Marie Petitjean
CS352 - Fall 2020

NAT Project

Motivation

Many individuals today have a personal computer, a smartphone, and perhaps an iPad or an Alexa connected to their home internet. Just in the United States, there are 328 million people, if we assume each of them have 3 devices, then there are almost a billion connected to the internet. When each of these devices are connected to the internet, they require a public IP address to identify it. While IPv4 can accommodate almost 4 billion different IP addresses, our 1 billion excludes Europe, Africa, Asia, and businesses networks. If we were to include all devices in the entire world, then we would quickly run out of IPv4 addresses. IPv6 addresses were developed to help solve these issues, but they are complicated, and it has been a slow process to gain support.

A Network Address Translation (NAT) enabled router allows the ability to map a group of devices connected to a home router to a single public IP address. This allows us to give a private IP address to each device connected to the home router, such as our phone, computer, and iPad. These 3 private IP addresses can then be mapped to a single public IP address when it connects to the internet. This means that rather than have 3 public IP addresses, each home only has 1. If 3 people live in a single home then rather than using 1 billion public IP addresses in the US, only 109 million would be used. This significantly reduces the number of public IP addresses while allowing more devices to be connected. This is immensely important because it prevents the depletion of IPv4 addresses. Another benefit of NAT is that it hides private IP addresses from the rest of the Internet. This adds an additional layer of security to protect ourselves and our devices.

Approach

The initial approach to solve this problem was to have a NAT program which listened on a single interface. It would have two NAT tables, one for inbound mappings and one for outbound mappings. The mappings would be from a pair of IP address and port to another pair of IP address and port. The NAT program would continuously listen for packets on the interface and would use their mac addresses to determine whether it was an inbound or outbound packet. If a mapping was found then it would either rewrite the source or destination of the packet. In order to be more compatible with the FPGA, functions to filter, read, and write packets were all implemented without the use of gopacket. In order to add and list mappings, the NAT program would listen for a specific IP and port which would be for control packets. The control packets were UDP on IPv4 and would have a special payload depending on the control type. A separate program was implemented that would easily create and send out well formed control packets.

Listening on a single interface did not fully mimick how a NAT functioned, so the NAT program was improved to listen and write on two interfaces. The first interface was a TUN interface which represented the LAN side of the NAT. The second interface was the ethernet interface which represented the WAN side of the NAT. In order to accomplish this, some of the functions to parse and write packets were modified to accommodate for the TUN packets not having an ethernet header. In addition, the water Go library was used in order to create the TUN interface and to read and write packets to it. Dynamic mappings were introduced so that if an outgoing packet from the LAN side did not have a mapping, it would create one from the source IP to the WAN IP, keeping the port the same for simplicity. In order to specify the WAN IP and

other important constants, the *config.yaml* file and parser were introduced to easily set global constants throughout the entire project.

Architecture

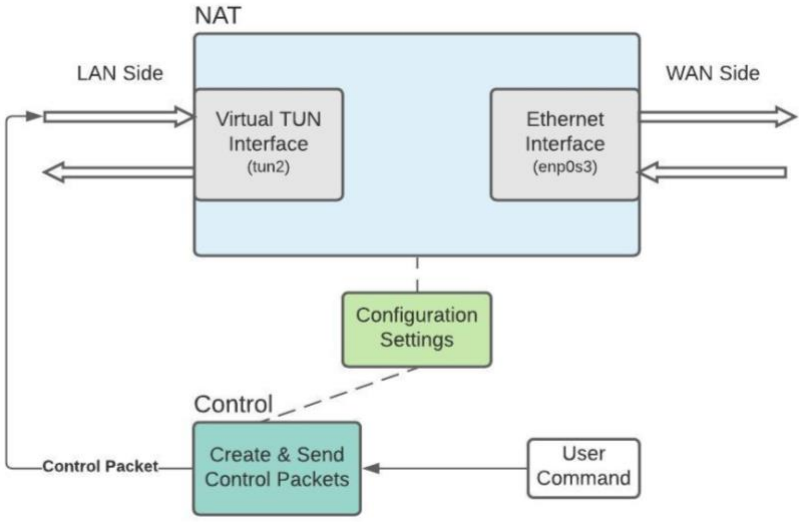


Figure 1: NAT Project Architecture

The NAT project comprises of three major components. The first component is the actual NAT which listens on two interfaces. The local side of the NAT, which is denoted as local area network (LAN), listens on a TUN interface. The other side of the NAT, which is denoted as the wide area network (WAN) listens on an ethernet interface.

