# Peer to peer Network / Onion Routing

Programming III Project Report

Hristijan Chupetreski
89211097@student.upr.si
UP FAMNIT
Koper, Slovenia

dr. Aleksandar Tošić
aleksandar.tosic@upr.si
UP FAMNIT
Koper, Slovenia

Domen Vake
domen.vake@famnit.upr.si
UP FAMNIT
Koper, Slovenia

## ABSTRACT

Onion routing, better known as 'TOR network', is a technique used to achieve anonymous communication over a network, and this project focuses on creating a secure, scalable and resilient system by leveraging containerization, hybrid cryptography, and a multi-tiered client architecture. All pointing towards the creation of a successfully implemented **P2P network** with onion routing. The core of the system is the message format, which employs a hybrid encryption scheme using **AES** and **RSA** to ensure message confidentiality and unlinkability. The infrastructure, built with Docker and a custom bridge network, facilitates the creation of a distributed and isolated environment. The report outlines the messaging protocol, the server-side architecture based on socket activation and a synchronized peer handler executor, and the three distinct client types. Testing results demonstrate the system's effectiveness in securely transmitting messages while preventing traffic analysis, confirming its potential as a foundation for a robust anonymous communication service.

## KEYWORDS

Mix Network, Anonymity, Onion Routing, Hybrid Encryption, Docker, AES, RSA, Socket Activation, Threads, Privacy, GSON

## 1 INTRODUCTION

Traditional communication channels, while often encrypted, are susceptible to traffic analysis, where a third party can infer information about the communication partners and their behavior even if the message content is hidden. A mix network addresses this vulnerability by sending a message through a bigger number of travel points ( mix nodes ) all having playing a role in the initial multi-layered encryption of the message and further on decryption. This report documents the implementation of a functional prototype of a mix network designed to be secure, extensible, and easy to deploy, providing a practical demonstration of its privacy-enhancing capabilities.

## 2 METHODOLOGY

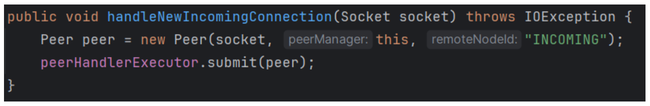
The project's methodology is centered on a modular and secure design, addressing all layers from infrastructure to application logic.

### 2.1 Containerization and Network Infrastructure

To ensure an isolated and reproducible environment for each component, Docker was chosen as the containerization platform. The network infrastructure was designed using a **Docker bridge network**. This approach provided a private, internal network that isolated the mix network's components (clients and mix nodes) from the host machine's network. This bridge network facilitated seamless, secure communication between containers without exposing internal ports to the public internet, a critical step for simulating a private network environment. Each service (e.g., a mix node) was run in its own container, allowing for independent scaling and management.

### 2.2 Server-Side Architecture

Each mix node was designed for high concurrency and resource efficiency. Instead of a continuously running server, **socket activation** was employed. This mechanism allows the operating system to listen for incoming connections and only start the server process when a connection is actually received. This "lazy loading" of server resources minimizes the idle footprint of each mix node. Once a mix node is activated, it utilizes a synchronized high class designed peer handler executor that runs with the command `.newCachedThreadPool` to handle new connections. This design prevents blocking, as each incoming request is assigned to an available thread, allowing the mix node to process multiple messages simultaneously without delay.



```
public void handleNewIncomingConnection(Socket socket) throws IOException {
    Peer peer = new Peer(socket, peerManager: this, remoteNodeId: "INCOMING");
    peerHandlerExecutor.submit(peer);
}
```

**Figure 1: Method for handling peer and container activation**

### 2.3 Client Architecture

The system was designed to support three distinct types of clients to manage the complexities of message flow:

- **Sender:** This client initiates communication by constructing the onion message. It is responsible for selecting a path through the mix network and performing the initial encryption process.
- **Recipient:** This client is the final destination. Its primary function is to receive the final message from the last mix node in the chain, unwrap the last layer of encryption, and retrieve the original plaintext message.
- **Path-flow/Peer handler:** The main purpose of this plays a logical part, in which after the user input of who the sender and receiver will be, we also have the path the message will follow to get to the receiver. Despite this, it holds the knowledge of the active mix nodes and has access to their public keys. This trusted centralized component ensures that clients can dynamically discover and communicate with the mix nodes.

