CS 354 Project 3: Network Address Translation

Group 41:

G. Mcgaffin (23565608@sun.ac.za)

J. P. Visser (21553416@sun.ac.za)

2 September 2022

Contents

1 Introduction

For this project we were tasked with implementing a simulated router with minimal NAT functionality, as well as a simple client.

The router processes paquets received from clients, who are either marked as internal (hosts in the local network) or external (hosts belonging to an external network).

Clients send simple paquets to one another, and the router has to process these paquets in an appropriate manner, depending on the following cases:

- **Internal → Internal**: The paquet is forwarded without changing its data.
- **Internal → External**: The paquet header is modified and an entry is added to the NAT table, or refreshed if it already exists, that binds the source IP/port to the destination address.
- **External → Internal**: The paquet is routed according to the entry in the NAT table. If there is no corresponding entry in the table for the source, the paquet is dropped and an error paquet is returned (unless something like port forwarding has been implemented).
- **External → External**: Paquets are dropped, as they should be routed by external networks.

Additional features we needed to implement were:

- **NAT table**. The address translation table should refresh dynamically.
- **DHCP**. Internal clients should automatically request and be assigned an address by the NAT server.

1.1 NAT-box Requirements

The NAT-box had the following requirements:

1. The NAT-box must be given a unique hardware (MAC) address.
2. The NAT-box must be given a unique IP address.
3. The NAT-box's MAC address and IP address must be communicated to clients.
4. Multiple clients must be able to be connected to a single NAT-box.
5. Internal clients must be assigned unique IP addresses by the NAT-box by using a minimal DHCP server implementation.
6. If an internal client disconnects its IP address must be released into the pool of available IP addresses.
7. The NAT-box must be able to process and modify the packets that it receives.
8. A NAT table must be present in the implementation.

9. The NAT table must be refreshed dynamically.
10. The NAT table refresh time must be configurable, as this facilitates testing.
11. Certain status paquets must be printed out, such as when a paquet has been routed, or could not be routed.
12. Clients are only able to communicate according to the rules of the chosen NAT-box implementation.

1.2 Client Requirements

The client implementation had the following requirements:

1. Each client must either be designated as an internal or external client, as discussed above.
2. A client's IP address should reflect its status as either an internal or external client.
3. Internal clients should use a minimal DHCP implementation to request an IP address.
4. Every client must have a unique (simulated) MAC address
5. A client should have the ability to send a simple paquet to a targeted client.
6. The client must provide a reply for every paquet received. This may take the form of an ACK or a similar type of packet.
7. Clients must minimally support the following simulated protocols and paquets:
   (a) ICMP paquet forwarding (error and echo/response paquets),
   (b) TU (TCP/UDP) paquet forwarding,
   (c) a minimal DHCP implementation.

1.3 Overview

In this document we will provide a complete view of our implementation, by discussing its design (§6), giving a breakdown of the files into which it is organized (§2), and providing a high level description, with more details where necessary, of the flow of execution of the two programs which it comprises (§3).

We are confident that our implementation meets all of the requirements. However, we did not implement any features beyond those listed in the requirements either. Therefore, we do not include sections dedicated to unimplemented or additional features, as it would be superfluous. We also did not encounter any serious or unexpected issues during the development process, so this topic, too, will be excluded.

Furthermore, we will discuss experiments we have conducted (§4), significant data structures (§5), compilation and execution of the router and client (§7 and §8), as well as the libraries we made use of (§9).

2 File Descriptions

2.1 NAT-box/Router

- `NAT.java`: This file is the NAT-box's main unit. Here the server/router listens for new connections from clients and assigns threads to handle each one's incoming and outgoing traffic.
- `ClientHandler.java`: This file implements a handler that takes care of a client's input and output streams. It creates two threads, one for sending and the other for receiving paquets.

2.2 Client

- `Client.java`: This file takes input from the user, which it uses to generate paquets. The paquets generated are sent to the NAT-box. It also receives paquets, sent by other clients, from the NAT-box and displays appropriate information about them to the user.

3 Program Description

3.1 NAT-box/Router

The following is a high-level outline of the NAT-box's flow of execution, along with some more detailed descriptions of its operation.