The second component is the control component. This component takes command line arguments from the user which specifies the kind of control packet that they would like to create. The control component will use this information to construct a well-formed packet and will send it on the TUN interface to be read and processed by the NAT.

The third component is the configuration settings. This component allows the user to specify configuration settings such as LAN and WAN IP addresses. Both the control component and the NAT component depend on the settings configurations component.

NAT Component

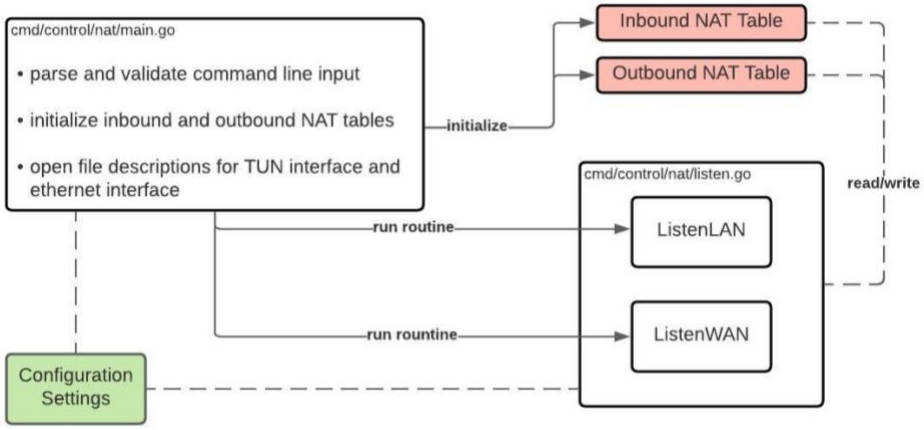


Figure 2: NAT Component Architecture

The NAT component first takes in command line arguments when it is started. These command line arguments can turn on static mappings only or turn on silent mode, which does not print when mappings are found to the console. It then initializes two instances of NAT table components; one is for inbound mappings and the other is for outbound mappings. It then opens the final descriptors for TUN and ethernet interfaces, which will used for listening and writing packets to these interfaces. Finally, ListenLAN and ListenWAN are run as Go routines indefinetly. These Go routines are responsible for listening for packets on their respective interfaces, and if a mapping is found, then rewriting the packet and sending it out on the other interface.

NAT Table Component

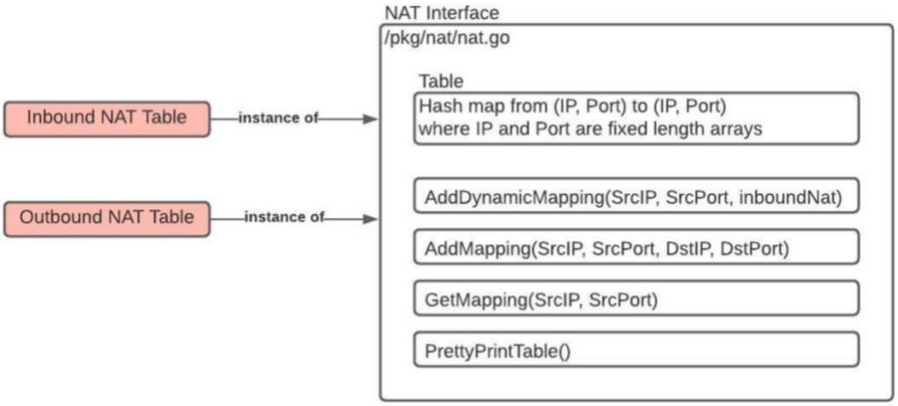


Figure 3: NAT Table Component and Functions

The NAT Table component contains the implementation of the NAT Table and a few functions that can be performed on it. The NAT Table is implemented as a hash map from a pair of IP address and Port to another pair of IP address and Port. The IP address and Ports are implemented as fixed length arrays in order to more compatible with the FPGA.

