Group 14 Project 3: NATBox

Dylan Callaghan: 21831599@sun.ac.za
Stephen Cochrane: 21748209@sun.ac.za

September 25, 2020

1 Introduction

The goal of this project was to create a simulated NATbox environment and implementation. The core purpose of the NATbox was to be able to translate incoming and outgoing packets, so as to maintain the appearance of a private local network. The implementation was intended to have functionality for sending multiple packet types (UDP/TCP, IMCP, DHCP), and be able to translate them accordingly. In addition to this, a minimal DHCP implementation was to be present which allowed the allocation of internal IP addresses to internal clients.

2 Unimplemented Features

We implemented all of the required features from the project spec. For the chosen NATbox technology, we have implemented traditional NAT.

3 Additional features

Apart from doing the simplistic traditional NAT implementation, for the rest of the project, we implemented a number of additional features. These include:

- Internet Server:
  The server for actually sending the messages between clients/NATboxes in our implementation mimics the realworld internet. As such, there are a number of additional features associated with it:
  - Effective Routing
    The server has an algorithm for sending packets through the simulated internet it has created, where packets only go through nodes directly on the way to the destination.
  - External client communication
    The server allows for external clients to communicate with each other (External to External) without sending messages to the NATbox

(done by the previous feature). This simulates the design of the internet.

- Multiple NATboxes
  The server allows multiple NATboxes to exist connected to it (like the real internet). Packets can be sent to and from NATboxes just like in the real internet. Essentially, this makes the NATboxes actual gateways to simulated local networks on the server, where each local network has it's own address space in the range for internal networks. A client connected to one NATbox, therefore, can send a packet to another client connected to another NATbox (provided the first client knows the mapped external IP address of the second client). The two NATboxes will translate their respective clients' addresses transparently, so that each client thinks it is sending to an external client.

- Advanced client shell:
  The client process has an advanced shell to allow for sending of many different types of packets, as well as various other commands. Any command can be run at any time, making the client side implementation more seemless. The commands available are (for more details, see the help command in the client):

  - `help`
    Provides a list of commands as well as a short description of each of them.
  - `help <command>`
    Provides a detailed description of a particular command including the values for the (optional) parameters.
  - `send <protocol> <dest ip>`
    Sends a paquet (encapsulated inside a network packet), to the given destination using the supplied protocol.
  - `whoami`
    Displays the IP information of the client. This includes it's IP address, as well as the NATBox IP address that it is connected to (for internal).
  - `iplist (<all>)`
    Displays IP addresses that are assigned on the server to some host. This command can be used to find clients to send to. Without the `<all>` option, only IP addresses on the local network (the whole "internet" for an external client), are shown. With the `<all>` option, both local and external IP addresses are shown (specific to a NATbox).
  - `rep`
    Repeats the last command

  - `ping <ip> <amount>`
    Sends `<amount>` number of ping packets to the given destination ip address.

- TCP Acks
  In addition to the required responses for all paquets routed to a client, a TCP paquet has an additional ACK packet response. This is to simulate the actual implementation of a TCP packet in the real internet, which requires ACKs to ensure reliability.

- Ping command
  The ping command from the client shell above allows for any number of ping packets to be sent. This mimics the actual functionality of the ping terminal command which continuously sends ping packets.

4 Description of files

The current classes included are,

- Server
- Client
- DHCP
- Packet wrappers (TCP, UDP, ICMP, DHCP, ACK, ARP, IP, Ethernet)
- NATtable
- NATbox
- optargs

The first item, Server, is the backbone of the simulation, and handles the routing of `packets` (containing *paquets*) to the correct socket. We decided to have a central server (as opposed to the server running on the natbox) as this allows us to have multiple natboxes (each with their own internal clients) to connect to the server and communicate with each other/ external clients.

The Client however, acts as a connection on the network, either connected to a natbox (in this case an internal client) or an external Client which simply connects directly to the network. The Client has a shell allowing for easier usage for the different functions. To see the available commands, simply invoke "help", and all the commands will be shown.

The DHCP object, when constructed, will generate IP's (first attempting to retrieve a non-used IP from a pool of free'd IP's) and give them to the requestee. On closing, the requestee can give back the used IP to the DHCP object, which

in turn adds it to the freely available IP's.

The packet wrappers are a collection of classes, that each represent a packet (sequence of bytes), and allowing the formatting of the specified packet in an easy way by just calling getters and setters, as opposed to working with the byte sequence directly. Each wrapper has three constructors, one taking the values for the packet, thus constructing a complete packet, another taking no arguments, which constructs an "empty" packet, and lastly, a constructor that takes a byte sequence and unpacks it into object form, allowing easy reading of the data.

The NATtable is an object representing the NAT table for a NAT box. It stores all the translation entries and allows thread safe retrieval and addition to the table. It contains a threaded timer to evict entries older than the specified refresh time as well.

