

# Mix Network

Jaka Ahlin

UP FAMNIT

Slovenia

89221140@student.upr.si

## ABSTRACT

In this paper, I present a Peer-to-peer network using Onion routing technique to provide anonymous communication between nodes. This Mix Network was designed using Docker to connect peers under different IP addresses. Entry Point is serving as a joining point for peers to be able to join the network and receive the needed information about other connected peers. Using asymmetric cryptography every message is being encrypted multiple times like a Russian doll and decrypted at each intermediate node at its path to the destination node.

## KEYWORDS

Encryption; peer; entry point; onion routing; peer-to-peer; protocol; routing; anonymity

## 1 INTRODUCTION

Project requires to build a working implementation of a peer-to-peer network with basic functionality of a mix network. Since the networking part of the project is inherently concurrent, and at the same time the system is distributed - a working implementation covers all three parts.

## 2 PROBLEM DESCRIPTION

Since each peers main functionality (in addition to have the ability of sending messages) in the network is to serve other peers as an intermediate node, we have to design a system that is able to handle and accept multiple connections concurrently to avoid unneeded delay when message gets anonymously routed through the peers in the path.

Asymmetric or Public-key cryptography is used to encrypt the important message information as the routing information and the message body itself. Each peer has its own private key to be able decrypting received messages and route the message forward if needed. Private keys have to be safely stored and should be kept a secret. In contrast to private keys, public keys are distributed between the peers and used in the encryption process.

If message is at the beginning encrypted  $X$  times it means that message will be passed between  $X-1$  intermediate nodes before reaching receiver - its final destination. An important note to give is that each node (excluding sender and receiver for obvious reasons) should only know node before them (from who they received message) and after them (to who they will route the message forward). This is a key to keep the receiver and sender anonymous and safe from various attacks.

## 3 IMPLEMENTATION

Whole project was implemented using only Java programming language in IntelliJ IDEA integrated development environment.

Docker was used to deploy application in containers. Version control was done using Git and can be seen on the following GitHub link: <https://github.com/ahlinj/mixnetwork>

### 3.1 Package overview

Java provides us with plenty of prebuilt packages to ease and fasten the production of our projects. In my project I used the following subpackages of java package:

- `io`: For exceptions and input/output streams.
- `net`: For sockets and server sockets.
- `security`: For asymmetric encryption.
- `util`: For scanner and hash maps.
- `time`: For capturing and managing time.

### 3.2 Project overview

Source files are organized in three folders - Users, EntryPoint and Common. In the root directory of this project there are also different docker files, PowerShell and bash scripts. Here is a quick overview:

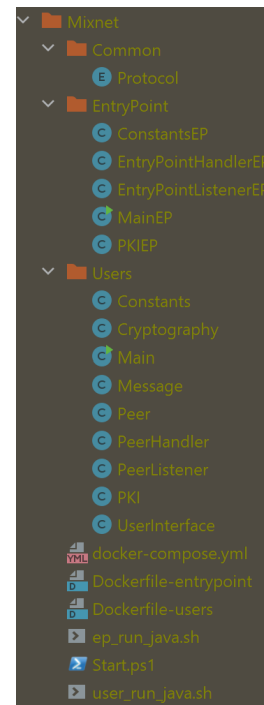


Figure 1: Project structure

#### 3.2.1 Common.

- `Protocol`: Serializable enum class listing all protocols.

### 3.2.2 Users.

- Main: Starting class.
- UserInterface: Provides interface for users.
- Peer: Manages peers operation.
- PeerListener: Listens for incoming connections from other peers.
- PeerHandler: Handles each individual connection.
- Cryptography: Encryption/decryption.
- PKI: Storing hash maps.
- Message: Message structure.
- Constants: Projects constants.

### 3.2.3 EntryPoint.

- MainEP: Starting EP class.
- EntryPointListenerEP: Listens for incoming connections from clients at the entry point.
- EntryPointHandlerEP: Handles each individual connection at the entry point.
- PKIEP: Storing hash maps.
- ConstantsEP: Projects constants.