There are two functions that can be performed on a NAT Table in order to introduce new mappings. The first function *AddMapping* simply takes in a pair of IP address and Port which will serve as the key and another pair of IP address and Port for the mapping. If the key is already in the table, then this new mapping will overwrite the previous one. The *AddDynamicMapping* adds a mapping on the current NAT Table from the given source IP address to the WAN IP address from the configuration settings. It also adds a mapping from the WAN IP address the source IP address to the inbound NAT Table which is passed into the function. The ports are kept the same through the mappings for simplicity.

There are also two functions which read from the NAT Table. The first function *GetMappings* take in a source IP address and port and returns the mapping if it found one. This function is implemented to treat the port of 0 as a wildcard, and only matches IP addresses. The function *PrettyPrintTable* is used to print out the mappings to the console.

Marie Petitjean
CS352 - Fall 2020

ListenWAN Component

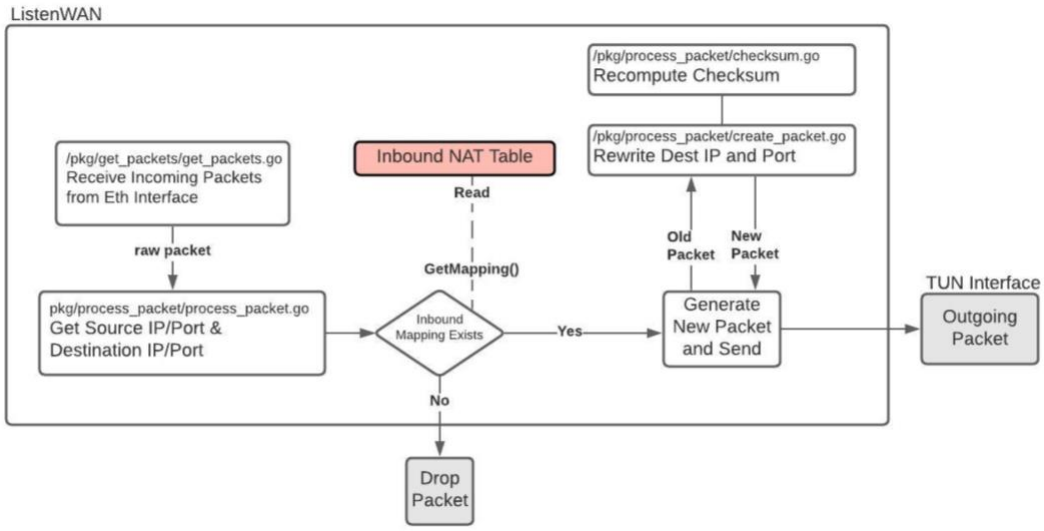
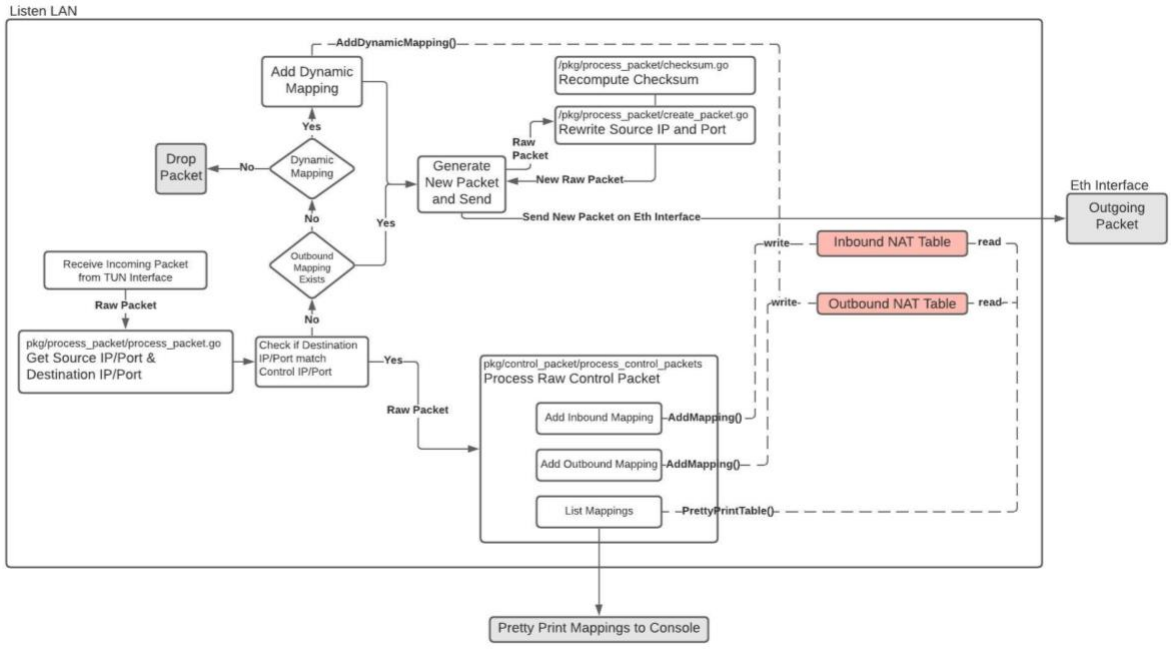