The NATbox class handles all functionality of the NAT box implementation (excluding server functions such as sending messages). It contains a NATtable object and interfaces with this to store address translations. It also keeps track of the connected clients and assigns internal IP addresses to them. The physical sending functionality of the NATbox is deferred to the server, as our implementation supports only the server sending messages.

5 Program description

When the server is started up, it starts a loop waiting for incoming connections. When a client connects (First we describe an external client, abbrevaited as EC), the server accepts the EC's connection, and allocates an external IP and sends it back to the EC. This is done using the external mode of the DHCP server which acts like an ISP (as for this implementation, the server simulates the internet). Once this initial "handshake" is completed the server forks this connection to a slave handler thread. On the EC, once started, it receives its IP and generates its MAC address. Then it starts its interactive shell, and prints any received messages (as well as executes instructions entered into the shell). To see the commands invoke "help".

An internal client (abbreviated IC) establishes a connection with the server similar to how an EC does in that the servers accepts the IC's connection. However the setup for an internal client includes a (minimal) DHCP request and response interaction with the NATbox to retrieve its' IP address. Also, after the initial setup, the forked thread has a link to the NATbox thread, and so any message that needs to be sent outside the internal network will be sent to the NATbox to be translated and sent on. On the client side, the client MAC is generated and then the DHCP interaction is executed, after which the shell is started.

When sending a `packet`, the packet is sent to the server, where the server peeks into the packet for the destination IP, and simply passes the entire `packet` to the correct socket (in a super simplified explanation, we simply map the sockets to their IP's).

When a client starts up, and after the initial "handshake", a thread is forked which handles all incoming `packets`, and simply prints them to standerd out (print output depends on whether the verbose flag is given).

6 Experiments

Most of the expirements we performed involved sending different *paquets* to clients (with varying payloads and payload sizes). Examples of *Paquet* types sent include:

- TCP
- UDP
- ICMP and
- DHCP

The "paths"/"routes" we sent these *paquets* over include:

- internal → external
- internal → internal (clients on the same NATBox)
- external → external (Our solution allows this since we have a central server)
- internal → internal (with both internal clients being on separate natboxes, with effectively double NAT taking place)

The results of these experiments saw that for internal → internal (on the same NAT box), the *paquets* remained unchanged by the NAT box, and so the IP's remained *internal* IP's.

For internal to external, we observed the *paquets* being routed through the associated NAT box first, and the header being translated accordingly. Also, an entry was added in the NAT table, and the *paquet* was then sent to the external host. This meant that when the external host replied with an automated response, the packet was sent back to the NAT box and then translated back using the NAT table entry before sending to the internal client. In conclusion, we saw that the external host had no knowledge of the internal host being internal, abiding by the transparency of a NAT box.

For internal to internal (on different NAT boxes), we saw an interesting but indeed correct response. The *paquets* being routed were passed through each of the two internal networks NAT boxes before being sent to and from the

clients, being translated as they went. This meant that each of the two clients were unaware of the other being on a private network connected to a NAT box. This astonishing result meant that our implementation was completely scalable, allowing for multiple NAT box support, and translating *paquets* correctly for all simulated network traffic.

Something cool we discovered during the testing, was due to our ping implementation, when we ping an IP that does not exits, an ICMP error *paquet* is returned and the ping terminates, this closely mimics real life ping implementations.

Our TCP implementation also has a built in ack system. And so, when a TCP message was sent, we observed an ACK being sent back to the sender. This mimicked the real implementation of TCP on the internet. When the ACK arrived at the sender again, the sender checked against its sent TCP packets, and asserted that the TCP message had been received by the client. This resulted in an output displaying each time that the TCP message had gone through. And hence in conclusion, we can state that TCP messages as payloads of the *paquets* are successfully routed.

The TCP Ack system makes use of an "ack assert" system, in short, when a TCP message is sent, the code hash generated for that TCP message is stored in a list/queue, and when a TCP Ack is recieved, we check the list/queue to see if that code exists, if it does, we remove the code ack from the list, and assert that the ack has been recieved. If the ack code cannot be found, the packet is simply dropped.

UDP (and hence DHCP) messages were experimented with similar to the TCP messages above. Since their implementation is similar to that of TCP (except that the stringent checks are not in place), we saw all of these *paquet* types being sent successfully like TCP.

Another experiment we conducted was to send *paquets* to host addresses on the network that do not exist (are not mapped to a host). We experimented with all of TCP, UDP, and ICMP ping *paquets* for this. In all of the cases, our system responded with an ICMP error response message, saying that the destination host was unreachable. This was the expected result, as on the real world internet, trying to send any packet to an unreachable host will return this packet type. We hence concluded that our ICMP implementation, as well as our server, correctly responded to any error condition.

7 Issues encountered

A major issue that we encountered early in the project was how we were going to send messages between clients and NATboxes. We wanted our implementation to have one central communications class (the server), so that we didn't have to complicate our code by having multiple classes being able to communicate. Our solution to this was to have one general server class which supported all communication. Any NATbox on the simulated network would then be apart of this server, and have it's own private "section" of the server. This solution meant

that the NAT boxes did not have to be able to send messages themselves, but rather relied on the server to simulate the "local connection" connecting them and the clients.