### 3.2.4 Other files.

- Start.ps1: Starting powershell script.
- user\_run\_java.sh: Users bash script.
- ep\_run\_java.sh: EntryPoints bash script.
- Dockerfile-users: Users dockerfile.
- Dockerfile-entrypoint: EntryPoints dockerfile.
- docker-compose.yml: Docker compose file.

## 3.3 Set-up and interface overview

Program was developed on a Windows 11 computer thus the starting script Start.ps1 is a PowerShell script and should be used to start the program. An important note is that the project structure should not be changed and also the script should be run from the "out" folder where compiled classes should be and not the source Java files. Make sure that before running the script, Docker desktop is also running.

Start.ps1 is responsible to build Docker images and creating containers based on a docker.compose.yml file. Script starts containers in a detached mode. For each Container we start a new PowerShell window where we open an interactive bash shell and run either the ep\_run\_java.sh or user\_run\_java.sh scripts. These two scripts are responsible to start the program for all users and the entry point. Everything up till this point should be automatic.

Once the scripts are finished running. You should be asked to enter your username for all users. After selecting username you will be greeted by the following message;

What do you want to do?

- 1: Send message
- 2: Show users in the network
- 3: Leave network and close the program

You should select one option by typing the number of your choice. Options are self explanatory.

## 4 DOCKER

As listed under Project overview there are 6 scripts in total that are responsible to build the network. I will explain what each file does

in the order in which they get executed. Process will be explained for creating a network with an entry point and three users.

### 4.1 Start.ps1

This is a starting powershell script that initializes the process.

```
docker-compose build
docker-compose up -d
$containers = @("entry-point-1", "user1-1", "user2-1", "user3-1")
foreach ($container in $containers) {
    if ($container -eq "entry-point-1") {
        Start-Process powershell -ArgumentList "docker exec -it mixnet-$container bash -c './ep_run_java.sh; exec bash'"
    } else {
        Start-Process powershell -ArgumentList "docker exec -it mixnet-$container bash -c './user_run_java.sh; exec bash'"
    }
}
```

*docker-compose build* and *docker-compose up -d* are responsible to build the Docker images and create containers based on the docker-compose.yml file, which we will look in more detail in the next paragraph.

*Start-Process powershell* will create a new PowerShell window for each Container (EntryPoint or User). We are adding *-ArgumentList* so we can attach an argument, which will be executed as a command when the process starts.

*docker exec -it mixnet-\$container bash -c './ep\_run\_java.sh; exec bash'* is the Docker command that we are passing as an argument. *Mixnet-\$container* is the name of the container. For each container we will run a new Bash shell with either *./ep\_run\_java.sh* as the command or *./user\_run\_java.sh* depending on if the container is EntryPoint or User. Exec bash keeps the terminal open.

### 4.2 docker-compose.yml

We will look at the docker-compose.yml file now that is responsible to build the network.

```
networks:
  mix-network:
    driver: bridge
    ipam:
      config:
        - subnet: 172.18.0.0/16
```

This can be found at the very end of the file and defines the network configuration. We use the default driver - bridge. We also have to define the subnet for our network to define the range.

Now let's look at how entry-point and users are defined.

```
entry-point:
  build:
    context: .
    dockerfile: Dockerfile-entrypoint
  volumes:
    - ./EntryPoint:/app/EntryPoint
    - ./Common:/app/Common
networks:
  mix-network:
    ipv4_address: 172.18.0.2
```

```
stdin_open: true
tty: true
```

We build the image based on dockerfiles that are different for EntryPoint and Users. We will look at them in more detail in the next paragraph. Dockerfiles should be located in the current directory (context: .). Volumes section is responsible to mount needed folders in out containers. We also have to define to which network service are we connecting and with which IP address.

```
user1:
  depends_on:
    - entry-point
  build:
    context: .
    dockerfile: Dockerfile-users
  volumes:
    - ./Users:/app/Users
    - ./Common:/app/Common
  networks:
    mix-network:
      ipv4_address: 172.18.0.3
  stdin_open: true
  tty: true
```

For users it is not much different. Main change is that we added section that states that we are depended on the entry-point, which means that first EntryPoint will get initialized and only after that users. We also have to change which folders we are mounting and of course our IP address and the Dockerfile.