Figure 4: ListenWAN Component

The ListenWAN component is a function that runs in a Go routine. Packets are read from the interface using *pcap.Handle* and are put into a Go channel. This was implemented to reduce latency by not using gopacket library and instead only relying on the pcap library. ListenWAN continuously picks packets off of this channel to process them.

The raw packet is sent to a few functions which extract the source and destination IP and port. If the packet does use TCP/UDP on top of IPv4 then the packet is dropped. If the destination IP and port have a mapping in the inbound NAT table, then the raw packet is sent to another function which rewrites the destination IP and port with the new values. Finally, the new packet is sent out on the TUN interface.

ListenLAN Component



Packets are read from the TUN interface using the water Go library. The raw packet is sent to a function which extracts the source and destination IP address and port. If the packet does not use TCP/UDP on top of IPv4, then it is dropped. By default, if the packet's source IP and port do not have a mapping in the outbound NAT table then a dynamic mapping is added. If dynamic mapping is turned off, then this packet is simply dropped. Afterwards, the packet is sent to a function which creates a new packet with the new source IP and port. The new raw packet is sent out on the WAN interface.

Control Component

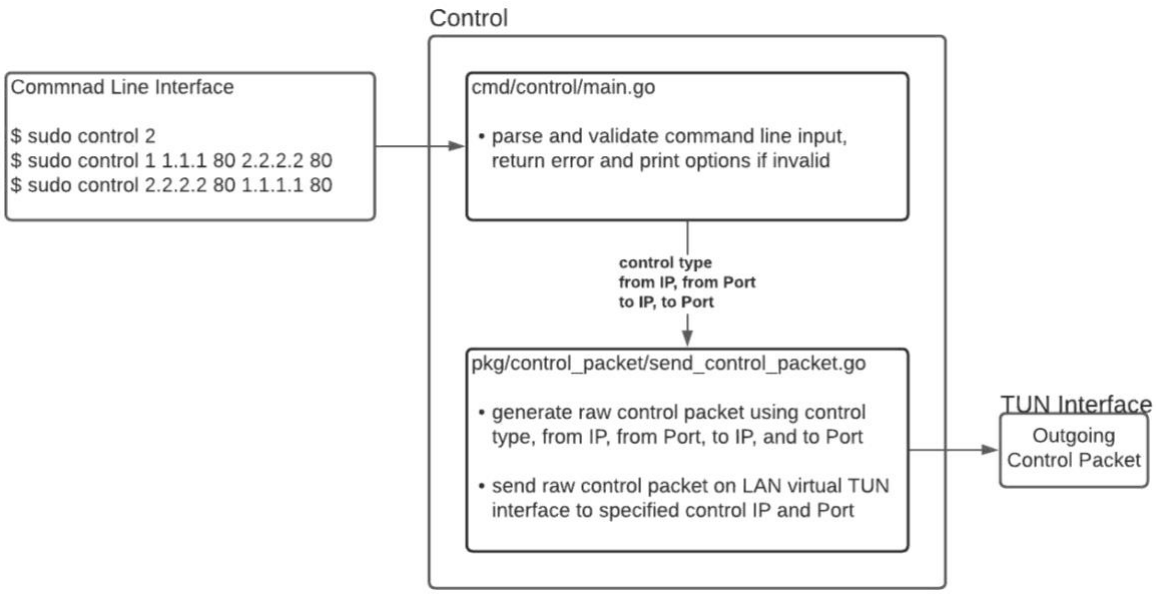


Figure 6: Control Component Architecture

