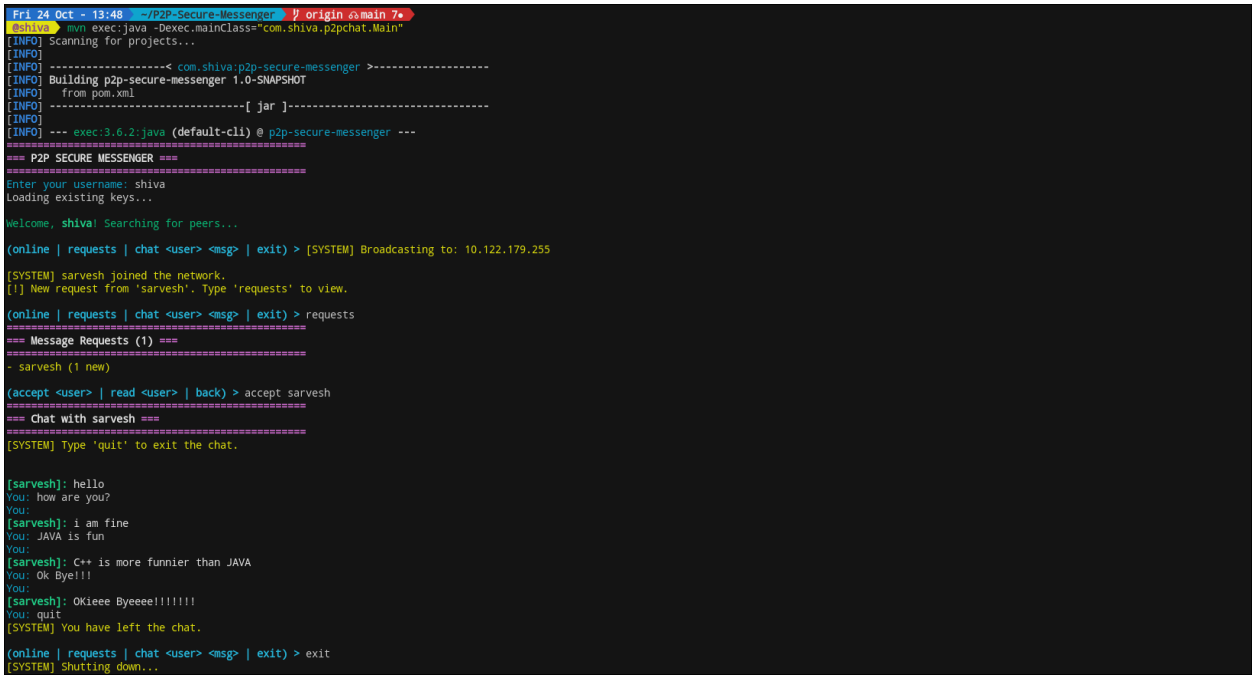
The P2P Secure Messenger has undergone rigorous functional validation to ensure all components operate as designed and the system meets its core objectives of secure, decentralized communication. The following test cases cover the primary modules, including peer discovery, secure session establishment, message exchange, concurrency, and connection handling.