### 4.3 Dockerfiles

In my project there are two Dockerfiles - Dockerfile-entrypoint and Dockerfile-users.

```
FROM openjdk:24-jdk-slim
WORKDIR /app
COPY /EntryPoint /app/EntryPoint
COPY /Common /app/Common
COPY ep_run_java.sh /app/ep_run_java.sh
RUN chmod +x /app/ep_run_java.sh
CMD ["bash"]
```

Dockerfiles are responsible to set the working directory and copy the directories in the container. We also make script executable. Dockerfile-users is similar to the entry point one, difference being only in what files we copy and which script we make executable.

### 4.4 Bash scripts

Entry Point and Users have each own simple two line script. They are only responsible to run the program.

```
#!/bin/bash
java -cp . EntryPoint.MainEP
```

Entry Point doesn't need any additional arguments.

```
#!/bin/bash
java -cp . Users.Main 172.18.0.2
```

Users need to put IP address of the EntryPoint as the program argument (172.18.0.2 in my example).







Name	Image	Status	CPU...
 <a href="#">mixnet</a>		Running (11/1)	1.06%
 <a href="#">user10-1</a> b12d7ece780a 	<a href="#">mixnet-user10</a>	Running	0.11%
 <a href="#">user5-1</a> fcdabc271452f 	<a href="#">mixnet-user5</a>	Running	0.09%
 <a href="#">user4-1</a> 13dae6ecd171 	<a href="#">mixnet-user4</a>	Running	0.1%
 <a href="#">user6-1</a> 53916725b19f 	<a href="#">mixnet-user6</a>	Running	0.1%
 <a href="#">user2-1</a> 800a24903716 	<a href="#">mixnet-user2</a>	Running	0.08%
 <a href="#">user7-1</a> 71886fceedf7 	<a href="#">mixnet-user7</a>	Running	0.11%
 <a href="#">user8-1</a> aa8a774fe8d0 	<a href="#">mixnet-user8</a>	Running	0.1%
 <a href="#">user1-1</a> 2ec591a0d17c 	<a href="#">mixnet-user1</a>	Running	0.1%
 <a href="#">user9-1</a> 51125a4f164e 	<a href="#">mixnet-user9</a>	Running	0.09%
 <a href="#">user3-1</a> b9599a7b6990 	<a href="#">mixnet-user3</a>	Running	0.09%
 <a href="#">entry-point-1</a> fbd4e5182bcd 	<a href="#">mixnet-entry-po</a>	Running	0.09%

Figure 2: Containers running in Docker desktop

## 5 TECHNICALITIES

### 5.1 Protocol

Protocol in my implementation of a mix network was simply defined and in fact Protocol.java is the only file that both the entry point and users share, in contrast to other files where there are differences for entry point and users respectively.

In Protocol class there are defined CONNECT, UPDATE and REMOVE enums. When users are connecting in the network they use CONNECT to exchange important information, like Public Keys and IP addresses. In similar fashion UPDATE works, but here users only receive public keys and IP addresses and not send them since they have already shared them at the start. REMOVE is used to inform entry point that we are disconnecting from the network and thus our information should be removed from the maps.

Users also have to serve other peers as intermediate nodes. But since their only job is to decrypt and route the message forward, there was no need to create new enum values like MESSAGE or FORWARD, but rather we directly start this process of decryption and routing.

## 5.2 Padding

To ensure that there is minimal meta data leakage, after each message there are `''` added at the end. This way all messages will be of the same length. Current message length is set to 256 characters, but can be easily changed in the Constants class.

### 5.3 Safety checks

There are plenty of safety checks throughout the program. For example when it displays users that are connected in the network, it will not show yourself and entry point as expected. Even if you later tried to enter one of those as a receiver or you enter username that doesn't even exist, it wouldn't let you do it and you would be returned to the starting screen.

Another example is that at the starting screen you are only able to choose one of the three options, otherwise if you decide to enter anything else, it wouldn't allow you.