The control component is a small program which creates and sends out control packets. This is completely separate from the NAT implementation and is used for convenience. It takes in command line arguments which represent the kind of control packet that needs to be created. Control packets are UDP packets on top of IPv4 which have a special destination IP and port that the NAT listens for. The packet's payload consists of at least 1 byte, which represents the control type. This can either be 2, to indicate print mappings, 1 to indicate adding a mapping to the inbound NAT table, or 3 to indicate adding a mapping to the outbound NAT table. For control types 1 and 3, the command line should also have the IP addresses and ports for the mapping, which will then be written in the payload of the packet. When the packet is constructed, it is sent out on the TUN interface to eventually be picked up by the NAT component.

Configuration Component

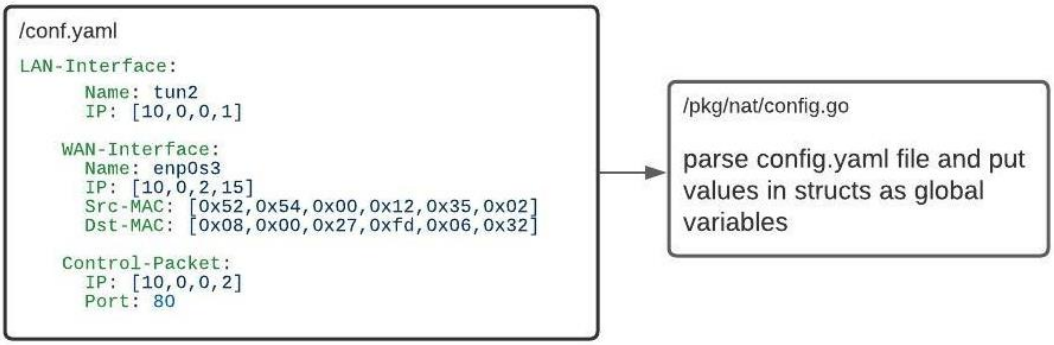


Figure 7: Configuration Component Files

The configuration component allows the user to set important constants in a single place that are used throughout the entire project by many of the components. This also ensures that all of the components are consistent in using the same constants so that they interact seamlessly with each other.

The user simply edits the *conf.yaml* file with the values that correspond to their system. Afterwards, when the NAT component or component are run, the parser for the config file is also run. The parser parses the yaml file and populates structs which become global variables.

Codebase

./cmd

The code for the project is divided into two large directories. The first directory is *cmd*. Within this directory there are two subdirectories *nat* and *control*. These directories correspond to the NAT component and the control component. These directories contain the high-level logic used for running the components.

