Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Telemática

Trabajo Fin de Grado

Developing a Network Virtualization Framework for Testing Network Resilience Techniques

ESCUELA POLITECNICA SUPERIOR

Autor: Pablo Collado Soto

Tutor/es: Iván Marsá Maestre

2021

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería Telemática

Developing a Network Virtualization Framework for Testing Network Resilience Techniques

Author: *Pablo Collado Soto*

Tutor: *Iván Marsá Maestre*

Comittee:

President: *José Manuel Giménez Guzmán*

First Member: *Luis de la Cruz Piris*

Tutor: *Iván Marsá Maestre*

23 June 2021

ACKs

This project marks the end of a key step in my academic and personal lives. I could not have tackled such a task without the help I have been gifted by many more individuals than I can really account for. However, I will try to do my best to thank as many as I possibly can.

I would like to thank my parents. Even though when asked about what I do they just say "there are a bunch of weird letters moving on screen" they have always been there for me along the way. Figures such as my uncle, *Chema Luzón* and my grandmother, *Josefina Álvarez* have also played a crucial role in my academic progression: they showed me, by example, the meaning and value of pure, raw knowledge.

If there is something I know for sure is I would not have made it here if it were not for all the teachers I have had, both good and bad. The best teachers have not only transmitted the technical knowledge I have made use of time and time again, but they also taught me how to think and how to appreciate knowledge for what it is. Teachers such as *Francisco José Álvarez*, *Antonio García*, *José Manuel Giménez*, *Javier Macías*, *Iván Marsá*, *Isaías Martínez*, *Manuel Moreno*, *Rosalino Pulido* and *Elisa Rojas* have given me more than they know.

It is quite difficult to understate the role my friends and loved ones have played in the years leading to this moment. Classmates such as *David Gaupp* and *Cristian Morales*, long time friends such as *Carlos Corbella*, *Gonzalo Martínez* and *Alberto Río* and my partner, *Alicia*, have made the road a thrill.

I would finally like to personally thank *David Carrascal*. He has not only provided direct help on this project (including this report) and many others, but he has also been one of the people we have shared the most with. Despite meeting towards the end of the degree he has proven to be, time and time again, the most knowledgeable person I know. His drive and passion for what he does has motivated me to try and improve every single day. Nonetheless and, more importantly, he has also been a great friend who makes you feel at ease with no matter what.

I am certain somebody is being left out. One of the best things of knowing nothing is that everybody can teach and show you something. Given there is not enough paper in the world to write down every single person's name I only ask that nobody feels ignored: everybody I have ever met has played a role, either big or small, in me being here. I cannot say anything but *thank you*.

Abstract

The object of the proposed Undergraduate Thesis is the development of a software system automating the deployment and management of fully virtual networks for their use as a testbed. The project sits within the scope of the *CloudWall*[1] project from the *Automatics Department* at the *University of Alcalá*, whose main focus is to develop a Cloud-enabled Resilience Framework tailored for the needs of the healthcare IT infrastructures to increase their capability to prevent and react to cyber attacks. Its intended purpose is serving as a validation mechanism for the techniques developed within said project.

The use of *docker* containers as virtual network nodes together with the possibilities offered by the *linux kernel* provide a huge amount of flexibility that we have respected and made available to the user. The logic implementing the network control functionalities has been written entirely with *python3*. A proof of concept proving the project's suitability for its intended use is also provided. What is more, given the technologies the project has been built upon its use cases are much broader than what was initially required. We consider its possible use as a teaching resource to be one of the most promising future applications.

Keywords: *docker* [1], *iproute2* [2], *namespaces* [3] *python3* [4].

[1]This work has been partially supported by project PID2019-104855RB-I00/AEI/10.13039/501100011033 of the Spanish Ministry of Science and Innovation.

Resumen

El objetivo del Trabajo de Fin de Grado propuesto es el desarrollo de un sistema *software* que automatice el despliegue y gestión de redes puramente virtuales para su uso como entorno de pruebas. El trabajo pertenece al ámbito del proyecto *CloudWall*[2] del *Departamento de Automática* de la *Universidad de Alcalá*, cuya meta primordial es el desarrollo de una infraestructura de resiliencia de red adapatada a las necesidades de los sistemas informáticos del sistema sanitario para así incrementar su capacidad de prevención y reacción ante ataques cibernéticos. Nuestro trabajo busca servir como mecanismo de validación para las ténicas obtenidas del proyecto *CloudWall*.

El uso de contenedores de *docker* como nodos de red virtuales junto a las posibilidades brindadas por el *kernel* de *linux* ofrecen una gran cantidad de flexibilidad que hemos respetado y proporcionado al usuario. La lógica que implementa las funciones de control de red se ha escrito íntegramente en *python3*. Asimismo, se ha desarrollado una prueba de concepto para demostrar la adecuación del resultado obtenido para el uso que se le pretendía dar en un principio. Además, dadas las tecnologías empleadas, las aplicaciones que se le pueden dar a este trabajo son más amplios de lo inicialmente requerido. Consideramos que una de las aplicaciones futuras más prometedoras es su posible uso como una herramienta de apoyo a labores docentes.

Palabras clave: *docker* [1], *iproute2* [2], *namespaces* [3] *python3* [4].

[2]Este trabajo ha sido parcialmente financiado por el proyecto PID2019-104855RB-I00/AEI/10.13039/501100011033 del Ministerio de Ciencia e Innovación.

I only know that I know nothing.

Socrates

Contents

Abstract **vi**

Resumen **vii**

1 Introduction and Background **1**
1.1 Project's Background 1
1.1.1 Simulation vs Emulation 2
1.2 Outlining the Implementation 2
1.2.1 Getting the Machines 2
1.2.2 Getting the Operating System 7

2 Used Technology Analysis **9**
2.1 The Network Stack 9
2.2 The Network Namespace 12
2.2.1 A Common HTTP Server 12
2.3 The iproute2 Suite 13
2.3.1 The Deprecated Solution: ifconfig 14
2.3.2 Describing a Network Interface 15
2.4 Instantiating Virtual Network Elements 15
2.4.1 Naming Network Elements 15
2.4.2 Bridges 16
2.4.3 Veths 19
2.4.4 Managing Network Namespaces 21
2.5 Addressing Layer 3 Network Devices 24
2.6 Adding Firewall Functionalities to Layer 3 Devices 26
2.6.1 Overview of iptables 26
2.6.2 Instantiating iptables Rules 28
2.6.3 A Foreword on Container Capabilities 29
2.7 A Small Caveat: Debugging Connectivity Issues in Virtual Bridges . 29
2.8 The Containers 31
2.8.1 Managing Docker 31
2.8.2 Container Images 31
2.8.3 Managing Containers 34

3 Manually Bringing Up a Simple Network **40**
3.1 A Note on the Chosen Private IP Range 40
3.2 Automating the Sample Topology 41
3.2.1 Running the Script 41
3.2.2 Checking the Script is Run by root 42
3.2.3 Automatically Tearing Down the Topology 42

3.3 Testing the Topology Is Working . . . 45

4 Automating the Deployment of Virtual Networks 47
4.1 High Level Overview . . . 47
4.1.1 External Dependencies . . . 48
4.1.2 User Manual . . . 49
4.2 Overview of the Project's Modules . . . 56
4.2.1 Conventions . . . 57
4.2.2 The virt_net Module . . . 58
4.2.3 The graph_interpreter Module . . . 59
4.2.4 The net_ctrl Module . . . 59
4.3 Working Topologies . . . 60
4.3.1 Topology Alpha . . . 60
4.3.2 Topology Beta . . . 60
4.3.3 Topology Gamma . . . 60
4.3.4 Topology ICS . . . 60

5 Proof of Concept 65
5.1 Description . . . 65
5.2 Running the Proof of Concept . . . 71
5.3 Results . . . 72
5.3.1 Running the Attack Against a Static Network . . . 72
5.3.2 Trying to Mitigate the Attack . . . 78

6 Closing Thoughts and Future Work 83
6.1 Future Work . . . 84
6.1.1 Possible Improvements . . . 84
6.1.2 Use as a Teaching Resource . . . 84

A Project Budget and Schedule 86
A.1 Project Schedule . . . 86
A.2 Budget . . . 86
A.2.1 Software Costs . . . 86
A.2.2 Hardware Costs . . . 86
A.2.3 Labour Costs . . . 87

B System Setup 88
B.1 Installing External Dependencies . . . 88
B.1.1 iproute2 . . . 88
B.1.2 Docker . . . 89
B.1.3 PIP . . . 89
B.1.4 Python Modules . . . 89
B.2 A note on Capabilities . . . 90
B.3 Interacting With Docker as a Regular User . . . 94
B.4 Acquiring the Project's Code . . . 95
B.4.1 Leveraging git . . . 95

C Comprehensive Module Analysis **97**
C.1 The virt_net Module 97
C.2 The graph_interpreter Module 138
C.3 The net_ctrl Module 145

Bibliography **146**

Listings

2.1 Instantiating a Virtual Network Bridge. 18
2.2 Instantiating a Virtual Ethernet Interface. 19
2.3 Connecting a Veth End to a Virtual Bridge. 20
2.4 Connecting a Veth End to a Host or Router. 21
2.5 Linking a Container's Network Namespace to `/var/run/netns`. . . . 23
2.6 Running `ip` on a Different Namespace. 24
2.7 Addressing an Interface. 25
2.8 Instantiating *iptables* Rules. 29
2.9 Disabling Bridge Calls to *iptables*. 30
2.10 *Dockerfile* for our Virtual Routers. 32
2.11 Building an Image from a *Dockerfile*. 33
2.12 Running Network Nodes. 37
2.13 Inspecting a Container's Processes. 38
2.14 Checking a Container's *IP* Address. 39
3.1 Running a *bash* Script. 41
3.2 Automatic Deployment of the Sample Topology. 42
3.3 Testing the Sample Topology. 45
4.1 Defining the Sample Topology as a *networkx* Graph. 49
4.2 Turning a Graph Into a Virtual Network. 51
4.3 Syntax for Specifying Firewall Rules. 52
4.4 Uni-directional vs. Symmetric Firewall Rules. 52
5.1 The Attack's Script. 67
5.2 Retrieved Data Points for the Attack on a Static Network. 76
5.3 Retrieved Data Points for the Attack on a Dynamic Network. 79
B.1 Commands for Installing Needed Dependencies. 89
B.2 A Malicious Node Name Exploiting Privileges. 91
B.3 Transformation of Capabilities During `execve()`. 91
B.4 Defintion of the `drop_cap()` Function. 92
C.1 Data Structure Containing All the Network's Routes. 143

List of Figures

1.1 VM Setup 4

1.2 Container Setup 6

2.1 Linux Kernel Structure 10
2.2 Container Lifecycle 35

3.1 A Sample Topology 41

4.1 The *Alpha Topology.* 61
4.2 The *Beta Topology.* 62
4.3 The *Gamma Topology.* 63
4.4 The *ICS (Industrial Control System) Topology.* 64

5.1 Definition of the Network's *QoS* 66
5.2 *QoS* on a Static Topology 75
5.3 Attack Progression 77
5.4 *QoS* on a Dynamic Topology 80
5.5 Attack Mitigation Effect vs. Baseline Case 80

A.1 Project's Development Phases Over Time 87

List of Tables

2.1 *net-tools* vs. *iproute2* 15
2.2 Basic *docker* Commands. 31

4.1 Node types. 50

6.1 Our Tool vs. VMs 85

A.1 Project's Budget 87

Chapter 1

Introduction and Background

> Beware of programmers carrying screwdrivers.
>
> *Chip Salzenberg*

1.1 Project's Background

Our work cannot be understood if we don't take into account our tutor's research group's current line of research: **network resilience** and how we can improve it. Their approach is broadly based on modeling a network infrastructure as a multilayered construct where the upper layers get closer and closer to reality as we continue climbing them up. This approach is quite similar to that of conceptual network stacks such as the ones running today's Internet. Over this model, they analyse threats to the network and propose alternative reconfigurations which improve network resiliency against cyber attacks.

The above line of work has produced several high-quality papers worth of theory. As engineers we feel our duty is to look for a real-world application for our solutions too. In an effort to somehow experimentally measure the effectiveness of the defense strategies proposed by the group's research, we have been tasked with the development of a testing *framework*. Said *framework* needs to *emulate* an arbitrary network topology on which we can operate in such a way that we can mimic real world attacks. By applying the researched mitigation strategies on said scenario we plan on being able to settle which perform best based on the current threat and network topology.

1.1.1 Simulation vs Emulation

These two terms are often used interchangeably when referring to tools whose mission is providing the user with a scenario resembling the real world in some sort of way. Even though both terms share the same purpose, the way in which the accomplish it is radically different.

Simulation leverages the theory capable of modeling real world phenomena. One of the clearest examples is physics. Physics let us model the world that surrounds us through mathematics. That is why we can leverage the pertaining equations to compute the outcome of any scenario we can describe. Thus, simulation **computes** an outcome based on the initial parameters and the model we have built for describing the system under study. This implies that simulation is limited by how accurate our models are. If an equation doesn't take an aspect into account, that means it won't affect the simulation's outcome which in turn can result in inaccurate results. Techniques used for simulating systems include discrete event and agent based simulations such as those used in AnyLogic [5].

Emulation takes a different approach and tries to recreate the system under study to then perform experiments on it. If we manage to craft a detailed enough model, we could even get a glimpse of unexpected behaviors we hadn't taken into account. Even though emulation tends to offer a more precise description of the system under study, complexity can render this approach unusable. This follows from the fact that it is harder to recreate a system than it is to describe its expected behavior. In our project we have nonetheless decided to leverage emulation so that we reaped the most useful information from our experiments.

1.2 Outlining the Implementation

Chapter 2 contains an in-depth analysis of the technologies we have decided to make use of. This section is concerned with justifying said decisions.

1.2.1 Getting the Machines

Given our objective, we need to come up with a way of emulating a whole network. Before settling on what software platform (i.e operating system) we are going to employ we need to decide which technology is going to provide the machines we run those operating systems on. These machines need not be "real": we can explore virtualization technology

as well as more modern approaches such as containers. We'll devote the next section to discussing the aspects for and against each alternative so as to reach a final decision on what technology to use.

The initial impulse we had was to leverage the current virtualization capabilities offered by tools such as *VirtualBox* [6] or *VMWare* [7]. We were also aware of other existing solutions such as the *container* technology offered by *Docker* [1], for instance. What is more, we also knew of the existence of orchestration tools such as *Kubernetes* [8] which were aimed at employing several containers in a cooperative and organized manner. Given we need to build a network, no matter what technology we end up using, one may believe *Kubernetes* to be an attractive option. We will later see that this statement is not as true as one might have expected. We'll walk through what each of the above offer and how they accomplish their goals in an effort to decide which of them to employ as the cornerstone for our work.

Virtual Machines

Technology firms often tend to lock their products up and make them incompatible with other industry solutions in an effort to lock their user based out of the reach of other companies. This has left many end users having to cope with running several operating systems (OSs) on a single machine so that they have access to particular programs such as the *COMNET III* GSM Network Simulator, for example.

Virtual machines (VMs) let us cope with this situation with ease. Put simply, VMs *emulate* a whole *guest* operating system (OS) within a *host* OS. In order to do so, virtualization solutions, like the ones we mentioned before, leverage the capabilities of a "middleperson" known as the *hypervisor*. This *hypervisor* may be implemented in hardware or software and it provides an interface letting *guest* operating systems share the available computing resources with the *host* OS.

We, however, are not concerned with aspects such as the amount of processor time or memory that would be devoted to our VM. We, on the contrary, are mostly concerned with what happens with the network infrastructure. After all, our aim is creating a virtualized network and so we need to know how the virtual network attached to the created VMs is set up. We know that the VMs actually have Internet connectivity through the *host* OS so there must be some sort of "infrastructure" supporting said connections. Nonetheless and before we consider VMs as a feasible solution, we need to think about how they will scale.

The network topologies we have been charged with virtualizing are not small. Our largest working topology consists of 45 nodes, of which 36 would need to be implemented

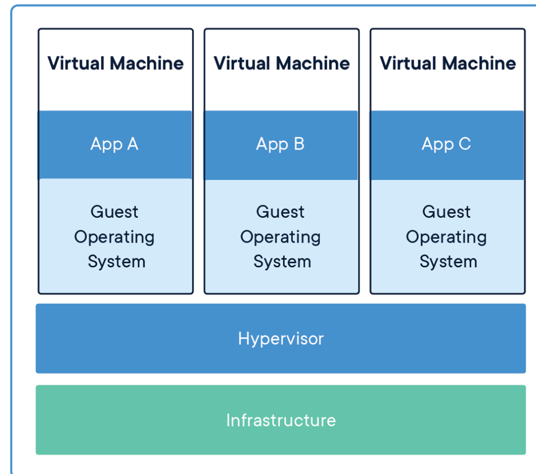


Figure 1.1: Image portraying the general architecture of a VM-oriented setup. [9].

as a VM instance. Given how resource intensive VMs are when compared to other solutions like containers, these numbers are too large to be handled in terms of virtual machines. If we just consider the amount of memory we would need to devote for them on our $8\ GB|_{RAM}$ machine we would be looking at roughly $234\ MB|_{RAM}$ for each of them and even then we would have no memory for the *host* OS at all. On top of that, the image for *Ubuntu 20.04* weighs $958,4\ MB$. It would be cumbersome to get it below that threshold as we would need to strip the *vanilla* (i.e. stock) version of any features we don't need and we would then have to re-package the result. Even then, we would be looking at around $72\ GB$ of used *HDD* space if we were to allocate $2\ GB$ of *HDD* space to each and every VM we were to bring up. Figure 1.1 graphically shows the general architecture characterizing a VM-oriented setup.

Even though the above requirements could be eventually met, it's easy to see how this approach wouldn't scale much further than it already has if running it on consumer grade platforms. We can then conclude how the fact that VMs are just "whole" operating system makes them too cumbersome to handle and too "big" to be instantiated all the times we need them. On top of that, and even though we didn't dig that deep into the underlying network infrastructure, it seems to be "darker" and less documented than that offered by other solutions like *docker*. Then, although VMs are a perfect fit for many scenarios, that is not our case. Knowing what set us back we will now analyse the *container* technology to find out how it is a perfect stepping stone for our project.

Containers

Before beginning to discuss whether containers are suitable for our purpose, we should begin by describing what a *container* really is as it can be a confusing actor in the virtualization realm. Please note this section has been purposefully written as a high-level description of container technology. As stated before, a more technical discussion is presented on section 2.8 belonging to chapter 2.

According to docker's documentation [10], "a *container* is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another". Now, this definition is a perfect representative of the main kind of problems we will have to face when trying to bend containers to our will throughout the development process. Containers are designed to try and increase the portability of applications. In today's Internet-centric society, where we all want uninterrupted services, it is critical to be able to change an application's environment at a moment's notice due to outages, ending support cycles for software, security breaches... That is why the container technology has developed at such a high pace. It provides the infrastructure for a reliable service delivery. Now, we could "twist" the definition if we were to think of applications as full-fledged operating systems. That is, if we run a whole OS within these containers we would be actually achieving our goal: each container will behave as a network node so, in other words, we would only need to cope with 36 concurrent containers: an easier task than doing the proper thing with VMs.

One of the questions we asked ourselves when trying to decide on a technology was: what makes a container different than a VM? If we go back to the section we devoted to virtual machines we will notice how they run a full-fledged operating system as a guest. This implies each VM has its own *kernel*. This is clearly shown on figure 1.1.

Kernels and their design have been the topic of many books and documents. We just need to know that the *kernel* is the piece of software "gluing" the hardware and userland software (programs such as browsers) together. That is, it allows applications to access the computing resources in an organized manner, enabling the sharing of resources amongst them. As we are telematics engineers we usually find the *kernel* concept easier to understand in terms of the abstractions it offers us; the network socket being the most familiar. Through it, our applications can leverage the networking capabilities of the machine they're running on for instance.

Unlike VMs, containers all share a single *kernel*. We can then think of containers as a way of isolating applications along with all their dependencies within a shared computing platform. We don't need "specialized" software agents such as an hypervisor; we would only need to be very meticulous and know the *kernel's* offered facilities very well to achieve

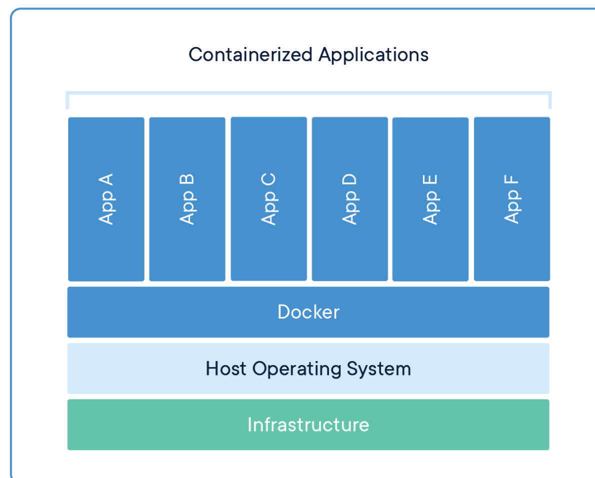


Figure 1.2: Image portraying the general architecture of a container-oriented setup. [12]

the end result offered by containers. Projects like *bocker* [11] prove one can achieve similar results to the ones provided by docker if only interested in a subset of the latter's capabilities. Nonetheless, one can regard containers as light VMs throughout the development as, even though it is not exactly true, it won't hinder our development approach. Figure 1.2 shows the overall architecture of a container-based approach. Comparing it to figure 1.1 can be truly revealing when comparing both virtualization approaches.

Containers and Docker Container technology can be thought of as a standard. However, the way that technology is implemented can vary. Then, docker offers an *implementation* for containers. This is a similar situation to that of VMs. The idea of a virtual machine has found two main implementations by VirtualBox and VMWare. This concept is similar to the situation posed by RFCs published by the IETF. They propose a standard and several people try to implement it according to their coding style, level of knowledge...

If we are to be entirely correct when it comes to nomenclature we would have to say that our solution is going to be based on docker's *container implementation*.

On top of container technology being lighter than that of VMs, which makes it more scalable, we also found the amount of documentation regarding the network infrastructure to be much larger. That gave us a strong foothold in our path to getting our framework up and running. Another positive aspect on containers we initially considered was the existence of *container orchestration tools* such as *Kubernetes* which we thought could make our job easier. We'll devote a few paragraphs to discussing why, in the end, *Kubernetes* wasn't that much of a fit for us.

Kubernetes

As stated in *Kubernetes'* own documentation [13], "*Kubernetes* is an open source container orchestration engine for automating deployment, scaling, and management of containerized applications". In other words, *Kubernetes* let's us define how we want our application to behave through a manifest provided in a `*.yaml` formatted file. Now, if we consider docker to be focused on offering "services", *Kubernetes* takes that objective to the next level. It's a product aimed at professionals that's engineered to be stable. That means that meddling with the internals in an effort to achieve our goals was bound to be an extremely difficult task. By employing *Kubernetes* we would be using an infrastructure that's already been laid for us, which is difficult to change and that does not behave how we want it to. We believe using a tool only to work against its basic principles is a wrong approach. That's why we decided to manually build the needed infrastructure from the ground up starting from docker containers.

Just to give a concrete example of what we mean by working against the pre-existing infrastructure we could take a look at how *Kubernetes* connects nodes. As it needs to offer some kind of service it will set up the internal network topology in such a way that all the nodes are able to connect among themselves. Given our requirement of implementing a firewall in the topologies we need to be working with, we can conclude it would be cumbersome to try to implement a firewall in an infrastructure whose primary concern is enabling communication links amongst all the network nodes. It's this diametrically opposite approach that made us dismiss *Kubernetes* as a feasible solution.

1.2.2 Getting the Operating System

After settling on an option providing the "hardware" our software is to run on, we need to decide which operating system is the best suited for the task we have been proposed. Given today's ecosystem we quickly thought of the three main contenders in the user market. These are *Microsoft Windows*, *macOS* and *Linux*. Before delving any deeper into the discussion about which to employ we would like to clarify our choice of words when referring to the last option.

Linux vs. GNU/Linux Strictly speaking, Linux is just the kernel of Linux-based distributions or *distros*. As we explained in the previous section, a kernel is the piece of software gluing the hardware and user software together. It accomplishes this non-trivial feat by offering applications an interface through which they can request services. This kernel will then fulfill these requests in an orderly manner so that applications can coexist

and cooperate towards a better overall usage of the system. These applications can in fact not even be aware that they are running alongside others.

The kernel is for many the most crucial piece of software within a full fledged operating system. It's the cornerstone on which everything else is built. Nonetheless, if a non-technical user were given just a kernel they would have a very hard time making any use of it. This is where applications or user programs come into play. The make use of the OS's services and they let an end-user get meaningful work done. These end user programs range form text editors such as *vim* or *Microsoft Word* to VoIP (Voice over IP) PBXs (Private Branch eXchanges) like *asterisk*.

The role of GNU in all this is providing many of these end user programs as free (as in freedom) software. Huge projects like the *gnome* desktop environment and *make* (which we are using for compiling this LaTeXfile) carry the GNU stamp. Given the above, GNU argues operating systems packaging their tools should be considered GNU/Linux systems as these two terms work cooperatively, i.e. they are software pieces with distinct purposes. While we consider this to be absolutely true for end-user systems such as *Ubuntu desktop* and *Debian* we feel this is not the real case with us.

We will be running our network nodes as docker containers and we will try to make the image ran by these containers as lightweight as possible. In doing that we will only rely on GNU's shell, *bash*, for running a very restricted collection of commands. That is why we will refer to our platform as Linux containers instead of GNU/Linux ones.

Knowing we have already restricted ourselves to the use of container technology we can discard *macOS* as an option right away as there is no official support for it. We could have used *Windows*-based containers but given our uncommon requirements we opted to use a *linux*-based one. Given we feel comfortable with it, we settled on using *Ubuntu* docker images for running our nodes. The version we employed during the testing phase was deemed *Bionic Beaver* and had version number *18.04 LTS* (Long Term Release) which boasts a higher stability than yearly versions. Given our requirements, all our work should run "just fine" on newer versions. We nonetheless recommend working to *LTS* releases as the yearly builds can sometimes behave unexpectedly. On top of our choice's flexibility, we mustn't forget about the price and licensing factors. Even though we haven't looked into these topics regarding Microsoft's OS, Windows licenses tend to be more restrictive than those attached to Ubuntu. As we plan on working at a very "low level" with the kernel's internal network infrastructure we aren't entirely sure if that would be allowed by Window's license terms. On top of that Ubuntu doesn't cost anything, so we won't have to be concerned about a monetary budget. Putting it all together justifies why we decided to run Ubuntu within our own containers.

Chapter 2

Used Technology Analysis

> If I have seen farther than others, it is because I was standing on the shoulders of giants.
>
> *Isaac Newton*

According to the discussion in the previous chapter we have settled on docker containers running Ubuntu for providing the backbone of our virtualized networks. This section is concerned with analyzing the different technologies that will play a role in supporting the virtual network infrastructures we are to create.

2.1 The Network Stack

A fascinating but rather overwhelming image can be seen on figure 2.1. If we pay close attention we will see how one of its columns is just devoted to networking. The software entities comprising this column is what we will refer to as *Linux's Network Stack*

The word stack is something that shows up time and time again in the area of networking. It helps us have a top-level view of how logical entities cooperate within a network. When we think about stacks we naturally begin to consider them in terms of the layers they are composed by, with each layer tackling a simple task and offering services to the layer above whilst using those provided by the layer below. It is not going to be any different with Linux; we can think of its network stack as a huge "blob" of code which all network packets reaching a Linux-based system traverse. Thus, if we can alter how Linux processes packets or build *virtual* connections between different network stacks we would be capable of constructing a de-facto virtual network tailored to our needs.

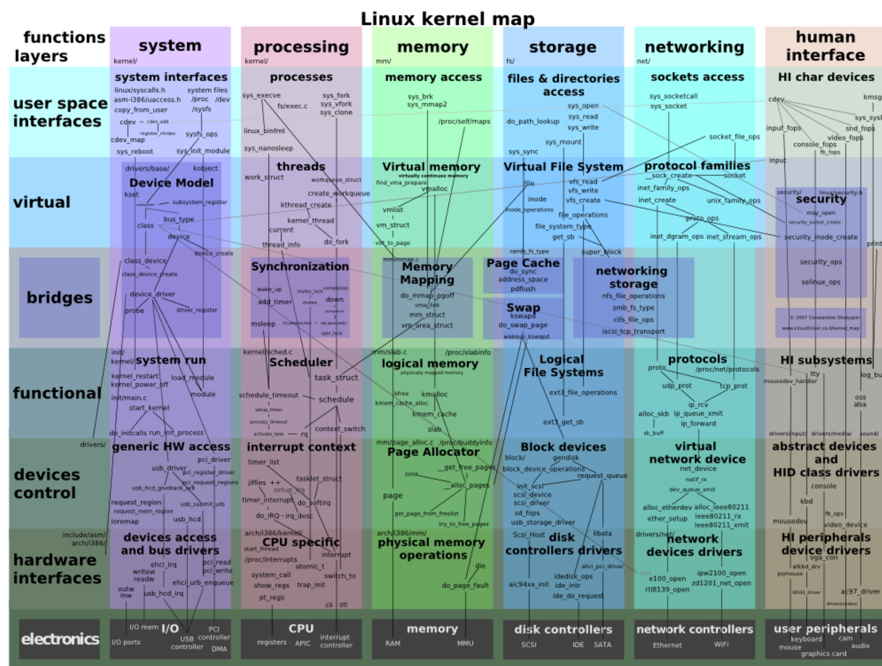


Figure 2.1: Linux's “Map”. [14]

Naming Packets Before we go on we need to shed some light on the naming we are going to use regarding the units of data exchanged through network links. Even though the term packet is tremendously generic we feel it is not a wrong one to turn to in our case. When we wire up several network nodes together we are looking for full connectivity, that is, connectivity at the application level. Thus, we are not really that interested on what layer the “packet” is at, we do not really care if the packet is a *segment*, *datagram* or *frame*. In the case a need for more specific naming arises, we will not hesitate to do so, but we prefer to keep the writing simple and avoid getting bogged down with technicalities whose benefit we feel is not that obvious.

A prime example of the above would be the use of the term *packet* instead of *link-layer frame* in the section's introduction. *Frame* is the correct term for referring to the data structure a *NIC* (Network Interface Card) hands to the kernel (albeit somewhat processed as the preamble and Frame Check Sequence of Ethernet frames are usually stripped from incoming frames by the *NIC* itself as seen on *WireShark's* documentation [15]).

If we think about network stacks, we would probably believe we need to have one per

machine. That is, every network-capable devices must have their own *data-path* which packets are to traverse. What may not be so simple is thinking that a machine *may* have more than one network stack. In order to get a firmer grasp on the implications of the above idea let us revisit the concept of a network interface.

When we first started learning about network architectures we where flabbergasted by the fact that a machine could have more than one NIC. This implied it had several different IP addresses “attached” to it, which provided redundancy amongst other several capabilities like traffic control. What astonished us the most was the raw power of not imposing a limit on the number of NICs a machine could have. That simple fact allowed routers to exist, for instance, and you could do seemingly useless stuff such as getting a packet through an interface and *echoing* it out the other. All in all, it provided a ton of flexibility to the whole system.

Now, if we apply the above to the concept of network stacks we could be talking about packets being interchanged in between them whilst residing on the same machine nonetheless. If we sit back and take a look at the larger picture, we can clearly see that packet below the application level are logically switched between network stacks belonging to different machines as it is the stack who is in charge of processing said data structures. Seeing matters in that light, and knowing we can have several network stacks on a single platform, talking about packets being interchanged within a machine does not seem that far fetched now.