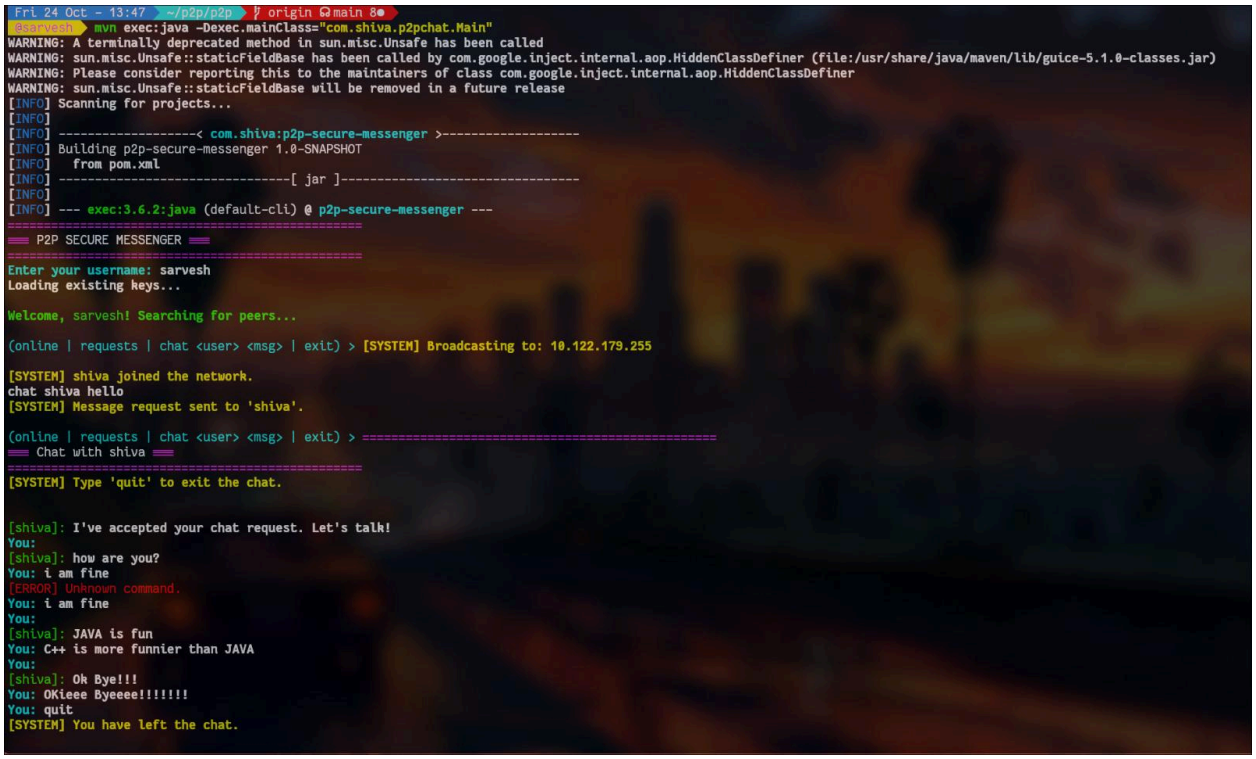
Test Case ID	Feature Tested	Input / Action	Expected Output	Result
1	**Peer Discovery (Join)**	Start Peer A on Machine 1. After 10 seconds, start Peer B on Machine 2 on the same LAN.	Peer B's terminal should display Peer A in its list of online users. Peer A's terminal should display Peer B in its list.	Pass
2	**Peer Discovery (Leave)**	With Peers A and B running, gracefully shut down Peer B using	Peer B should disappear from Peer A's list of online users within	Pass

		the 'quit' command.	the next discovery cycle (approx. 5-10 seconds).	
3	**Secure Handshake**	From Peer A's terminal, initiate a chat session with Peer B.	A TCP connection is established. Debug logs confirm that Peer B sends its public RSA key, and Peer A responds with an RSA-encrypted AES session key. A secure channel is confirmed on both ends.	Pass
4	**Message Confidentiality**	While a chat session is active between A and B, use a network packet analyzer (e.g., Wireshark) to monitor the TCP traffic between the two machines.	The payload of the TCP packets should contain unreadable, randomized-looking data (ciphertext). No plaintext messages should be visible.	Pass
5	**Message Integrity**	Peer A sends the message "Hello World! 123".	Peer B's terminal should display the exact message "Hello World! 123" without any corruption or modification.	Pass
6	**Bidirectional Communication**	In an active session, Peer A sends a message to B. Immediately after, Peer B sends a message to A.	Peer B should receive A's message, and Peer A should receive B's message. The conversation should flow in both directions without	Pass