## 2.4 Message Construction: From Plaintext to Onion Message

The journey of a message begins with a simple plaintext format. Before sending, this message must be transformed into an **Onion Message** to ensure anonymity.

```
public static Message buildOnionMessage(byte[]
originalMessageBytes, List<String> fullPath,
Map<String, PublicKey> publicKeyMap)
```

This is a multi-layered process, constructed from the inside out. The **GSON** library is used to serialize the message payload into JSON format before encryption. For a path consisting of three mix nodes (Mix 1, Mix 2, Mix 3) and a final recipient, the process is as follows:

(1) **Original Message & Final Instructions**
The sender composes the plaintext message $M$, then creates the innermost layer containing the recipient's address and the encrypted payload. This layer is encrypted using a new symmetric key (AESR_key), which is itself encrypted with the recipient's public key (RSAR_pub).

**Layer 1 (Innermost):**
- Payload: $E_{\text{AESR\_key}}(M + \text{Recipient's Address})$
- Key: $E_{\text{RSAR\_pub}}(\text{AESR\_key})$

(2) **Layer for Mix 3**
The entire Layer 1 is now encrypted again with a new symmetric key (AES3_key). That key is encrypted with Mix 3's public key (RSA3_pub), along with forwarding instructions.

**Layer 2:**
- Payload: $E_{\text{AES3\_key}}(\text{Layer 1})$
- Key: $E_{\text{RSA3\_pub}}(\text{AES3\_key} + \text{Next Hop: Recipient})$

(3) **Layer for Mix 2**
Repeat the wrapping process with AES2_key and Mix 2's public key.

**Layer 3:**
- Payload: $E_{\text{AES2\_key}}(\text{Layer 2})$
- Key: $E_{\text{RSA2\_pub}}(\text{AES2\_key} + \text{Next Hop: Mix 3})$

(4) **Layer for Mix 1 (Outermost)**
The final wrapping uses AES1_key and Mix 1's public key, preparing the full onion message for transmission.

**Layer 4 (Outermost):**
- Payload: $E_{\text{AES1\_key}}(\text{Layer 3})$
- Key: $E_{\text{RSA1\_pub}}(\text{AES1\_key} + \text{Next Hop: Mix 2})$

The result is a fully wrapped "onion" message where only the outermost layer is visible to the first mix node. Each node peels off a layer, forwards the decrypted content to the next node, and never sees the original plaintext or full path.

We first send the message as a constructed object using the following code:

```
Message onionMessage = ClientMessageBuilder
    .buildOnionMessage(
        content.getBytes("UTF-8"),
        fullPath,
        publicKeysFromPathNodes
);
```

At the final destination, the message is received and reconstructed like this:

```
Message finalMessage = new Message(fullPath,
encodedPayload, encodedIv, encodedSymmetricKey);
Logger.log("Client: Built onion message"
+ finalMessage, LogLevel.Info);
return finalMessage;
```

This final object contains all necessary information to begin the decryption process at the destination. The mix network ensures that no intermediate node can access or trace the original message or its intended recipient.

## 2.5 Hybrid Encryption Scheme

A **hybrid encryption scheme** was implemented to balance both security and performance, as neither pure AES nor pure RSA is ideal on their own for this application. At the core of this approach lies a **centralized encryption scheme** for key management. All keys are initially generated and stored in a shared file.

This is achieved by creating a dedicated key generator service, managed through Docker, which produces the keys and makes them accessible to all containers on the network before any services are launched.

```
key-generator:
  build: .
  container_name: mixnet-key-generator
  hostname: key-generator
  networks:
    - my-mixnet-dev-net
  command: java -jar app.jar GenerateKeys
  volumes:
    - ./keys:/app/keys
```

These keys are then distributed to the Directory Authority, which stores and serves the public keys for all mix nodes and clients. This centralized key distribution process is essential to support the onion encryption system, as senders must have access to the correct public keys for every node in the routing path.

**Encryption Breakdown:**

- **Symmetric Encryption (AES):** The Advanced Encryption Standard (AES) is used to encrypt the actual message content. AES is highly efficient and well-suited for encrypting large payloads. A unique, randomly generated AES key is created for each message.
- **Asymmetric Encryption (RSA):** RSA is used to encrypt the AES key itself. Each AES key is encrypted using the public key of the intended recipient (or mix node). Since RSA is computationally expensive, it's only used for small data like keys — optimizing performance without compromising security.

**Why Not Use Just One?**

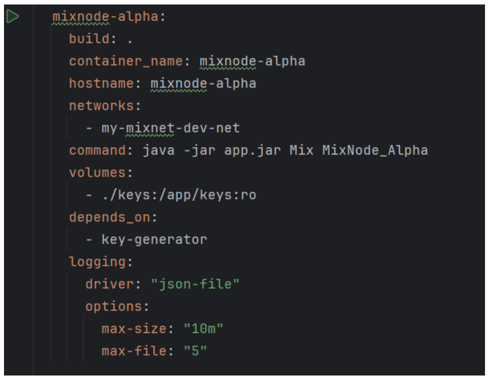
- **Pure RSA:** Encrypting entire messages with RSA is impractical. RSA is designed for small inputs and is significantly slower than AES. Using it to encrypt full payloads would drastically degrade performance and overwhelm system resources.
- **Pure AES:** AES requires a shared secret between both parties. In a privacy-focused network with no pre-established trust relationships, this is infeasible. Clients don't have pre-shared keys with every possible mix node, making pure symmetric encryption unscalable and insecure in this context.