Given the previous discussion we can now clearly see the base on which everything else is built upon. The ability to have several coexisting network stacks on a machine, as well as being able to connect them as we please, is such a powerful tool that our work is only scratching the surface of the capabilities enabled by this kind of technology. Linux’s network stacks are nothing short of an ode to code modularity.

All the previous discussion is related with the theoretical or conceptual realm of matters. We will now delve into how we can translate these ideas into a working virtual network. After going through that process we will also look into why we decided to do everything manually once we have a broader technical background on the subject.

Finally, we feel we need to clarify that, even though it may be clear at this point, all the network traffic we are to generate originates from within our own system. In other words, our network would be perfectly capable of working without any Internet access.

2.2 The Network Namespace

Many of the programming languages dominating today's market are object oriented. This, very roughly, means that the programmer is expected to generate *classes* representing "real-life" entities to some extent. These classes are defined by their *attributes* (characteristics) and their *methods* (what they can do). Now, a class by itself cannot do anything, we need to *instantiate* it so that we create an *object* of that class. We'll then be capable of using the newly created object as we please.

This concept of instantiation is actually quite powerful as it appears continuously in many areas of engineering. Now, we could say that Linux's network namespaces work in a similar fashion to classes. We can think of the network stack as the class and what we call *network namespaces* as the objects.

As seen on [3], namespaces are not only seen when dealing with networking, they are a feature of the Linux kernel employed in many other areas. These namespaces let us partition kernel resources so that each process sees a resource that is only for it, it is not shared. When applied to networking, we can see how each network namespace [16] represents an entire network stack. Then, if we set up several network namespaces (namespaces from now on as this is the only type of namespace we will deal with) we have effectively housed several network stacks within the same machine. We then need to look into how we can interconnect them.

Our system will model a network-capable machine as a simple process with its very own network namespace. This sentence can be quite abstract. That is why we believe a more concrete example can help the reader grasp what we are trying to entail.

2.2.1 A Common HTTP Server

The *Internet* has revolutionized society in ways little people could have predicted. It supports many different application protocols such as *SMTP* and *FTP*, but one of the most popular (if not the most) is *HTTP(S)*. This protocol provides the backbone for websites and many other applications that are employed by huge amounts of people on a daily basis.

Like many other protocols, *HTTP(S)* leverages the *client-server* architecture. The client role is usually "played" by web browsers such as *Firefox* or *Safari*, whilst the server part is relegated to programs such as *Apache* and *Nginx*. From a networking point of view, a process is univocally addressed through an *IP address* and a *port number*. The former identifies a given machine within a network whilst the latter targets a process running within that machine. Then, we can have different servers running on a *physical* machine

as long as their port numbers differ. We begin to see how, from a strict networking point of view, we can have several *logical* entities supported by a single *physical* one.

Now, imagine that we want or need to run two *HTTP(S)* servers on a single machine and a single port number. That would not be possible, would it? Now, if we leverage the concept of the network namespace we can circumvent this limitation. We could, for example, instantiate two different namespaces for each of the server processes and then bind each of them to the same port **within the respective namespaces**. Then, a network-aware process together with its own network namespace is, in a way, a full-fledged logical entity within a network: it is addressed by its own *IP address* and it has its own pool of independent *port numbers*. This is exactly what our framework exploits: each virtual host is a simple process running within its very own namespace.

The above was one of the main reasons we vouched for *containers* as a technology before. Each container ships with its own network namespace when it comes to networking. Whilst it is true that *docker* does bring up some network infrastructure with a new container, we can explicitly avoid that to be given a “pure” namespace with each new container. We can then manually instantiate any other network elements we might need to complete the network topology we are to build.

Managing namespaces and instantiating the virtual network elements, such as *veths*, gluing them together is attainable with the help of the *iproute2* suite of tools. We will then look into how to leverage *iproute2* for our needs.

2.3 The iproute2 Suite

The time we spend on terminal emulators has exponentially increased as we worked our way through our bachelor’s. Even though they provide quite a fast way to “get around” the computer they can be a little abstract at times. We are always aware that the OS “under” us has many running processes and provides us with services that can be used at our request. Nonetheless, it is crucial to leverage the help of programs that can query and show the system’s status when required. This is the case for programs like *ps*, which reports the status of the system’s processes, or *w*, which tells us about who is currently logged into the system as well as what they are up to. When using a shell we are aware of the fact that other processes are running and that other users may be logged in but we nonetheless have tools that let us consult the current status of the system.

Where networking is concerned, *iproute2* [2] is like a “Swiss Army Knife”. It is a *suite* or collection of tools that lets us do everything from inspecting the current interfaces and

routes to generating *IPv6/IPv4* tunnels. We will only be concerned with the *ip* command in our case which is in charge of *"showing/manipulating routing tables, network devices, interfaces and tunnels"* as seen in *ip's* manpage (i.e. [2]). The uses one can give to *iproute2* are seemingly endless, but before getting into them we believe it would be useful to compare the *iproute2* suite to its predecessor: *net-tools*.

2.3.1 The Deprecated Solution: ifconfig

Querying a system's interfaces is a very common task in the day of a network engineer. Whenever Internet connectivity is not behaving as expected or network connections do not seem to be working as intended, our initial step is to check interfaces are configured as they should. On *Unix*-based systems such as *macOS* and *GUN/Linux* this was traditionally accomplished with the *ifconfig* command. Before *iproute2* was implemented, network-related tools fulfilling the same purpose were provided by the *net-tools* suite. Among the most well-known tools provided by it we can mention *ifconfig* and *netstat*. The former displayed information on the system's network interfaces whilst the latter shed light on the open sockets in the system. When working with network-centric applications such as web-servers, *netstat* was a superb way of finding out whether some process had *binded* (i.e. began listening) to the 80 or 443 ports, each being the default for the *HTTP* and *HTTPS* protocols, respectively. With time, *iproute2* became available and it *deprecated* [17] the *net-tools* suite. We have nonetheless observed throughout our studies that a non-negligible amount of people are reluctant to abandon *ifconfig* and related tools.

What we are trying to achieve with this project is possible if leveraging *net-tools* instead of *iproute2*. However, the former's documentation and examples tend to be more cryptic and harder to digest. What is more, even though people resist to stop using it, *net-tools* is bound to disappear. That is why using the newer network tools confers a longer life prospect to our work.

Even though *net-tools* has been deprecated on *linux-based* systems, it's still the selected network tool suite for systems such as *macOS* where *iproute2* is **not available**. Some programs such as *iproute2mac* do exist for this operating system, but they just *parse* (i.e. process) the output provided by *net-tools* utilities and present it following *iproute2's* format. We must not be deceived by looks: *iproute2* is a part of *linux* systems. We compare both suites on table 2.1.

net-tools	**iproute2**
`arp`	`ip neigh`
`ifconfig`	`ip addr`, `ip link`, `ip tunnel`
`netstat`	`ss`, `ip maddr`
`route`	`ip route`

Table 2.1: *net-tools* utils vs. *iproute2's*. [18]

2.3.2 Describing a Network Interface

In a previous section we already provided some comments on what NICs are. Knowing that interfaces create "bridges" between different systems we can clearly see how these network interfaces connect digital systems to a communication network; they let these digital machines leverage the communication capacities of computer networks.

Whilst almost all engineers could recognize what a NIC physically is, the matter is not that simple regarding how the OS treats these interfaces. We find it quite helpful to think of what we would do if we had to implement some software solutions that had to employ the network's capabilities. There must be a "way" that these hardware components can be used from the application perspective, that is, the OS needs to offer some kind of abstraction through which we can use the NIC itself. This abstraction is what we will call a OS-level NIC or network interface. Then, applications need only be concerned with opening *sockets* that are then supported by one of the system's network interfaces.

Knowing a bit about what *iproute2* is, we believe walking through some examples showing how to instantiate virtual network elements will prove rather revealing.

2.4 Instantiating Virtual Network Elements

As *iproute2* offers us total control over a machine's internal network infrastructure, we believe the best way to showcase what we can do is through example. Before getting down to the "command level" of matters, let us establish some naming conventions.

2.4.1 Naming Network Elements

The networks we will work with are rather simple when it comes to the type of elements they are composed of. Even though they can grow quite large, from a network architecture point of view they will reuse the same components over and over. These components

are:

- ***Link-level bridges*:** These *layer 2* devices will forward *link-layer frames* within a subnet based on the destination *MAC address*. These are **transparent** to *layer 3* (i.e. *network*) protocols such as *IP*. These are implemented as a *linux bridge* which, in turn, manifests itself as a network interface. They'll provide the backbone for each of the subnets we are to instantiate as they provide their very own broadcast domain. One can find documentation regarding these bridges on [19].
- ***Network-level routers* and *firewalls*:** These *layer 3* devices will forward packets between subnets based on the destination *IP* address. These will be implemented as *containers* running a regular *Ubuntu* image with minor additions. Given these routers are located between subnets, we will also implement any required firewall functionality within them as well. We will delve deeper into how this can be accomplished in due time, but we can already say that these firewalls are based on *iptables*.
- ***End hosts*:** These are the end systems in the network. They'll be implemented as *docker containers* running a lightly modified Ubuntu image as well.
- ***Veths*:** These are *virtual Ethernet interfaces* which we will use to wire the entire network together. These virtual interfaces can be regarded as "virtual wires" with two ends. Any frame coming into one end comes out the other and vice-versa. We can then conclude they behave exactly like real-world wires. We will then "insert" one end of a *veth* into a network device and the other end into another one to effectively "wire them together".
- ***Network namespace*:** We would like to make it absolutely clear that *namespaces* are **not** network devices per-se: they have no real world counterpart. As we will later see, each *container* spawns its own *namespace*: we have a one-to-one correspondence between hosts/routers and namespaces. This will be crucial when connecting network elements with *veths*, as they will have to be associated to a particular network namespace for things to work.

Now that we have settled the naming scheme we will use, it is time to break down some commands allowing us to instantiate virtual network elements as we please.

2.4.2 Bridges

Just like we stated before, *bridges* are *layer 2* devices. We will find how, due to their operation, no configuration is needed beyond bridge creation. These devices will automatically learn link-level routes as needed whilst in operation. As we are concerned with

the link layer, we will be dealing with physical or *MAC* (Media Access Control) addresses. These are very characteristic in the sense that they are represented as a series of *hexadecimal* digits separated in groups of 2 by colons (`:`). Given the intricacies of a $base - 16$ numbering system, each of these colon-delimited groups is equivalent to a single *byte* of data. Anyhow, an example of a *MAC* address would be: `8c:2d:aa:56:e2:7b`. As a curiosity, we can indicate that different *MAC* prefixes are allocated to manufacturers by the *IEEE*. That means that we can know the manufacturer of a pice of equipment through its *MAC* address (assuming it has not been tampered with). If one were to check the above address, he or she would find it to belong to a device manufactured by *Apple Inc.*. After all, it is the *MAC* of the *iMac* we are writing this document on.

The bottom line to the above is that a bridge's *MAC* forwarding table need not be configured explicitly, be it through a protocol or a human operator. As soon as the bridge is brought up, it'll broadcast an incoming link-layer frame on all of its interfaces but the one the frame arrived on. The key aspect here is that the bridge will include an entry in its forwarding table associating this frame's **source *MAC*** to the interface the frame arrived on. If that original frame's *MAC* address was, say, `01:23:45:67:89:AB` and the frame arrived on port 0, the bridge would automatically learn to send any incoming frame with destination `MAC 01:23:45:67:89:AB` through port 0. After being in operation for some time, the bridge will throttle back the number of frames it broadcasts, thus operating ever more efficiently. An aspect to keep in mind with this way of automatically configuring the forwarding tables is that these entries need some way of being deleted. Just like with learning, this will be an implicit process requiring no explicit signaling whatsoever. The learnt entries will be associated to a timer. When this timer runs out, the route will be assumed to be stale and be deleted from the table. This mechanism allows us to remain true to the real network topology whilst being stable enough in the face of network changes.

We believe the above discussion to be relevant as a way of justifying why we need not be concerned with any configuration whatsoever. The very design of real *bridges* (and thus of its virtual counterparts) takes care of this for us. We will later see how this is not the case for network level routers. We'll need to manually populate the routing tables so that we attain the desired topology.

We would also like to state that we did have to overcome a slight but hard-to-spot problem related to these bridges. It involves a setting controlling whether to involve *iptables* in frame forwarding at each of the virtual bridges. As this interaction has several important implications, such as violating a "pure" layered architecture we will delve deeper into it in section 2.7.

Even though the previous paragraphs might make bridge creation look cumbersome,

it could not be any simpler. Getting a working bridge is a matter of executing the lines found on listing 2.1 from a shell. As listing 2.1 contains commands that ought to be run in a shell we would like to discuss the necessary execution permissions so that someone trying to recreate this work can do so in a satisfactory way.

Capabilities for Network Infrastructure Handling

Unleashing *iproute2's* full potential implies that we can wreck havoc on a system's network infrastructure. We could leave the entire system without network connectivity or alter the way traffic is processed in such a way that we could snoop on user's data, for example. This implies that not everybody using a system should be able to perform these actions.

One of the ideas at the core of any system is *access control*. It lets us control what system users can and cannot do in such a way that we protect the system's stability and security. Each user will have different capabilities to perform actions on the system. One can read on them on [20], but the bottom line is that not every user will be able to use all of *iproute2's* features without them being granted some capabilities. Given the configuration overhead this entails, we opted for issuing restricted commands as the system's *root* user whose *user id* (*UID*) is 0. This implies it bypasses all capability checks: that is, *root* has unrestricted access to *iproute2's* features. On *UNIX*-based systems one can either opt for prepending `sudo` to commands such as the ones on listing 2.1 or just run all of them as *root* directly. The latter can be achieved by running `su -` to switch users. These are by no means the only ways of running the commands we include in this document. One can grant the required capabilities to an arbitrary user to perform the same tasks whilst being more elegant with respect to capability handling. Again, the information found on [20] is a wonderful stepping stone for more complex capability-oriented setups.

We will revisit capabilities when discussing the method of creating *docker containers* later down the road as it's a crucial step for enabling the management of a container's networking infrastructure.

```
1 # Create a new virtual bridge named foo-brd.
2 ip link add foo-brd type bridge
3
4 # Bring it up (i.e. turn it on).
5 ip link set foo-brd up
6
7 # Check the bridge is indeed up.
8 ip link
9
```

```
10 # This should provide an output like the following:
11 # 4: foo-brd: <BROADCAST,MULTICAST,UP,LOWER_UP>\
12     # mtu 1500 qdisc noqueue state UNKNOWN mode DEFAULT
13     # group default qlen\
14     # link/ether 72:8a:77:68:42:b1 brd ff:ff:ff:ff:ff:ff
```

Listing 2.1: Instantiating a Virtual Network Bridge.

We would like to draw the reader's attention to the `LOWER_UP` string found on `ip link`'s output on listing 2.1, as it is letting us know that the newly created *bridge* is indeed active. On top of that, we would also like to note that the backslash (\) characters found on `ip link`'s output are **not** present on the "real" output. We have included them out of necessity, given the output lines were too long to be displayed on a single line in this document. We decided to indicate where we had broken up the line in an effort to be true to the original text whilst making it more visually appealing. This small "tweak" has been applied on other listings such as 2.2.

2.4.3 Veths

Now that we know how to create *bridges* it is time to instantiate the "wires" connecting all of them together [21]. Just like we explained before, these "wires" are supported on *veths*. The creation process is very similar to that of bridges in the sense that we will leverage `ip link`'s functionality. Listing 2.2 shows how these *veths* are created. From *iproute2's* perspective, these *veths* manifest as two different interfaces that are nonetheless very intimately related. Just like we explained before, everything that enters one *veth* end will go out the other and vice-versa. This relation is made clear by how these *veths* are displayed when querying the network configuration as seen on listing 2.2.

```
1 # Create a new veth whose ends are named veth-end-x and
2     # veth-end-y.
3 ip link add veth-end-x type veth peer name veth-end-y
4
5 # Bring BOTH ENDS up.
6 ip link set veth-end-x up
7 ip link set veth-end-y up
8
9 # Check the veth ends are indeed up.
10 ip link
11
12 # This should provide an output like the following:
13 # 5: veth-end-y@veth-end-x: <BROADCAST,MULTICAST,UP,\
14     # LOWER_UP> mtu 1500 qdisc noqueue state UP mode\
15     # DEFAULT group default qlen 1000\
16     # link/ether 5e:b1:40:3f:e7:53 brd ff:ff:ff:ff:ff:ff
```

```
17 # 6: veth-end-x@veth-end-y: <BROADCAST,MULTICAST,UP,\
18     # LOWER_UP> mtu 1500 qdisc noqueue state UP mode\
19     # DEFAULT group default qlen 1000\
20     # link/ether de:16:80:f6:c3:64 brd ff:ff:ff:ff:ff:ff
```

Listing 2.2: Instantiating a Virtual Ethernet Interface.

When looking at listing 2.2 we find that, just like in listing 2.1, the `LOWER_UP` string is telling us that both *veth* ends are up and running. We would also like to bring the reader's attention to the `@veth-end-[x/y]` suffix in both *veth* names. This is just telling us the name of the other *veth* end associated with this one. In other words, `veth-end-y@veth-end-x` tells us that the *veth* whose configuration we are currently inspecting (i.e. `veth-end-y`) is attached to `veth-end-x`. Nonetheless, we need not specify this suffix when referring to the *veth* ends during configuration. Note we didn't include it on lines 5 and 6 on the same listing.

Adding Veths to Network Namespaces

With what we have seen on the previous section we are only capable of creating the *veths* themselves. In order for them to be useful we need to somehow connect them to other network elements so that they are "physically" connected. The process of connecting *veth* depends on what element we are to connect it to. Even though it only takes a single command to do, the connection of *veths* is a more conceptually intricate process that requires us to recall what network namespaces were.

When dealing with *bridges* we just need to issue a simple command as seen on listing 2.3. Said instruction would be the "virtual" equivalent of walking up to a real *bridge* and plugging an *Ethernet* wire in, for instance.

```
1 # Connect veth end veth-end-x to bridge foo-brd
2 ip link set veth-end-x master foo-brd
```

Listing 2.3: Connecting a Veth End to a Virtual Bridge.

Now, connecting a *veth* to a host is a more intricate process. Even though it is true that it only requires a single command, we need be clear about the "conceptual domain" backing said connection up. Doing so requires recalling what *network namespaces* were. In the realm of networking we stated that a network namespace could be regarded as an independent network stack within a single physical machine. Thus, each process that was granted its own would "believe" to have its own, independent network stack. We also teased that each of our network nodes (both hosts and routers) would run as a docker container and that each of them would have an associated namespace. Thus, connecting a *veth* to a host or router is a synonym for making said *veth* a part of that namespace.

This can be accomplished through a single command as seen on listing 2.4. We would like to stress the fact that in order for said command to behave as expected we must ensure *iproute2* can "see" the namespace we are trying to add the *veth* to. This is a topic we will delve into in the following section.

```
1 # Connect veth end veth-end-y to a host whose associated
2     # namespace is host-a-ns
3 ip link set veth-end-y netns host-a-ns
```

Listing 2.4: Connecting a Veth End to a Host or Router.

2.4.4 Managing Network Namespaces

Given the framework we have designed we will need to circumvent some small limitations to leverage the *iproute2* suite. We will explained what these are and how to quench them before explaining how to work with *iproute2* in a "multi-namespace" setup.

Making iproute2 Aware of Container Namespaces

The command we presented on listing 2.4 depends on *iproute2* being aware of the existence of the `host-a-ns` network namespace. This will not be an issue if the namespace itself is created with the same suite of tools. However, this is **not** our case: we are trying to interact with network namespaces our docker containers are managing themselves. This implies that we need to manually perform some actions to let *iproute2* manage these namespaces too.

Working with virtual network infrastructure and namespaces is rather abstract. Thus, commands letting us inspect the current state of affairs are tremendously helpful. In this case we can leverage `ip netns`. This will list all the namespaces the *iproute2* suite is currently aware of (and thus, those it can currently manage and modify). If we *run* a docker container and compare the output of the `ip netns` command before and after doing so we **will not** appreciate any difference. This follows from the fact that, as we stated before, the associated namespace is managed by *docker* itself. After performing the actions we detail on the next paragraph one will be able to check how the output shown by `ip netns` does reflect the newly created namespace.

The key idea to keep in mind is that "a named network namespace is an object at `/var/run/netns/NAME` that can be opened" as stated on [22]. Note that in order to be sure that we are obtaining information relevant for our platform we must make sure this *manpage* is queried on a machine running *Ubuntu* or on a site offering *Ubuntu's manpages*.

All in all, we find that in order for *iproute2* to be aware of network namespaces these must be available at `/var/run/netns`. If we create a network namespace ourselves with `ip netns add foo-ns` we would find that the `ip netns` command now shows `foo-ns` on its output and that there would be a new empty file at `/var/run/netns/foo-ns`. The latter is the "manifestation" of the newly created network namespace on our filesystem. As expected, *running* a *docker* container **does not** place any file under `/var/run/netns` and so *iproute2* cannot manage that namespace. Nonetheless, the fact that a container is associated with its own namespace **implies** a file similar to `foo-ns` must also exist somewhere in the filesystem once the container is active. We only need to find out were it is and link it to `/var/run/netns` so that we can work with the container's namespace.

Even though we will devote a section to the management of containers, we will look a bit into the `docker inspect` command. This instruction lets us query information regarding a particular container. As of now, we are interested in the containers *PID* (Process Identifier; remember a container *virtualizes a process*: it's a single process with its own *PID*). We can easily retrieve said identifier based on the container's name through `docker inspect -f .State.Pid <container_name>`. This *PID* will let us locate the container namespace under the `/proc` directory. The filesystem mounted at `/proc` is by no means a regular one. We will provide a very short description of its purpose and some example uses in the next section.

The procfs Interface Filesystems were initially envisioned to manage data in the form of files and directories. Nonetheless, the mechanisms provided by them can be leveraged to present other types of information. This is just what the *Proc Filesystem (procfs)* accomplishes. It presents information on the system's process and other characteristics in a hierarchical manner akin to how files are organized in a common filesystem such as *ext4*. This in turn provides a ways of communication between the *user* and *kernel* spaces that's leveraged by tools such as *ps* (*GNU's* implementation doesn't use *system calls*, it just queries the *procfs* to obtain the necessary information). As containers are just processes, they also "manifest" under `/proc`. Through a container's *PID* we can locate the relevant information and obtain the empty file granting access to its namespace (this file is located at `/proc/<container_pid>/ns/net` for a given container). This information has been extracted from [23].

In the light of all the above, we can summarize the process of making a container's namespace visible to *iproute2* in 3 steps. We are also including a code snippet on listing 2.5. If we run `ip netns` after carrying these out we must be able to find the container's namespace in the output. We are now ready to modify it as we please.

1. Obtain the container's *PID*.
2. Find the file representing its network namespace under `/proc`.
3. Create a link to said file under `/var/run/netns`.

```
# Find out the PID of container foo-cont
cont_pid=$(docker inspect -f {{.State.Pid}} foo-cont)

# Create the link under /var/run/netns. The namespace will
    # show up as foo-cont when running ip netns. The name
    # we'll find is that of the link we create. It is NOT
    # compulsory to use the container's name, but it
    # does make namespace management easier.
# Options used with ln:
    # -s: Create a symbolic link
    # -f: Force link creation (i.e. overwrite an existing link)
ln -sf /proc/$(cont_pid)/ns/net /var/run/netns/foo-cont

# This can also be accomplished in a single line
# ln -sf /proc/$(docker inspect -f {{.State.Pid}} foo-cont)/ns/net\
    # /var/run/netns/foo-cont
```

Listing 2.5: Linking a Container's Network Namespace to `/var/run/netns`.

Running Commands on Different Namespaces

If we recall listing 2.4, we notice how connecting a *veth* end to a host is equivalent to "moving" that interface to the host's *namespace*. However, we did not delve much deeper into the implications of this action. Once an interface is sent to a *namespace* different than the default one (i.e. the *root namespace*) we'll find that running commands like `ip link` or `ip addr` won't show that interface anymore.

We need to be aware that, even though we don't usually specify it, every `ip XYZ` command is running within a given *namespace*. Up until now, these were being executed on the *default* namespace as we weren't specifying the opposite. We then need to, after sending an interface to a different namespace, somehow instruct subsequent commands to run within that new namespace instead of the default one. This can be easily accomplished thanks to the `-n` flag accepted by commands from the *iproute2* suite. Listing 2.6 contains a series of commands that portray this behaviour. Please note that for the sake of brevity we assume the *veths* created on listing 2.2 still exist.

```
# Create the host-a-ns namespace
ip netns add host-a-ns
```

```
3
4 # Move veth-end-y to the host-a-netns namespace
5 ip link set veth-end-y netns host-a-ns
6
7 # Analyze the current interfaces on the default
8     # namespace
9 ip link
10
11 # This should provide an output like the following:
12 # 6: veth-end-x@if5: <BROADCAST,MULTICAST,UP,\
13     # LOWER_UP> mtu 1500 qdisc noqueue state UP mode\
14     # DEFAULT group default qlen 1000\
15     # link/ether de:16:80:f6:c3:64 brd ff:ff:ff:ff:ff:ff
16
17 # Do the proper thing on the host-a-ns namespace
18 ip -n host-a-ns link
19
20 # This should produce an output in line with the following:
21 # 5: veth-end-y@if6: <BROADCAST,MULTICAST,UP,\
22     # LOWER_UP> mtu 1500 qdisc noqueue state UP mode\
23     # DEFAULT group default qlen 1000\
24     # link/ether 5e:b1:40:3f:e7:53 brd ff:ff:ff:ff:ff:ff
```

Listing 2.6: Running `ip` on a Different Namespace.

Notice how after moving a *veth* ned to a different namespace on listing 2.6, we can no longer "see it" when running `ip link` on the *default namespace*. We would also like to point out that we had to manually create the *host-a-ns namespace* for demonstrative purposes but this **will not** be the case when we are dealing with the framework we have developed. Namespaces will be managed by *docker* in their entirety so we don't have to be concerned about their creation and latter deletion.

2.5 Addressing Layer 3 Network Devices

In previous sections we have covered how to instantiate different virtual network elements. Even though we haven't exactly looked into **how** to leverage *docker* to generate virtual hosts in the form of *containers* we can safely assume that is something we can indeed achieve. What's more, we have discovered how, from the networking point of view it suffices to create a network namespace: we don't need anything else more than that to generate a presence on the network. It is true that a namespace is a lifeless being in the sense that it won't generate any traffic or reply to any request: it only provides the network backbone that the containers will indeed use. Nonetheless, we already have all the information we need to tackle the topic of addressing.

This section is concerned with *layer 3* of the *OSI Model* (i.e. the *network layer*). The network stacks we are dealing with implement *IP* at this level, so when we refer to addressing a host it's equivalent to assigning it an *IP address*. Now, we need to be aware of the fact that we are **not** addressing *hosts*: we are addressing *interfaces* belonging to that host. From an end user's perspective it's quite common for a given network-aware machine to contain a single network interface. The fact that there sometimes exists a one-to-one correspondence between machine and interface **does not imply** that addressing a host is the same as addressing an interface. The clearest example of this would be the way we deal with *layer 3 packet switches* (i.e. *routers*). These will (usually) contain more than one interface where each of them belongs to a different *subnet*. Then, each of them needs a different network address. Now, what's the router's *IP* address? We know it has at least two different addresses, so what's the correct answer? This example shows how the misconceptions surrounding *IP* addressing can be easily dismantled.

In the previous paragraph we found out how we are actually addressing *interfaces*, not *systems* themselves. Once we have cleared the "conceptual" air we will find out how it's rather simple to address interfaces thanks to the `ip addr`. Listing 2.7 shows the process of assigning a given *IP* address to a given interface. This listing assumes the interfaces created on listing 2.2 still exist.

```
1 # Assign the 192.168.1.1/24 address to the veth-end-x veth end. This
    command will automatically
2     # assign a broadcast address based on the provided subnet mask (i.e.
      /24)
3 ip addr add 192.168.1.1/24 brd + dev veth-end-x
4
5 # Verify the changes were applied
6 ip a
7
8 # We should see something like the following:
9 # 5: veth-end-x@if4: <NO-CARRIER,BROADCAST,MULTICAST,\
10    # UP> mtu 1500 qdisc noqueue state LOWERLAYERDOWN\
11    # group default qlen 1000
12    # link/ether de:16:80:f6:c3:64 brd ff:ff:ff:ff:ff:ff\
13    # link-netns host-a-ns
14    # inet 192.168.1.1/24 brd 192.168.1.255 scope global\
15    # veth-end-x
16        # valid_lft forever preferred_lft forever
17    # inet6 fe80::cd4:77ff:fe8a:2ec0/64 scope link
18        # valid_lft forever preferred_lft forever
19
20 # We can of course address interfaces on other namespaces
21    # with the -n flag. The following would assign address
22    # 192.168.1.2 to the veth-end-y present on the
23    # host-a-ns namespace.
```

```
24 ip -n host-a-ns addr add 192.168.1.2/24 brd + dev veth-end-y
```

Listing 2.7: Addressing an Interface.

We would like to mention that assigning addresses like we do on listing 2.7 will also affect the route table of the machine the interface belongs to. We'll delve deeper into this topic when we discuss routing in a later section.

2.6 Adding Firewall Functionalities to Layer 3 Devices

The network topologies we are to instantiate come with several restrictions regarding the connections network elements are allowed to establish. These policies will be enforced at the *network layer* in the different routers we will work with. What's more, we will leverage the *iptables* tool to filter the different datagrams based on predefined criteria. Given *iptables* complexity we believe a short discussion on the tool's architecture is appropriate. Please note that the following is heavily based on the contents of [24].

Before diving into *iptables* we would like to justify our use of the term *packet* in the following paragraphs. Even though we intend to be as technically precise as possible, doing so when discussing a *firewall's* operation can prove to be exasperating. Firewalls are not constrained to a single layer in the *OSI model*: they can match packets based on the source and destination *network addresses* of the incoming *datagram*, the *transport layer* protocol to which the input *segment* belongs to or even the *MAC* address associated with the incoming *frame*. As these rules can leverage information relative to many architectural layers we feel it is not correct to fit it within a single one of them. On the other hand, explicitly differentiating between the term describing a packet based on the criteria we were using at the time would bog the reader time in a myriad of technicalities that we felt wasn't offering a better understanding in return. Thus, we settled on the use of the term packet for the following discussion: after all, this kind of situations is where packet's "ambiguity" excels.

2.6.1 Overview of iptables

Just as its name implies, *iptables* is organized as a set of tables, each containing rules that are to be applied to incoming packets. Depending on the incoming packet's nature, *iptables* will use one table or other. In our case we will only be dealing with the default *filter* table. These tables are further organized into *chains*. The *filter* table contains the

following predefined *chains*:

1. *INPUT*: Its rules are applied to packets destined for **local sockets**.
2. *OUTPUT*: Its rules are applied to **locally generated packets**.
3. *FORWARD*: Its rules are applied to **packets being routed through the machine**.

A table may contain user defined chains as well: if there is something characterizing *iptables* it would be its versatility.

Even though the rules we'll be needing are quite simple, these can get extremely convolved. Rules are composed by a certain *criteria* and a *target*. The criteria will be applied to each incoming packet and, if it matches it, the target will be made effective. *iptables* applies rules within a chain in a sequential manner: if a rule doesn't match a packet the next one will be applied until there are no more rules left in the chain. If that's the case, the chains *default policy* will be enforced. We comment a bit more on these policies on a later paragraph. We should end by listing the two *targets* we will be using in our rules:

1. *DROP*: According to [24] this target "*DROPS* the packet to the floor". That's not "accurate" in the sense that nothing goes to the floor really, but the effect is the same: the packet itself will be **discarded**.
2. *ACCEPT*: This target will cause the packet to the let through.