One of the biggest issues we discovered later in devolpment was the problem of mapping the socket of an external client to an internal client so the external can send a `packet` to the internal client. We fixed this by having a the local IP + natbox IP map to the internal client, which allowed us to optain the physical socket that belonged to the specific internal client when attempting to do external → internal communication.

8 Significant Data Structures

- NAT table
  Our implementation uses a number of parallel arrays to represent the NAT table, all stored in an object with methods to act on the NAT table. The reasoning behind this approach was to prioritise fast searching and retrieval of the NAT table entries, as this would be the main functionality of the NAT table. This aproach was used instead of an array of objects to minimise the overhead for each NAT table entry, as depending on the number of clients and number of NAT boxes, the additional overhead could grow in size. Also, an array was used as we can assume that the NAT table will only ever be able to mapp as many entries as it has external IP addresses. This means that we have a finite size, and so we can use an array which allows for faster searching time.

- Server Routing
  For server Routing, there are multiple list structures which are used to allow for routing between different nodes on the simulated internet. The two types of lists are:

  - A list of connected external clients
  - A list of internal clients

  The reason for the separation is so that internal clients can be connected to a particular NATbox, and packets can be routed through that NATbox instead of directly to the actual clients.

9 Design

The design of our program and some decisions we made regarding it are very interesting. Some of these include having a global server acting as an internet or network connecting hosts, another being the implementation of DHCP that we used. Some of these are outlined below:

- Central Server
  We deicided to have a central server running for the entire simulation, and

clients must connect to this central server. The central server can host multiple NATBoxes, allowing us to effectivly simulate the internet. The server consists of a master thread (which simply accepts new connections) and multiple slave threads, that were forked from the master thread. These slave threads will handle reading bytestreams from the sockets, and then will forward the bytestream to the destination slave thread (the thread allocated to the destination IP of the *paquet*, so they can send it to their respective clients.

- The DHCP server
  Our implementation of the DHCP server is quite an interesting one, and using it allowed efficient access to IP addresses. The DHCP server can run in two modes, namely internal and external mode. The external mode of the DHCP server allows for it to function like an ISP does when assigning external IP addresses to clients. The internal mode on the other hand, works like that of a DHCP server on a network, assigning internal IP addresses to internal hosts. Both implementations make use of a pool and generation algorithm. The generation algorithm generates new IP addresses for the DHCP server to allocate. This function is called when more IP addresses are needed by the clients. The pool algorithm is the DHCP server's way of reusing IP addresses. The pool is a data structure containing any IP address that had been generated, but was freed up by a client no longer using it. When a new client requests an IP address from the DHCP server again, the pool will be consulted first, and any IP addresses in there will be reused. If there are no available IP addresses, then generation will take effect again.

- The NAT box – server connection
  The strategy used for connecting the server and NAT box was to have the server *contain* an amount of NAT boxes, and have the NAT boxes run from within the server. This interesting implementation was used to avoid the NAT boxes from having to replicate the functionality of the server (send and receive messages). The implementation allows for clients to be connected to a certain part of the server hosting that particular NAT box, and send messages directly to that NAT box. It also intuatively allows for multiple NAT boxes, an impressive feature specific to this implementation.

- The NAT boxes external addresses mapping
  A NAT box in the real world has a set of IP addresses that it has been allocated by the ISP, and which it uses to allow for external communication between it's internal clients and the external network. In our implementation, we simulated this accurately, using the DHCP server and NAT box implementation. It works by having the server's DHCP server allocate each NAT box a certain amount of IP addresses. These IP addresses are then given to the NAT box to be used to map to internal clients when needed (for connecting to external hosts). The IP addresses are then given to a special case of another DHCP server, which takes in only these

addresses, and can only give out these (it stores them in its' pool and disables generation). This means that when the NAT box adds a new entry into it's NAT table, it can request from this special DHCP server, an IP address that the NAT box has been allocated. When the NAT table entry is evicted, it's external IP is given back to the special DHCP server to be reused in the table again.

10 Compilation

10.1 Compiling

Simply run,

```
$ make
```

11 Execution

11.1 Server

To see all available arguments, run,

```
$ java -cp src/ Server -h
```

To run the server, run,

```
$ java -cp src/ Server [optional args]
```

11.2 Client

To see all available arguments, run,

```
$ java -cp src/ Client -h
```

11.2.1 Internal Client

```
$ java -cp src/ Client -i <natbox ip to connect to> [optional args]
```

11.2.2 External Client

```
$ java -cp src/ Client [optional args]
```

12 Libraries used

- java.net.*;
- java.io.*;
- java.util.Scanner;
- java.util.ArrayList;
- java.nio.file.Files;
- java.nio.ByteBuffer;