1. Upon startup, the NAT-box processes the command-line arguments passed to it. The arguments passed to it are: the port on which it must listen for new client connections, the maximum number of threads it is allowed to use to process clients, and the amount of time in milliseconds to wait before refreshing the NAT table.
2. After the command-line arguments have been successfully processed, a listener socket is set up for clients to connect to.
3. Once a new client has connected to the listener socket, a new thread is created for handling the client. The client is also added to the client list.
4. A NAT table entry is created for the client's IP and port that it uses to listen for incoming data.
5. The new thread then starts its execution.
6. After handing over the client to the `ClientHandler`, the NAT-box then goes back to waiting for new client connections.
7. The `ClientHandler`, once generated, creates a unique IP address for the client. It then sends this new IP address, the NAT's MAC address, and client's port number, to the client.
8. The `ClientHandler` then waits to receive paquets sent by the client.
9. Once a paquet is received, it gets processed according to the type of client interaction taking place:

(a) **Internal → Internal**: The paquet is forwarded with no translation.

(b) **Internal → External**: The paquet's source address is translated to its assigned global address, and it is then forwarded to the external client.

(c) **External → Internal**: The paquet's destination address is translated to an internal address, and the paquet is forwarded to the internal client (the owner of the internal address).

(d) **External → External**: The paquet is dropped.

If an address cannot be found, an ICMP paquet carrying a designated error message is sent to the client from which the paquet containing the unknown address has been received.

10. When a client disconnects from the NAT-box, the client's socket, as well as input and ouput streams are closed; the client is removed from the client list; and its NAT table entry is discarded.

11. When the NAT-box is terminated, each `ClientHandler` closes all resources used to communicate with the client that was assigned to it.

3.2 Client

1. On startup the client creates a unique MAC address for itself.

2. It receives its assigned internal IP address, the NAT-box's MAC address, and the port number to use to use to communicate with the NAT-box from the NAT-box.

3. Two threads are started, one for sending paquets, and another for receiving paquets.

4. The receiver thread receives paquets, processes them, and displays relevant information to the user.

5. The sending thread allows the user to select a type of message to send. The user can choose to request the client list from the NAT-box, send a ping to another client, or send a message to another client.

6. Once the user has chosen an action, the appropriate paquet is generated and sent to the router.

7. When the client loses connection to the NAT-box, it informs the user by printing a message, and terminates.

8. On termination, all resources are closed.

4 Experiments

4.1 Duplicate IP address and Port Number

4.1.1 Experiment

Determine what happens when the NAT-box assigns the same IP address and port number to two different internal clients.

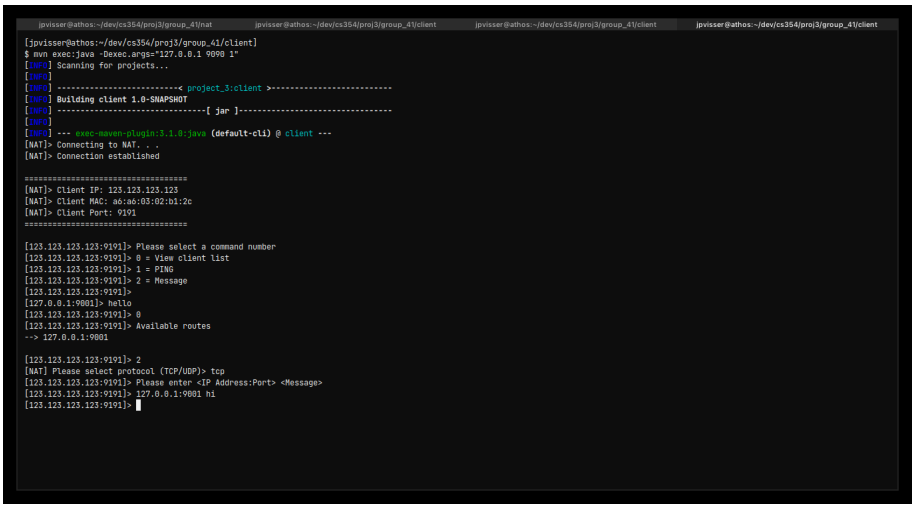

Figure 1: Message sent from the external client to the NAT-box after a connection between the internal and external clients has been established.