One can query the current rules at any time by issuing the `iptables -L` command or the more specific `iptables -L <chain-name>` to query the rules for a particular chain. These commands can always be combined with the `-t <table-name>` option, should we want to work with a table other than *filter*. An example would be the *nat* table, used for packets creating a connection (such as a *TCP SYN* segment).

As we hinted before, these chains will also have an associated *policy*. The way *iptables* work is that it will try to apply the most restrictive rule for a given packet. If there is no such a rule, it will fall back to the *chain's policy*. End user systems usually have a default policy *DROP*ping packets that are to be forwarded, for instance. This follows from the fact that a normal host should not be acting as a *layer 3* router in any case. This can of course be altered thanks to the `-P` flag that we can use with *iptables*. Thus, running `iptables -P FORWARD ACCEPT` would allow any packets traversing the host through. We then find how the logic behind firewall rules shifts depending on the policy applied to a

given chain. If the default policy *ACCEPT*s packets, we'll instantiate more specific rules to *DROP* some of them. On the other hand, if the configured policy *DROP*s packets, we'll then add rules letting some of them through.

With this background information, we can now analyze the syntax of the rules we will be instantiating on the *layer 3 routers* that act as firewalls that we will use in our virtual topologies.

2.6.2 Instantiating iptables Rules

When discussing the main architecture of *iptables* in the previous section we stated that rules within a chain are *sequentially* applied to a given packet. This implies that the **order** we add these rules in is crucial: it will determine the fate of the packet. Even though every situation differs from one another, the general rule says that rules are to be added in such a way that "generality increases with each rule". In other words, the rules should get more specific as we add them.

Our scenario is a bit simpler however: none of our rules **overlap**. By overlapping we mean that, given the criteria for matching packets we will be using, only a **single rule within a chain** will be applied to a given packet. This implies that, **in our particular case**, the order we instantiate the rules in is not relevant. What's more, we'll only discriminate packets according to the destination and source *IP addresses*. This allows us to state that we'll leverage *iptables* as a *layer 3 firewall* that's in charge of filtering *datagrams*.

We showcase how to add rules to a given chain in listing 2.8. We have added comments within the listing explaining the effect of the different options. The following enumeration puts the rules' effects into words:

1. Accept traffic egressing from the host with *IP 10.0.0.3* and destined to the host with *IP 10.0.5.4* no matter the *transport layer* protocol [1] in use.
2. Stop traffic egressing from hosts other than *10.0.0.3* and destined to host *10.0.5.4*. As it's not specified, this rule applies to every *transport layer protocol* too.

```
1 # Rule 1:
2     # -I FORWARD: Insert the rule at the beginning of the
3         # FORWARD chain.
```

[1] *ICMP* is also considered a transport layer protocol in this case despite not "fully qualifying" as one (it is not associated with a port number).

```
4     # -j ACCEPT: Matching packets will be accepted.
5     # -p all: Match any transport layer protocol
6         # (including ICMP).
7     # -s 10.0.0.3: Match packets with a source IP of
8         # 10.0.0.3.
9     # -d 10.0.5.4: Match packets with a destination
10        # IP of 10.0.5.4.
11 iptables -I FORWARD -j ACCEPT -p all -s 10.0.0.3 -d 10.0.5.4
12
13 # Rule 2:
14    # -A FORWARD: Append the rule at the end of the
15        # FORWARD chain.
16    # -j ACCEPT: Matching packets will be accepted.
17    # ! -s 10.0.0.3: Match packets whose source IP is
18        # NOT 10.0.0.3.
19    # -d 10.0.5.4: Match packets whose destination IP
20        # is 10.5.0.4.
21 iptables -A DROP -j ACCEPT ! -s 10.0.0.3 -d 10.0.5.4
```

Listing 2.8: Instantiating *iptables* Rules.

With what we have discussed in this section we can tackle the instantiation of the pertinent firewall rules when they are needed.

2.6.3 A Foreword on Container Capabilities

We usually assume the *root* user can perform any actions on a given system. However, this is not "entirely true" for docker containers. We will need to manually grant them the *NET_ADMIN* capability so that these firewall rules can indeed be applied. We will look into *capabilities* in more detail when discussing containers, but we wanted to point out that **this** is the only configuration aspect that requires something other than the defaults.

2.7 A Small Caveat: Debugging Connectivity Issues in Virtual Bridges

When we began exploring and testing small virtual network models such as the one we showcase on 3 we stumbled upon a disconcerting issue. Thanks to WireShark [25] we managed to analyze the packet traces at each of the bridges in an effort to correct the connectivity problems we were facing. It is important to note that the virtual bridges **remain in the root namespace**. Remember we are only moving *veths* to the container's network namespace. The other ends will always be added to a virtual bridge that's run-

ning within the default namespace belonging to the host.

Our analysis showed how *ethernet frames* were indeed reaching the bridges, **but** they weren't egressing from them. This situation is rather troubling in the sense that nobody really expects a firewall to be effective in a *layer 2* device. After looking around we stumbled with the *bridge-nf-call-iptables* system option. This option, which is enabled by default, forces frames traversing virtual bridges to be "filtered through" *iptables*. If one scrutinizes *iptables' manpage* he or she will find the *physdev match extension*. This module comes with the *–physdev-is-bridged* option that would allow a user to configure *iptables* in such a way that every frame being switched within virtual devices is allowed through. Just like the original writer of [26], we find this to be tremendously counter intuitive.

Our approach was a bit harsher than configuring the host's *iptables* implementation: we altogether disabled the filtering of *ethernet frames* traversing the virtual bridges through *iptables*. This can easily be accomplished through the *procfs* interface with the command shown on listing 2.9. Said listing also contains an option allowing for the permanent change of this setting within a system. We nonetheless chose the former approach so that changes can be reverted in a "worst-case scenario" by rebooting the host running the virtual network infrastructure.

```
1 # Disable bridge calls to ipatbles through the procfs
2     # interface. We leverage tee to work around shell
3     # redirection characters such as > not having
4     # elevated privileges when echo is run with sudo.
5 echo 0 | sudo tee \
6     /proc/sys/net/bridge/bridge-nf-call-iptables > /dev/null
7
8 # The above is equivalent to running:
9 sysctl -w net.bridge.bridge-nf-call-iptables=0
10
11 # Make a permanent change to /etc/sysctl.conf and apply it
12 echo "net.bridge.bridge-nf-call-iptables = 0" | \
13     sudo tee -a /etc/sysctl.conf
14 sysctl -p
```

Listing 2.9: Disabling Bridge Calls to *iptables*.

Our tool will make sure this kernel feature is disabled before instantiating any virtual network devices.

Command	Description
`docker ps -a`	List the status of all existing containers.
`docker build ...`	Build an image from a *Dockerfile*.
`docker exec ...`	Execute a command within a **running** container.
`docker run ...`	Create and run a new container.
`docker start ...`	Start an existing stopped container.
`docker stop ...`	Stop a running container.
`docker rm ...`	Remove an stopped container.

Table 2.2: Basic *docker* Commands.

2.8 The Containers

We have previously stated that a container can be regarded as the virtualization of a process. An initial discussion about containers, what they are and what they are not can be found on section 1.2.1. This section "fills in the holes" left by the aforementioned one.

2.8.1 Managing Docker

We have chosen to leverage docker's *CLI* (Command Line Interface) to manage all the containers we will be dealing with. The set of tools that's offered to us can also be used to query the currently active containers, build images and query container logs among other actions. We are including a non-comprehensive list of commands we will commonly use on table 2.2 so that it can be used as a reference for the rest of the section.

Commands listed on table 2.2 will be expanded on as we work our way through this section.

2.8.2 Container Images

The application a docker container runs together with its dependencies is packed into an *image*. Then, *a docker container runs an image*. We can once again leverage the *class-instance* concept soaking the object oriented programming paradigm and regard the *images* as the container classes and the running containers themselves as instances of the *image* they are running. In other words, the images are run as containers within what we call the *Docker Engine* [27]. The fact that this Docker Engine is abstracting us from the underlying operating system is what makes containers run in exactly the same way across machines.

Images themselves need to be built before a container can run them. The process docker is to follow to build a given image is specified in a *Dockerfile*. We are including the one we use for building the images for our routers on listing 2.10. It is rather common to build and image *based on* a preexisting one. That is exactly what we do: we use a “vanilla” (i.e. plain) *Ubuntu image* and then tweak it to our needs as can be seen on listing 2.10. Even though it is not our case, it is worth mentioning one can use the *docker pull* command to download prebuilt images from sites such as Docker Hub [28].

```
1  # Pull a "vanilla" Ubuntu image.
2  FROM ubuntu
3
4  # Install the required dependencies:
5      # iputils-ping -> Install the ping command both for
6          # testing and demonstration purposes.
7      # openssh-server -> Allow incoming SSH connections.
8          # The client is installed by default.
9      # iptables -> Allow turning the routers into firewalls.
10     # daemonize -> Turns any program (ping in our case)
11         # into a daemon.
12 RUN \
13     apt-get update && \
14     apt-get install -y iputils-ping && \
15     apt-get install -y openssh-server && \
16     apt-get install -y iptables && \
17     apt-get install -y daemonize && \
18
19     # Make the /run/sshd directory needed by the SSH daemon.
20     mkdir /run/sshd && \
21
22     # Allow others to log in as root into this machine.
23         # Note the default user is indeed root him/herself.
24     echo "PermitRootLogin yes" >> /etc/ssh/sshd_config
25
26 # Set root's password to 1234.
27     # echo's -e option allows the use of escape sequences (\n).
28 RUN ["/bin/bash", "-c", "echo -e '1234\n1234' | passwd root"]
29
30 # Copy necessary files into the container.
31 ADD moving_adjustments.sh /moving_adjustments.sh
32
33 # Set root's home directory.
34 ENV HOME /root
35
36 # Default command at startup (i.e. run the SSH daemon).
37     # The -D option prevents sshd from daemonizing.
```

```
38 CMD ["/usr/sbin/sshd", "-D"]
```

Listing 2.10: *Dockerfile* for our Virtual Routers.

Building the Images

As seen on the second command on table 2.2, we need to use the *docker build* command. We just need to be aware of two arguments.

The `-t` option allows us to **name** the built image. This is the name we will later use when selecting this image to be run within a new container. We can optionally **tag** this image if we use the *name:tag* syntax with the `-t` option. This is handful for building several different versions of the same image. We, however, haven't made use of this feature.

We also need to specify the path to the directory containing the *Dockerfile* itself. This is commonly specified as `.` (i.e. the *working directory*), which implies we are running `docker build` within the same directory our *Dockerfile* is in. We will, however, run the command from a directory other than the one holding the *Dockerfile*, which calls for the explicit path to the *Dockerfile* being passed as an argument to the `-f` option.

All in all, our command is shown on listing 2.11. The images our containers will run are:

- *ubuntu_node*: Image run by regular virtual end systems (i.e. hosts).
- *ubuntu_router*: Image run by our virtual routers. This image is built from the *Dockerfile* displayed on listing 2.10.

```
1 # This command is to be run form the directory containig
2     # any files we are ADDing or COPYing to within the
3     # Dockerfile.
4 docker image build -t <image-name> -f /path/to/Dockerfile .
```

Listing 2.11: Building an Image from a *Dockerfile*.

Managing Built Images

Once images are built we can leverage the *docker images* command to list the ones currently available to us. If everything went well, the image we built with the command shown on listing 2.11 should show up in this list identified by the name provided to the `-t` option. This command's output will also show the image's *ID*: an hexadecimal string

univocally identifying it. This can often be used instead of the image's name.

We can also use the `docker rmi` command to delete a built image. We can specify the image we want to remove through either its name or id.

2.8.3 Managing Containers

A Container's Lifecycle

Understanding the different states in a container's lifespan is the key to managing them. The following enumeration walks the reader through the natural steps a container will follow: from its creation to its removal. The same information is also contained on figure 2.2 for the more visual readers.

1. A container is *created* with an associated image.
2. An existing container is *started* so that it begins running a program.
3. A container can be *run*: this will just *create* and *start* it.
4. A currently running container's state can be altered in several ways:
    (a) A running container can be *paused* and then *unpaused*.
    (b) A running container can be *stopped* or *killed*: this will effectively stop the program running within it.
    (c) A running container can be *restarted*.
    (d) A running container's associated program can also exit, thus *killing* the container.
    (e) A running container can also exhaust the resources it's been granted, effectively *dying*.
5. A stopped or created container can *removed*. This step is non-reversible: the container's data will be deleted with it.

We feel it is extremely important to draw the reader's attention to entry 4d in the enumeration describing a container's lifecycle. The fact that the exit of a container's main process effectively stops it implies that we can indeed stop containers through means not depending on the `docker stop` command. Given we are running an *ssh* daemon within the containers, we can log in as *root* and then issue `kill 1` to effectively stop the container. After all, `docker stop` will send the *SIGTERM* signal to the process running

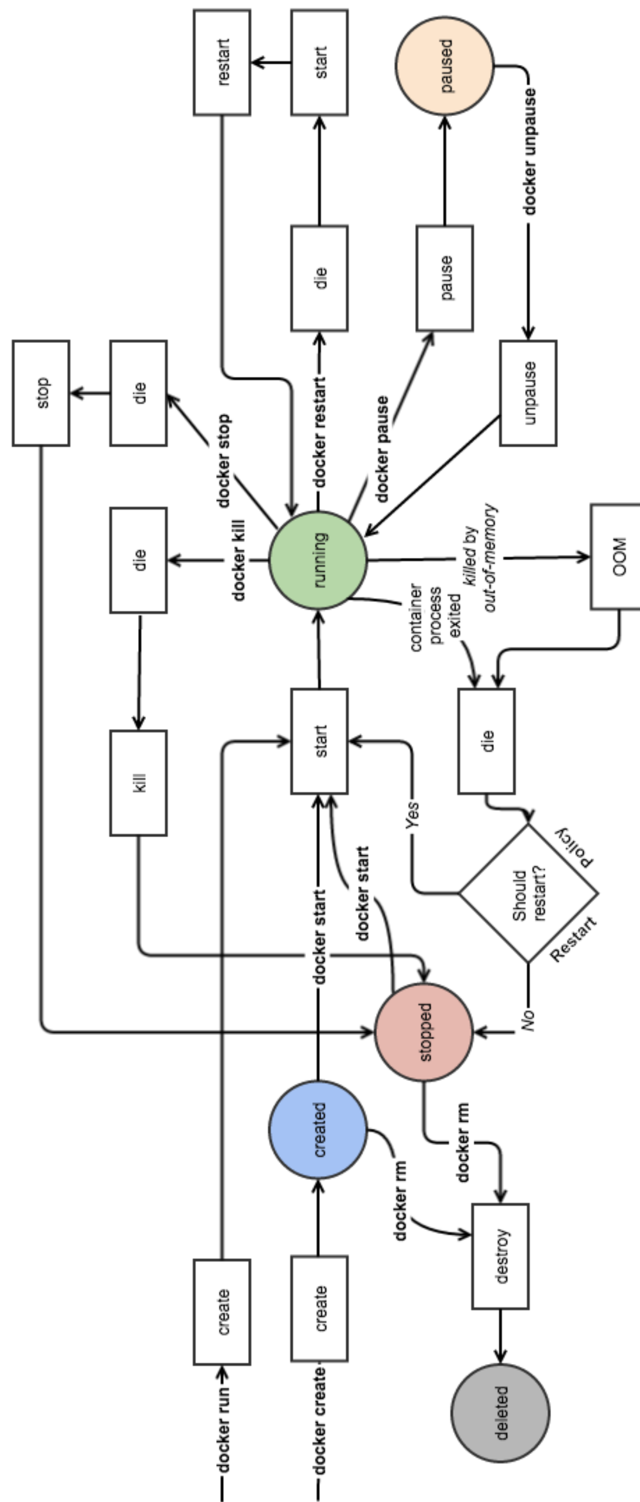


Figure 2.2: Image portraying a *container's* lifecycle. [29]

within the container and, should it not behave appropriately, *SIGKILL* after a configurable delay. Even though this subtlety will not impact the use we make of the virtual environments we are to work with it is a crucial aspect to bear in minf to fully understand the lifecylce of containers.

Container Capabilities

Due to the actions we want to perform inside our containers we need to provide them with some capabilities beyond the default ones. In the case of the regular nodes, we need to provide the *SYS_ADMIN* capability so that they can change their own hostname with the `hostname` command. Routers will also need to be granted the *NET_ADMIN* capability so that they can configure their own *iptables* rules. These capabilities are granted to containers when they are run thorough the `--cap-add` option.

What is more, we need to allow the routers to forward packets through them. Besides instantiating the proper rules in *iptables* we also need to allow the *kernel* to process these packets. We can do so through the `--sysctl` option when running the container, thus enabling *ip forwarding* with the `net.ipv4.ip_forward=1` parameter. This would be equivalent to running `echo 1 > /proc/sys/net/ipv4/ip_forward` within an active container. The catch is that `/proc/sys` is mounted as *read-only* within containers. This would imply that we need to grant many more capabilities to said container so that we could run `mount -o remount rw /proc/sys` and then overwrite the option manually. We believe it is more elegant to just configure the options we need when starting the container and that is what we willl do.

In any case, these options can be seen in the commands shown on listing 2.12.

Docker's Internal Networks

The avid reader might have noticed how, after installing *docker* on an end system the output of the `ip addr` command within the machine has a different output: the *docker0* interfce is added as a gateway for the containers allowing them to reach the "outside world". What's more, after installing *docker* the *iptables* rules on the machine change as well. Running `iptables -L` shows a myriad of new *chains* with names such as *DOCKER-USER* and things along the lines of *DOCKER-ISOLATION-STAGE-**. This approach to networking does simplify the issues one might run into when trying to make a "normal" use of a network, that is, when someone wants to either isolate a container from the external network or provide outwards connectivity. This is by no means our endeavour: we want to work with arbitrarily complex network topologies.

In the realm of technology one often finds that simplifying the operation of a given software system implies making assumptions and avoiding exposing configuration parameters to the end user. Whilst this is a totally reasonable approach it is **not** one suited to us. This justifies the fact that we discarded the preexisting *docker network* model. This translates into us passing the `--network none` option to commands such as `docker run`, for instance. Just like when we discussed the role *kubernetes* plays when orchestrating containers, we feel it is pointless to use a tool only to go against it time and time again.

Our Approach

Our framework makes extensive use of the `docker run` command to both *create* and *start* the containers with the same action. The network topologies we are to set up contain a myriad of nodes, so allocating a *tty* for each of them makes handling things tremendously unwieldy. That is why we will always use the `-d` option, thus *detaching* the container. This translates into *docker* instantaneously returning control of the shell session we are currently using as soon as the container itself is up and running. What's more, we will also run the container with **no** network connections: remember we will manually set the entire networking infrastructure up. The commands we use to run both types of images, that for routers and nodes, is shown on listing 2.12.

```
1 # Running regular network nodes.
2 docker run -d --name <node-name> --network --cap-add SYS_ADMIN none
    ubuntu_node
3
4 # Running routers.
5 docker run -d --name <router-name> --network none --cap-add SYS_ADMIN --
    cap-add NET_ADMIN --sysctl net.ipv4.ip_forward=1 ubuntu_router
```

Listing 2.12: Running Network Nodes.

Running the container in the background poses the question of how we are to "get inside the container" to act as a regular user. In other words, how can we log into a *detached* container? Given *docker's* architecture, and the fact that we are running an *SSH* daemon as the container's main process, we have two options to attain this result.

Executing Commands Within the Container As shown in table 2.2, we can always use the `docker exec` command to execute a command within a running container. If what we want to do is start a *bash* shell within the container (Ubuntu's default shell) we can just execute the command found on listing 2.13. We can also take a look at the processes running within the container to confirm how *sshd's PID* is indeed 1 (it was the

first process to spawn in the container). Aside from that process, we will only see the current *shell* session together with the *ps* program too. This approach can of course be leveraged to run single-non interactive commands within the containers. We could for instance run `docker exec <container-name> ps ax` to list a containers processes, for instance.

```
1  # Spawning our test container.
2  docker run -d --name test_node ubuntu_node
3
4  # Running a shell within it:
5      # -i: Interactive command (i.e. keep STDIN open).
6      # -t: Allocate a pseudo-TTY (i.e. a software
7          # emulated terminal).
8  docker exec -it test_node bash
9
10 # Listing the existing processes:
11     # a: Show all processes with an allocated TTY.
12     # x: Include processes without an allocated TTY.
13     # These two options effectively select every process.
14 ps ax
15
16 # Sample output:
17     # PID TTY       STAT   TIME COMMAND
18     # 1   ?         Ss     0:00 sshd: /usr/sbin/sshd -D \
19         # [listener] 0 of 10-100 startups
20     # 13  pts/0     Ss     0:00 bash
21     # 23  pts/0     R+     0:00 ps ax
```

Listing 2.13: Inspecting a Container's Processes.

Logging in Through SSH We previously explained why our containers were going to run an *ssh* daemon as their main process. This not only "keeps them alive", but it also allows external connections to be made from the outside world. Thus, once we know the *IP* address associated to a given container we can log into it from the host running *docker* as long as we do have connectivity with the container itself. If we spawn a container using the same command as the one shown in listing 2.13 we can verify how it has been given an *IP* by the docker daemon with the command found on listing 2.14. Then, we can just run `ssh root@<container-ip>` to log into that machine as we would normally do in any other system. The password is, of course, *1234* as specified in the *Dockerfiles* we used to build the images these containers will be running.

This aspect shows one of the great strengths of our approach when instantiating the entire networking infrastructure ourselves: we can choose whether to isolate the entire virtual scenario from the outside or not. This would permit or forbid an external user to

log into our containers for instance. All we would have to do is add a *veth* connecting one of the virtual routers and the machine running the containers: as simple as that.

```
1 docker inspect -f {{.NetworkSettings.IPAddress}}\
2     <container-name>
```

Listing 2.14: Checking a Container's *IP* Address.

Attaching to the Container We can theoretically leverage a third option: *attaching* a terminal session to the running container. This will **not** behave as we would expect however. In our case, attaching a shell to our container would make us "see" the *ssh* daemon we are running as the container's initial process. This would't allow us to perform any actions whatsoever: we wouldn't be attaching ourselves to an interactive process, so it is of very little use. We nonetheless felt it was worthwhile to mention this option did indeed exists: it might prove to be useful under different circumstances.

The framework for instantiating virtual networks leverages the different technologies we have discussed throughout this chapter. The next one shows a "toy example" demonstrating the use of all of them to bring up a simple topology comprising two containers and a bridge mediating between them.

Chapter 3

Manually Bringing Up a Simple Network

> If the vendors started doing everything right, we would be out of a job. Let's hear it for OSI and X! With those babies in the wings, we can count on being employed until we drop, or get smart and switch to gardening, paper folding, or something.
>
> *C. Philip Wood*

Figure 3.1 showcases a simple topology we will instantiate by applying the contents described in chapter 2. Instead of relying on our framework to perform the task, we will manually carry out all the needed steps and detail them in a code listing. This demonstrates how are tool is nothing more than an additional "layer" in charge of managing the subtleties we will manually deal with once the topologies' complexity begins to increase.

3.1 A Note on the Chosen Private IP Range

As seen on section 3 of [30] we could have chosen private addresses within one of 3 private network address spaces. Doing so guarantees we will not run into any collisions should we want to let our virtual hosts and routers communicate with the public Internet. We should nonetheless be aware of the fact that, if we are pursuing a completely isolated virtual network we need not adhere to these reserved address ranges: no collisions could probably occur.

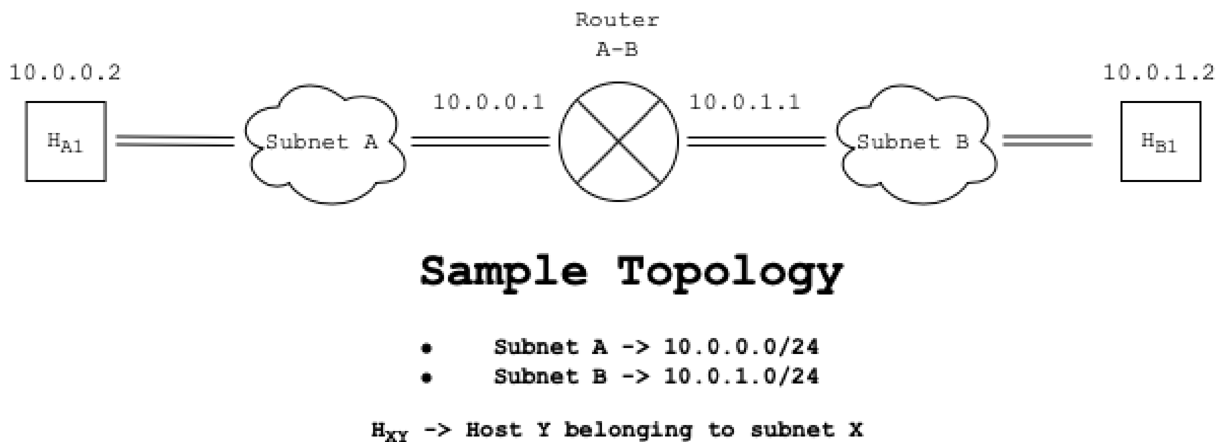


Figure 3.1: The Sample Topology We Will Set Up Throughout Chapter 3.

Given we did not have a firm reason not to choose one of the reserved private address spaces, we decided to make use of them. As our home network uses the *192.168.1.0/24* range and, in our experience, the *172.16.0.0./12* space tends to be more common, we decided to assign addresses within the *10.0.0.0/8* pool. These are the ones that will be included in any figures and listings that need to explicitly include IP addresses.

3.2 Automating the Sample Topology

Listing 3.2 contains the shell script automating the deployment of the network showcased on figure 3.1. The script itself contains messages that are to be printed to the user which clarify its operation to a great extent.

3.2.1 Running the Script

As seen on *line 1* of listing 3.2 this script is to be tun within a *bash* shell. This can be accomplished in one of two ways as shown on 3.1. As we will later look into, the script is to be run by *root* so that it can carry out the needed operations. This amounts to prepending the command we choose to run the script by `sudo`.

```
1 # Making the script executable and running it.
2 chmod +x <script-name> && sudo ./<script-name>
```

```
3
4 # Spawning a privileged shell and running the script in it.
5 sudo bash <script-name>
```

Listing 3.1: Running a *bash* Script.

3.2.2 Checking the Script is Run by root

The privileged user in *UNIX*-based systems is identified by a *UID* (User ID) of 0. When a program is run, the associated process will have the *UID* of the user who launched it. Some program's permissions allow it to run with a different *UID* than that of the user executing it: this is wha we call the *EUID* (Effective *UID*)). In any case, if the *EUID* of a process evaluates to 0 it is being run by *root*. This is what is being checked by *lines 3-7* on listing 3.2. If the user running the script is not *root* we will simply abort execution whilst warning the user.

3.2.3 Automatically Tearing Down the Topology

As seen in *lines 9-17*, the script shown on listing 3.2 is prepared to dismantle the network topology it brings up when invoked a second time. The trigger for this behaviour is passing a parameter when running the script. Any parameter will cause the removal of the entire virtual network: this is not the best practice when writing code but it simplifies handling the arguments passed to the script. We are concerned with clearly portraying the technologies discussed in chapter 2, not with teaching the user how to write proper shell scripts. In any case, running `sudo bash <script-name> quit` will dismantle the sample virtual topology.

```
1 #!/bin/bash
2
3 if [ $EUID -ne 0 ]
4 then
5     echo "Run me as root..."
6     exit
7 fi
8
9 if [ $# -eq 1 ]
10 then
11    echo "Tearing down the sample network..."
12    echo -e "\t(If one didn't exist before this script will silently
     fail)."
13    docker rm $(docker stop h-a-1 h-b-1 r-a-b 2>/dev/null) > /dev/null
     2>&1
14    ip link del subnet-a-brd 2>/dev/null
```

```
15     ip link del subnet-b-brd 2>/dev/null
16     exit
17 fi
18
19 echo "Setting up the topology seen on figure 3.1\n"
20 echo -e "\tDisabling bridge calls to iptables (section 2.7)"
21 echo 0 > /proc/sys/net/bridge/bridge-nf-call-iptables
22
23 echo -e "\tEnabling IP forwarding within the host\n"
24 echo 0 > /proc/sys/net/ipv4/ip_forward
25
26 echo -e "\tSetting up the bridges"
27 echo -e "\t\tSetting up subnet A's bridge"
28 ip link add subnet-a-brd type bridge
29 ip link set subnet-a-brd up
30
31 echo -e "\t\tSetting up subnet B's bridge\n"
32 ip link add subnet-b-brd type bridge
33 ip link set subnet-b-brd up
34
35 echo -e "\tSpawning the hosts"
36 echo -e "\t\tSpawning H-A-1"
37 docker run -d --name h-a-1 --network none --cap-add SYS_ADMIN
       ubuntu_node > /dev/null
38
39 echo -e "\t\t\tLinking its network namespace to /var/run/netns"
40 ln -sf /proc/$(docker inspect -f {{.State.Pid}} h-a-1)/ns/net /var/run/
       netns/h-a-1
41
42 echo -e "\t\t\tSetting the hostname"
43 docker exec h-a-1 hostname h-a-1
44
45 echo -e "\t\tSpawning H-B-1"
46 docker run -d --name h-b-1 --network none --cap-add SYS_ADMIN
       ubuntu_node > /dev/null
47
48 echo -e "\t\t\tLinking its network namespace to /var/run/netns"
49 ln -sf /proc/$(docker inspect -f {{.State.Pid}} h-b-1)/ns/net /var/run/
       netns/h-b-1
50
51 echo -e "\t\t\tSetting the hostname"
52 docker exec h-b-1 hostname h-b-1
53
54 echo -e "\t\tSpawning Router-A-B"
55 docker run -d --name r-a-b --network none --sysctl net.ipv4.ip_forward=1
        --cap-add NET_ADMIN --cap-add SYS_ADMIN ubuntu_router > /dev/null
56
57 echo -e "\t\t\tLinking its network namespace to /var/run/netns\n"
```

```
58 ln -sf /proc/$(docker inspect -f {{.State.Pid}} r-a-b)/ns/net /var/run/
     netns/r-a-b
59
60 echo -e "\t\t\tSetting the hostname"
61 docker exec r-a-b hostname r-a-b
62
63 echo -e "\tSetting up necessary veths"
64 echo -e "\t\tSetting up veth H-A-1 <--> Subnet A Bridge"
65 ip link add veth-h-a-1 type veth peer name veth-brd-h-a-1
66 ip link set veth-h-a-1 netns h-a-1
67 ip link set veth-brd-h-a-1 master subnet-a-brd
68 ip -n h-a-1 link set veth-h-a-1 up
69 ip link set veth-brd-h-a-1 up
70
71 echo -e "\t\tSetting up veth Subnet A Bridge <--> Router-A-B"
72 ip link add veth-r-a-b-a type veth peer name brd-r-a-b-a
73 ip link set veth-r-a-b-a netns r-a-b
74 ip link set brd-r-a-b-a master subnet-a-brd
75 ip -n r-a-b link set veth-r-a-b-a up
76 ip link set brd-r-a-b-a up
77
78 echo -e "\t\tSetting up veth H-B-1 <--> Subnet B Bridge"
79 ip link add veth-h-b-1 type veth peer name veth-brd-h-b-1
80 ip link set veth-h-b-1 netns h-b-1
81 ip link set veth-brd-h-b-1 master subnet-b-brd
82 ip -n h-b-1 link set veth-h-b-1 up
83 ip link set veth-brd-h-b-1 up
84
85 echo -e "\t\tSetting up veth Subnet B Bridge <--> Router-A-B\n"
86 ip link add veth-r-a-b-b type veth peer name brd-r-a-b-b
87 ip link set veth-r-a-b-b netns r-a-b
88 ip link set brd-r-a-b-b master subnet-b-brd
89 ip -n r-a-b link set veth-r-a-b-b up
90 ip link set brd-r-a-b-b up
91
92 echo -e "\tAddressing the interfaces"
93 echo -e "\t\tAddressing H-A-1"
94 ip -n h-a-1 addr add 10.0.0.2/24 brd + dev veth-h-a-1
95
96 echo -e "\t\tAddressing Router-A-B's interface on subnet A"
97 ip -n r-a-b addr add 10.0.0.1/24 brd + dev veth-r-a-b-a
98
99 echo -e "\t\tAddressing H-B-1"
100 ip -n h-b-1 addr add 10.0.1.2/24 brd + dev veth-h-b-1
101
102 echo -e "\t\tAddressing Router-A-B's interface on subnet A\n"
103 ip -n r-a-b addr add 10.0.1.1/24 brd + dev veth-r-a-b-b
104
105 echo -e "\tAdding necessary routes to hosts"
```

```
106 echo -e "\t\tAdding default route to H-A-1 through Router-A-B"
107 ip -n h-a-1 route add default via 10.0.0.1
108
109 echo -e "\t\tAdding default route to H-B-1 through Router-A-B\n"
110 ip -n h-b-1 route add default via 10.0.1.1
111
112 echo "Done setting up the network!"
113 exit
```

Listing 3.2: Automatic Deployment of the Sample Topology.