./pkg/control_packet

The high-level logic depends on several smaller components and functions. These are held in the *pkg* directory. There are 4 main groups of functions which are split over the *control_packet, get_packets, nat,* and *process_packet* directories. The *control_packet* directory contains files which relate to the operation of control packets within the control component and for the NAT component. The file *send_control_packet.go* is only used by the control component to create control packets. It uses the gopacket library for simplicity since it is not intended to run on the FPGA. The file *process_control_packet.go* is used by the NAT component to process the payload of a control packet to perform the necessary operations on the NAT tables. Finally, a test file *control_packet_test.go* is used to make sure that the parsing and construction of control packets is correct.

./pkg/get_packets

The *get_packets* directory contains one file, *get_packets.go*. This file contains code that uses the pcap library in order to receive packets from an interface and put them in a Go channel. This means that we do not depend on the gopacket library for reading in packets and instead only relies on the pcap library.

./pkg/nat

The *nat* directory contains two files that relate to the implementation of the NAT table component. The implementation is held in *nat.go* and the corresponding test file is *nat_test.go*. This directory also contains *config.go* which is the parser of the yaml file and contains global variables that are used throughout the project.

./pkg/process_packet

Finally, there is the *process_packet* directory which contains all of the functions that parse and rewrite packets. This project does not depend on gopacket so a lot of functionality for manipulating packets has been implemented. The file *process_packet.go* handles filtering packets such that we only have TCP/UDP packets over IPv4 and extracting the source and destination IP addresses and ports. The file *checksum.go* implements calculating checksum from a raw byte array for both the IP header and TCP/UDP headers. The file *create_packet.go* contains functions which rewrite the source and destination IP and ports of a raw byte array packet. Finally, in order to better comply with the need of the FPGA, *copy_functions.go* implements some copy functions to use fixed length arrays instead of byte slices. The test file *process_packet_test.go* verifies the functionality of these files by comparing against the result of the gopacket library.

Evaluation

Navigating IP Addresses

An important aspect of a successful NAT is being able to navigate public and private IP address where packets pass through the NAT. The project was successful in achieving this by creating two interfaces which represent that LAN and WAN sides of the NAT. Its functionality can be demoed by either browsing through lynx or through curl commands. By having tcpdump running on the two interfaces, it can also be verified that the packets travel between the two interfaces and that the source and destination IP addresses and ports are properly rewritten.

It has been verified to run in a VM on the iLab machines and on a Mac. The readme files in the github repository (https://github.com/mpetitjean22/nat_project) contains directions on how to run and use each component. Scripts facilitate settint up the environment and running the demo.

Minimizing Latency

In addition to the NAT being able to properly manage packets passing in and out of the two interfaces, it is also important that it is able to do so efficiently. This implementation should be fast enough so that it is reasonable to use for browsing and downloading files from the internet. It should not introduce so much latency that it would be ineffective to use.

Downloading a large file from the internet would indicate how well the NAT would be able to process and send out packets on a single connection. The NAT was tested on how well it would be able to download the kernel using the following command.

```
time wget -O /dev/null https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/snapshot/linux-5.10-rc4.tar.gz
```

The results showed that the NAT had an average rate of download of 3.51 MB/s whereas without the NAT had an average rate of 3.95 MB/s. The difference of total time between with

and without the NAT was 5 seconds. Since the file was 176.80 MB, the NAT introduced 28 milliseconds per byte. This amount of latency is reasonable and shows that the NAT is able to perform well enough to be useful and effective. For more information about how the tests were run, the github repository contains the file *public_download.md* which specifies how a user could reproduce the results.

Minimize Errors and Dropped Connections

A NAT is not only responsible for passing through many packets, but it must also be able to support creating many connections and maintaining mappings. In order to test this, a local HTTP server was created and httperf was used to do a benchmark test. The httperf test created a total of 8,000 connection at a rate of 100 connections per second.

Over the 8,000 connections, the NAT introduced on average only 42.1 milliseconds per connection. The reply rate with and without the NAT were very similar and showed an average of 100 replies per second. This matches the rate of connections being sent out which indicates that the NAT is not dropping any connections even with a large number of connections being made. Overall, the test found that no connections were dropped or failed. This indicates that the NAT introduced minimal latency and errors when thousands of connections were being made.

For more information about the exact data from the benchmark test and how to reproduce the tests check out the *performance* directory in the github repository.