4.1.2 Hypothesis

Because the NAT-box uses a hash table to associate IP addresses and port numbers with internal clients, it is expected that when an IP address and port number that are already assigned to an internal client is assigned again to new internal client, the new association will simply overwrite the original one. The NAT-box and all active clients will remain operational, but there will be no way for the original client to receive communications from any of the other clients, since there will no longer be a NAT table entry associated with it.

4.1.3 Test

To test this hypothesis, introduce some small changes to the `ClientHandler`'s source code, so that it will not generate a new IP address and port for each new internal client, but will instead assign the same address and port to every internal client. Then recompile and run the NAT-box. Start three clients: two internal and one external. Then, for each client request the client list from the NAT-box, and inspect and compare the lists. Finally, try to send a message from the external client to the internal clients and see what happens.

4.1.4 Findings

Our hyphothesis was correct. The last internal client to connect to the NAT-box takes over the IP address and port number from the client that connected just before it. When sending a message to the NAT-box from the external client, only the last connected internal client receives it. See Figures 1 to 3.

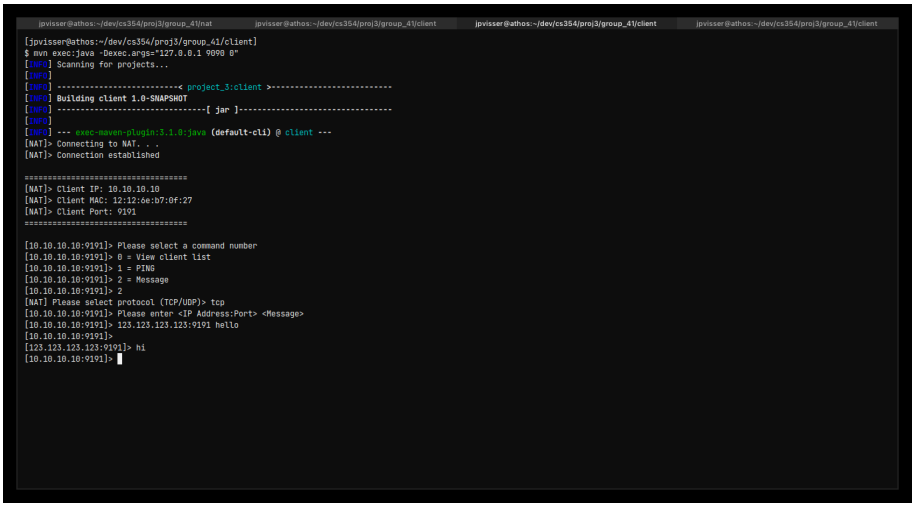

Figure 2: The last connected internal client displaying the message received from the external client.

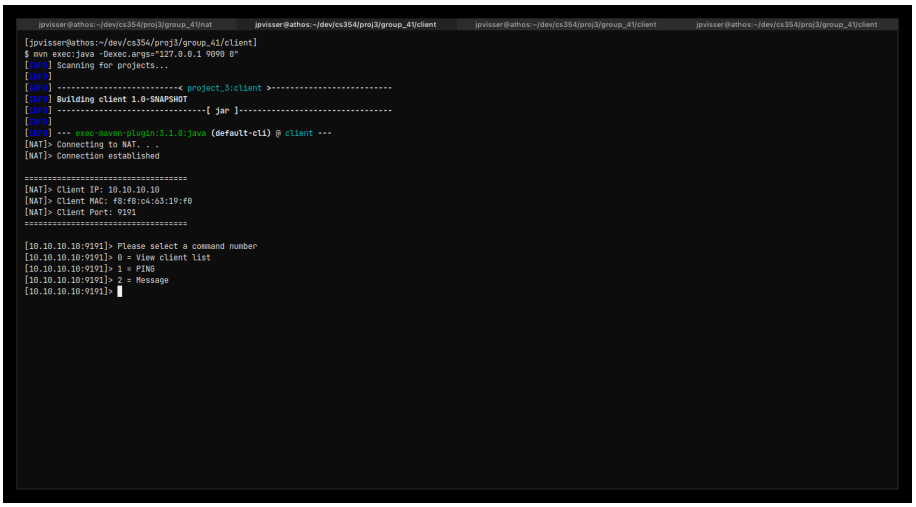

Figure 3: The client that first connected to the NAT-box shows no message, because the last connected client took over the IP address and port number from it.