3.3 Testing the Topology Is Working

Once the script shown on listing 3.2 has been run we should have a full-fledged virtual network at our disposal. The simplest way to check we do have full connectivity is checking one of the hosts can "see the other". we can accomplish our objective by running a shell within one of the hosts and then trying to *ping* the other. We can extract the appropriate *IP* addresses from figure 3.1. Listing 3.3 shows how we can open up a shell within host H_{A_1} and launch *ping* against host H_{B_1}.

```
1  # Spawn a shell within H-A-1
2  docker exec -it h-a-1 bash
3
4  # Ping H-B-1 3 times from H-A-1
5  root@h-a-1:/$ ping -c 3 10.0.1.2
6
7  # It should produce an output similar to:
8      # PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data.
9      # 64 bytes from 10.0.1.2: icmp_seq=1 ttl=63 time=0.036 ms
10     # 64 bytes from 10.0.1.2: icmp_seq=2 ttl=63 time=0.057 ms
11     # 64 bytes from 10.0.1.2: icmp_seq=3 ttl=63 time=0.062 ms
12     #
13     # --- 10.0.1.2 ping statistics ---
14     # 3 packets transmitted, 3 received, 0% packet loss,\
15         # time 2049ms
16     # rtt min/avg/max/mdev = 0.036/0.051/0.062/0.011 ms
17
18 # We can also run traceroute
19 root@h-a-1:/$ traceroute 10.0.1.2
20
21 # It should produce an output similar to:
22     # traceroute to 10.0.1.2 (10.0.1.2), 30 hops max, 60\
23         # byte packets
24     # 1  10.0.0.1 (10.0.0.1)  0.029 ms  0.039 ms  0.007 ms
```

```
25      # 2  10.0.1.2 (10.0.1.2)  0.018 ms  0.023 ms  0.041 ms
```

Listing 3.3: Testing the Sample Topology.

Now that we have shown how to "manually" set up a simple network we begin to discover the challenges that will surely arise as their complexity grows. What is more, we also need to add additional functionalities such as moving end nodes around the network in a way that does not mangle the network's operation. The next chapter is devoted to providing a high level overview of our tool's design and operation together with a comprehensive collection of the actions it can and cannot perform.

Chapter 4

Automating the Deployment of Virtual Networks

> One of my most productive days was throwing away 1,000 lines of code.
>
> *Ken Thompson*

4.1 High Level Overview

As seen in chapter 3, bringing a virtual network up entails an organizational overhead that is not easily handled. That is why we have developed a complete software system capable of handling these intricacies in an automatic fashion. Then, a user need only provide a *graph* describing the desired topology and our project will be able to read, interpret and instantiate said network.

Due to its simple yet rich syntax, we have decided to leverage the *Python* [4] programming language to develop the entire system. We will be using version *3.x* given *python's 2.7* release has been deprecated as of *January 2021*. One of the external dependencies we will make use of is the *NetworkX* [31] network analysis module. This software bundle was recommended by the research group we have collaborated with and it is distributed as a *python package*. Thus, we felt even more inclined towards *python*. Given we will interact with the *docker engine* through its *CLI API*, its use will not impose any restrictions on our choice either. This implies that we have nothing more than reasons supporting the use of *python* for our development.

4.1.1 External Dependencies

One of the main objectives pursued throughout the development was reducing the number of external dependencies to the maximum extent. We did manage to only require the presence of *docker*, *iproute2* and *python3* for an initial and fully functional version. Not leveraging *NetworkX* implied we had to manually route all the nodes within the network, which amounted to be a rather complex task. Due to the research group's suggestions we settled on taking advantage of *NetworkX* for both modeling the different topologies and routing them. Then, we have designed two independent solutions that accomplish the same task. One of them **does not** require *NetworkX* whist the other one **does**. The reasoning behind including a dependency that is not strictly needed is that it greatly simplifies our code and it will surely avoid some of the most common pitfalls our own solution can incur into under complex circumstances.

The following enumeration briefly explains the use of each of the required dependencies. Please bear in mind that the installation instructions for each of them are detailed in the document's appendix.

1. ***Python3*****:** With *python3* being an interpreted language, we need to make use of the interpreter that is going to execute our code.

2. ***Docker*****:** The different nodes in our network are modeled as *docker containers*, which implies we indeed depend on *docker* for making our system work.

3. ***iproute2*****:** We need the *iproute2* suite of tools to manage the virtual networking infrastructure "gluing" all our nodes together.

4. ***NetworkX*****:** As explained above, we *are not forced* to use *NetworkX* and we have developed a version that does not depend on it at all. Nonetheless, it does simplify big portions of code and so we decided to include it in our final, sharper version.

5. ***Matplotlib*****:** *NetworkX* is capable of graphically representing our topologies through *graphs*. In order to "draw them", *NetworkX* depends on the *matplotlib* module we have also installed as a dependency. This module is however **not mandatory**: they rest of the program will work as intended, it will just be unable to graphically represent the topology. In a later chapter we will devote our time to looking into a *proof of concept* we have developed. Said experiment generates a series of time-tagged events that our program is capable of representing as a regular graph. The lack of the *matplotlib* module implies this graph will not be available either. All in all, it is up to the user to decide whether they want this functionality or not: the program itself will carry out is primary task either way.

6. ***Docker Python SDK*:** We **have not** used the *docker python SDK* (Software Development Kit) in our project. We have decide to leverage *docker's CLI* interface from our code through calls to `os.system()`. However, someone deciding to use our project as a basis for something else might feel more comfortable interacting with *docker* through a pure-*python API* (Application Programming Interface). Changing our code to work in said fashion is a rather simple task should it have to be done.

4.1.2 User Manual

Throughout the development of our tool we have tried to simplify the use of the project as much as possible. The end result is a user-side workflow that only requires them to import a single project module to which they **must** provide a *NetworkX graph*. After doing so, they will be presented with a simple *CLI* letting them modify the currently live virtual network.

User Permissions Given the project will make use of *iproute2* the program needs to be run with administrative privileges (i.e. prepended by `sudo`). Even though this is the easiest approach and everything will "just work" it does have some security implications (mainly command injection) we will showcase in the appendix. Instead of choosing to run the entire blob of code as *root* we can also grant certain capabilities to the user who is to run the code, namely the *NET_ADMIN* capability (the same we need to grant containers). We should also mention that the user running the program must be able to interact with the *docker engine*. This can be ensured by adding said user to the *docker group* within the system, even though *root* will also be able to interact with and manage containers. This paragraph is intended as a warning, please refer to the appendix for a deeper discussion.

Depending on how the users decide to provide the required *graph* they might need to import additional modules. If they have stored a live graph (i.e. an instance of a graph) as a *pickle* [32] they will then need to import the *pickle* module to *un-pickle* the graph, for instance. In our examples we will define the graphs "on-the-fly", which requires us to import *networkx* itself. Listing 4.1 shows how one would define the topology found on figure 3.1 as a *networkx* graph.

```
1 # Import the networkx module so that we can define a graph.
2 import networkx
3
4 # Instantiate the netowrkx.Graph class.
5 sample_net = networkx.Graph(net = 'Sample Topology')
6
7 # Add host H-A-1 and the bridge for subnet A.
```

Node Type	Description
node	A regular host.
bridge	A link-layer switch that represents an entire subnet.
router	A net-layer router joining two or more subnets together.

Table 4.1: Node types.

```
8  sample_net.add_node('h-a-1', type = 'node')
9  sample_net.add_node('subnet-a-brd', type = 'bridge', subnet = '
       10.0.0.0/24')
10
11 # Add host H-B-1 and the bridge for subnet B.
12 sample_net.add_node('h-b-1', type = 'node')
13 sample_net.add_node('subnet-b-brd', type = 'bridge', subnet = '
       10.0.1.0/24')
14
15 # Add the R-A-B router with NO firewall rules.
16 sample_net.add_node('r-a-b', type = 'router', fw_rules = {})
17
18 # Connect H-A-1 to the bridge for subnet A.
19 sample_net.add_edge('h-a-1', 'subnet-a-brd')
20
21 # Connect H-B-1 to the bridge for subnet B.
22 sample_net.add_edge('h-b-1', 'subnet-b-brd')
23
24 # Connect router R-A-B to both bridges.
25 sample_net.add_edge('subnet-a-brd', 'r-a-b')
26 sample_net.add_edge('subnet-b-brd', 'r-a-b')
```

Listing 4.1: Defining the Sample Topology as a *networkx* Graph.

Creating Well-formed Graphs

Our tool expects the nodes on a *networkx* graph to adhere to certain constraints so that it can assemble the requested topology. Given the bi-directional nature of network links (data is sent in both directions), we have decided to model our networks as **undirected graphs**. Then, these will always be composed by a set of *nodes* interconnected by a set of *edges*. The types of nodes one can use are listed on table 4.1.

When adding a new node one must make sure that the following constraints **are respected**. Otherwise, all sorts of undefined behavior can and will be experienced: anything from a network that cannot be started to the presence of routing loops can happen. Given the restrictions we are imposing, these will be *bipartite graphs* [33]. If we assume U and V as the two sets in the *bipartite graph*, set V would contain bridges whilst set U would contain both nodes and routers. This property **has not** been leveraged in our code, but

it could prove to be useful if this work is developed further.

1. **Rules regarding node definitions:**
   (a) Each node's name **MUST** be a **unique** `string`.
   (b) Each node **MUST** contain a `type` attribute whose value is a `string`.
   (c) The value of the `type` attribute **MUST** be one of `"node"`, `"router"` or `"bridge"`.
   (d) Each bridge **MUST** contain a *subnet* attribute.
   (e) Every value for a *subnet* attribute **MUST** be specified as a `string` with the `"A.B.C.D/E"` format, where $A,\ B,\ C,\ D \in [0,\ 255];\ E \in [0,\ 30]$.
   (f) The subnet ranges associated to each bridge through the *subnet* attribute **MUST** be **unique**.
   (g) Each router **MUST** contain a `fw_rules` attribute.
   (h) Each `fw_rules` attribute **MUST** be set to a `dictionary` complying to the specifications laid out in section 4.1.2.
2. **Rules regarding edge definitions:**
   (a) The `strings` used to identify the nodes to be joined by an edge **MUST** refer to previously defined nodes.
3. **Rules regarding the topology:**
   (a) Each subnet **MUST** be composed by a **single** switch and an arbitrary number of nodes and/or routers.

Running a Graph

Once we have defined a graph as we have done on listing 4.1 we just need to run it to turn it into a virtual network. This can be accomplished though the `launch_net()` function we have defined on the *net_tools* module. We will of course analyze these in a later section.

Listing 4.2 shows how a use can run an existing *networkx* graph like the one defined on listing 4.1.

```
1 # Importing our module to launch the virtual network
2 from net_tools import net_ctrl
3
4 # This line will trigger the virtual network's creation.
```

```
5     # The second parameter controls whether we enable
6     # the configured firewalls or not.
7 net_ctrl.launch_net(sample_net, fw_on = False)
```

Listing 4.2: Turning a Graph Into a Virtual Network.

Configuring Firewalls

When configuring router `R-A-B` on listing 4.1 we passed an empty `dictionary` {} to the `fw_rules` parameter, thus effectively disabling the firewall of said router.

In order to define appropriate rules, the user needs to provide a `dictionary` adhering to the syntax specification shown on listing 4.3. Note that the order in which these rules are specified is the order in which they will be instantiated. This **will not** affect our topologies, but it can have an impact on the logical connections supported on the virtual network infrastructures.

```
1 # Note '|' is to be read as 'OR'
2 fw_rules = {
3     'POLICY': 'DROP' | 'ACCEPT',
4     'ACCEPT': [(RULE_1), (RULE_2), ..., (RULE_N)],
5     'DROP':   [(RULE_A), (RULE_B), ..., (RULE_Z)]
6 }
7
8 # Each rule has a syntax of the form
9 ('origin_node', 'destination_node', True | False)
```

Listing 4.3: Syntax for Specifying Firewall Rules.

Rule Syntax The first 2 elements are `strings` containing the names of the origin and destination nodes, respectively. The third parameter acts as a flag controlling whether the rule is uni or bi-directional. If set to `False`, we will only instantiate a rule affecting traffic going from the origin to the destination. If it is `True` however we will also instantiate a symmetric rule allowing for two-way communication. Even though the flag is not *explicitly* needed it does reduce the configuration specification tremendously, as the topologies we worked with always made use of symmetric rules. Listing 4.4 shows how one can accomplish the same configuration with two rules instead of one if not using said flag.

```
1 # Enable two-way communication x <--> y.
2 ('x', 'y', True)
3
4 # This accomplishes the same result with two triplets.
```

```
5 ('x', 'y', False)
6 ('y', 'x', False)
```

Listing 4.4: Uni-directional vs. Symmetric Firewall Rules.

Dictionary Syntax The `dictionary` contains 3 key-value pairs. The first one defines the policy for the *FORWARDING* chain as a `string`. The second one contains a `list` of rules for *ACCEPT*ing packets. If the default policy is set to *ACCEPT* these rules will be meaningless... The third one contains a `list` of rules for *DROP*ping packets. If the default policy is to *DROP* them these rules will have no effect either.

Execution Modes

Chapter 5 is devoted to analyzing the *proof of concept* we have developed. Said *proof of concept* will produce a series of *CSV* files containing data that characterizes how the experiment progressed. These files can also be analyzed by our program to provide a nicely formatted output so that the end user can make the most of the results.

Instantiating a full-fledged virtual network when the user only wants to load some *CSV* files to graph or analyze them can prove to be a rather time and energy consuming process. That is why we have developed the so called *report mode* on top of the *normal* or *network mode*.

The former will cause the program to parse a graph defining a network topology and then instantiate it. This mode can of course carry out the same analysis on files as the *report mode*. The *report mode* on the other hand will just present the user with the *CLI* where he or she will be able to invoke a subset of all the commands. These are: `ld-atk-data`, `atk-graph`, `c | clear`, `quit | exit | x` and `CTRL + C`.

The advantage *report mode* has over the *normal mode* is that it need not be concerned with instantiating a virtual network. This makes its startup time almost negligible.

In order to enable *report mode*, one can either specify the `report_mode = True` parameter on the call to `launch_net()` as seen on listing 4.2 or just run the `net_ctrl.py` file directly with `python3 net_ctrl.py`. The latter approach will trigger an `if name == "__main__:` clause, thus causing the *report_mode* parameter to be set to `True`. Please note the different files and their purposes will be explained in a later section.

Available Commands

As stated before, our tool will allow the users to modify the virtual network once it is up and running through a *CLI* interface. The following enumeration contains a list of the available commands together with a short description of what they can achieve.

1. `mvsubn <affected_subnet> <destination_subnet>`: This command will move the `<affected_subnet>` and attach it to the `<destination_subnet>`. These identifiers should be the ones provided by the `lssubn`. If either subnet does not exist, the command will fail with an error message.

2. `mvnode <affected_node> <destination_subnet>`: This command will move the `<affected_node>` to the `<destination_subnet>`. These identifiers should be the ones provided by `lsnode` and `lssubn`, respectively. If either element does not exist, the command will print an error message.

3. `lssubn`: This command will print a list of all the currently active subnets. These identifiers are the ones to be provided to the `mvsubn` and `mvnode` commands.

4. `lsnode`: This command will print a list of all the currently active nodes. These identifiers are the ones to be provided to the `mvnode` command.

5. `lsnet`: This command will graphically represent the current network topology. Please note this call is blocking, so the user will not be able to issue any other command until he or she closes the image. This command will require the installation of the *matplotlib* dependency. It is also worth mentioning that the generated image can be stored as a *PNG* file for later inspection.

6. `lscnx`: This command will show a "higher level graph" capturing the logical connections set up through firewall rules within the routers belonging to the network. If firewalls have not been enabled, the command will just print an informative message on screen as the resulting graph would be the same as the one shown by the `lsnet` command, given no logical connections are hampered by firewall rules. Even though one can generate this graph, the user should regard it as more of a "debugging" feature. We are internally the data structure from which the graph is derived as means of making dynamic firewall reconfiguration easier.

7. `dump-atk-data [path]`: Running the attack on the scenario generates output through files within the network nodes. This command is in charge of reaping all this data an dumping it both to a file and to a `dictionary`. Please note that, as the main program is ran as *root*, the generated files will belong to said user. One can manually change permissions with *chown* later on. These files contain a list of comma

separated values (that is, these are *CSV* files) that are human readable. Nonetheless, the primary intention of this files is to allow the user to later inspect them and generate graphs through the program itself. In order to do so, we have prepared the so called *report-mode*. As seen in the command description, one can optionally provide a path to save the file to under `../proof_of_concept/generated_data/`. This will usually be just a filename. If this path is not provided, the default name `last_data.csv` will be used. Please note that results will overwrite themselves unless the user changes the output file's name.

8. `ld-atk-data [path]`: This command will load the attack data from the default `../proof_of_concept/generated_data/last_data.csv` file or the one specified through the optional `path` parameter. Please note that `../proof_of_concept/generated_data` will be prepended to whatever argument is provided. This command will fail if the provided path leads to a non existent file.

9. `atk-report`: This command will read the `dictionary` containing the attack's results and print a nicely formatted table to *STDOUT* showing the times the *ping* processes went either up or down. In order for it to work, the user must either dump the data previously through `dump-atk-data` or load it from a file with `ld-atk-data`. If these steps have not been fulfilled, a nice reminder will be printed to the screen.

10. `atk-graph`: This command will show a graph displaying the evolution of the number of *ping* processes in the network against time. As before, the data must either be dumped or loaded beforehand.

11. `check-atk`: This command was written to aid in the debugging of the attack script. It will display the number of times the attack has run on each network node. An attack that is behaving as expected will run only once within each node.

12. `launch-pings`: This command will launch a *daemonized ping* process in each node. *Daemonizing* the *pings* allows them to keep on running after the script moves to a new network node (i.e. it allows ping to run without a controlling *TTY*).

13. `reset-net`: The attack we have written relies on several output files it generates to keep track of its current state. Thus, running the attack twice may result in some unexpected behaviour. This command will get rid of said files so as to effectively restore the network nodes to their original state.

14. `clear | c`: This command will clear the screen to allow for a more comfortable user experience. If running on a compliant shell, the `CTRL + L` combination will have the same effect. Note this command can be invoked either via `clear` or just `c` as seen in the syntax specification.

15. `quit | exit | x`: This command will dismantle the network and exit the program. Note `exit` and `x` are *aliases* for `quit`.

16. `CTRL + C`: This key combination will send the *SIGINT* signal to the process which will be handled, causing the program's termination.

4.2 Overview of the Project's Modules

One of the principles driving software development is *modularity*. We have then tried to make our code components as independent as possible whilst allowing them to cooperate so that they can be used for other purposes besides the ones that we originally intended.

The development effort culminated on 3 different modules fulfilling each a set of tasks:

1. ***virt_net***: This module is concerned with the instantiation of the different network elements (*veths*, *bridges* and *nodes*) together with their addressing and configuration. The module contains classes representing everything from the entire network to a single *veth*.

2. ***graph_interpreter***: This module acts as an intermediary translating the graphs provided by users to instances of the different classes defined in the *virt_net* module. It will also implement the high-level functionality provided by user commands such as `mvsubn` and `mvnode`. On top of that, it will leverage *networkx's* functionality to route the entire network and instantiate said routes in the routers once they have been brought up.

3. ***net_ctrl***: This module implements the *CLI* users running the tool will be presented with. It will resolve issued commands to calls to functions defined in the *graph_interpreter* module. This module serves as the user's entry point to the functionality offered by our tool.

The design we have just specified allows other users to "swap" the modules they do not desire to use for their own or third-party ones. One can, for instance, decide not to use our *graph_interpreter* and manually instantiate a network through calls to the *virt_net* module alone and that would be completely feasible. We will now include the documentation for each of the modules down to the *function-level*.

Given the extension of the module's description we have decided to include them on appendix **??**. The following sections provide some general conventions, background and the reasoning behind some of our decisions during the development.

4.2.1 Conventions

Data Type Specifications

The *C programming language* is by no means user friendly. Nonetheless, there is one aspect we really missed when developing in *python*: data type specification. In an effort to make the understanding of our code easier we have decided to specified each attribute's and variable's data types in the documentation composing this section. The syntax we will use to describe the used types is:

- `Strings`: `string`.
- `Lists`: `list/value_type`.
- `Dictionaries`: `dictionary/key_type/value_type`.
- `Boolean`: `boolean`.
- `Instance of class Foo`: `Foo_inst`.

Defining Private Methods

One of the main advantages of the object-oriented programming is that it provides "access control" to methods and attributes of a class. In *C++*, this is accomplished by classifying them as *private* or *public*, for instance. On the other hand *python* does not *enforce* this behaviour: a user will be able to call any method or access any attribute of an instance.

In order to "circumvent" this issue, the convention has it that method and attribute names prepended by an underscore (_) are to be treated as private. If a user decides to explicitly call or access these methods and attributes he or she is then exposed to undefined behaviours.

Calling Methods Through the Module Name

We have chosen to always call functions and instantiate classes external to the current file through their full name (i.e. including the name of the external file in the call). This does increase the length of code lines but it allows the reader (and ourselves) to know where the functions and classes are effectively defined. We feel like the increase in line length is completely justified.

The __init__.py File

Each of our modules contains an empty `__init__.py` file so that they become *regular packages*. This allows us to comfortably import the modules where their contents are needed. More information on *python's* import system can be found on section 5.2.1 of [34].

Constructors and Destructors

When a *class* is instanced and it becomes an *object* one special method will be automatically invoked for us: the *constructor*. In *python* this method is **always** named `__init__()` and it **cannot** return anything.

The counterpart of the *constructor* is the *destructor*. *Python* is *garbage collected* in the sense that, once there are no more references to an object, it will schedule it for deletion. When an object is being deleted its *destructor* will be automatically called as well. This *destructor* is always defined as `__del__()` in *python* and it accepts a single argument: a reference to the instance that is being deleted (i.e. `self`). Just like the *constructor*, it **cannot** return anything.

These methods will be defined for each of our classes and, given the `__init__()` and `__del__()` names can come across as rather cryptic we have decided to denote them as *constructor* and *destructor*, respectively, in the following sections.

4.2.2 The virt_net Module

This module encapsulates every kind of virtual network devices in their own class. Thus, one can theoretically instantiate a full-fledged virtual network through the sole use of this module. Given this module **does not** offer any routing logic, the end user would be responsible for providing it. In our case, it is implemented as part of the *graph_interpreter* module. This module serves as the "backbone" supporting the latter two and it is mostly concerned with interacting with *docker* and *iproute2*. Thus, the code itself is not logically complex but it does require knowledge of the aforementioned technologies to understand. These contents have already been covered in chapter 2.

Please refer to section C.1 for a comprehensive description of the module.

4.2.3 The graph_interpreter Module

This module is concerned with interpreting a graph and translating its structure to calls to methods defined in the previous module. In order to accomplish this, we have had to implement the “thickest” methods which are in charge of routing the network, dealing with the movement of nodes and subnets and parsing the initial graph. As we stated before, the use of this module is not mandatory: one can perform “raw calls” to the *virt_net* module and manually handle the routing within the generated network. This would also imply features like moving nodes once the network is operational would be lost. Even though this module’s purpose is not complex, the implementation of several features proved to be rather challenging.

Leveraging NetworkX We strongly believe it is better to stress one’s work flaws than its strong aspects. In an effort to ease the development process whilst enhancing the outcome’s resiliency we settled on leveraging the *NetworkX* module.As we have previously hinted, *NetworkX’s* main purpose is handling graphs. By modeling our networks as undirected graphs we can leverage already implemented functionalities such as checking whether the graph is completely connected (i.e. there are no dangling nodes) or even routing algorithms. Given we have a complete view of the network we decided to apply *Dijkstra’s Algorithm* (section 24.3 of [35]). *NetworkX* offers several methods through which we can obtain the shortest path between any two nodes. Combining this information with the complete list of network nodes allows us to completely route the network in an elegant way. We would like to point out once again how we manually implemented our solution to this issue through our own routing algorithm. Nevertheless, we discarded it in favour of *Dijkstra’s Algorithm* when we settled on relying on *NetworkX*.

Please refer to section C.2 for a comprehensive description of the module.

4.2.4 The net_ctrl Module

This module is composed by a single file whose main goal is offering a *CLI* to a user so that he/she can interact with a network that is currently operational. The function defined by this module is the “entry point” to the rest of the modules, as it is the one that should be called and passed the graph a user generates that represents the desired topology.

Please refer to section C.3 for a comprehensive description of the module.

4.3 Working Topologies

Developing a project such as this one can at times seem daunting during the initial steps. One finds him or herself writing a lot of code without being certain of whether it is correct or not. That is why we decided to test our work as soon as possible to verify everything was marching as intended. In order to do so we designed some simple topologies, each being more complex than the previous one. We tried to instantiate them and manually check that the connectivity between all the nodes was the desired one.

This section includes these test topologies together with the most complex one we have been capable of working with.

4.3.1 Topology Alpha

This was the first topology we tested. The most valuable piece of information we extracted from it was that we were able to communicate with nodes belonging to *Subnet C* from *Subnet B*. This proved that routing packets through a subnet was behaving as intended. This topology is included on figure 4.1.

4.3.2 Topology Beta

This second topology added another subnet to the previous to push our routing procedures a little further. It is shown on figure 4.2.

4.3.3 Topology Gamma

This third topology, shown on figure 4.3, added yet another subnet to the previous one. After successfully instantiating this topology we felt confident our design was capable of handling significantly more complex topologies.

4.3.4 Topology ICS

This fourth topology, portrayed on figure 4.4, is modeled after *figure 4* of [36]. It is the most complex one we have worked with and the chosen scenario for the proof of concept. We would like to note how, unless otherwise specified **all the firewalls drop packets** (i.e. only hosts listed on the *FW Conf* sections on figure 4.4 can communicate).

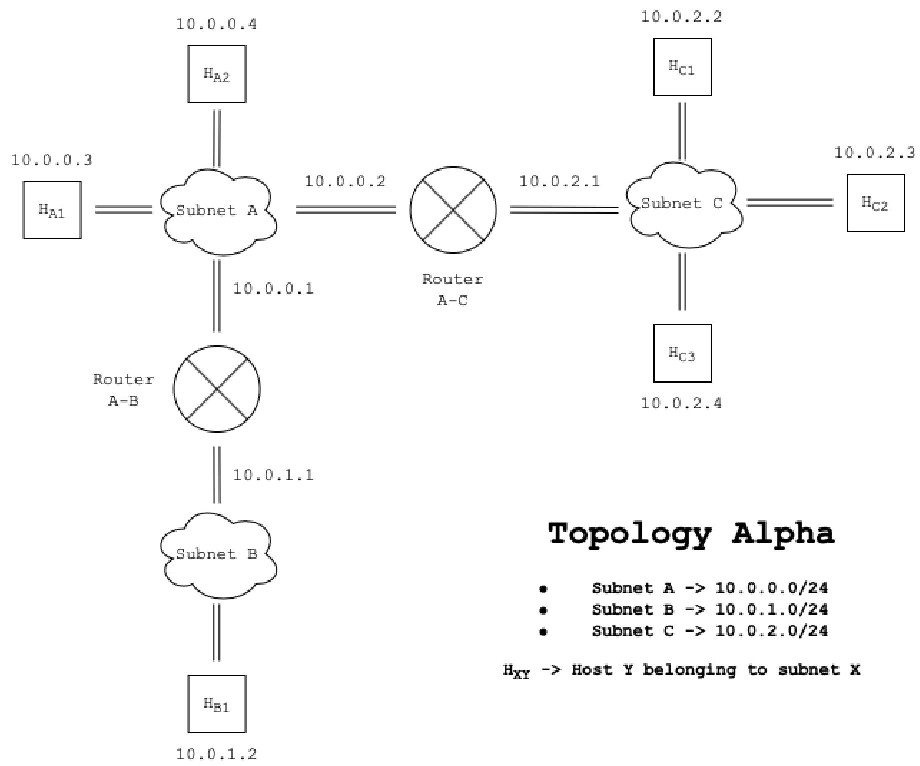


Figure 4.1: The *Alpha Topology*.

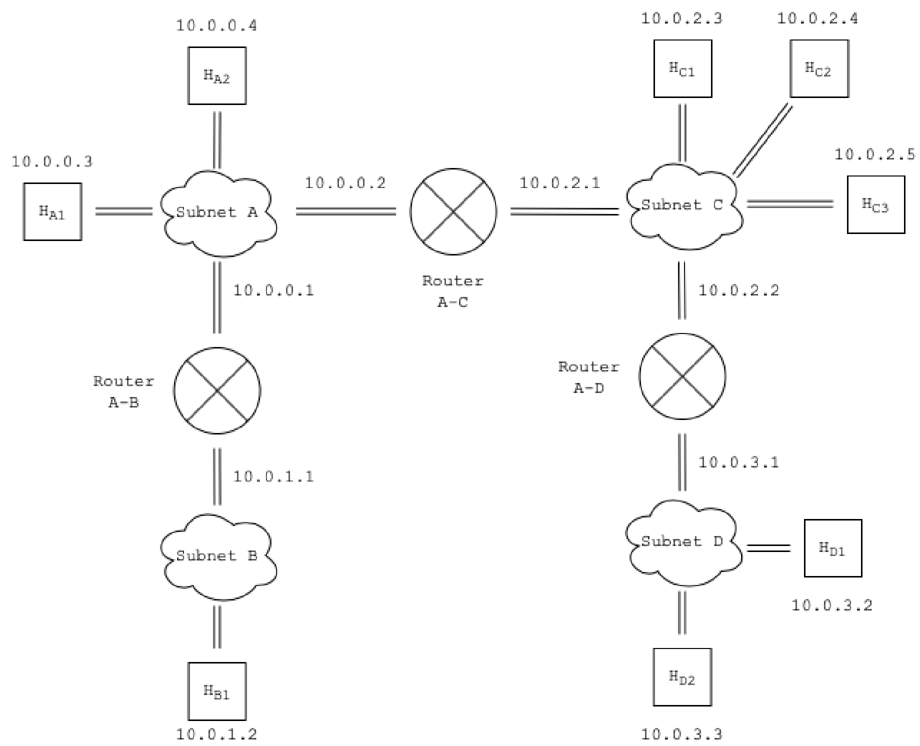


Figure 4.2: The *Beta Topology*.