Maintainable

Another important aspect of this project was that it would be easy to maintain and to expand upon. In order to accomplish this, there have been several test file developed in order to verify that everytime a component of the project has been modified or improved that the functionality still remained. It was also important that the project was easy to use. In order to accomplish this, the control component was developed which allowed an easy way for the user to construct control packets and send them out. There are also a collection of scripts which allow users set up their development environment quickly and easily. The configurations yaml file also allows the user to set constant variables, which depend on their environement, for the entire project. This allows the NAT to be set up on any system with ease.

Reduce Dependency on gopacket Library

The gopacket library is a Go library that parses and creates packets. Since this project was originally designed to run on an FPGA, using the gopacket library would introduce too much complexity. The files in *./pkg/process_packet* implement the function need to filter, parse, and write packets. They also use custome copy functions which reduce the use of slices and instead use fixed length arrays. These efforts make it more feasible to run on the FPGA.

The branch *fpga* in the github repository contains a breakdown of how the code ran through the argo2verilog parser. Overall, the code that was initially anticipated to run on the FPGA ran well through the parser. Some minor adjustements were made and can be seen in each of the files within this branch. In addition, I found some improvements to the parser that would make it so that most of the code would be able to run through it.

Marie Petitjean
CS352 - Fall 2020

Future Work & Enhancements

Multiple Go Routines

The current implementation runs the *ListenWAN* and *ListenLAN* functions in two different Go routines. Read and write locks have been implemented to allow the ability to run more than one routine of *ListenWAN* and *ListenLAN*. The program was able to run 10 Go routines for each of *ListenWAN* and *ListenLAN*, however, it was less efficient than running a single Go routine for each. One possible explanation is that the Go routines were not efficiently picking packets off of their respective Go channels. Future work could look into why this is the case and perhaps implement a way such that the packets in the Go channel are evenly distributed among the Go routines. Taking advantage of having multiople Go routines could improve performance even further. Another possible explanation is that the locks were causing contentions. It would be beneficial to do a more in depth analysis of how the locks interacted and whether improvements could be made.

More Advance Dynamic Mapping

An improvement to the current implementation would be to introduce a more complex dynamic mapping. At the moment, we only have one IP in our pool of WAN IP address, the implementation can be improved by allowing a pool of WAN IP addresses. There are also no rules to purge mappings after a period of time, which could lead to running out of IP and port mappings in a scenario with many connections. An improvement would be to create rules for when to purge a mapping.

Retrospective

What worked, and what did not?

Overall the project worked rather well. Listening on two interfaces was an effective way of simulating LAN and WAN sides of the NAT. The control component was very useful in creating and sending out control packets to be read by the NAT component which was helpful for debugging. Throughout the semester, the project was developed iteratively, starting with a very basic implementation and building on top of it. This development process worked rather well. The code base was divided into several different directories which made it easy to change certain implementations without having to refactoring multiple parts of the codebase. The test files also verified that any changes would not break functionality.

Since this project was originally designed to work on an FPGA, I implemented things that could have been provided by a Go library. This enhanced by understanding of certain networking topics, but did take a lot of time to implement.

What was challenging?

One of the most challenging aspects of this assignment was understanding networking topics such as how packets are structured and being able to properly create and read them. There were times in which my packets were malformed and I had to learn how to properly use wireshark in order to figure out why they were not being sent out.

Another challenging aspect of this project was using the TUN interface. I had to learn how to read packets from the TUN interface and send packets onto the TUN interface, which was a different process than simply using the pcap library. I had to understand how to use routing

tables effectively so that I could direct packets to the TUN interface instead of the default interface.

Knowing what you know now, how would you do something differently? If fellow students would attempt similar projects, how you advise them?

If this project did not need to run on an FPGA then I think that I would have spent less time implementing functions which parsed and created packets, and instead would have used the gopacket library. Although, implementing these functions solidified by understanding of networking topics. I would advise students to take advantage of all the Go libraries if they are able to because they are very powerful.