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

Asymmetric encryption; peer; entry point; onion routing; peer-to-peer; protocol; routing; anonymous

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 park 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 though 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 begging 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

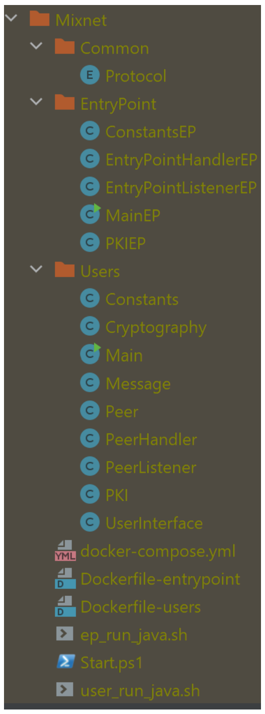
3.1 Package overview

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:



```
Mixnet
  Common
    Protocol
  EntryPoint
    ConstantsEP
    EntryPointHandlerEP
    EntryPointListenerEP
    MainEP
    PKIEP
  Users
    Constants
    Cryptography
    Main
    Message
    Peer
    PeerHandler
    PeerListener
    PKI
    UserInterface
  docker-compose.yml
  Dockerfile-entrypoint
  Dockerfile-users
  ep_run_java.sh
  Start.ps1
  user_run_java.sh
```

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 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 users 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 server 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 TECHNICALITIES

5.1 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.

5.2 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.

5.3 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.

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 thought the network. In the following tests timer 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

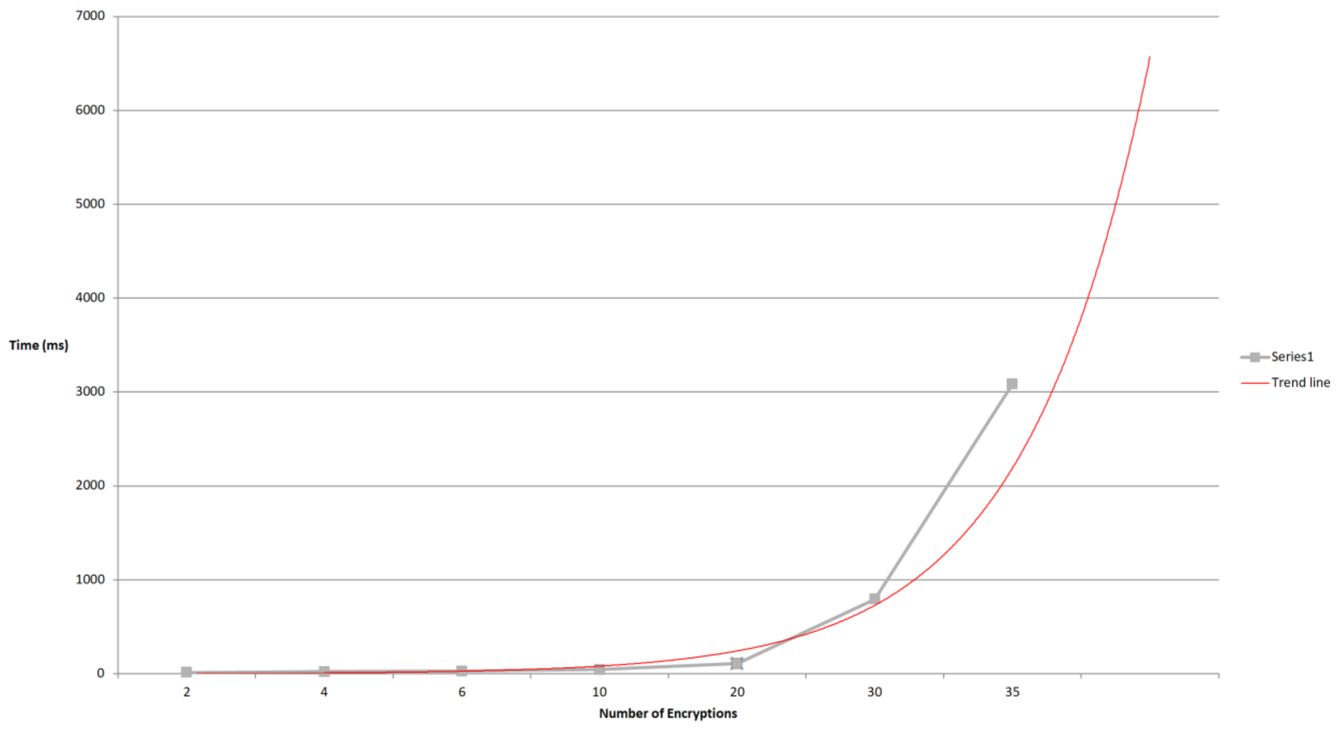


Figure 2: Time (ms) based on number of encryptions

program crashed. It can be clearly seen that time is exponential in relation to the number of encryptions.

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

7 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.