## 5.4 Routing

Similarly to how we encrypt message in a Russian doll style, we do that with routing information, but it is a little more complex. At each step of the encryption process we have to add to the encrypted part also the IP address of the next peer in the path. Similarly we will have to remove these in the decryption process. Its important to note that in the decryption process when a user spots '-1' instead of an IP address it means that they are the last peer in the path or in other words that they are the receiver.

## 5.5 Onion message

To understand how the onion message is constructed we first need to look at the Message class. There are four important parts: body, sender, timestamp and route. Body is the actual message that we are sending, which sender enters through the interface. Timestamp gets automatically added using `System.currentTimeMillis()` and is later transformed into a readable format on the receivers end. Sender also gets automatically set by using our username that we have set up at the begging. Starting route information is at the beginning set up to -1. Padding is added as explained in the Padding paragraph.

```
Before encryption:
Body: ExampleMessage
Sender: user4
Route: -1
```

**Figure 3: Example message after adding padding and before encryption**

First layer of encryption is always made using receivers public key.

[illegible]

**Figure 4: Example message after the first encryption using receivers public key**

After encryption, routing information is attached as explained in the Routing paragraph.

```
Added route info: 172.18.0.6::sjfRRc3Ux+MfxlAOqGUbrg==
```

**Figure 5: Route after adding receivers IP address at the beginning**

With this the first step of the encryption process is finished. Now a random user from the network will be selected as an intermediate node and their public key for encryption.

Random peer in the network: user9

**Figure 6: Selecting a random peer from the network**

Now we repeat the encryption process using the already once encrypted messages from before.

[illegible]

**Figure 7: Example message after the second encryption**

We add routing information of our intermediate node (in our example IP address of user9).

```
Added route info: 172.18.0.4::yCyFabAvUPvae3ZXe+Hq3FhtQx1e9eewDAERmrH1H80rpBZYzVvKkOVI,
ePvQvm
```

**Figure 8: Route after adding IP address at the begging**

Now this process is repeated until we get the final onion message that is then routed throughout the network and decrypted by each intermediate node and finally received by the receiver.

## 5.6 Cryptography

In my Cryptography class I have used a hybrid approach, which uses both symmetric and asymmetric encryption algorithms. For the asymmetric encryption algorithm I have used RSA (Rivest-Shamir-Adleman) and for symmetric I used AES (Advanced Encryption Standard). This approach was necessary, since using only asymmetric encryption would result in a very inefficient system. I encountered the problem when I was using only RSA, that I wouldn't be able to encrypt large messages due to the restriction of a key size. Introducing AES solved this problem. Now I am able to first encrypt the message using AES secret key, which doesn't have any message size limitations and it's relatively fast. Then I only encrypt the secret AES key using the RSA encryption and attach the RSA-encrypted secret AES key to the message for the person on the other side to be able to decrypt it. This is doable, since secret AES key is always of the same size - 256, which you are able to encrypt using RSA of a key size 2048. AES key is regenerated every time when function is called.