The hybrid approach combines the strengths of both algorithms — AES for fast, secure message encryption, and RSA for safe key exchange. This results in a system that is both practical and resilient in a dynamic, decentralized network.

## 2.6 Message Flow and Decryption

The onion message, structured as a JSON package, is sent to the first mix node (Mix 1) in the path. Each mix node performs a similar, two-step decryption process before forwarding the message. Using the example from Section 2.4, the message flow is as follows:

(1) **Mix 1:** Receives the outermost layer. It uses its private RSA key to decrypt the AES key and forwarding instructions. It then uses this AES key to decrypt the payload. The decrypted content is `Layer 3`, containing instructions for Mix 2. Mix 1 forwards this new, smaller onion message to Mix 2.
(2) **Mix 2:** Receives the message from Mix 1. It performs the same process, using its private RSA key to decrypt the AES key and forwarding instructions. It then decrypts the payload to reveal `Layer 2` and forwards it to Mix 3.
(3) **Mix 3:** Receives the message from Mix 2. It again uses its private RSA key to decrypt its layer, revealing `Layer 1` and the recipient's address. It then forwards this final encrypted payload to the recipient client.



```
mixnode-alpha:
  build: .
  container_name: mixnode-alpha
  hostname: mixnode-alpha
  networks:
    - my-mixnet-dev-net
  command: java -jar app.jar Mix MixNode_Alpha
  volumes:
    - ./keys:/app/keys:ro
  depends_on:
    - key-generator
  logging:
    driver: "json-file"
    options:
      max-size: "10m"
      max-file: "5"
```

**Figure 2: Example of a mix-node being generated in docker-compose.yml**

(4) **Recipient:** Receives the final payload from Mix 3. It uses its private RSA key to decrypt the AES key, then uses this key to decrypt the final message and its original content.

This "peeling" process ensures that each mix node only knows its immediate predecessor and successor, and only the final recipient can access the plaintext message.

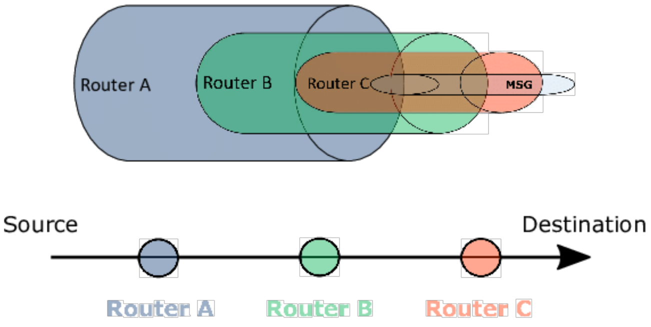
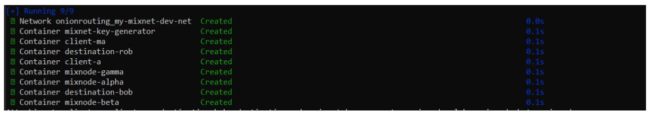


**Figure 3: The journey of the message package**

## 3 TESTING

The testing phase focused on validating the functionality, security, and performance of the system.

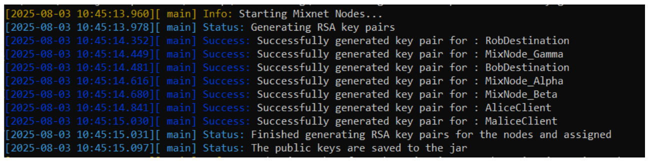


```
[+] Running 9/9
 ▯ Network onionrouting_my-mixnet-dev-net  Created                                        0.0s
 ▯ Container mixnet-key-generator          Created                                        0.1s
 ▯ Container client-ma                     Created                                        0.1s
 ▯ Container destination-rob               Created                                        0.1s
 ▯ Container client-a                      Created                                        0.1s
 ▯ Container mixnode-gamma                 Created                                        0.1s
 ▯ Container mixnode-alpha                 Created                                        0.1s
 ▯ Container destination-bob               Created                                        0.1s
 ▯ Container mixnode-beta                  Created                                        0.1s
```

**Figure 4: Docker Activation of Containers**

[H]