4.2 Too Many Clients; Too Few Threads

4.2.1 Experiment

Determine what happens when too many clients try to connect to the NAT-box. When the NAT-box is run, one of its command-line arguments specifies the maximum number of threads to use to handle clients. When the NAT-box is instructed to use n threads, but $n+1$ clients attempt to connect, what is the result?

4.2.2 Hypothesis

It is expected that the last client to attempt to connect to the NAT-box will "hang" and do nothing until one of the other clients disconnects from the NAT-box, freeing a thread to handle the waiting client. The waiting client will then connect to the NAT-box.

4.2.3 Test

To conduct this experiment, run the NAT-box, and specify that it must use no more than 2 threads to handle clients. Then start up three clients. It does not matter which types of clients are used, but we opted to use only internal clients. One of the clients should be unable to establish a connection with the NAT-box. Then close one of the connected clients, and observe what happens with the aforementioned client that was unable to connect.

4.2.4 Findings

The test disproved our hypothesis. Although the client that could not initially connect (i.e., the third client) did "hang" (Figure 4), once one of the other two clients disconnected, it did not successfully connect to the NAT-box (Figure 6). Also out of line with our expectations, when the third client started up and tried to connect to the NAT-box, it caused the NAT-box to throw a `ArrayIndexOutOfBoundsException` (Figure 5).

5 Significant Data Structures

The only data structure of note that we used for this project is the `ConcurrentHashMap`. The `ConcurrentHashMap` is a thread safe `HashMap` implementation. It was used to implement the NAT table and the client list. These hash tables are accessed from multiple threads, so race conditions can occur. The `ConcurrentHashMap` prevents this from happening by means of synchronization.

6 Design

An interesting design decision we made, was to represent paquets using byte arrays. We took this approach, because it allows for a more realistic, "low-level" implementation of the RFC specifications of the different types of paquets. It resulted in an minimalistic implementation, that is efficient, and also quick to adapt if/when necessary.

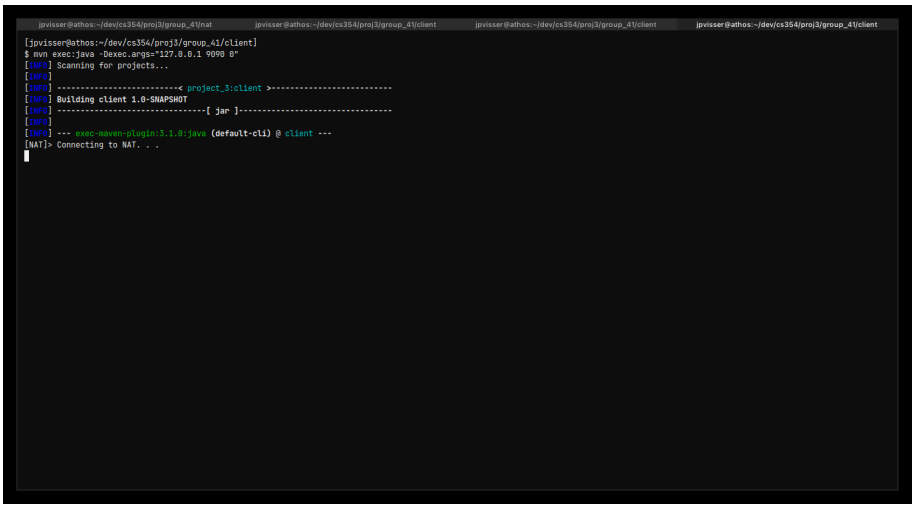

Figure 4: Third client "hanging", unable to establish a connection with the NAT-box.