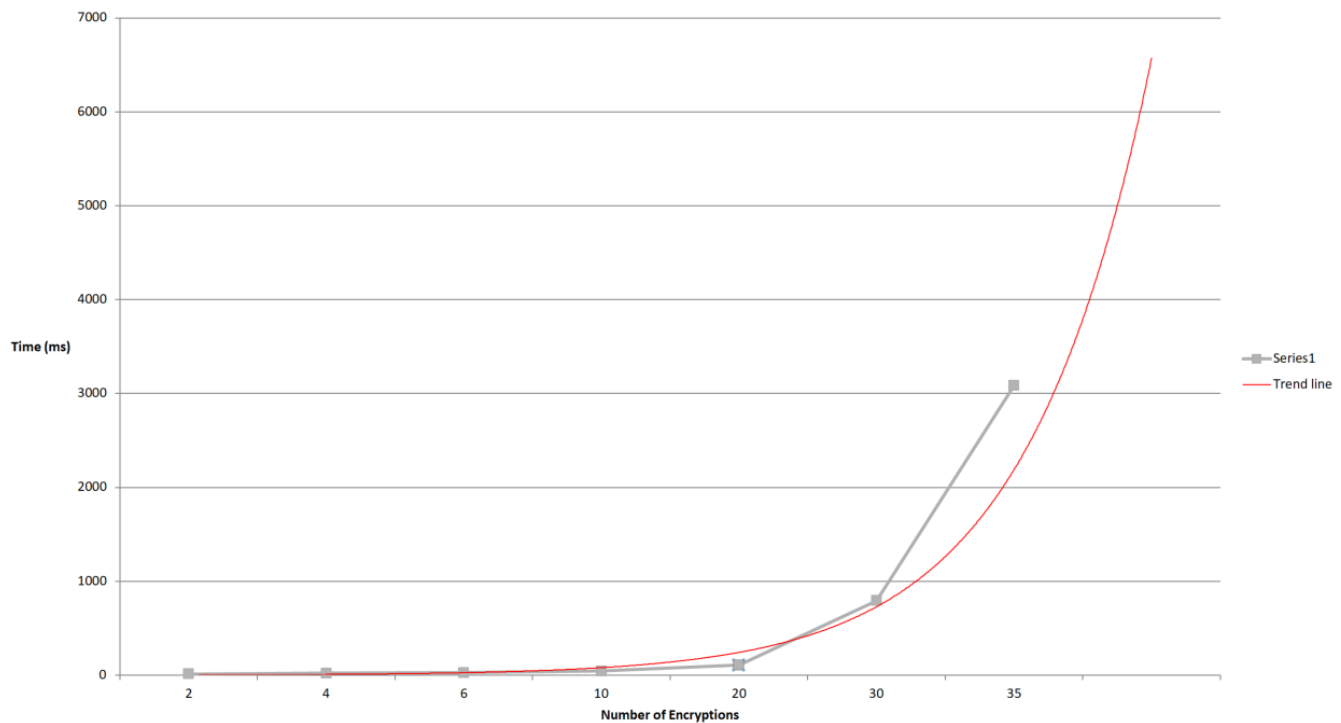


Figure 9: Time (ms) based on number of encryptions

## 6 PERFORMANCE ANALYSIS

To test the performance of our system we have to measure the time it takes for the message to be encrypted and routed through the network. In the following tests timer was started before the first encryption was done and stopped the moment receiver received the message. To ensure fairness, tests were done on a same computer at the same conditions. As it can be seen from the Figure 2 graph 35 encryptions, 35 decryptions and routing between nodes took around three seconds at average. When testing with 40 encryptions program crashed. It can be clearly seen that time is exponential in relation to the number of encryptions.

The sweet spot seems to be somewhere between ten and twenty encryptions. For ten encryptions we needed on average 45 milliseconds, while for twenty it took around 108 milliseconds. If the number of users in the network is large enough so that using ten encryptions would be logical, we should do that.

## 7 LIMITATIONS

Like every program, this one also has its limitations. Statistical disclosure attacks are a big threat. By analyzing and metadata leaks you would maybe be able to deanonymize peers. So called timing attacks could be used to observe when does a message enter a mix network and when does it leave it. Adding a delay to when the message is sent could improve this issue, but would add latency to when the whole process.

There should be a trust system implemented that would check if the profiles actually belong to the person behind them. This way

we could limit fake identities that someone could use to control a part of the network in the sybil attack.

## 8 CONCLUSION

While these peer-to-peer networks may not be known to an average internet user, they are most definitely a big game changer for more niche computer geeks. Main challenge of this project was to create a concurrent system that would allow users to send and receive multiple messages at once, which was possible by extending Thread classes. This allowed for each received message to be processed by its own thread, rather than sequentially.

Big challenge was also to create a clearly defined protocol, that allows us to handle each peers request. User interface was kept minimal, that way also an average user will be able to use the program.

Most important thing about mix networks is anonymity. I think that using the encryption process and message routing, that I described in the paragraphs above I was able to create a system that would provide a secure messaging platform while also not adding any unnecessary latency. Only some of the more complex attacks will show limitations of the program.