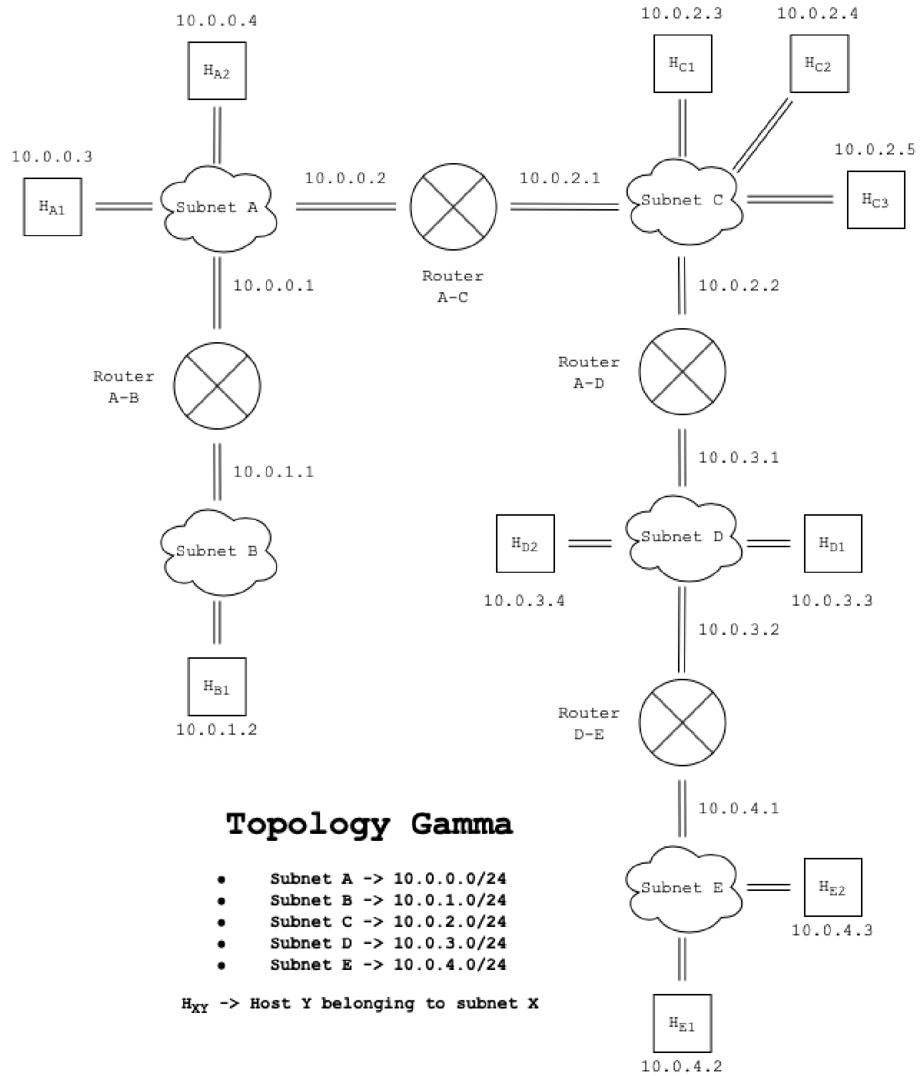


Figure 4.3: The *Gamma Topology*.

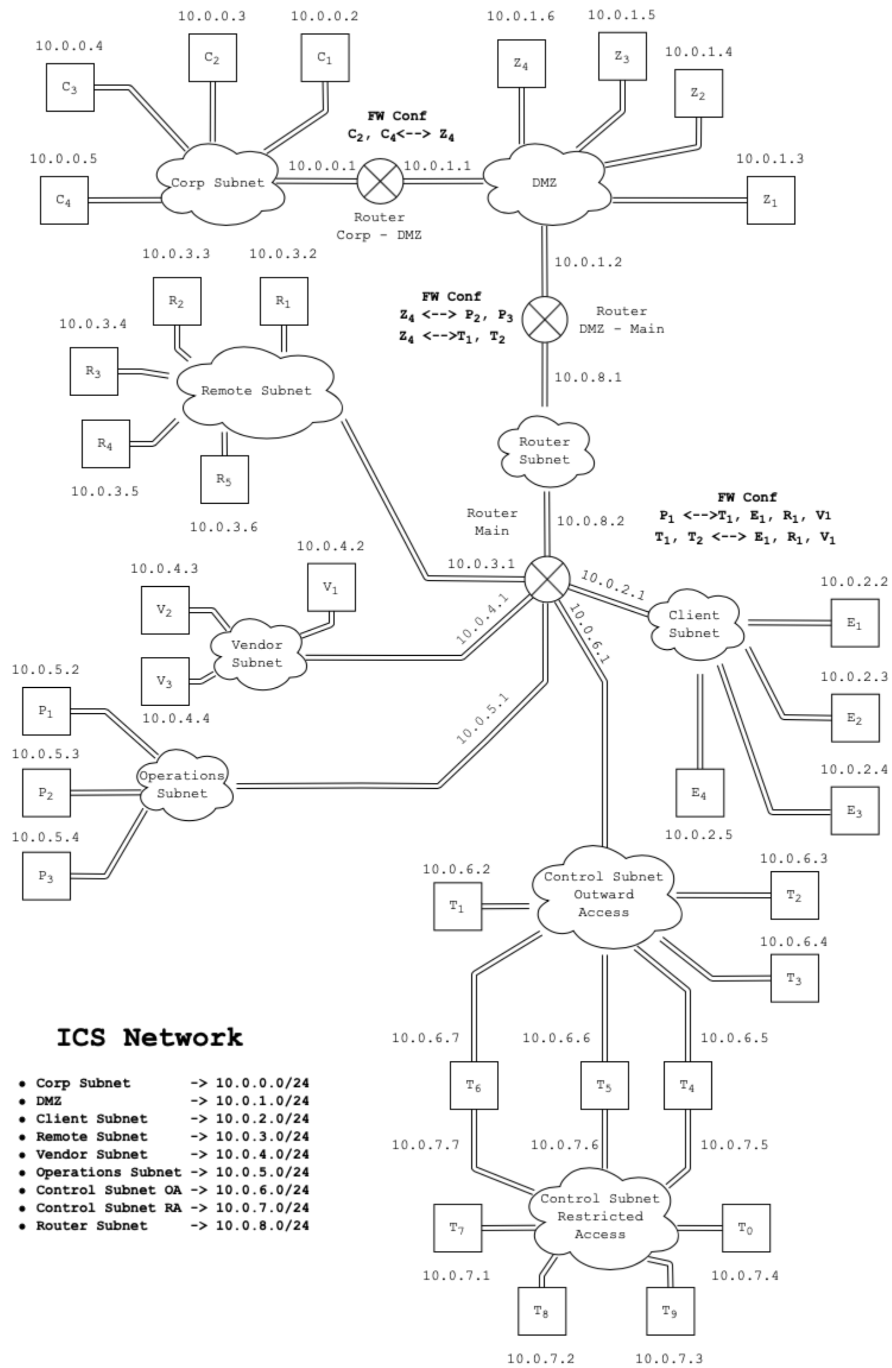


Figure 4.4: The *ICS (Industrial Control System) Topology.*

Chapter 5

Proof of Concept

> Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.
>
> *Edsger Dijkstra*

5.1 Description

The main goal of this project is serving as a benchmark tool for research purposes. In order to demonstrate this is a feasible use of our work we have prepared a proof of concept in which we simulate an attack and monitor the network's state.

The initial scenario is composed by a set of nodes running a *ping* process against themselves (i.e. against *IP 127.0.0.1*). We then define the *Quality of Service (QoS)* of the network through the function shown on equation 5.1. Function $pings(t)$ represents the number of currently active *ping* processes in the network. Thus $pings(0) = Maximum\ Number\ of\ Ping\ Processes$; in other words, our initial scenario is offering the best possible QoS (i.e. $QoS(0) = 1$).

The attack we have written manifests itself as a "virus" that will replicate throughout the network. Whenever it reaches a machine, it will *kill* its associated *ping* process and then jump to another one. This implies that, as the attack progresses, we will experience how $pings(t)$ decreases as t increases. This amounts to $QoS(t)$ diminishing with time as well. By scrutinising the plots of $QoS(t)$ we will try to judge how much of an impact altering the network's topology has on the attacks progression.

$$QoS(t) = \frac{pings(t)}{pings(0)};\ Qos(t) \in [0,\ 1]\ \forall\ t \in [0, \infty)$$

Equation 5.1: Definition of the Network's QoS

The Exploit

Like many attacks, our own leverages a vulnerability in the network elements' configuration. The reader might recall the *Dockerfile* we presented on listing 2.10 in which we set *1234* as *root's* password. Our "virus" assumes the password is already known, which implies it is capable of accessing every machine of the network. Note that, even though we have not included it explicitly, the *Dockerfile* defining the images run by nodes also configure *1234* as *root's* password.

It is true that this assumption can be considered as giving an unfair advantage to a potential attacker. However, we are interested in mitigating an attack once it occurs, not in preventing it. Thus, answering how an attacker gains a foothold in the network is not as interesting to us: we want to discover how we can minimize the attack's impact.

Attack Dependencies

The attack itself requires the *ssh* daemon and associated tools such as *scp* as well as on standard utilities like *ping*. However, these are also required for the container's correct operation, so we are not considering them strict attack dependencies. On the other hand, we depend on the *sshpass* binary to carry out the attack in an automatic fashion.

Programs such as *ssh* will only read input such as passwords from their *controlling terminal* (i.e. the file descriptor returned by `open("/dev/tty")`) rather than from *STDIN* itself. This renders shell redirections such as `echo "1234" | ssh 10.0.1.3` unusable. This "limitation" can be circumvented by programs such as *sshpass*. This binary will `fork()` a process and run *sshpass* within it. However, before doing so it will consider the parent process (i.e. *sshpass* itself) as the child's controlling terminal. This allows *sshpass* to "feed" the password to *ssh* in a totally automated way.

We quoted the term "limitation" because the design of the *ssh* tool and its associated utilities is not design to be frustrating. When passing passwords on the command line, these can be "seen" on the output of programs such as *ps*, which exposes credentials that should always remain private. Thus, when we invoke *sshpass* we are letting other users

logged into the system know our *1234* password. Nonetheless, this is an attack: we are not concerned with security. That is why we believe the use of tools such as *sshpass* is justified.

The *sshpass* binary was compiled on our own host from its sources [37] in a *static* fashion. This produces a larger output, but the resulting executable does not depend on any shared libraries. Given how slim container images are, we preferred to make sure no errors provoked by missing libraries could take place.

The Attack Script

Listing 5.1 contains the *bash* script implementing the attack on the network. Given *bash* can be a harsh language we have littered the code with comments clarifying some of the most convoluted lines and obscure assumptions that might not be at all clear by just reading through the code.

```
1 # Invoke a bash shell to interpret the following script
2 #!/bin/bash
3
4 # Print the hostname of the machine we are currently running
5     # on so that we can follow the script's "infection path".
6 echo "Running @ $(hostname)"
7
8 # Global variables:
9     # rootpwd: Password used when SSHing to machines.
10    # debug: Causes the script to dump the number of times
11        # it is executed to a file. It MUST BE commented out
12        # to be disabled.
13 rootpwd="1234"
14 debug="on"
15
16 # Check the times the script is run on a node when debugging.
17 if [ ! -z $debug ]
18 then
19     # If the file '/n_runs' exists
20     if [ -f /n_runs ]
21     then
22         # Read it and increment its value by 1.
23         # Arithmetic is evealuated when enclosed by
24             # an Arithmetic Expansion -> $(())
25         # The '>' redirection operator will either
26            # overwrite or create a file.
27         echo $(( $(cat /n_runs) + 1 )) > /n_runs
28     else
29         echo 1 > /n_runs
30     fi
31 fi
```

```
32
33 # Define an array containing the IPs for each interface.
34 iface_ips=($(ip a | awk '$1 ~ /inet/ && $2 !~ /127.0.0.1\/8/' | awk '{
       print $2}'))
35
36 # Store the number of IPs (i.e. the length of the iface_ips array).
37 n_ips=${#iface_ips[@]}
38
39 # Store the PID associateds to the ping process within the
40     # container. We could have also used the 'killall'
41     # utility that 'kills' processes by name. This
42     # increased the number of needed dependencies however,
43     # and we decided to do it ourselves.
44 ping_pid=$(pgrep ping)
45
46 # If we found a ping process (i.e. 'ping_pid' variable is
47     # not empty).
48 if [ ! -z $ping_pid ]
49 then
50     # Terminate the ping process and note down the time it
51         # was taken down in the '/p_deaths' file. Note the
52         # '>>' redirection operator appends data to a file
53         # or creates it if it doesn't exist.
54     kill $ping_pid
55     echo "$(date),$(date +%s)" >> /p_deaths
56 else
57     echo -e "\tNo ping found. Quitting..."
58     exit
59 fi
60
61 # If we are currently running on a node (we only have a
62     # single network interface) and we are not the
63     # "entrypoint" for the attack we will exit. Some
64     # other machine will carry on with the attack.
65     # Deciding whether we are the entrypoint amounts
66     # to checking whether the /entrypoint file exists.
67 if [ $n_ips -eq 1 ] && [ ! -f entrypoint ]
68 then
69     echo -e "\tRegular node. Quitting..."
70     exit
71 fi
72
73 # Inform the user what type of machine the attack is
74     # currently at.
75 if [ ! -f /entrypoint ]
76 then
77     echo -e "\tWe are a router!"
78 else
79     echo -e "\tWe are at the entrypoint!"
```

```
80 fi
81
82 # Run the following on each interface through a for loop.
83 for (( i=0; i<${n_ips}; i++ ))
84 do
85     # Leverage awk to find the interface's associated
86         # subnetwork's network address and our own IP
87         # address witin it.
88     net_addr=$(echo "${iface_ips[i]}" | awk '{split($0,foo,"/"); split(
    foo[1],fuu,"."); print sprintf("%s.%s.%s.", fuu[1], fuu[2], fuu[3])}'
    )
89     our_ip=$(echo "${iface_ips[i]}" | awk '{split($0,foo,"/"); print foo
    [1]}')
90
91     # If we are debugging, print the discovered network
92         # and IP addresses.
93     if [ ! -z $debug ]
94     then
95         echo -e "\tNet_addr: $net_addr\tIP: $our_ip"
96     fi
97
98     # If this subnetwork has no been attacked yet (i.e. it is
99         # not contained in the '/victims' file) proceed to
100        # attack it. The '/victims' file is copied to each
101        # newly infected machine an contains a list of the
102        # subnetwork's that have been attack so that we do
103        # not attempt it again. This serves as a base case
104        # for recursivity so that the attack does not run
105        # indefinitely.
106    if [ -z $(grep $net_addr /victims 2> /dev/null) ]
107    then
108        # Append this subnet to the victims list right away
109            # as we are going to proceed to attack it.
110        echo $net_addr >> /victims
111
112        # As we know this is a '/24' subnetwork we know the
113            # valid addresses range from X.X.X.1 to
114            # X.X.X.254.
115        for j in {1..254}
116        do
117            # We can generate the current victim's IP address
118                # by appending the current ending ($j) to the
119                # network address ($net_addr).
120            if [ $net_addr$j != $our_ip ]
121                then
122                echo -e "\tCopying to -> $net_addr$j"
123
124                # Copy the script itself, the '/victims' file
125                    # and the 'sshpass' binary to the next
```

```
126                     # victim with scp. We are using the '-o
127                     # StrictHostKeyChecking=no' option with
128                     # 'scp' because we want to avoid the "Do
129                     # you trust this host..." or ECDSA prompt
130                     # as this script is NOT interactive.
131                 /sshpass -p $rootpwd scp -o StrictHostKeyChecking=no /
    victims /p_stopper.sh /sshpass root@$net_addr$j:/ > /dev/null 2>&1
132
133                 # Run the script within that victim. This
134                     # will cause the script to recur
135                     # indifenitelyuntil there are no more
136                     # subnetworks to attack. At that point,
137                     # the attack will begin dismantling
138                     # ping processes in a backwards fashion
139                     # of compared to the order in which the
140                     # attack itself was copied between
141                     # machines.
142                 /sshpass -p $rootpwd ssh root@$net_addr$j "bash /
    p_stopper.sh" < /dev/null  2> /dev/null
143
144                 # Get an updated copy of the subnets which
145                     # have been attacked to avoid attacking
146                     # subnetworks twice.
147                 /sshpass -p $rootpwd scp -o StrictHostKeyChecking=no
    root@$net_addr$j:/victims /victims > /dev/null 2>&1
148
149                 # In order to check whether we have
150                     # connectivity with the current
151                     # IP we will ping it once (thanks
152                     # to the '-c' option) whilst
153                     # redirecting the output of both
154                     # STDOUT and STDERR to /dev/null.
155                     # We'll then look at ping's return
156                     # code (through the $? variable
157                     # the shell mantains which is the
158                     # last return code) to check
159                     # whether the current IP is alive
160                     # or not. As seen in ping's manpage
161                     # (man ping) one can indeed use this
162                     # return code for this purpose. In
163                     # other words, if ping returns with
164                     # a 0 code then the host is alive.
165                     # If the code is either 1 or 2 the
166                     # current IP is not associated with
167                     # any host and we will move on to
168                     # attack another subnet. This
169                     # assumes all the nodes reside in
170                     # the lower end of the subnetwork's
171                     # address space. This attack would
```

```
172                     # work in the same way if this
173                     # assumption were not feasible: we
174                     # would just remove this last section
175                     # chekcing whether the host is alive
176                     # or not and just blindly continue
177                     # until all the subnetwork's addresses
178                     # were exhausted. However, given the
179                     # size of /24 subnetworks this would
180                     # be a time consuming process that
181                     # we believe would add no value to
182                     # the proof of concept.
183                 ping -c 1 $net_addr$j > /dev/null 2>&1
184 
185                 # If the return code is not 0.
186                 if [ $? -ne 0 ]
187                 then
188                     # Break from the current for loop and try
189                         # to attack the next subnetwork.
190                     echo -e "\tBreaking..."
191                     break 1
192                 fi
193                 # After all the subnetworks have been
194                     # attacked and we return to this
195                     # node print it on screen so that
196                     # we can follow the attack's path.
197                 echo -e "Back @ $(hostname)"
198             fi
199         done
200     fi
201 done
202 
203 # Clean the victims state unless we are debugging and we want
204     # to take a look at it later on. Invoking 'rm' with the
205     # '-f' flag prevents it from outputting an error message
206     # in case the specified files don't exist.
207 if [ -z $debug ]
208 then
209     rm -f /og_net /victims
210 fi
```

Listing 5.1: The Attack's Script.

5.2 Running the Proof of Concept

As a prerequisite to run the proof of concept the user needs to bring up the network the attack will be run on as explained on section 4.1.2. Once this has been accomplished the following must be done in order:

1. Execute the *launch-pings* command on the *CLI* that is offered after the network has been instantiated. This will run docker -d exec <container-name> ping 127.0.0.1 against each node so that we can assure the $pings(0) = Maximum\ Number\ of\ Ping\ Processes$ condition is met. Note the -d option launches the command in the background (i.e. as a *daemon*).

2. Copy the attack script together with the *sshpass* binary to the node the start will start from. This can be achieved by running docker cp p_stopper.sh <initial-container>:/ and docker cp sshpass <initial-container>:/. Note these files **must be** executable. The user should run chmod +x <filename> on both of them either before or after copying them to the container. In the latter case the same effect can be achieved by running docker exec <initial-container> chmod +x <filename>.

3. If the initially infected node **is not** a router, the script relies on am empty file named *entrypoint* that must be present. It can be generated by means of the docker exec <initial-container> touch /entrypoint.

4. The attack script must be run, either from a *bash* shell opened through docker exec -it <initial-container> bash by executing ./p_sto pper.sh or directly through docker exec <initial-node> p_stopper .sh.

5. The attack takes a non-negligible amount of time to finish. After it is done it will return control of the shell the attack was started from back to the user.

5.3 Results

We applied the steps described in the previous section to the *ICS Topology* we presented on figure 4.4. We initially launched the attack script on the network "as-is" so that we would have a reference against which to compare the effectiveness of proposed mitigation strategies. In all of the cases below, host *c-1* was the attack's entry point.

5.3.1 Running the Attack Against a Static Network

Figure 5.2 depicts the evolution of the *QoS* over time on the network as time progresses. The initial point is $(0, QoS(0)) = (0, 1)$ and it has been added so that the initial reference for any subsequent attacks is equivalent. The times present on the abscissa axis are relative to this initial condition. This graph has been generated **exclusively** based on the data points present on listing 5.2 which were recovered after the attack had concluded through the dump-atk-data command. Given we **are not** taking any explicit action to

try and halt the attack's progression we can regard this as the worst case scenario.

The plot shown on figure 5.2 is rather distinctive. Due to how the attack operates, it will suffer non-negligible delays when trying to *ping* an *IP* address that is not assigned. In other words, the *ping* timeout is large when compared to how fast the attack propagates on the lower end of the subnets. The reason is that the "virus" will continue taking *ping* processes down as long as it finds victims within a subnet without any apparent delay. We must not forget the entire virtual network is housed within the same host. This fact implies network connections are extremely fast and thus they do not become a "bottleneck" for the attack's operation. These delays manifest themselves as regions with minuscule slopes when compared with the steeper ones. Figure 5.2 contains 6 of them.

The smooth regions alternate with others characterised by a rather tumultuous slope. These are generated by the attack quickly propagating throughout a subnet and it is easy to spot 7 of them on figure 5.2. We can then see how the attack has an almost "periodic" behaviour. It will "bomb" the lower address space within each subnet it encounters. When it runs out of victims, it will suffer a noticeable timeout to then resume its operation on a new subnet.

Figure 5.3 contains a simplified network schematic representing the topology shown on figure 4.4 and the attack's propagation over the network. By combining the contents of this image with those of figure 5.2 one can understand why and how the attack behaves the way it does.

The tags found on figure 5.3 are explained in the following enumeration:

1. The attack begins propagating from the C_1 node.
2. The attack propagates to the *Corp - DMZ* Router.
3. The attack makes its way to the *DMZ - Main* Router.
4. The attack reaches the *Main Router*.
5. The attack terminates the *ping* processes on the entire *Client* subnet.
6. The attack returns to the *Main Router* and then proceeds to infect the *Remote* subnet.
7. The attack goes back to the *Main Router* and then assaults the *Vendor* subnet.
8. After going back to the *Main Router*, the attack infects the *Operations* subnet.

9. The attack returns to the *Main Router* and then attacks the *Control Subnet's Outward Access Region*. It will take down the ping processes on hosts $T_1 \rightarrow T_4$, inclusive.

10. From T_4, the attack propagates through the *Control Subnet's Restricted Access Region*, taking nodes $T_5 \rightarrow T_0$ with it.

11. The attack returns to the *Main Router*, and given no more nodes on the *Router* subnet remain to be attacked, it goes back to the *Corp - DMZ Router*.

12. The attack now takes down the entire *DMZ* subnet.

13. And it finally kills the *ping* process on machines belonging to the *Corp* subnet. This concludes the attack.

Given the information expressed through the figures can come across as "dense" we have also decided to describe the attack's overall beahviour in a textual manner. In both scenarios the initially infected node will be C_1. The attack will then propagate to the router connecting the *Corp* and *DMZ* subnets to then "jump" to the topology's main router. From there it will begin attacking each subnet in an ascending order in terms of their network address. That is, it will begin taking down processes on the *Client* subnet and then make its way up to the *Control Subnet's Outward Access Region*. Once node T_4 becomes infected, the attack will propagate to the *Restricted Access Region* of the aforementioned subnet. Upon termination of the attack on this last subnet, it will follow its initial path in the opposite direction by taking down nodes on the *DMZ* subnet and finally targeting machines belonging to the *Corp* subnet. At these point all the *ping* processes will have been taken down, thus concluding the attack. Please refer to figure 5.3 for a more detailed walkthrough of the attack's evolution.

The tags found on figure 5.2 are explained in the following enumeration:

1. Initial state.

2. This decrease in *QoS* is due to the *ping* processes being killed in nodes C_1, R_x, R_y, R_z and the entire *Client* subnet (i.e. the E_n nodes).

3. Timeout suffered after infecting the entire *Client* subnet.

4. The attack proceeds to the *Remote* subnet (R_n nodes).

5. Timeout suffered after infecting the entire *Remote* subnet.

6. The attack continues and targets the *Vendor* subnet (V_n nodes).

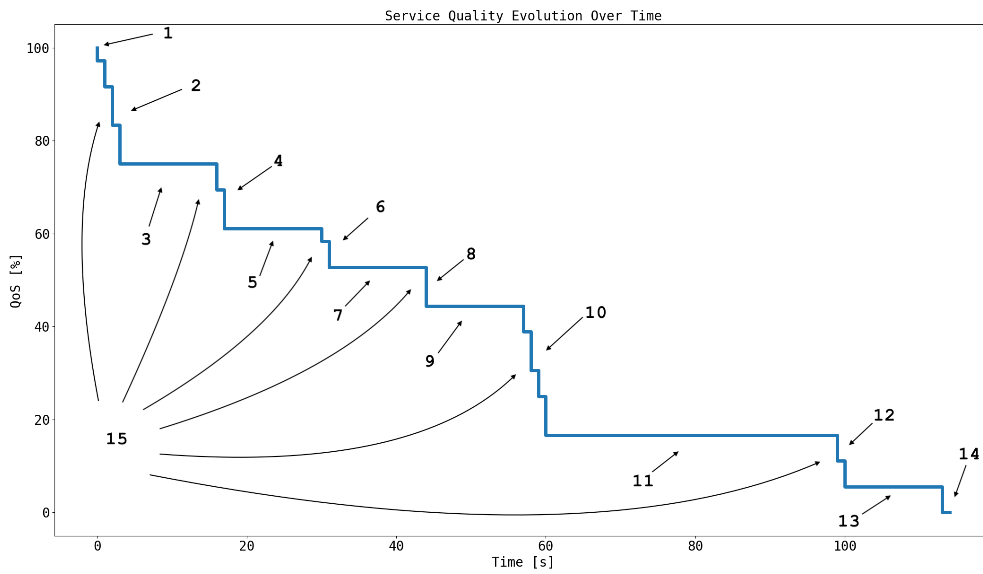


Figure 5.2: Evolution of the *QoS* Over Time for a Static Topology.

7. Timeout suffered after infecting the entire *Vendor* subnet.

8. The attack goes on and attacks the *Operations* subnet (P_n nodes).

9. Timeout suffered after infecting the entire *Operations* subnet.

10. The attack makes its way to the *Control Subnet's Outward Access Region*. Once it infects node T_4, it will gain access to the *Restricted Access Region* as well. It will then put *ping* processes on both T_5 and T_6 out of commission as if they only belonged to the *Restricted Access Region*. Thus, the characteristic delay experienced after taking an entire subnet down will now only be suffered after the *ping* processes are actually killed.

11. This delay roughly equates to 3 of the ones suffered up to now. These correspond to the *Control Subnet's Restricted Access Area*, the *Control Subnet's Outward Access Area* and the *Router* subnet, respectively. In the last of the three there are actually no more nodes to take down: both *ping* processes on the R_y and R_z routers were terminated during the first *QoS* fall.

12. The attack proceeds to the *DMZ* subnet (Z_n nodes).

13. Timeout suffered after infecting the entire *DMZ* subnet.

14. The attack targets the *Corp* subnet (C_n nodes) and then finishes.

15. Note how the magnitude of the reduction in terms of the QoS is proportional to the size of each of the affected subnets in terms of hosts.

```
Init_time: 2021-06-14 15:45:39.707455
c-1,Mon Jun 14 15:47:29 UTC 2021,1623685649,kill
c-2,Mon Jun 14 15:49:21 UTC 2021,1623685761,kill
c-3,Mon Jun 14 15:49:21 UTC 2021,1623685761,kill
c-4,Mon Jun 14 15:49:22 UTC 2021,1623685762,kill
z-1,Mon Jun 14 15:49:07 UTC 2021,1623685747,kill
z-2,Mon Jun 14 15:49:07 UTC 2021,1623685747,kill
z-3,Mon Jun 14 15:49:08 UTC 2021,1623685748,kill
z-4,Mon Jun 14 15:49:08 UTC 2021,1623685748,kill
e-1,Mon Jun 14 15:47:30 UTC 2021,1623685650,kill
e-2,Mon Jun 14 15:47:31 UTC 2021,1623685651,kill
e-3,Mon Jun 14 15:47:31 UTC 2021,1623685651,kill
e-4,Mon Jun 14 15:47:31 UTC 2021,1623685651,kill
r-1,Mon Jun 14 15:47:44 UTC 2021,1623685664,kill
r-2,Mon Jun 14 15:47:44 UTC 2021,1623685664,kill
r-3,Mon Jun 14 15:47:45 UTC 2021,1623685665,kill
r-4,Mon Jun 14 15:47:45 UTC 2021,1623685665,kill
r-5,Mon Jun 14 15:47:45 UTC 2021,1623685665,kill
v-1,Mon Jun 14 15:47:58 UTC 2021,1623685678,kill
v-2,Mon Jun 14 15:47:59 UTC 2021,1623685679,kill
v-3,Mon Jun 14 15:47:59 UTC 2021,1623685679,kill
p-1,Mon Jun 14 15:48:12 UTC 2021,1623685692,kill
p-2,Mon Jun 14 15:48:12 UTC 2021,1623685692,kill
p-3,Mon Jun 14 15:48:12 UTC 2021,1623685692,kill
t-1,Mon Jun 14 15:48:25 UTC 2021,1623685705,kill
t-2,Mon Jun 14 15:48:25 UTC 2021,1623685705,kill
t-3,Mon Jun 14 15:48:26 UTC 2021,1623685706,kill
t-4,Mon Jun 14 15:48:26 UTC 2021,1623685706,kill
t-5,Mon Jun 14 15:48:26 UTC 2021,1623685706,kill
t-6,Mon Jun 14 15:48:27 UTC 2021,1623685707,kill
t-7,Mon Jun 14 15:48:27 UTC 2021,1623685707,kill
t-8,Mon Jun 14 15:48:28 UTC 2021,1623685708,kill
t-9,Mon Jun 14 15:48:28 UTC 2021,1623685708,kill
t-0,Mon Jun 14 15:48:28 UTC 2021,1623685708,kill
r-x,Mon Jun 14 15:47:29 UTC 2021,1623685649,kill
r-y,Mon Jun 14 15:47:30 UTC 2021,1623685650,kill
r-z,Mon Jun 14 15:47:30 UTC 2021,1623685650,kill
```

Listing 5.2: Retrieved Data Points for the Attack on a Static Network.