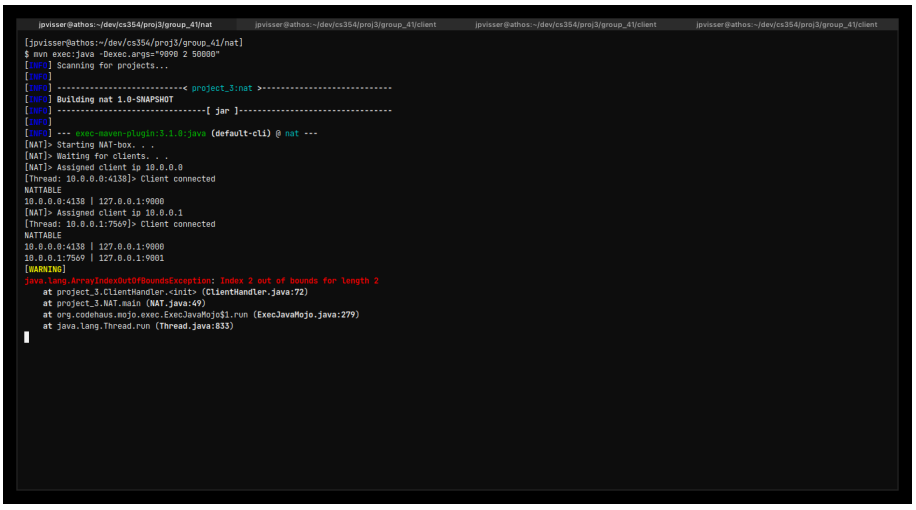

Figure 5: `ArrayIndexOutOfBoundsException` thrown by the NAT-box when the third client tries to connect to it.

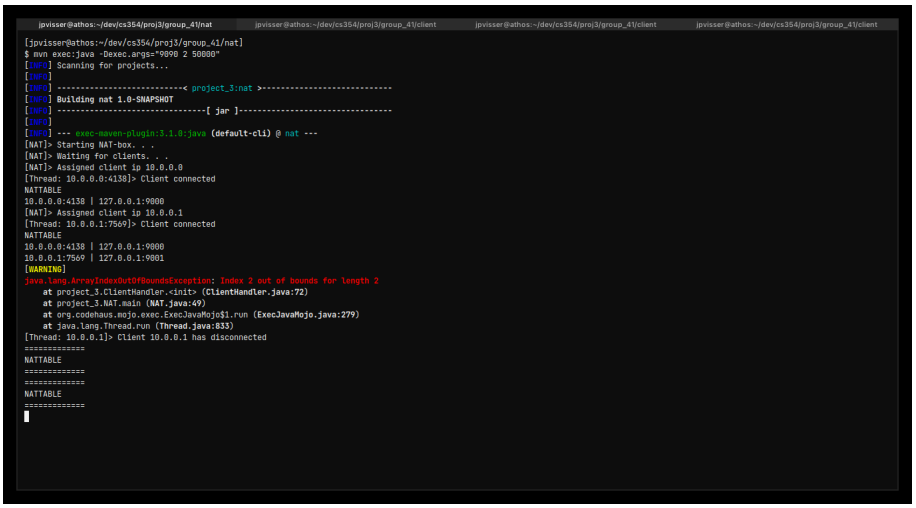

Figure 6: NAT-box output after one of the other clients disconnects.

7 Compilation

It is assumed that the project will be run on Linux from a Bash shell.

7.1 Dependencies

In order to compile and run the NAT-box and client programs, the following dependencies must be installed and available on the `PATH`:

- OpenJDK 18.0.2
- Apache Maven 3.8.6

The versions listed are what we used, but older versions should work as well.

7.2 NAT-box

1. `cd` into the `nat` directory.
2. Compile using `mvn compile`.

7.3 Client

1. `cd` into the `client` directory.
2. Compile using `mvn compile`.

8 Execution

8.1 NAT-box

Run the NAT-box using the command `mvn exec:java -Dexec.args="<port> <n> <timeout>"`, where:

- `<port>` is the port number that should be used to listen for new client connections,
- `<n>` is the maximum number threads to create to handle clients,
- `<timeout>` is the amount of time in miliseconds to wait before refreshing the NAT table.

8.2 Client

Run the client using the command `mvn exec:java -Dexec.args="<ip> <port> <type>"`, where:

- `<ip>` is the IP address of the host that the NAT-box is running on;
- `<port>` is the port number of the NAT-box;
- `<type>` is the client's type, where 0 indicates an internal client, and 1 indicates an external client.

9 Libraries

We used only the Java Standard API for this project.