- **Unit Tests:** Individual tests were conducted for the encryption and decryption modules to ensure the hybrid AES/RSA scheme worked correctly.
- **Key Generator:** Before starting and activating the containers we first run the *Key Generator* in order to create the file with the keys for the nodes the client will use in order the encrypt the message.
*docker-compose run key-generator*



```
[2025-08-03 10:45:13.960][ main] Info: Starting Mixnet Nodes...
[2025-08-03 10:45:13.978][ main] Status: Generating RSA key pairs
[2025-08-03 10:45:14.352][ main] Success: Successfully generated key pair for : RobDestination
[2025-08-03 10:45:14.449][ main] Success: Successfully generated key pair for : MixNode_Gamma
[2025-08-03 10:45:14.481][ main] Success: Successfully generated key pair for : BobDestination
[2025-08-03 10:45:14.616][ main] Success: Successfully generated key pair for : MixNode_Alpha
[2025-08-03 10:45:14.680][ main] Success: Successfully generated key pair for : MixNode_Beta
[2025-08-03 10:45:14.841][ main] Success: Successfully generated key pair for : AliceClient
[2025-08-03 10:45:15.030][ main] Success: Successfully generated key pair for : MaliceClient
[2025-08-03 10:45:15.031][ main] Status: Finished generating RSA key pairs for the nodes and assigned
[2025-08-03 10:45:15.097][ main] Status: The public keys are saved to the jar
```

**Figure 5: Key Generator Running**

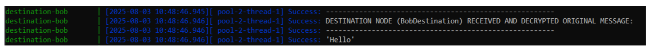
[H]

- **CMD Usage:** The crucial testing was done by activating a CMD Prompt for each and individual node. By accessing the containers logs on the mix nodes and destination nodes.
*docker logs -f mixnode-alpha / docker logs -f destination-bob*
And also by attaching on the client (sender) nodes in order to send out a message with the help of the PrintWriter we have implemented in them and flushing the user's input.
*docker attach client-a*
- **Integration Tests:** A crucial part of the testing involved an end-to-end simulation of a message being sent through a

specific, pre-defined path. For example, a test was conducted with a sender, a recipient, and a path of three mix nodes (Mix 1, Mix 2, Mix 3). The test confirmed that:

(1) The sender successfully constructed the multi-layered onion message.
(2) The message was routed from Mix 1 to Mix 2, then to Mix 3, and finally to the recipient.
(3) Each mix node correctly decrypted its layer and forwarded the message.
(4) The final recipient successfully unwrapped the last layer and recovered the original, uncorrupted plaintext message.

- **Security Analysis:** The system was tested to confirm that an eavesdropper listening on any segment of the network could not link the sender and recipient, a fundamental requirement of a mix network. This was validated by inspecting network traffic at each hop.
- **Performance Benchmarking:** The system's throughput and latency were measured under different loads to assess its scalability and the efficiency of the socket activation and thread pool design.



```
destination-bob       | [2025-08-03 10:48:46.945][ pool-2-thread-1] Success: ----------------------------------------------------
destination-bob       | [2025-08-03 10:48:46.946][ pool-2-thread-1] Success: DESTINATION NODE (BobDestination) RECEIVED AND DECRYPTED ORIGINAL MESSAGE:
destination-bob       | [2025-08-03 10:48:46.946][ pool-2-thread-1] Success: ----------------------------------------------------
destination-bob       | [2025-08-03 10:48:46.946][ pool-2-thread-1] Success: 'Hello'
```

**Figure 6: Destination Node Logs**

## 4 RESULTS

The implementation successfully demonstrated the core principles of a mix network. Messages were reliably delivered, and the hybrid encryption scheme proved effective. The Docker-based infrastructure provided a flexible and isolated testing environment, allowing for rapid deployment and modification of the network topology. The socket activation mechanism significantly reduced the resource footprint of idle mix nodes.

## 5 DISCUSSION

The implementation of this prototype validates the theoretical concepts behind mix networks. The modular design, particularly the use of Docker, makes the system highly scalable and easily adaptable for different network sizes. The hybrid encryption scheme offers a robust security model that is both computationally efficient and difficult to break.. Future work could explore a decentralized, peer-to-peer approach for node discovery to eliminate this single point of trust. Additionally, further security analysis would be needed to guard against more sophisticated attacks, such as replay attacks or active manipulation of message headers by a malicious mix node.

## 6 CONCLUSION

The project successfully developed a functional prototype of a privacy-preserving mix network. By implementing a layered architecture with a secure hybrid encryption scheme, the system effectively protects communication metadata and ensures the anonymity of its users. The use of modern technologies like Docker and socket activation with multi-layered security of both AES and RSA provides a robust, reliable and scalable foundation. This project serves as a strong proof-of-concept, laying the groundwork for more advanced and resilient anonymous communication systems.