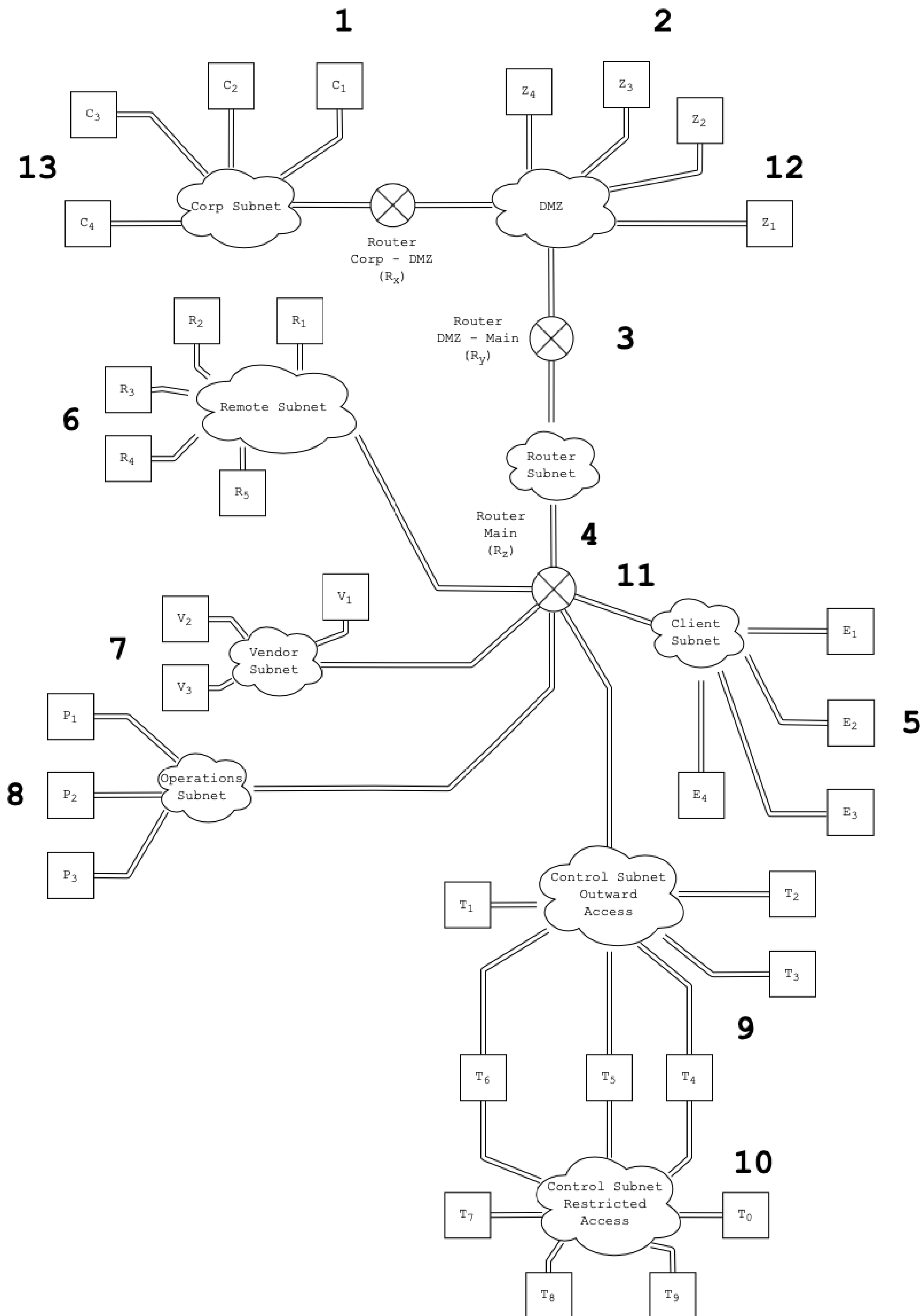


Figure 5.3: Attack Progression Over the Network.

5.3.2 Trying to Mitigate the Attack

Given the contents of figure 5.3 we are certain that the last hosts that are to be attacked are $C_2 \rightarrow C_4$. We can then try to move several other hosts to the *Corp subnet* (figure 4.4) so that they suffer the attack's consequences at a later time than they normally would. Given the "virus" will still be subject to a delay after sweeping each subnet, this strategy should provide a bit of leeway for the hosts we are to move.

In order to run this experiment we defined the `mitigate-atk` command on top of those already discussed on section 4.1.2. It will basically call the `mvnode` command repeatedly and move the following hosts to the *Corp Subnet*: $E_1 \rightarrow E_4$; $R_1 \rightarrow R_5$; $V_1 \rightarrow V_3$; $P_1 \rightarrow P_3$. We decided to add this auxiliary order due to the speed with which the attack propagates. If we were to manually move the 15 nodes we would be tarnishing the data, making it harder to interpret in a satisfactory manner. Then, after launching the attack exactly as described in the previous section we issue the `mitigate-atk` command, thus triggering the movement of the nodes we have just specified. As soon as said order is invoked we need not do anything more: we will gather the data through the `dumpt-atk-data` once it has terminated. The data we gathered and then used to draw the plots is included on listing 5.3

Given the "manual" aspect of having to explicitly issue a command the data presented in this section might not be **exactly reproducible**. Nonetheless, as long as the user issues the `mitigate-atk` command in a reasonable time the data should be tremendously similar.

Even though we are moving hosts while the attack is underway, the steps described in figure 5.3 are totally applicable: if we regard the network as a collection of subnets its topology **is not altered** at all by the `mitigate-atk` command. This explains why the attack's behaviour is the same despite the times at which the events take place not being exactly alike.

Figure 5.4 portrays the evolution of the *QoS* when we try to hinder the attack's progress. Figure 5.5 superimposes the evolution of the *QoS* in both cases. Studying it reveals how in the latter case the *QoS* is higher during the entirety of the attack. Thus, we can rest assured our mitigation mechanism is provoking its intended effect. The tags in said figure are described in the following enumeration:

1. Initial state.
2. The attack's progression is somewhat obfuscated due to the mitigation procedure. The movement of nodes causes *ping* processes to be killed and spawned, which has

transient effect on the *QoS* level.

3. After the attack is launched, we manually move every node within the *Client*, *Remote*, *Vendor* and *Operations* subnets and attach them to the *Corp* subnet. In other words, we are moving all the E_n, R_n, V_n, and P_n nodes and turning them into C_n nodes. The movement implies that the *ping* processes will be taken down and then spawned, which is what causes the "jittery" behaviour seen here.

4. The attack experiments a delay that is roughly 4 times the one that a single subnet causes. These are provided by the *Client*, *Remote*, *Vendor* and *Operations* subnets, respectively.

5. This decrease in the *QoS* is due to the dismantling of the T_n nodes' *ping* processes.

6. The attack suffers a total delay that is the product of those provided by the *Control Subnet's Restricted Access Region*, the *Control Subnet's Outward Access Region* and the *Router* subnet. Thus, it is roughly 3 times larger than the delay caused by a single subnet.

7. The *DMZ* subnet (Z_n nodes) is attacked.

8. The attack experiences a delay after infecting the entire *DMZ* subnet.

9. The attack proceeds to the *Corp* subnet. It houses all the C_n nodes and all the nodes we displaced to mitigate the attack, which amounts to a grand total of 19 hosts. The ping processes will be nonetheless terminated on only 18, due to the fact that node C_1 was the attack's entry point. Once the *Corp* subnet is attacked, the network's *QoS* drops to 0 %: the attack is finished.

```
1  Init_time: 2021-06-14 16:10:09.066786
2  c-1,Mon Jun 14 16:11:58 UTC 2021,1623687118,kill
3  c-2,Mon Jun 14 16:13:44 UTC 2021,1623687224,kill
4  c-3,Mon Jun 14 16:13:45 UTC 2021,1623687225,kill
5  c-4,Mon Jun 14 16:13:45 UTC 2021,1623687225,kill
6  z-1,Mon Jun 14 16:13:30 UTC 2021,1623687210,kill
7  z-2,Mon Jun 14 16:13:31 UTC 2021,1623687211,kill
8  z-3,Mon Jun 14 16:13:31 UTC 2021,1623687211,kill
9  z-4,Mon Jun 14 16:13:31 UTC 2021,1623687211,kill
10 e-1,Mon Jun 14 16:11:58 UTC 2021,1623687118,kill
11 e-1,Mon Jun 14 16:13:45 UTC 2021,1623687225,kill
12 e-1,Mon Jun 14 16:11:59 UTC 2021,1623687119,spawn
13 e-2,Mon Jun 14 16:11:59 UTC 2021,1623687119,kill
14 e-2,Mon Jun 14 16:13:46 UTC 2021,1623687226,kill
15 e-2,Mon Jun 14 16:11:59 UTC 2021,1623687119,spawn
16 e-3,Mon Jun 14 16:11:59 UTC 2021,1623687119,kill
17 e-3,Mon Jun 14 16:13:46 UTC 2021,1623687226,kill
```

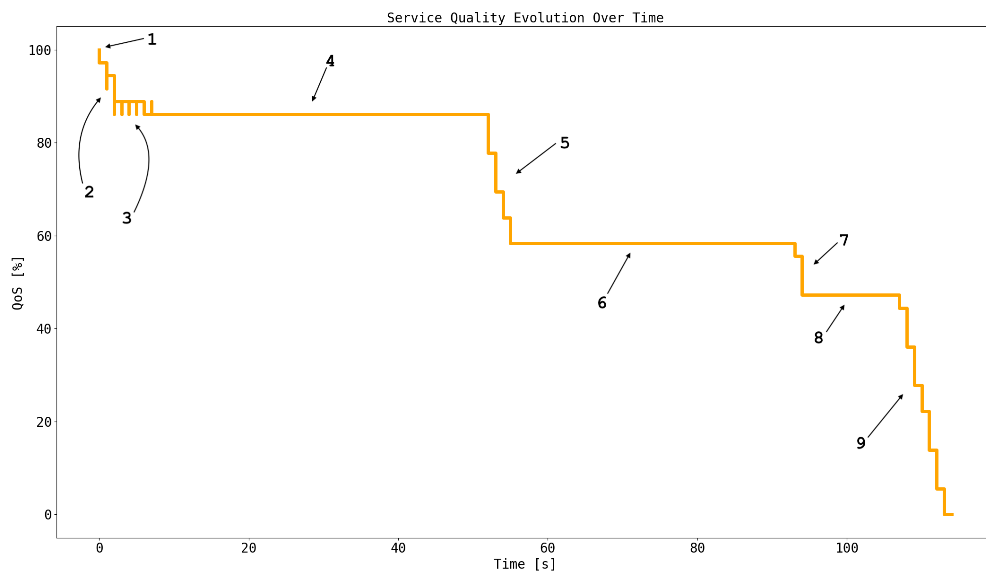


Figure 5.4: Evolution of the *QoS* Over Time for a Dynamic Topology.

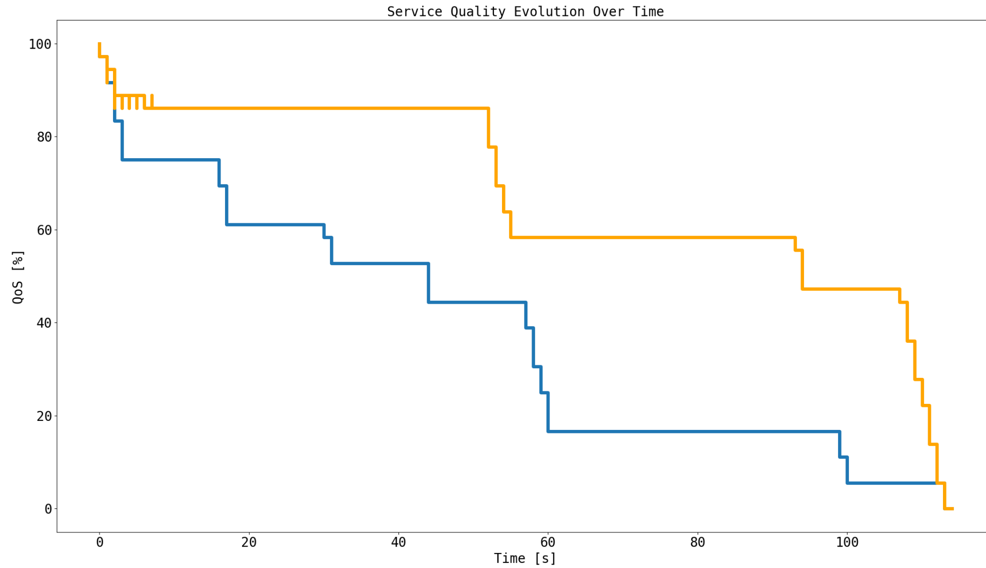


Figure 5.5: Comparison of the Evolution of the *QoS*. *Blue* - Base Case. *Orange* - Mitigated Attack.

```
18 e-3,Mon Jun 14 16:12:00 UTC 2021,1623687120,spawn
19 e-4,Mon Jun 14 16:12:00 UTC 2021,1623687120,kill
20 e-4,Mon Jun 14 16:13:46 UTC 2021,1623687226,kill
21 e-4,Mon Jun 14 16:12:00 UTC 2021,1623687120,spawn
22 r-1,Mon Jun 14 16:12:00 UTC 2021,1623687120,kill
23 r-1,Mon Jun 14 16:13:47 UTC 2021,1623687227,kill
24 r-1,Mon Jun 14 16:12:00 UTC 2021,1623687120,spawn
25 r-2,Mon Jun 14 16:12:00 UTC 2021,1623687120,kill
26 r-2,Mon Jun 14 16:13:47 UTC 2021,1623687227,kill
27 r-2,Mon Jun 14 16:12:01 UTC 2021,1623687121,spawn
28 r-3,Mon Jun 14 16:12:01 UTC 2021,1623687121,kill
29 r-3,Mon Jun 14 16:13:48 UTC 2021,1623687228,kill
30 r-3,Mon Jun 14 16:12:01 UTC 2021,1623687121,spawn
31 r-4,Mon Jun 14 16:12:01 UTC 2021,1623687121,kill
32 r-4,Mon Jun 14 16:13:48 UTC 2021,1623687228,kill
33 r-4,Mon Jun 14 16:12:01 UTC 2021,1623687121,spawn
34 r-5,Mon Jun 14 16:12:01 UTC 2021,1623687121,kill
35 r-5,Mon Jun 14 16:13:48 UTC 2021,1623687228,kill
36 r-5,Mon Jun 14 16:12:02 UTC 2021,1623687122,spawn
37 v-1,Mon Jun 14 16:12:02 UTC 2021,1623687122,kill
38 v-1,Mon Jun 14 16:13:49 UTC 2021,1623687229,kill
39 v-1,Mon Jun 14 16:12:02 UTC 2021,1623687122,spawn
40 v-2,Mon Jun 14 16:12:02 UTC 2021,1623687122,kill
41 v-2,Mon Jun 14 16:13:49 UTC 2021,1623687229,kill
42 v-2,Mon Jun 14 16:12:02 UTC 2021,1623687122,spawn
43 v-3,Mon Jun 14 16:12:02 UTC 2021,1623687122,kill
44 v-3,Mon Jun 14 16:13:49 UTC 2021,1623687229,kill
45 v-3,Mon Jun 14 16:12:03 UTC 2021,1623687123,spawn
46 p-1,Mon Jun 14 16:12:03 UTC 2021,1623687123,kill
47 p-1,Mon Jun 14 16:13:50 UTC 2021,1623687230,kill
48 p-1,Mon Jun 14 16:12:03 UTC 2021,1623687123,spawn
49 p-2,Mon Jun 14 16:12:03 UTC 2021,1623687123,kill
50 p-2,Mon Jun 14 16:13:50 UTC 2021,1623687230,kill
51 p-2,Mon Jun 14 16:12:03 UTC 2021,1623687123,spawn
52 p-3,Mon Jun 14 16:12:04 UTC 2021,1623687124,kill
53 p-3,Mon Jun 14 16:13:51 UTC 2021,1623687231,kill
54 p-3,Mon Jun 14 16:12:04 UTC 2021,1623687124,spawn
55 t-1,Mon Jun 14 16:12:49 UTC 2021,1623687169,kill
56 t-2,Mon Jun 14 16:12:49 UTC 2021,1623687169,kill
57 t-3,Mon Jun 14 16:12:49 UTC 2021,1623687169,kill
58 t-4,Mon Jun 14 16:12:50 UTC 2021,1623687170,kill
59 t-5,Mon Jun 14 16:12:50 UTC 2021,1623687170,kill
60 t-6,Mon Jun 14 16:12:50 UTC 2021,1623687170,kill
61 t-7,Mon Jun 14 16:12:51 UTC 2021,1623687171,kill
62 t-8,Mon Jun 14 16:12:51 UTC 2021,1623687171,kill
63 t-9,Mon Jun 14 16:12:52 UTC 2021,1623687172,kill
64 t-0,Mon Jun 14 16:12:52 UTC 2021,1623687172,kill
65 r-x,Mon Jun 14 16:11:58 UTC 2021,1623687118,kill
66 r-y,Mon Jun 14 16:11:59 UTC 2021,1623687119,kill
```

```
67 r-z,Mon Jun 14 16:11:59 UTC 2021,1623687119,kill
```

Listing 5.3: Retrieved Data Points for the Attack on a Dynamic Network.

The figures we have presented in this section display the evolution of an extremely quick attack. What is more, we have only explored a single quick response to an attack which we have found to be clearly more effective than not taking any action at all. Nonetheless, the real value of this tool is serving as a testing ground for more complex and efficient techniques that are being developed as part of associated research projects. We have decided to prepare this short demonstration in an effort to showcase our tool's potential and we have not even scratched the surface.

Chapter 6

Closing Thoughts and Future Work

> I am not a visionary. I'm an engineer. I'm happy with the people who are wandering around looking at the stars but I am looking at the ground and I want to fix the pothole before I fall in.
>
> *Linus Torvalds*

This project has posed a huge challenge. Given I have had to implement it entirely from the ground up I have had to grapple with several different technologies. In the case of *docker* I have had to push its conception a little bit further in an effort to achieve my purpose. This need implied I had to circumvent several existing limitations derived from the tool itself not being designed for the use I was making of it.

Aside from *docker*, I have also had to get acquainted with the entire *iproute2* suite. Up to now I had only used it to carry out small fixes on my own machines when the networking configuration broke down in some place or another. The situation now called for a deeper understanding of the entire tool collection as well as its interaction with other programs such as *iptables*. All the necessary information is intimately related with the kernel itself, which makes it somewhat harsh and arid. Even though these characteristics do not work towards the documentation's readability I have personally found it to be hugely precise and helpful.

Once I felt comfortable with all the technologies I was to leverage, I began developing the tool itself to automate the deployment of virtual networks. This was the first project I tackled using *python*, and I am aware of the fact that anybody else could have made a much better job, but I managed to produce a working version. Even though the entire development process can be summarized in a single sentence, it was part of the project

that took the longest to complete.

One might argue that this project has not pushed the associated group's research in a particular direction, and it is true. My work is aimed at generating a tool that can be used as an auxiliary resource for the group's ongoing work. It is meant to be used as a validation mechanism to aide in as many articles and experiments as it possibly can. I personally believe that, as engineers, we should not forget to turn our work into tangible products, at least as tangible as software can be. I hope this work contributed to that ideal.

In any case, I am tremendously thankful for having been given the opportunity to devote my time and effort to this project. As in any other case I have had my ups and downs, but if there is one thing I can be sure about is that I have learnt and become a better engineer at every turn of the road.

6.1 Future Work

6.1.1 Possible Improvements

The proof of concept I presented on chapter 5 is only concerned with two different scenarios. In one of them the network remains static whilst in the other I am actively altering the topology in an effort to mitigate the attack. Even though both experiments show clear differences I can still devise many other techniques to try and monitor the network's response to a threat. What is more, I could even apply some of the techniques discussed in [38] to discern how well they respond to the attack. Even though I only presented a couple of situations, I believe they strongly exhibit how the project can withstand much more demanding tests.

Trying to use this project for new techniques might call for some enhancements to the *CLI* whose commands I discussed on section 4.1.2. Given how the tool has been designed, this component is fully independent from the rest, which implies future patches and or features should be easy to integrate. One of the main reasons behind this modularized design is facilitating future work on the tool.

6.1.2 Use as a Teaching Resource

I have a strong teaching tendency. Despite hurtful quotes like the one stating that "he who is not good enough, teaches" I sincerely believe teaching to be a necessary aspect of the academic life. What is more, I consider teaching to be an extremely potent engine

	Our Tool	**VMs**
Network Control	Easy and Total	Hard and Total
Impact on the Host's Resources	Low	High
Ease of Deployment	High	Low
Preexisting Configurations for Nodes	A lot	Not that many

Table 6.1: Comparison of Both Approaches as Teaching Platforms

for social change and, in these times of unrest, that fact becomes more apparent than ever. This feeling is what prompted the idea of repurposing this tool into a framework that could be leveraged as a resource in practical lessons involving the manipulation of network-aware machines and topologies.

I had a course in which I had to work with several *virtual machines* within a given network to then configure routing protocols such as *OSPF* [39] and analyze traffic on the different machines through tools such as *WireShark* [25]. My tool could play the same role *VMs* had in that scenario, and given the existence of tools such as *tcpdump* [40], the same kind of information could be easily extracted. On top of my tool being a more lightweight approach to generating arbitrary topologies it also offers just that: arbitrarily complex networks. From my experience working with virtual machines I can confirm that configuring the underlying network is close to impossible. What is more, they usually require an installation process and when transferring them around they manifest as large files in terms of storage. Table 6.1 summarizes this paragraph neatly.

If this option were to be pursued, the already existing code could easily be turned into a *python package* [34] and be distributed through package indices such as *PyPI* [41], making the deployment of the tool trivially easy on end systems.

Appendix A

Project Budget and Schedule

A.1 Project Schedule

The *Gantt Diagram* found on figure A.1 summarizes the different periods the development process has traversed. We would like to point out that each week is assumed to last 20 *hours*, which brings the project's total duration up to 320 *hours*.

A.2 Budget

Like in any other profession, a developer needs a set of tools and a given amount of time to carry out his or her work in an effective manner. These incur into expenses we are detailing in this section.

A.2.1 Software Costs

We have strived to use only cost-free software in our development. That is why we have incurred in **no** software related costs.

A.2.2 Hardware Costs

Given we are working with virtual infrastructure we only need to account for the cost of our workstation when it comes to budgeting. We have developed the project with the help of a *MacBook Pro* computer sporting a *4 Core Intel i5 CPU at 1,4 GHz* and a *16 GB RAM*. Given we have found the machine's trackpad incredibly comfortable we have not had to acquire any external accessories such as a mouse or external keyboard. The

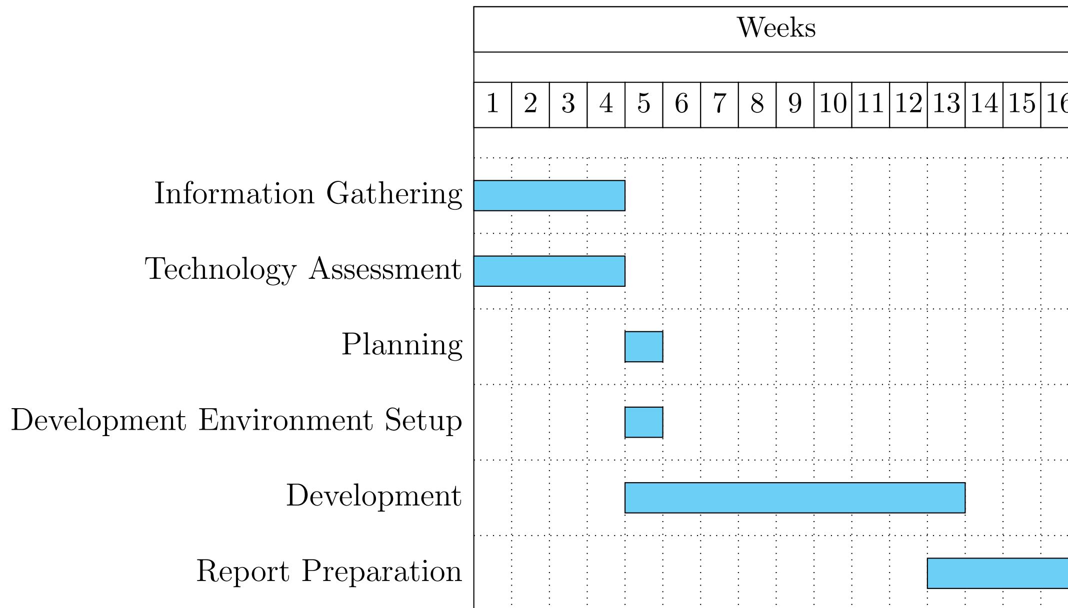


Figure A.1: Project's Development Phases Over Time

Concept	**Units**	**Unit Cost**	**Total Cost (VAT Included)**
Software	0	0 €	0 €
Hardware	1	1200 €	1200 €
Fixed Expenses per Week	16	85 €	1360 €
Salary per hour	320	18 €	5760 €
Total:			8320 €

Table A.1: Project's Budget

computer's cost was 1200 €.

A.2.3 Labour Costs

We have developed the entire project ourselves. Given the common salary of a graduated engineer (18 $\frac{€}{h}$) and the number of ours employed on the project, as provided on figure A.1, the total cost amounts to 5760 €.

When accounting for indirect expenses such as the cost of electricity and internet access, and factoring in all of the above, we can easily obtain the project's budget as detailed on table A.1.

Appendix B

System Setup

One of the main aspects we sought when designing the project was making the installation procedure as simple as possible. The following sections describe how to install and configure all the necessary tools. We will assume the target system is running a *Debian-based* distribution. If that is not the case, we encourage the reader to query his or her distribution's documentation to find out what package manager to use instead of *apt*. Common examples are *dnf* and *pacman* for *Fedora* and *Arch-based* distributions, respectively. Some package names might slightly differ as well: beware.

B.1 Installing External Dependencies

We will describe how one can check whether the necessary tools are present or not. After that, we include listing B.1, which contains the necessary commands to install each of the components in case they are not already present. Note several of the commands contain a leading *sudo* to signify they will require elevated privileges. This might however not be needed in case it is *root* him or herself who is running these commands.

B.1.1 iproute2

This project **requires** a machine running the *linux kernel*. Most modern linux distributions will ship with the *iproute2* suite preinstalled, but if that is not the case it needs to be installed. In order to determine whether *iproute2* is available one can run `which ip`. If the command's output is **not** empty, the necessary tools are present.

B.1.2 Docker

Once can run `docker --version` to determine whether it is present on the system or not. We are including the necessary commands to acquire it on listing B.1, but we encourage the reader to visit [42] as it contains a more comprehensive explanation on the process.

B.1.3 PIP

PIP is *python's* package manager. Running `python3 -m pip --version` aides in determining whether it is present on the system or not. If it is, a message containing information on *PIP's* version will be printed.

B.1.4 Python Modules

Our tool depends on both the *matplotlib* and *networkx* modules. Once *PIP* is installed on the system we can run `python3 -m pip list | grep <module-name>` to determine whether module *¡module-name¿* is present on the system or not. If the previous command shows no output, the specified module is not present on the system. Note listing B.1 explicitly uses *sudo* when installing these modules. We delve a little deeper into that fact in a later section.

```
1  # If any packages are to be installed with apt,
2      # update the repositories.
3  sudo apt update
4
5  # Installing iproute2
6  sudo apt install iproute2
7
8  # Installing docker
9      # Auxiliary packages
10     sudo apt install \
11         apt-transport-https \
12         ca-certificates \
13         curl \
14         gnupg \
15         lsb-release
16
17     # Obtaining Docker's repository GPG key
18     curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
19         | sudo gpg --dearmor -o \
20         /usr/share/keyrings/docker-archive-keyring.gpg
21
22     # Adding Docker's repository to the system
```

```
23     echo \
24         "deb [arch=amd64 signed-by=/usr/share/keyrings/ \
25             docker-archive-keyring.gpg] https://download \
26             .docker.com/linux/ubuntu $(lsb_release -cs) \
27             stable" | sudo tee /etc/apt/sources.list.d/ \
28             docker.list > /dev/null
29
30     # Updating the repositories
31     sudo apt update
32
33     # Installing docker itself
34     sudo apt install docker-ce docker-ce-cli containerd.io
35
36 # Installing PIP
37 sudo apt install python3-pip
38
39 # Installing python modules matplotlib and networkx
40     # Note the can be installed separately.
41 sudo python3 -m pip install matplotlib networkx
```

Listing B.1: Commands for Installing Needed Dependencies.

B.2 A note on Capabilities

On section 4.1.2 we already introduced the term *capability*. As seen on [20], *capabilities* allow for a finer control so as to what a process can and cannot do. What is more, a process spawned by a non-privileged user can take actions traditionally reserved for those started by a privileged user as long as it is granted the necessary capabilities. The reader might recall listing 2.12 in which we were explicitly assigning several capabilities to the containers we were spawning so that they could perform several needed actions. One of the benefits of using containers is that they allow for a tighter control on what the process running within them can and cannot do. That is why they do are not granted all the capabilities by default.

Capabilities are a feature of the linux kernel, and thus they can be applied to other scenarios. One of the conditions for running our program set forth in section 4.1.2 was that it needed to be executed as *root*. The limitation was imposed by the need to leverage the *iproute2* suite for the configuration of the network interfaces associated to the containers. Now, if we apply the concept of capabilities to this particular case we can free ourselves from such a limitation. Given we are passing strings to the `os.system()` function we are exposing the host to injection attacks through, for example, malicious node names. Listing B.2 contains one such example. Even though it is tru careful parsing of the arguments passed to `os.system()` can prevent these kind of attacks we believe the

approach involving capabilities to be much more robust.

```
1 # Note the node name is passed to os.system() "as is".
2     # With the ';' character we will try to run a malformed
3     # command and then execute whatever comes after it. We
4     # are being nice in our example, but if someone tried
5     # to run 'rm -rf /' we could be in deep trouble...
6 net.add_node('; echo "Running with UID = $UID!', type = 'node')
```

Listing B.2: A Malicious Node Name Exploiting Privileges.

A key piece of information extracted from [20] is shown on listing B.3. It shows how the capabilities of a process are transformed when a call to `execve()` [43] takes place. Given how shells such as *bash* [44] work, they rely on `execve()` for launching other programs, which implies the rules set forth on listing B.3 are of the utmost importance.

```
1 P'(ambient)     = (file is privileged) ? 0 : P(ambient)
2 P'(permitted)   = (P(inheritable) & F(inheritable)) |
3                     (F(permitted) & P(bounding)) | P'(ambient)
4 P'(effective)   = F(effective) ? P'(permitted) : P'(ambient)
5 P'(inheritable) = P(inheritable)     [i.e., unchanged]
6 P'(bounding)    = P(bounding)        [i.e., unchanged]
7 where:
8     P()         denotes the value of a thread capability set before
9                     the execve(2)
10    P'()        denotes the value of a thread capability set after
11                    the execve(2)
12    F()         denotes a file capability set
13    &           denotes the AND logical operation between sets.
14    |           denotes the OR logical operation between sets.
15    A ? B : C denotes an IF-ELSE close that can be read as: if
16                    A is TRUE then B ELSE C.
```

Listing B.3: Transformation of Capabilities During `execve()`.

Given the contents of listing B.3 one might feel overwhelmed. The key idea is to somehow make sure that the processes effective set (i.e. `P'(effective)`) contains the capabilities we want to grant. In our case we seek granting the *CAP_NET_ADMIN* capability to the host's *iproute2* binary. This would imply that we would no longer need to run our program as *root*[1], thus denying a huge attack vector.

We must begin by locating the `ip` binary within the system. This can easily be accomplished by running `readlink $(which ip)`. This command will either show the real path to the `ip` or show no output. In the latter case, the binary's real path is given by `which`

[1]Disabling the calls to *iptables* as discussed on section 2.7 would still require privileges. This can nonetheless be done manually without impacting the tool's main use.

`ip`. In our case on a machine running *Ubuntu 20.04* `which ip` returns `/usr/sbin/ip` which is a symbolic link to `/bin/ip`, the real executable.

Now that we have located the program itself we need to make sure that *CAP_NET_ADMIN* makes its way to the `P'(effective)` set. This implies the *effective* (i.e. `F(effective)`) flag must be set for `/bin/ip` and that *CAP_NET_ADMIN* should belong to `P'(permitted)`. In order for this to be true, the *CAP_NET_ADMIN* capability must belong to the file's *permitted* (i.e. `F(permitted)`) set and the same capability must belong to the `P(bounding)` set.

These changes can be made effective through the `setcap` [45] command. Running `sudo setcap cap_net_admin+ep /bin/ip` does indeed configure the capabilities as shown by the output of `getcap /bin/ip` [46]. On top of that, we can leverage the `capsh --print` [47] command to query the current process' (i.e. the *shell's*) *bounding* capabilities (i.e. `P(bounding)`) to indeed check that *CAP_NET_ADMIN* belongs to it. To the extent of what we know this would need to be sufficient to be able to leverage `ip` to carry out network-related tasks. To our uttermost surprise, it was not.