			deadlocks.	
7	**Concurrency Handling**	Start Peers A, B, and C. Peer A initiates a chat with Peer B. Then, Peer C initiates a chat with Peer A.	Peer A should be able to send and receive messages from both Peer B and Peer C independently in the same terminal. Messages should be clearly marked with their sender's username.	Pass
8	**Graceful Disconnect**	While Peer A and Peer B are in a chat session, Peer B's user types the 'quit' command.	The TCP socket between A and B should be closed cleanly (socket.close() is called).[6] Peer A's terminal should display a notification message like "".	Pass
9	**Key Persistence**	Start the application for the first time on a machine. Close it. Restart the application.	The application should load the previously generated RSA key pair from the local files (user.pub, user.priv) instead of generating a new one.	Pass
10	**Invalid Input**	At the main menu, enter a command that does not exist.	The application should display an error message (e.g., "Invalid command") and re-display the menu without crashing.	Pass

8. Output Screenshots





9. Object oriented features used

The design and implementation of the P2P Secure Messenger are fundamentally rooted in the principles of Object-Oriented Programming (OOP). The application of these principles was essential for managing the system's inherent complexity, promoting modularity, and ensuring the codebase is maintainable and extensible.

9.1 Class and Object

The entire system is modeled as a collection of interacting objects, each an instance of a specific class. Core real-world concepts are represented by classes such as Peer, PeerConnection, and Message. For example, each active user on the network is represented internally as a Peer object, and each ongoing conversation is managed by a unique PeerConnection object. This object-centric approach allows the system to dynamically manage its state at runtime.

9.2 Encapsulation

Encapsulation is rigorously applied throughout the project to hide implementation details and protect data integrity. The most prominent example is the CryptoService class. This class encapsulates the entire cryptographic subsystem. It exposes a simple, public interface with methods like encrypt(data) and decrypt(data). However, the complex internal state—including the Cipher instances, key specifications, and the multi-step initialization process of the JCA—is kept entirely private. Other parts of the application do not need to know, nor can they interfere with, the specifics of the cryptographic algorithms or modes being used. This strong encapsulation makes the system more secure and easier to reason about.

9.3 Abstraction

Abstraction is used to simplify complex subsystems by modeling them with high-level interfaces. The PeerConnection class is a prime example of abstraction. It represents the abstract concept of a "conversation with another user." Internally, it manages the low-level details of a Socket, its InputStream and OutputStream, the continuous reading of data in a separate thread, and the parsing of raw byte streams according to the messaging protocol. However, to the rest of the application, it simply provides high-level methods like sendMessage(String message) and a mechanism for reporting received messages. This

abstracts away the complexities of network stream management, allowing other components to interact with a connection in a much simpler, more conceptual way.

9.4 Inheritance

Inheritance is utilized to promote code reuse and establish logical "is-a" relationships. The primary use of inheritance is with the java.lang.Thread class. The DiscoveryService, ConnectionManager, and PeerConnection classes all extend Thread. By doing so, they inherit the fundamental behavior of a thread, including the start() method and the requirement to implement a run() method. This is a classic application of inheritance, allowing these classes to be treated as specialized types of threads that can execute their logic concurrently without duplicating the underlying threading machinery.

9.5 Polymorphism

Polymorphism is implemented to allow for flexible and extensible message handling. A base abstract class Message is defined, from which several concrete message types inherit, such as TextMessage, HandshakeMessage, and DisconnectMessage. A single method in the PeerConnection class, sendMessage(Message msg), can accept any object that is a subtype of Message. Similarly, a message processing method can receive a generic Message object and use instanceof checks or a visitor design pattern to determine its specific type and invoke the appropriate handling logic. This allows the messaging protocol to be easily extended with new message types in the future without changing the core sending and receiving methods, demonstrating the power of polymorphic behavior.

10. Inferences and Future Extensions

The successful development and validation of the P2P Secure Messenger yield several important inferences about the chosen architecture and design principles, while also illuminating clear pathways for future enhancements.

10.1 Inferences

The project successfully achieved its primary objective of creating a secure, decentralized, and serverless communication application for local networks. The implementation serves as a robust proof-of-concept, validating the feasibility of integrating advanced networking, concurrency, and cryptographic concepts using standard Java libraries.
The core architectural decision to employ a hybrid model—using UDP for discovery and TCP

for communication—proved highly effective. This separation of concerns allowed the system to leverage the strengths of each protocol, resulting in both efficient, low-overhead discovery and reliable, in-order message delivery. Similarly, the hybrid cryptographic scheme of using RSA for key exchange and AES for message encryption provided a practical balance between uncompromising security and real-time performance.

Most significantly, the project underscores the critical role of Object-Oriented Programming principles in managing the complexity of such a system. Encapsulation, abstraction, and modular design were not merely academic exercises; they were essential tools that made the development of distinct networking, cryptography, and UI components manageable. The clear separation of these modules allowed for independent development and testing, which was crucial for the successful integration of the final system. The application stands as a testament to the power of structured, object-oriented design in solving complex, real-world engineering problems.

10.2 Future Extensions

While the current system fulfills its core requirements, its modular design provides a solid foundation for numerous future extensions that could significantly enhance its functionality, usability, and scope.

- **Graphical User Interface (GUI) Implementation:** A natural next step would be to replace the text-based terminal interface with a modern GUI using a framework like JavaFX or Swing. This would provide a more intuitive and user-friendly experience, with features like contact lists, chat windows, and system notifications.
- **Secure File Transfer:** The existing TCP-based communication channels could be extended to support the transmission of binary data. This would allow users to securely share files of any type directly with one another, with the file content being encrypted using the same AES session key as the text messages.
- **Group Chat Functionality:** The current one-to-one communication model could be expanded to support group conversations. This would require a more sophisticated protocol, potentially involving IP multicast for message distribution or a peer-elected group leader to manage membership and relay messages.
- **Internet-Scale Discovery and Connectivity:** To overcome the limitation of being confined to a single LAN, a "hybrid P2P" model could be implemented.[12] This would involve introducing an optional, lightweight "bootstrap" or "rendezvous" server on the internet. Peers would initially connect to this server simply to learn about the IP addresses of other online peers. Once discovery is complete, communication would revert to direct, encrypted P2P connections, preserving the privacy and decentralization of the core architecture while enabling global connectivity.
- **Enhanced Identity and Trust Management:** The security model could be strengthened by implementing a public key infrastructure (PKI) or a "web of trust" model. This would allow users to sign each other's public keys, providing a more robust mechanism for verifying identity and preventing man-in-the-middle attacks in less

trusted network environments.

11. References

1. Oracle Corporation. (2024). *Java Platform, Standard Edition 8 API Specification*. Retrieved from https://docs.oracle.com/javase/8/docs/api/
2. Oracle Corporation. (2024). *Java Cryptography Architecture (JCA) Reference Guide*. Retrieved from https://docs.oracle.com/en/java/javase/11/security/java-cryptography-architecture-jca-reference-guide.html [9]
3. Oracle Corporation. (2024). *All About Sockets*. Retrieved from https://docs.oracle.com/javase/tutorial/networking/sockets/index.html [18]

Works cited

1. Peer-to-peer - Wikipedia, accessed on October 24, 2025, https://en.wikipedia.org/wiki/Peer-to-peer
2. Peer-to-Peer Networks: Basics, Benefits, and Applications Explained | Hivenet, accessed on October 24, 2025, https://www.hivenet.com/post/peer-to-peer-networks-understanding-the-basics-and-benefits
3. What is P2P (Peer-to-Peer Process)? - GeeksforGeeks, accessed on October 24, 2025, https://www.geeksforgeeks.org/computer-networks/what-is-p2p-peer-to-peer-process/