If one browses *iproute2's* source code on [48] line 181 on file *iproute2/ip/ip.c*[2] has a suspicious looking name. The definition of the `drop_cap()` function called on it is found on lines 1571 → 1597 of file *iproute2/lib/utils.c*[3] and is included on listing B.4 for convenience. The key aspect to note is that the function checks whether the *CAP_NET_ADMIN* capability belongs to the process' *inheritable* set (i.e. `P'(inheritable)`) as we have already `execve()`ed from the shell. Given this set remains unchanged, we can rest assured `ip` is effectively checking whether the *CAP_NET_ADMIN* capability belonged to the shell's *inheritable* set (i.e. `P(inheritable)`). As that was **not** the case (we had not modified the shell's binary) then the check on line 18 of listing B.4 evaluates to `true` (i.e 1), which makes makes `ip` drop **all** the capabilities we have explicitly provided through `setcap`.

```
1  void drop_cap(void)
2  {
3  #ifdef HAVE_LIBCAP
4    /* don't harmstring root/sudo */
5    if (getuid() != 0 && geteuid() != 0) {
6      cap_t capabilities;
7      cap_value_t net_admin = CAP_NET_ADMIN;
8      cap_flag_t inheritable = CAP_INHERITABLE;
9      cap_flag_value_t is_set;
10
11     capabilities = cap_get_proc();
```

[2] https://github.com/shemminger/iproute2/blob/main/ip/ip.c#L181
[3] https://github.com/shemminger/iproute2/blob/main/lib/utils.c#L1571-L1597

```
12     if (!capabilities)
13       exit(EXIT_FAILURE);
14     if (cap_get_flag(capabilities, net_admin, inheritable,
15         &is_set) != 0)
16       exit(EXIT_FAILURE);
17     /* apps with ambient caps can fork and call ip */
18     if (is_set == CAP_CLEAR) {
19       if (cap_clear(capabilities) != 0)
20         exit(EXIT_FAILURE);
21       if (cap_set_proc(capabilities) != 0)
22         exit(EXIT_FAILURE);
23     }
24     cap_free(capabilities);
25   }
26 #endif
27 }
```

Listing B.4: Defintion of the `drop_cap()` Function.

The check on line 18 opens up a door for executing `ip` with the aid of capabilities. When we read the `drop_cap()` function we began wondering the motivation behind it. Given our approach of adding the *CAP_NET_ADMIN* capability to both the *effective* and *permitted* sets for `/bin/ip`, we are effectively allowing any user modify the machine's network configuration. This poses a potential security risk. In our quest for a solution to this limitation we encountered a solution that circumvents this issue whilst allowing a more fine grained approach to the management of capabilities.

The *Linux Pluggable Authentication Modules (PAM)* [49] provides an interface for programs to access several authentication schemes. In our case we **are not** concerned with the authentication facilities *PAM* provides: we just seek the ability to somehow modify a shell's *inheritable* capabilities set so that these can be passed on to `/bin/ip` when run from the shell. In order to do so we can leverage the *pam_cap PAM module* [50] which in turn relies on the `/etc/security/capability.conf` file to set the current process' inheritable capabilities. We just need to use the *pam_cap* module within one of the configuration files under the `/etc/pam.d` directory. We settled on using the `/etc/pam.d/login` file as it offered a service required by login shells like the one we will launch our program from. One just needs to add the `auth required pam_cap.so` line so that the module becomes effective. By then specifying a line such as `cap_net_admin <username>` on `/etc/security/capability.conf` we would be making processes spawned by user `username` that relied on the *login* service contain the *CAP_NET_ADMIN* capability in its *inheritable* set (i.e. `P(inheritable)`). This can indeed be checked if we run `capsh --print` from a new shell after having modified the *PAM*-related configuration.

Now that the process that will be launching `/bin/ip` contains the *CAP_NET_ADMIN* capability in its *inheritable* set we just need to recall the contents of listing B.3. As capa-

bilities will not be dropped by `drop_cap()` by the test on line 18 evaluating to `true` we just need to make sure *CAP_NET_ADMIN* will make its way into `P'(effective)`. In order to achieve that we need to activate the *effective* flag for `/bin/ip` and add said capability to its *inheritable* set (i.e. `F(inheritable)`), as it will be *AND*ed with the parent process' *inheritable* capability set. In other words, we need to run `setcap cap_net_admin=ei /bin/ip`.

With the configuration we have just described user `username` will be capable of executing any *iproute2* command without depending in `sudo`. As seen on [20], capability sets are maintained across `fork()`s. Given `system()` relies on `fork()` we can rest assured this approach will be compatible with how we are interacting with the *iproute2* suite. Capabilities might come across as confusing at first sight, but they are a great mechanism to gain a firmer grasp on a system's security policies.

B.3 Interacting With Docker as a Regular User

After following the installation procedure outlined before the user might find he or she cannot execute *docker* commands. This is due to the fact that only *root* can do that by default. However, given permissions are based on both *users* and *groups* on *Unix*-like systems we can easily grant any user the ability to interact with the *docker engine*. By running `sudo usermod -aG docker <username>` we would be granting user `username` the ability to use *docker's CLI*. The `usermod` [51] command will just add user `username` to the `docker` group. This group is automatically created during *docker's* installation. Please note that the changes will not come into effect until the user in question logs in again after being added to the `docker` group (i.e. the user will need to spawn another *shell*).

If we combine this setup with the one outlined in the previous section we can avoid having to run our tool as *root*, thus providing stricter security measures and a more elegant design.

B.4 Acquiring the Project's Code

This document has only described the functions, classes and methods we have defined to provide a tool capable of automating the deployment and management of full virtual networks. This section exposes how one can gain access to the entire source code so that he or she can run the code and use it as they please.

B.4.1 Leveraging git

As per [52], "*git* is a fast, scalable, distributed revision control system". We have decided to leverage its capabilities to aide in our development even though we were working on our own. It has allowed us to manage our code in a resilient way, keep several synchronised backups and keep our source tree tidy at all times.

In our opinion, one of *git's* best features is that it allows users to quickly obtain a copy of a project. The next section is devoted to explaining how anybody can obtain the product of our work and how they can also propose changes in such a way that handling them is an almost effortless procedure.

Cloning the Repository

Git is concerned with managing repositories. Our code exists as a *remote repository* that a user may *clone* to obtain a local copy of it. Thus, by issuing the `git clone https://github.com/pcolladosoto/cld_wall.git` command a directory named `cld_wall` will be created in the current working directory. Said directory is just a *local copy* of the *remote repository*. It allows the user not only access ti the current version of the code, but also to its entire history. This is by no means a comprehensive guide on how to use *git*. We encourage the reader to read through [52] and also to consider reading [53] if he or she finds the topic interesting. We have also prepared a small presentation (albeit in Spanish) that can be found in [54] covering the basics of *git*.

Proposing Changes

Given the project's source code is hosted on https://github.com one can leverage the mechanisms it provides for handling a project's lifespan. We personally find *issues*[4] to be quite intuitive and we encourage any user wanting to propose an enhancement or point out a *bug* to open an issue at https://github.com/pcolladosoto/cld_wall/issues. We will try to do our best to reply to any requests we might receive.

[4] https://guides.github.com/features/issues/

Appendix C

Comprehensive Module Analysis

C.1 The virt_net Module

Constants.py

In an effort to reduce the number of dependencies we decided to leverage *ANSI Escape Codes* to colour the different output messages. This module then defines the different `strings` controlling the colours.

Imported Libraries None as there are no external dependencies.

Global Variables

1. `terminal_escape_sequences (dictionary/string/string): Dictionary` keyed by color names whose associated values are *ANSI Escape Sequences* changing the terminal's output color to the one specified. This solution is **not** to be considered portable. It **should** nonetheless work on any regular terminals supporting colors. We can assure it works with the following terminal emulators: *tilix*, *kitty* and *Visual Studio Code's Embedded Terminal Emulator*.

Interface.py

This class will be in charge of providing a node's network functionality. An interface represents a connection with a computer network and thus is defined by a key parameter: the IP address. We should stress how an IP address is associated with an interface, **not** with the host itself. This subtlety will be extremely relevant when dealing with routers.

Another key aspect of interfaces is the role they play in network routes. Even though it is not entirely correct, in our case we can talk about the interface's routes. If we are

being precise we would consider routes to be associated to a machine's network stack, not to a given interface. Nonetheless, as the graphs we are to instantiate will always define topologies in such a way that only an interface per host will belong to a given subnet we can indeed associate a route with an interface for a given route will only egress through a specific interface. This distinction will allow us to quickly and univocally query a node's route configuration.

Once an interface is configured it will just support a machine's connections. It is also crucial to note that even though an instance of the `interface` class is **not** the same thing as a *veth*, they are intimately related. Instantiating an `interface` will trigger the creation of a *veth* and deleting the former will also remove the latter. All in all, an interface's lifecycle can be described as:

1. An interface instance is created and the associated *veth* is created as well.
2. The interface is assigned an IP address.
3. One or more routes are assigned to or deleted from the interface.
4. The interface sits idle until it returns to 2 or 3 or goes to 5.
5. The interface is removed, which will also remove the corresponding *veth*.

Imported Libraries

1. `os`: This module enables the execution of commands through a *sh* shell through the `os.system()` method. One can check the shell being spawned is indeed *sh* by running `os.system("echo $0")` or by querying [55].
2. `constants`: Grant access to the `terminal_escape _sequences dictionary` containing `ANSI escape sequences` enabling colored output.

Global Variables

1. `t_colors - dictionary/string/string`: This is a synonym for the `terminal_escape_sequences dictionary` we me mentioned before. It is used within calls to `print()` so that we can alter the terminal text's color allowing for a more visual information representation.

The interface Class This class represents a host's interface.

Class Attributes

1. `self.host_name - string`: Name identifying the host this interface belongs to.
2. `self.ip_range - string`: Interface's IP and subnet mask in *CIDR* [56] format. Ex: *10.0.0.1/24*.
3. `self.ip - string`: Interface's IP address.
4. `self.if_name - string`: Interface's name. This is equivalent to traditional names like `enp2s0` or `wlp3s0` in Linux-based systems. In our case the names will follow a pattern given by `veth_+_-` where `+` and `-` are the names of the connected nodes. Note either `+` or `-` will be the same as `self.host_name`. We would finally like to point out that `+` and `-` can be either upper or lowercase, depending on which is the first node to be attached to the links we create.
5. `self.subn_gateway - string`: IP address of this subnet's gateway, following the same format as `self.ip`.
6. `self.subn - string`: Interface's subnet given as the network address for said subnet together with the subnet mask in `CIDR` notation. Ex: *10.0.0.0/24*.
7. `self.outbound_interface - boolean`: `True` if the interface's host is allowed to connect to external subnets. `False` otherwise. Due to how some ICS networks need to isolate some equipment, such as reactor's control systems, we need to know whether a given host is allowed to "see" the outside world. Having this information encoded into interfaces allows for a cleaner design when routing the network.

The Constructor

1. **Parameters:**
   (a) `self - interface_inst`: The actual interface instance.
   (b) `h_name - string`: The host the interface belongs to.
   (c) `if_name - string`: The interface name
   (d) `subnet - string`: The subnet the interface belongs to.
   (e) `h_type - boolean`: `True` if the interface belongs to a regular node, `False` otherwise.
   (f) `out_interface - boolean - optional`: `True` if the node is allowed to "see" other subnets, `False` otherwise.
2. **Returns:** Nothing.
3. **Description:** This method will initialize the members of an `interface_inst` and call the `_activate_iface()` method.

The _activate_iface() Method

1. **Parameters:**
   (a) `self - interface_inst`: The actual interface instance.
2. **Returns:** Nothing.
3. **Description:** This method leverages the `os.system()` function to activate a *veth* interface. The `string` we pass `os.system()` is built though the concatenation of several instance attributes. As we usually do, we check whether `os.system()`'s return code was 0 to show an informative message to the user.

The get_if_subnet() Method

1. **Parameters:**
   (a) `self - interface_inst`: The actual interface instance.
2. **Returns:** A `string` containing the subnet the interface belongs to in *CIDR* notation.
3. **Description:** Not applicable.

The get_if_ip_range() Method

1. **Parameters:**
   (a) `self - interface_inst`: The actual interface instance.
2. **Returns:** A `string` containing the interface's IP and subnet mask in *CIDR* format.
3. **Description:** Not applicable.

The get_if_ip() Method

1. **Parameters:**
   (a) `self - interface_inst`: The actual interface instance.
2. **Returns:** A `string` containing the interface's IP.
3. **Description:** Not applicable.

The get_if_name() Method

1. **Parameters:**
   (a) `self - interface_inst`: The actual interface instance.
2. **Returns:** A `string` containing the interface's name.
3. **Description:** Not applicable.

The get_routes() Method

1. **Parameters:**
   (a) `self - interface_inst`: The actual interface instance.
2. **Returns:** A `dictionary/string/string` containing the routes that have been assigned through this interface.
3. **Description:** Not applicable.

The set_if_name() Method

1. **Parameters:**
   (a) `self - interface_inst`: The actual interface instance.
   (b) `if_name - string`
2. **Returns:** Nothing.
3. **Description:** Assigns the value of the `if_name` parameter to the `self.if_name` attribute.

The assign_addr() Method

1. **Parameters:**
   (a) `self - interface_inst`: The actual interface instance.
   (b) `ip_range - string`: The interface's IP address together with its subnet mask.
   (c) `gw_addr - string - optional`: IP address of the subnet's gateway (i.e. router).
   (d) `subnet - string - optional`: The subnet the interface belongs to in *CIDR* format.
2. **Returns:** Nothing.

3. **Description:** This method lets the caller assign an IP address to the interface. The method will check that the interface name is valid and that the interface had no previous address to avoid errors. Note that when a node moves within the network its interface **will not** be reconfigured: a new one will be added to the node. Once the addressing has been successfully completed a message stating that will be printed.

The assign_route() Method

1. **Parameters:**
   (a) `self - interface_inst`: The actual interface instance.
   (b) `dest_subnet - string`: Destination subnet for the route in *CIDR* format.
   (c) `gw_router - string`: IP of the gateway for the interface's subnet.
2. **Returns:** Nothing.
3. **Description:** After checking the interface has been assigned an IP address and that it is allowed to communicate with other subnets through the `self.outbound_interface` attribute it will add the route defined by the parameters to the `self.routes dictionary`.

The reset_interface() Method

1. **Parameters:**
   (a) `self - interface_inst`: The actual interface instance.
2. **Returns:** Nothing.
3. **Description:** This method will wipe the `self.routes dictionary` and remove the interface's IP address. This will also delete any preexisting routes. As usual, an informative message will inform of the operation's success.

The remove_interface() Method

1. **Parameters:**
   (a) `self - interface_inst`: The actual interface instance.
2. **Returns:** Nothing.
3. **Description:** This method will delete the associated interface, together with all the routes, and print an informative message upon completion.

The Destructor

1. **Parameters:**
   (a) `self - interface_inst`: The actual interface instance.
2. **Returns:** Nothing.
3. **Description:** This method will just call `remove_interface()`.

Subnet_Machines.py

This file contains the class definitions for the elements belonging to a subnet: *bridges* and *nodes*. We would like to clarify that we refer to regular *hosts* as *nodes* interchangeably.

We have purposefully decided **not to consider** *routers* as part of a subnet. The logic behind the decision is that a *router* belongs to at least two subnets in our topologies. Thus, defining its class in a file whose context is that of a single subnet did not seem appropriate.

Once again, we should point out how an instance of the `k_bridge` class **is not** the same thing as the actual *bridge*, but they are intimately related. The following explains the lifecycle of both classes defined in this file.

1. The real instance, either a *bridge* or a *host*, is brought up.
2. The object representing said instance is created.
3. The object sits idle. Several of its parameters can be altered within this state.
4. At some point, the object will be dismantled.
5. The release of the object will trigger the removal of the associated real instance.

Imported Libraries

1. `os`: This module enables the execution of commands through a *sh* shell through the `os.system()` method. One can check the shell being spawned is indeed *sh* by running `os.system("echo $0")` or by querying [55].
2. `constants`: Grant access to the `terminal_escape_sequences dictionary` containing *ANSI escape sequences* enabling colored output.
3. `sys`: This module allows us to print error messages to *STDERR* instead of *STDOUT* so that they can be easily redirected later on if needed with `2>/path/to/log`.

4. `subprocess`: This module enables the execution of commands through a shell **and** allows the caller to retrieve the command's output to *STDOUT* on top of its return code. This will let us retrieve a container's associated *PID*.

5. `interface`: This module will let us instantiate and add `interface_inst` to the nodes we create.

Global Variables

1. `t_colors - dictionary/string/string`: This is a synonym for the `terminal _escape_sequences dictionary` we me mentioned before. It is used within calls to `print()` so that we can alter the terminal text's color allowing for a more visual information representation.

The k_bridge Class This class represents a virtual bridge. Note the `'k'` stands for kernel as these bridges are part of the kernel itself.

Class Attributes

1. `self.type - string`: Object type identifier containing the `''bridge'' string` for bridges. It is used within functions to check the type of object it is currently dealing with.

2. `self.name - string`: Bridge's name.

3. `self.up - boolean`: `True` if the bridge is currently up (i.e. "switched on"). `False` otherwise.

4. `self.subnet - string`: Bridge's associated subnet given as the network address for said subnet together with the subnet mask in *CIDR* notation.

The Constructor

1. **Parameters:**

   (a) `self - k_birdge_inst`: The actual bridge instance.

   (b) `name - string`: The bridge's name.

   (c) `subnet - string`: The subnet associated with this bridge.

2. **Returns:** Nothing.

3. **Description:** This method will initialize the members of an `k_bridge_inst` and call the `_activate_bridge()` method.

The _activate_bridge() Method

1. **Parameters:**
   (a) `self - k_bridge_inst`: The actual bridge instance.
2. **Returns:** Nothing.
3. **Description:** After the bridge has been instantiated it must be brought up so that it can be operated normally. We do so through the `ip link` command, printing a message on success. Once the bridge is activated, the `self.up` attribute will be set to `True`.

The get_name() Method

1. **Parameters:**
   (a) `self - k_bridge_inst`: The actual bridge instance.
2. **Returns:** A `string` containing the bridge's name.
3. **Description:** Not applicable.

The get_type() Method

1. **Parameters:**
   (a) `self - k_bridge_inst`: The actual bridge instance.
2. **Returns:** The `‘‘bridge’’ string`.
3. **Description:** Not applicable.

The get_subnet() Method

1. **Parameters:**
   (a) `self - k_bridge_inst`: The actual bridge instance.
2. **Returns:** A `string` containing the bridge's associated subnet in *CIDR* format.
3. **Description:** Not applicable.

The remove() Method

1. **Parameters:**

   (a) `self - k_bridge_inst:` The actual bridge instance.

2. **Returns:** Nothing.

3. **Description:** This method shuts the bridge down. It will check it is indeed activated before doing so in an effort to prevent errors provoked by calling `ip link` with incorrect parameters (i.e. such as by trying to shutdown a bridge that is already down). This method will print a message to *STDOUT* informing whether the operation was a success or not.

The Destructor

1. **Parameters:**

   (a) `self - k_bridge_inst:` The actual bridge instance.

2. **Returns:** Nothing.

3. **Description:** This method will just call `remove()`.

The d_node Class This class represents a node implementing a virtual host. Note the `'d'` stands for docker.

Class Attributes

1. `self.type - string:` Object type identifier containing the `''node'' string` for nodes. It is used within functions to check the type of object it is currently dealing with.

2. `self.name - string:` Node's name.

3. `self.pid - int:` The associated container's *PID*. It is used when linking the container's network namespace to `/var/run/netns` so that we can easily access it afterwards.

4. `self.interfaces - list/interface_inst:` A list containing the instances of the interfaces belonging to this node.

The Constructor

1. **Parameters:**
   (a) `self - d_node_inst`: The actual node instance.
   (b) `name - string`: The node's name.
2. **Returns:** Nothing.
3. **Description:** This method will just initialize several members with the passed parameters whilst assigning sane defaults to others. It leverages the power of the `docker inspect` command to find the associated container's *PID*. It will also call `_link_net_namespace()` to allow an easier handling of the container's namespace through the `ip` command. It finally calls `_set_hostname()` to configure the node's identity from the network's perspective.

The _link_net_namespace() Method

1. **Parameters:**
   (a) `self - d_node_inst`: The actual node instance.
2. **Returns:** Nothing.
3. **Description:** This method will create a symbolic link to the container's network namespace under `/var/run/netns` where the `ip` command will "look for" existing named namespaces. This method leverages the `self.pid` attribute that is initialized in the constructor to find the original location of the container's namespace so that it can later be linked.

The _set_hostname() Method

1. **Parameters:**
   (a) `self - d_node_inst`: The actual node instance.
2. **Returns:** Nothing.
3. **Description:** This method will just set the container's hostname. It is essential to distinguish between the container's name and its hostname. The former is an identifier that only concerns docker, whilst the latter will be the machine's identifier from the point of view of the network. This method will make the latter the same as the former. What is more, this operation made us change the underscores (`_`) in the original node names by hyphens (`-`): the former were not allowed on network hostnames.

The get_name() Method

1. **Parameters:**
   (a) `self - d_node_inst:` The actual node instance.
2. **Returns:** A `string` containing the node's name.
3. **Description:** Not applicable.

The get_type() Method

1. **Parameters:**
   (a) `self - d_node_inst:` The actual node instance.
2. **Returns:** The `‘‘node’’ string`.
3. **Description:** Not applicable.

The get_subnets() Method

1. **Parameters:**
   (a) `self - d_node_inst:` The actual node instance.
2. **Returns:** A `list/string` containing the different subnets this node is connected to in *CIDR* format.
3. **Description:** This method will iterate over the `interface_inst` contained in the `self.interfaces` attribute to then provide a list containing the subnets each of these `interface_inst` belong to. Even though nodes will usually belong to a single subnet this method allows for a more extensible design that can accommodate for nodes belonging to different subnets in more complex topologies.

The get_interface() Method

1. **Parameters:**
   (a) `self - d_node_inst:` The actual node instance.
   (b) `subnet - string - optional:` Specifies the subnet for which an associated `interface_inst` should be returned.
2. **Returns:** The following are evaluated in order.

(a) A `list/interface_inst` if the `subnet` parameter was not specified.

(b) An `interface_inst` associated to the subnet specified in the `subnet` parameter.

(c) `None` if there is no interface associated with the subnet specified through the `subnet` parameter.

3. **Description:** As seen on the **returns** section, the `subnet` parameter behaves like a switch. A `list` of `interface_inst` will be returned if it is not specified by the caller. In case it is provided, it will return either an `interface_inst` or `None` if there is no interface associated with the specified subnet.

The create_interface() Method

1. **Parameters:**

   (a) `self - d_node_inst:` The actual node instance.

   (b) `if_name - string:` Name of the interface to create.

   (c) `if_subnet - string:` Subnet associated with the interface to create.

   (d) `o_if - boolean:` Flag indicating whether the interface to create should be allowed to "see" other subnets.

2. **Returns:** Nothing.

3. **Description:** This method will instantiate a new interface and add it to the node's `self.interfaces list`. The method's parameters will be passed directly to the `interface` class constructor.

The reset_interfaces() Method

1. **Parameters:**

   (a) `self - d_node_inst:` The actual node instance.

2. **Returns:** Nothing.

3. **Description:** This method erases all the interfaces associated with the node through the `clear()` [57] method of the `self.interfaces list`. This method will *implicitly* call the destructor for each interface, thus effectively deleting the underlying *veths*.

The stop() Method

1. **Parameters:**

   (a) `self - d_node_inst`: The actual node instance.

2. **Returns:** A `boolean` indicating whether the operation succeeded or not.

3. **Description:** This method is in charge of stopping the instance's associated docker container. We can do so thanks to the `docker stop` command. This method leverages a redirection to `/dev/null` so that the output generated by `docker stop` does not interfere with our own. This method will print a message both upon success and error.

The remove() Method

1. **Parameters:**

   (a) `self - d_node_inst`: The actual node instance.

2. **Returns:** Nothing.

3. **Description:** This method is in charge of entirely removing the associated container. It will call the `stop()` method as a container must be stopped before it can be removed through the `docker rm` command. First of all, the method will get rid of the interfaces as soon as possible so that, even if something goes wrong when deleting the container, it has no connection to a network that might still be alive. This will in fact isolate the container so that possible errors have no impact on the rest of the virtualized scenario. We will also delete the link to the container's network namespace living under `/var/run/netns`. As usual we will print information relative to the command's outcome.

The Destructor

1. **Parameters:**

   (a) `self - d_node_inst`: The actual node instance.

2. **Returns:** Nothing.

3. **Description:** This method will call the `remove()` which will in turn delete the associated container. Given the definition of the `remove()` method this will later allow us to dismantle the entire network with a single order.

Subnet.py

This file defines a class representing an entire subnet. This class acts as the gateway through which bridges and nodes can be instantiated. In an effort to attain a firmer logical structure, instances of this class are the **only** means of adding more bridges and nodes to the network.

When a subnet is instantiated it will contain no nodes or bridges, these will be added over time. We would like to stress how, even though a subnet will keep track of the routers it contains, it will by no means control them. That is, routers will not be added to or deleted from a subnet through a subnet instance. These will be instead controlled from the net class we will delve into later.

A subnet's lifecycle is summarized by the following enumeration:

1. The subnet is instantiated.
2. Bridges and nodes are created and/or deleted through this instance.
3. The subnet is removed from the overall network instance when the network is being dismantled.
4. Upon deletion a subnet will remove all its associated bridges and nodes.

Imported Libraries

1. `os`: This module enables the execution of commands through a *sh* shell through the `os.system()` method. One can check the shell being spawned is indeed *sh* by running `os.system("echo $0")` or by querying [55].
2. `constants`: Grant access to the `terminal_escape_sequences dictionary` containing *ANSI escape sequences* enabling colored output.
3. `sys`: This module allows us to print error messages to *STDERR* instead of *STDOUT* so that they can be easily redirected later on if needed with `2>/path/to/log`.
4. `subnet_machines`: This module will let us instantiate and add `k_bridge_inst`s and `d_node_inst`s to the subnets we create.

Global Variables

1. `t_colors - dictionary/string/string`: This is a synonym for the `terminal_escape_sequences dictionary` we me mentioned before. It is used within calls to `print()` so that we can alter the terminal text's color allowing for a more visual information representation.

The d_subnet Class This class represents an entire subnet. Note the `'d'` stands for docker, just like with the `d_node` class. We would also like to mention that we decided to leverage `lists` instead of `dictionaries` for the attributes holding references to the different subnet components which we will analyze below. The reason behind it is that using a node or bridge name as a `dictionary` key when it is also an attribute of said instances seemed redundant. The reader might notice how the `get_bridges()` method is missing. We purposefully decided not to define it given it did not prove to be necessary under any circumstances.

Class Attributes

1. `self.subnet_addr - string:` subnet's address given as the network address as well as the subnet mask in *CIDR* format.
2. `self.bridges - list/k_bridge_inst:` A `list` of the active bridges in the subnet. Our topologies only have a single bridge per subnet but we nonetheless decided to allow for more bridges within a subnet in case a user wanted to expand on our work.
3. `self.nodes - list/d_node_inst:` A `list` of the nodes belonging to this subnet.
4. `self.routers - list/d_router_inst:` A `list` containing references to the routers that have an interface belonging to this subnet.

The Constructor

1. **Parameters:**
   (a) `self - d_subnet_inst:` The actual subnet instance.
   (b) `subnet - string:` The subnet's address.
2. **Returns:** Nothing.
3. **Description:** This method will just initialize the `self.subnet_addr` with the passed parameter and the rest with empty `lists`.

The get_subnet_addr() Method

1. **Parameters:**
   (a) `self - d_subnet_inst:` The actual subnet instance.
2. **Returns:** A `string` containing the subnet's address in *CIDR* format.
3. **Description:** Not applicable.

The get_nodes() Method

1. **Parameters:**
   (a) `self - d_subnet_inst:` The actual subnet instance.
2. **Returns:** A `list/d_node_inst` containing the instances of the nodes belonging to this subnet.
3. **Description:** Not applicable.

The get_routers() Method

1. **Parameters:**
   (a) `self - d_subnet_inst:` The actual subnet instance.
2. **Returns:** A `list/d_router_inst` containing the instances of the routers with at least one interface belonging to this subnet.
3. **Description:** Not applicable.

The add_node() Method

1. **Parameters:**
   (a) `self - d_subnet_inst:` The actual subnet instance.
   (b) `node - d_node_inst:` The node we are to add to the subnet.
2. **Returns:** Nothing.
3. **Description:** This method will add an **existing** node to this subnet. This method's main use is reconfiguring the state of the network when a node is moved.

The add_router() Method

1. **Parameters:**
   (a) `self - d_subnet_inst:` The actual subnet instance.
   (b) `node - d_router_inst:` The router we are to add to the subnet.
2. **Returns:** Nothing.
3. **Description:** This method will add an **existing** router to this subnet. This method's main use is reconfiguring the state of the network when an entire subnet is moved.

The create_bridge() Method

1. **Parameters:**
   (a) `self - d_subnet_inst`: The actual subnet instance.
   (b) `name - string`: The name of the bridge we are to create.
2. **Returns:** The created `k_bridge_inst` or `None` in case of failure.
3. **Description:** This method will create a "real instance" of a bridge together with the object logically representing it. The bridge's name is the passed parameter. It does so through the `ip link` command. A message will be printed on both success and failure.

The create_node() Method

1. **Parameters:**
   (a) `self - d_subnet_inst`: The actual subnet instance.
   (b) `name - string`: The name of the node we are to create.
2. **Returns:** The created `d_node_inst` or `None` in case of failure.
3. **Description:** The method will just create a docker container and the associated `d_node_inst` which will be appended to the `self.nodes` attributes. The name for both the container and the instance will be the passed parameter.

The remove_bridge() Method

1. **Parameters:**
   (a) `self - d_subnet_inst`: The actual subnet instance.
   (b) `name - string`: The name of the bridge we are trying to remove.
2. **Returns:** Nothing.
3. **Description:** This method will remove a bridge instance from the `self.bridges` attribute, thus removing the "real instance" in the process as well. A message will be printed upon both a successful and unsuccessful completion.

The remove_node() Method

1. **Parameters:**
   (a) `self - d_subnet_inst`: The actual subnet instance.
   (b) `name - string`: The name of the node we are trying to remove.
2. **Returns:** Nothing.
3. **Description:** This method will remove a node instance from the `self.nodes` attribute, thus removing the "real instance" in the process as well. A message will be printed upon both a successful and unsuccessful completion.

The remove_node_instance() Method

1. **Parameters:**
   (a) `self - d_subnet_inst`: The actual subnet instance.
   (b) `node_inst - d_node_inst`: The node we are to remove.
2. **Returns:** Nothing.
3. **Description:** This method will remove a node instance form the subnet's `self.nodes` attribute but it **will not** remove the node itself. It is extremely important to grasp this subtle difference. This method is invoked when moving nodes around the net and would not be strictly needed in a context where the virtualized network is totally static. It will help us maintain an accurate network representation upon network changes. If this method were not to exist it would imply that our network representation would deviate from the actual virtual network whenever we altered the topology.

The remove_router() Method

1. **Parameters:**
   (a) `self - d_subnet_inst`: The actual subnet instance.
   (b) `name - string`: The name of the router we are trying to remove.
2. **Returns:** Nothing.
3. **Description:** This method will remove a router instance from the `self.routers` attribute. However, said router instance will always be referenced somewhere else. This implies that this deletion will not trigger the removal of the "real instance". Given this method is being called quite often, we decided not to print any messages related to the operation's outcome as they proved to be more of a nuisance than a helpful feature.

The Destructor

1. **Parameters:**
   (a) `self - d_subnet_inst`: The actual subnet instance.
2. **Returns:** Nothing.
3. **Description:** Due to how we have implemented the destructors in both the `d_node` and `k_bridge` classes we can just empty the lists containing all the references to the subnet's elements. This will trigger the orderly deletion of every expendable component. This destructor is not explicitly needed: when an instance of the `d_subnet` class is deleted, the attributes holding the references to the nodes and bridges will be deleted as well. That will in turn have the same effect as explicitly freeing all the attributes but we believe it is alwys better to be explicit when writing code tan relying on "obscure" mechanisms.

Veth.py

When we began this project we decided to represent each *veth* as its own object. This approach proved to be quite cumbersome once we implemented the functionality allowing the user to modify the network topology. At that point we decided to define the `interface` class and implement the functionality connecting *veths* as a standalone function. Given an interface is the manifestation of a *veth* once we move it to a different namespace we felt this slight change did not go against the pre-existing philosophy.

Like we mentioned before, we did not define a function altering the *veth's* status (i.e. we will not disconnect and reconnect them). When moving nodes around we will altogether delete and create *veths* as needed. We can work in such a manner because we can rely on the kernel's interface management to a great extent. Given how *veths* are implemented we can be sure that as soon as we remove one of its ends through `ip link` the associated termination will be seamlessly removed too. This allows for a "fire and forget" approach in the sense that we need not carry out the elaborate bookkeeping that we would otherwise have to. Stepping on functionality implemented by others is a great way to greatly simplify the code, so we decided to leverage it as long as we could be positive no unexpected behaviour would take place.

After the above discussion let us dive into how this crucial function is implemented. Please note we are now referring to a *function* and not a *method* because the former is not associated to a class whilst the latter is.

Imported Libraries

1. `os`: This module enables the execution of commands through a *sh* shell through the `os.system()` method. One can check the shell being spawned is indeed *sh* by running `os.system("echo $0")` or by querying [55].

2. `constants`: Grant access to the `terminal_escape_sequences dictionary` containing *ANSI escape sequences* enabling colored output.

3. `sys`: This module allows us to print error messages to *STDERR* instead of *STDOUT* so that they can be easily redirected later on if needed with `2>/path/to/log`.

Global Variables

1. `t_colors - dictionary/string/string`: This is a synonym for the `terminal_escape_sequences dictionary` we me mentioned before. It is used within calls to `print()` so that we can alter the terminal text's color allowing for a more visual information representation.

The connect_veth_end() Function

1. **Parameters:**

   (a) `node - k_bridge_inst | d_node_inst | d_router_inst`: The network-aware machine we are to connect the *veth* end to. Note the `'|'` character is to be read as *OR*.

   (b) `veth_name - string`: The name of the *veth* end we are to connect to the network-aware machine specified by the `node` parameter.

   (c) `subnet - string - optional`: The subnet the connected *veth* end will belong to in *CIDR*. This parameter **must** be specified is the network-aware machine we are attaching the *veth* to is either a node or a router.

2. **Returns:** A `boolean` indicating whether the operation completed successfully (`True`) or failed `False`.

3. **Description:** This function will connect the *veth* end specified by the `veth_end` parameter to the network machine whose reference is passed through the `node` parameter. In case the latter is either a node or a router, the subnet the resulting interface will belong to is specified in the `subnet` parameter. This function relies on the `get_type()` method we have defined for every class representing a network-aware machine to resolve how to proceed. The function prints a message informing about the outcome of the operation besides returning a `boolean` encoding the same

information. Please note that attaching a *veth* end to a bridge **will not** trigger the instantiation of an object whilst an `interface_inst` will be created if it is attached to either a node or a router. What is more, this is the only method capable of creating *veths*. This approach has proven to allow for faster changes throughout the codebase when they were required.

Net_machines.py

This file plays a role quite similar to that of the `Subnet_machines.py` file we described above. This file contains the definition of the class representing the routers we will use in our topologies. Even though routers, like nodes, are implemented as docker containers they behave in a slightly different way.

Routers will implement the different firewalls we are to use through *iptables* and they will always contain at least two interfaces (in contrast with nodes that will usually contain a single one). The differences between these two classes are motivated by these slight discrepancies in terms of functionality. We would like to point out that the syntax that one **must** adhere to when defining firewall rules has been set forth in section 4.1.2.

The lifecycle of routers is practically the same as that of nodes. The most noticeable difference might be how a router will usually assist to more interface additions and deletions throughout its lifespan.

1. The real instance is brought up.
2. The object representing said instance is created.
3. The object sits idle. Several of its parameters can be altered within this state, such as the number of interfaces.
4. At some point, the object will be dismantled.
5. The release of the object will trigger the removal of the associated real instance.

Imported Libraries

1. `os`: This module enables the execution of commands through a *sh* shell through the `os.system()` method. One can check the shell being spawned is indeed *sh* by running `os.system("echo $0")` or by querying [55].
2. `constants`: Grant access to the `terminal_escape_sequences dictionary` containing *ANSI escape sequences* enabling colored output.

3. `sys`: This module allows us to print error messages to *STDERR* instead of *STDOUT* so that they can be easily redirected later on if needed with `2>/path/to/log`.

4. `subprocess`: This module enables the execution of commands through a shell **and** allows the caller to retrieve the command's output to *STDOUT* on top of its return code. This will let us retrieve a container's associated *PID*.

5. `interface`: This module will let us instantiate and add `interface_inst` to the nodes we create.

Global Variables

1. `t_colors - dictionary/string/string`: This is a synonym for the `terminal _escape_sequences dictionary` we me mentioned before. It is used within calls to `print()` so that we can alter the terminal text's color allowing for a more visual information representation.

The d_router Class This class represents a virtual router. Note the `'d'` stands for docker.

Class Attributes

1. `self.type - string`: Object type identifier containing the `''router'' string` for routers. It is used within functions to check the type of object it is currently dealing with.

2. `self.name - string`: Router's name.

3. `self.pid - int`: The associated container's *PID*. It is used when linking the container's network namespace to `/var/run/netns` so that we can easily access it afterwards.

4. `self.interfaces - list/interface_inst`: A list containing the instances of the interfaces belonging to this router.

5. `fw_rules - dictionary/string/list`: Firewall rules configured for this router. Note that even though the "logical" format is exactly the same as the one presented on section 4.1.2, the node names are translated to *IPs* by the caller when this member is initialized. This implies that the "real" entries we are to work with internally are of the form `("source_ip", "destination_ip", bidirectional?)` instead of `("source_node", "destination_node", bidirectional?)`.

The Constructor

1. **Parameters:**

   (a) `self - d_router_inst`: The actual router instance.

   (b) `name - string`: The router's name.

2. **Returns:** Nothing.

3. **Description:** This method will just initialize several members with the passed parameters whilst assigning sane defaults to others. It leverages the power of the `docker inspect` command to find the associated container's *PID*. It will also call `_link_net_namespace()` to allow an easier handling of the container's namespace through the `ip` command. It finally calls `_set_hostname()` to configure the router's identity from the network's perspective.

The _link_net_namespace() Method

1. **Parameters:**

   (a) `self - d_router_inst`: The actual router instance.

2. **Returns:** Nothing.

3. **Description:** This method will create a symbolic link to the container's network namespace under `/var/run/netns` where the `ip` command will "look for" existing named namespaces. This method leverages the `self.pid` attribute that is initialized in the constructor to find the original location of the container's namespace so that it can later be linked.

The _set_hostname() Method

1. **Parameters:**

   (a) `self - d_router_inst`: The actual router instance.

2. **Returns:** Nothing.

3. **Description:** This method will just set the container's hostname. It is essential to distinguish between the container's name and its hostname. The former is an identifier that only concerns docker, whilst the latter will be the machine's identifier from the point of view of the network. This method will make the latter the same as the former. What is more, this operation made us change the underscores (`_`) in the original node names by hyphens (`-`): the former were not allowed on network hostnames.

The get_name() Method

1. **Parameters:**
   (a) `self - d_router_inst:` The actual router instance.
2. **Returns:** A `string` containing the router's name.
3. **Description:** Not applicable.

The get_type() Method

1. **Parameters:**
   (a) `self - d_router_inst:` The actual router instance.
2. **Returns:** The `"router" string`.
3. **Description:** Not applicable.

The get_subnets() Method

1. **Parameters:**
   (a) `self - d_router_inst:` The actual router instance.
2. **Returns:** A `list/string` containing the different subnets this node is connected to in *CIDR* format.
3. **Description:** This method will iterate over the `interface_inst` contained in the `self.interfaces` attribute to then provide a list containing the subnets each of these `interface_inst` belong to.

The get_interface() Method

1. **Parameters:**
   (a) `self - d_router_inst:` The actual router instance.
   (b) `subnet - string - optional:` Specifies the subnet for which an associated `interface_inst` should be returned.
2. **Returns:** The following are evaluated in order.
   (a) A `list/interface_inst` if the `subnet` parameter was not specified.

(b) An `interface_inst` associated to the subnet specified in the `subnet` parameter.

(c) `None` if there is no interface associated with the subnet specified through the `subnet` parameter.

3. **Description:** As seen on the **returns** section, the `subnet` parameter behaves like a switch. A `list` of `interface_inst` will be returned if it is not specified by the caller. In case it is provided, it will return either an `interface_inst` or `None` if there is no interface associated with the specified subnet.

The create_interface() Method

1. **Parameters:**

   (a) `self - d_router_inst:` The actual router instance.

   (b) `if_name - string:` Name of the interface to create.

   (c) `if_subnet - string:` Subnet associated with the interface to create.

   (d) `o_if - boolean:` Flag indicating whether the interface to create should be allowed to "see" other subnets.

2. **Returns:** Nothing.

3. **Description:** This method will instantiate a new interface and add it to the node's `self.interfaces list`. The method's parameters will be passed directly to the `interface` class constructor.

The reset_interfaces() Method

1. **Parameters:**

   (a) `self - d_router_inst:` The actual router instance.

   (b) `subnet - string - optional:` The subnet associated to the interface that should be reset.

2. **Returns:** A `boolean` indicating whether the operation completed successfully or not.

3. **Description:** If the caller does not specify a `subnet` parameter, this method erases all the interfaces associated with the router whilst also removing the associated "real instances" from the host machine. This can easily be accomplished through the `clear()` [57] method defined for `lists`, such as the `self.interfaces` attribute due to the destructor we defined for the `interface` class. If the caller specifies a

subnet, the method will look for the interface belonging to that subnet and `remove()` it from the `self.interfaces` attribute. This will, as before, trigger the deletion of the "real interface". The method will return `True` if either all the interfaces were deleted (i.e. no `subnet` was specified) or if the one specified through the `subnet` parameter was found and deleted as well. If the caller specified an interface belonging to a specific subnet and it was not found the method will return `False`, signaling an error.

The set_fw_rules() Method

1. **Parameters:**
   (a) `self - d_router_inst`: The actual router instance.
   (b) `fw_rules - dictionary/list/string`: Firewall rules to be applied to the router.
2. **Returns:** Nothing.
3. **Description:** This method will update the contents of the `self.fw_rules` attribute that will serve as the cornerstone for the following methods.

The apply_fw_rules() Method

1. **Parameters:**
   (a) `self - d_router_inst`: The actual router instance.
2. **Returns:** Nothing.
3. **Description:** This method will parse (i.e. process) the rules and policy contained in the `self.fw_rules` attribute and then proceed on to make them effective. In order to do so, it relies on the `self.instantiate_fw_rule()` and `set_fw_policy()` methods we describe below. If said attribute is an empty `dictionary` the method will exit promptly and allow the router to forward all the traffic flowing through it, thus effectively disabling the firewall within the associated container. This method will only print a message when there are no rules configured, as the `self.instantiate_fw_rule()` method will print a message for each rule it processes.

The instantiate_fw_rule() Method

1. **Parameters:**
   (a) `self - d_router_inst`: The actual router instance.

(b) `target - string`: The *target* for the rule, either ``ACCEPT'' or ``DROP''.

(c) `source - string`: The source *IP* address for the rule.

(d) `dest - string`: The destination *IP* address for the rule.

(e) `bi_dir - boolean`: `True` if the rule is "bi-directional" (i.e. an inverse rule should be instantiated). `False` otherwise.

2. **Returns:** Nothing.

3. **Description:** This method leverages the `docker exec` command to instantiate the *iptables* rules specified by its parameters on the associated container. The `bi_dir` parameter will cause the instantiation of a symmetric rule (i.e. one that swaps the source and destination *IP* addresses) as seen on section 4.1.2. An informative message will be printed for each rule that is successfully added.

The set_fw_policy() Method

1. **Parameters:**
   (a) `self - d_router_inst`: The actual router instance.
   (b) `policy - string`: The policy to apply to the router.

2. **Returns:** Nothing.

3. **Description:** This method will apply the policy specified through the `policy` parameter to the router's *FORWARDING* chain. *iptables* is capable of handling both upper and lowercase policy specifications. This method relies on the `docker exec` command to configure the container associated to the instance whose method we are invoking. The method will also print a message if the operation succeeds.

The add_n_apply_fw_rule() Method

1. **Parameters:**
   (a) `self - d_router_inst`: The actual router instance.
   (b) `trgt - string`: The *target* for the rule, either ``ACCEPT'' or ``DROP''.
   (c) `src - string`: The source *IP* address for the rule.
   (d) `dst - string`: The destination *IP* address for the rule.
   (e) `b_dir - boolean`: `True` if the rule is "bi-directional" (i.e. an inverse rule should be instantiated). `False` otherwise.

2. **Returns:** Nothing.

3. **Description:** This method has been written to support "dynamic firewalls". As nodes are moved around the network the need for changes in firewall rules arises in order to maintain the exact same logical topology. This method will just add the rule specified through the parameters to the `self.fw_rules` attribute and pass them to the `instantiate_fw_rule()` method to make them effective.

The remove_fw_rule() Method

1. **Parameters:**
   (a) `self - d_router_inst`: The actual router instance.
   (b) `moving_node_ip - string`: The *IP* address whose associated rules we are to delete.
2. **Returns:** Nothing.
3. **Description:** This method has been written to support "dynamic firewalls" as well. On top of adding new rules upon a node's movement we also need to delete stale ones so that firewall configurations do not become cluttered over time. Given we are reusing freed *IP* addresses, leaving unused rules behind could have unintended side effects that would be difficult to debug. This method will iterate over a router's firewall rules through the `self.fw_rules` attribute and delete those whose source or destination *IP* is the one specified by the `moving_node` parameter. The reason behind the definition of this method is what justifies the chosen parameter name. This method relies on the `docker exec` command and it will also print an informative message upon rule deletion so that we can be sure every task was carried out successfully.

The stop() Method

1. **Parameters:**
   (a) `self - d_router_inst`: The actual router instance.
2. **Returns:** A `boolean` indicating whether the operation succeeded or not.
3. **Description:** This method is in charge of stopping the instance's associated docker container. We can do so thanks to the `docker stop` command. This method leverages a redirection to `/dev/null` so that the output generated by `docker stop` does not interfere with our own. This method will print a message both upon success and error.

The remove() Method

1. **Parameters:**

   (a) `self - d_router_inst`: The actual router instance.

2. **Returns:** Nothing.

3. **Description:** This method is in charge of entirely removing the associated container. It will call the `stop()` method as a container must be stopped before it can be removed through the `docker rm` command. First of all, the method will get rid of the interfaces as soon as possible so that, even if something goes wrong when deleting the container, it has no connection to a network that might still be alive. This will in fact isolate the container so that possible errors have no impact on the rest of the virtualized scenario. We will also delete the link to the container's network namespace living under `/var/run/netns`. As usual we will print information relative to the command's outcome.

The Destructor

1. **Parameters:**

   (a) `self - d_router_inst`: The actual router instance.

2. **Returns:** Nothing.

3. **Description:** This method will call the `remove()` which will in turn delete the associated container. Given the definition of the `remove()` method this will later allow us to dismantle the entire network with a single order.

Net.py

This file contains the definition for the class representing the entire network topology. Said class gathers all the existing subnets as well as the routers interconnecting them. It will also instantiate the `address_manager` class to handle of addressing. This class is the only place from which *veths* can be created. This speeds up development when changes are needed. Given it is the base on which the rest of the infrastructure is built upon, the lifecycle of this class' instance is that of the virtual network.

1. The `d_net` class is instantiated.

2. The entire virtual network is brought up.

3. The instance continues to exist as long as the program is running.
4. Once the instance is released, the entire virtual network is dismantled.
5. The program exits.

Naming Veths Consistently *Veths* are the most common element in the network. This implies that we need to somehow name them in such a way that we can easily reference them and guarantee no names will collide. Given the node names **must be** unique as specified in section 4.1.2, and that there will only be a single connection between any two nodes, these *veths* will be named according to the nodes they connect. The restrictions we imposed on naming guarantee these names will effectively be unique. Given a *veth* is composed of two different ends we decided to share the name across the two and take advantage of *iproute2* being case-sensitive. Thus, a *veth* end will be named according to the *veth_¡node_x¿_¡node_y* template where each node name follows the *¡character¿-¡number — character¿* pattern (the `'|'` character is to be read as *OR*). The other end will have the **exact same** name with the node names in upper case instead of lower case. Thus, the ends of a *veth* could be named *veth_c-1_c-b* and *veth_C-1_C-B*, respectively.

Imported Libraries

1. `os`: This module enables the execution of commands through a *sh* shell through the `os.system()` method. One can check the shell being spawned is indeed *sh* by running `os.system("echo $0")` or by querying [55].
2. `constants`: Grant access to the `terminal_escape_sequences dictionary` containing *ANSI escape sequences* enabling colored output.
3. `sys`: This module allows us to print error messages to *STDERR* instead of *STDOUT* so that they can be easily redirected later on if needed with `2>/path/to/log`.
4. `address_manager`: This module provides access to the `addr_manager` class, which provides helper methods to address the network.
5. `subnet`: This module will let us instantiate the `d_subnet` class as needed in order to add subnets to the network.
6. `net_machines`: This module will let us instantiate and add `d_router_inst`s to the network.
7. `veth`: This module provides access to the `connect_veth_end()` function so that we can invoke it when required.

Global Variables

1. `t_colors - dictionary/string/string`: This is a synonym for the `terminal _escape_sequences dictionary` we me mentioned before. It is used within calls to `print()` so that we can alter the terminal text's color allowing for a more visual information representation.

The d_net Class This class represents an entire network. Note the '`d`' stands for docker.

Class Attributes

1. `self.subnets - dictionary/string/d_subnet_inst`: A dictionary containing an instance of each active subnet as the value associated to a key that is the subnet's address in *CIDR* format.
2. `self.routers - dictionary/string/s_router_inst`: A dictionary containing an instance of each active router as the value associated to a key that is the router's name.
3. `self.net_addr_manager - addr_manager_inst`: An object in charge of managing *IP* addresses and addressing the entire network.
4. `self.router_bridge - string`: Initial placeholder name for a an auxiliary bridge that might be needed when moving subnets within the network. This will be clarified below.
5. `self.aux_router - string`: Initial placeholder name for a an auxiliary router that might be needed when moving subnets within the network. This will be clarified below.

The Constructor

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
2. **Returns:** Nothing.
3. **Description:** This method will just initialize the network's parameters with sane defaults and call the `_disable_iptables()` method to prevent the host's kernel from interfering with the virtual network.

The _disable_iptables() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
2. **Returns:** Nothing.
3. **Description:** This method will disable the *bridge-nf-call-iptables* kernel parameter to prevent the host kernel from interfering with the network as explained on section 2.7. It will also make sure the `/var/run/netns` directory exists through the `mkdir -p` command to make sure the host is capable of correctly supporting the virtual network.

The get_addr_manager() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
2. **Returns:** The `addr_manager_inst` referenced by the `seld.addr_manager` attribute.
3. **Description:** Not applicable.

The req_brd_name() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
2. **Returns:** A `string` containing the name to be given to a bridge that needs to be automatically created.
3. **Description:** This method might be needed when moving entire subnets within the virtual network. This method will also update the `self.router_bridge` attribute to gracefully handle a subsequent calls. In order to modify new names we leverage *python's* `ord()` [58] and `chr()` [58] functions. The generated names will follow the `''z-rb'', ''y-rb'', ..., ''a-rb''` progression. The current method definition only allows for 26 different names. This limitation has not manifested yet but we should nonetheless note it exists.

The req_r_name() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
2. **Returns:** A `string` containing the name to be given to a router that needs to be automatically created.
3. **Description:** This method might be needed when moving entire subnets within the virtual network. The method will also update the `self.aux_router` attribute to gracefully handle a subsequent calls. In order to modify new names we leverage *python's* `ord()` [58] and `chr()` [58] functions. The generated names will follow the `''ra-z'', ''ra-y'', ..., ''ra-a''` progression. The current method definition only allows for 26 different names. This limitation has not manifested yet but we should nonetheless note it exists.

The create_subnet() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
   (b) `subn - string`: The subnet address for the subnet to create, specified in *CIDR* format.
2. **Returns:** The created `d_subnet_inst`.
3. **Description:** This method is in charge of instantiating each and every subnet composing the network. If the subnet address is unique and the `subn` parameter is not `None` the method will instantiate the `d_subnet` class and add the instance to the `self.subnets` attribute. The instance referenced by the `self.addr_manager` method will also be informed of the new subnet so that it can assign addresses belonging to it when needed. The fact the the method returns the created `d_subnet_inst` makes it compatible with almost any third-party overlay a user might want to add on top of our code. Instead of imposing restrictions on how the network elements should be created we tried to let the user decide that for him or herself.

The create_router() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
   (b) `name - string`: The router's name.

2. **Returns:** The created `d_router_inst`.

3. **Description:** This method is the only way of adding new routers to the network. Like in the previous method, the generated router instance will be returned as well as stored on the `self.routers` attribute. The method will also print messages informing of whether it managed to carry out its operation successfully or not.

The connect_nodes() Method

1. **Parameters:**

   (a) `self - d_net_inst`: The actual network instance.

   (b) `node_x - k_bridge_inst | d_node_inst | d_router_inst`: One network-aware machine we are to connect a *veth* end to. Note the '`|`' character is to be read as *OR*.

   (c) `node_y - k_bridge_inst | d_node_inst | d_router_inst`: The other network-aware machine we are to connect a *veth* end to. Note the '`|`' character is to be read as *OR*.

   (d) `cnx_subnet - string`: The subnet on which the network aware machines are being connected.

2. **Returns:** Nothing.

3. **Description:** This method handles the creation and connection of *veths* whenever they are needed. It will generate the names for both *veth* ends following the rules described at the beginning of the section, and call the `veth.connect_veth_end()` function to connect those ends the appropriate destinations. This method will print messages informing on the outcome.

The add_router_to_subnet() Method

1. **Parameters:**

   (a) `self - d_net_inst`: The actual network instance.

   (b) `router - string`: The name of the router to add to a given subnet.

   (c) `subnet - string`: The subnet address identifying the subnet we are to add the router to in *CIDR* format.

2. **Returns:** Nothing.

3. **Description:** This method will allow us to add a router to any subnets we connect it to. This method will let us maintain an accurate picture of the network state at all times.

The addressing_time() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
2. **Returns:** Nothing.
3. **Description:** This method triggers the addressing process for the entire network through the `address_net` method defined in the `address_manager` class.

The Destructor

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
2. **Returns:** Nothing.
3. **Description:** The destructor will undo the changes caused by the `_disable_iptables()` method and empty the `self.subnets` and `self.routers` attributes. This will trigger the destructors for all the instances contained in them, thus effectively dismantling the entire network. Even though the explicit deletion of the aforementioned attributes is not mandatory, carrying it out explicitly provides an easier understanding of the inner workings of the code. Being able to dismantle the network with just a few lines is the consequence of carefully crafting the destructors for each of our classes as we have explained in the previous sections.

Address_manager.py

If one were to summarize the steps our task requires to the greatest extent he or she would find that there are only three of them. We first need to bring all the necessary network components up, then address them and finally add the routes enabling communication between them. All the ground we have covered up to now has been devoted to dealing with the first of the three. This section tackles the second aspect, and the last one will be related to a later section.

The class defined in this file will deal with **everything** related to *IP* addressing within our network. It will keep track of the assigned and free addresses, reclaim those that are freed and manipulate them as needed in order to ensure an orderly and functional addressing of the network. In order to facilitate these actions we have decided to work with a text-based and integer-based representation of the *IPv4* addresses we have to deal with.

Representing IP Addresses As seen on the enumeration on section 4.1.2, *IP* (note we are always referring to *IPv4* addresses) addresses follow the *A.B.C.D/X* pattern. These addresses are traditionally represented as `strings` within programs, but this data type quickly becomes cumbersome once we need to modify these addresses. We will later see how our address manager will hand out monotonically increasing *IPs*. This implies we must somehow modify a given address and craft a new one. Even though this would be feasible and, under certain restrictions, not terribly hard to implement with `strings`, this approach lacks the flexibility offered by an integer-based representation. *IPv4* addresses are "just 32-bit numbers", just like `integers` [1]: after all, that is what is included in the header of network layer datagrams. The text-based representation became widespread because us, as humans, prefer to deal with it rather than with a 32-digit long chain of 1s and 0s. It has even made its way into the interfaces network operators use to configure network equipment: remember *iproute2* accepts *IP* addresses with the format we have set forth at the beginning of the paragraph. Nonetheless, the former approach is tremendously easy to modify: we can literally add 2 to a given address and obtain a new address. We can also subtract one address from other to find out how many lie between them. This is what motivated us to create a set of methods capable of converting between both representations. We can use the text-based representation to "communicate" with *iproute2* and print informative messages whilst we can leverage the integer-based one to easily modify and alter the addresses as desired.

Given how our network topologies are not extremely complex, we decided to associate a */24* (i.e. *255.255.255.0*) subnet mask to every subnet. This restriction would make it feasible to skip the integer-based representation and just rely on a textual one. We nonetheless decided to implement the latter in an effort to make our code as extensible as possible so that it can also be effective under more "extreme" circumstances.

Unlike previous classes, the one defined in this file has no "real" lifecycle. As soon as the `d_net` constructor is called this class will be instantiated and it will continue to exist until the program is taken down. We would like to point out how only one instance of this class can be brought up. We have not written the logic capable of coordinating two or more address managers, so if a user decides to work in such a way it is their own job to accommodate all the issues that might arise.

Imported Libraries None.

[1] In *python* we can deal with 64-int integers as shown by executing `math.log(sys.maxsize + 1, 2)` on a *python* interpreter, which returns 63.0. This line shows that an `unsigned integer's` maximum value is $2^{63} - 1$. [59]

Global Variables None. We decided this module should not print any information as that task was already being carried out from other methods the class defined in this file relies on.

The addr_manager Class This class is in charge of all the addressing for a given network.

Class Attributes

1. `self.next_subn_addr - dictionary/string/int:` This `dictionary` is keyed by a subnet address in *CIDR* format. The value is the next free *IP* address within that subnet.

2. `self.freed_subn_addr - dictionary/string/(list/string):` This `dictionary` is also keyed by a subnet address in *CIDR* format. The value will now be a `list` of the free *IPs* belonging to the subnet specified in the key. These are stored as `strings` instead of as `ints`.

3. `self.router_subn - string:` This is the subnet address to be assigned to an automatically created subnet in the process of moving other subnets or nodes around. This address will be modified in place as needed.

The Constructor

1. **Parameters:**
   (a) `self - addr_manager:` The actual address manager instance.
2. **Returns:** Nothing.
3. **Description:** The constructor will just initialize the instance's attributes.

The add_subn() Method

1. **Parameters:**
   (a) `self - addr_manager:` The actual address manager instance.
   (b) `subnet - string:` The subnet address that the address manager needs to start keeping track of.
2. **Returns:** Nothing.
3. **Description:** The method will first check the subnet the caller is trying to add is indeed new and, if that is the case, it will find the first assignable address for that subnet. A new entry for that subnet will also be added to the `self.next_subn_addr` attribute.

The revoke_ip() Method

1. **Parameters:**

   (a) `self - addr_manager`: The actual address manager instance.

   (b) `node - d_node_inst | d_router_inst`: The network-aware machine whose *IP* for a given subnet we are to revoke.

   (c) `subnet - string`: The subnet to which the *IP* we are to revoke belongs to.

2. **Returns:** Nothing.

3. **Description:** This method will revoke the *IP* assigned to a *layer-3* device belonging to the `subnet` subnet. This method is the only one populating the `self.freed_subn_addr` attribute. As stated before, these addresses are `strings`: we are certain they are free, so they can be assigned "as-is", without any need of modifying them.

The req_r_subnet() Method

1. **Parameters:**

   (a) `self - addr_manager`: The actual address manager instance.

2. **Returns:** A `string` containing the subnet address for an automatically generated subnet, in *CIDR* format.

3. **Description:** This method will return the current `self.router_subn` attribute and modify the subnet address for subsequent calls. The new subnet address will be the */24* subnet "below" the one returned. If the returned subnet address were *10.0.254.0/24* the next would become *10.0.253.0/24*, for instance.

The _request_ip() Method

1. **Parameters:**

   (a) `self - addr_manager`: The actual address manager instance.

   (b) `subnet - string`: The subnet for which we are requesting an *IP* address.

2. **Returns:** A `string` containing the next free *IP* address for the requested subnet together with the subnet mask (i.e. */24*).

3. **Description:** This method will also update the instance's attributes so that the assigned *IP* address is no longer considered free.

The _addr_to_binary() Method

1. **Parameters:**
   (a) `self - addr_manager`: The actual address manager instance.
   (b) `addr - string`: The *IP* address whose representation we want to alter together with its associated subnet mask.
2. **Returns:** An `int` containing the equivalent *IP* address to the one specified in the `addr` parameter.
3. **Description:** This method makes constant use of *bitwise operators* such as `OR` (`'|'`) and `shifts ('<<', '>>')`. Given an *IP* address such as `A.B.C.D`, its binary representation can be written as `A << 24 | B << 16 | C << 8 | D`.

The _binary_to_addr() Method

1. **Parameters:**
   (a) `self - addr_manager`: The actual address manager instance.
   (b) `bin - int`: The *IP* address whose representation we want to alter.
2. **Returns:** A `string` containing the equivalent *IP* address to the one specified in the `bin` parameter.
3. **Description:** This method makes constant use of *bitwise operators* such as `AND` (`'&'`) and `shifts ('<<', '>>')` as well. Given an *IP* as an `int` (i.e. the `bin` parameter) and assuming a `/24` mask we can write the equivalent textual address as `"...".format(bin >> 24 & 0xFF, bin >> 16 & 0xFF, bin >> 8 & 0xFF, bin & 0xFF)`.

The _get_net_addr() Method

1. **Parameters:**
   (a) `self - addr_manager`: The actual address manager instance.
   (b) `subn - string`: The subnet whose network address we want to unravel in *CIDR* format.
2. **Returns:** An `int` containing the subnet's network address.
3. **Description:** This method makes constant use of *bitwise operators* as well. It will build the subnet mask as a chain of 1s and 0s from the one specified through the `subn` parameter (i.e. */24*). It will then apply it to said parameter with an `AND` (`'&'`) operation to obtain the desired network address.

The _get_brd_addr() Method

1. **Parameters:**
   (a) `self - addr_manager`: The actual address manager instance.
   (b) `subn - string`: The subnet whose broadcast address we want to unravel in *CIDR* format.
2. **Returns:** An `int` containing the subnet's broadcast address.
3. **Description:** This method makes constant use of *bitwise operators* as well. It will build the **exact same** mask as the `_get_net_addr()` method and then invert it with the `NOT (' ')` operator. It will finally combine it with the passed parameter with an `OR ('|')` operation so as to obtain the desired result.

The address_net() Method

1. **Parameters:**
   (a) `self - addr_manager`: The actual address manager instance.
   (b) `routers - dictionary/string/d_router_inst`: The routers that require addressing.
   (c) `routers - dictionary/string/d_subnet_inst`: The subnets that require addressing.
2. **Returns:** Nothing.
3. **Description:** This method will address the entire network. It will iterate over the provided `dictionaries` and assign addresses based on the subnets the interfaces belong to. We are addressing the routers before the subnets (and therefor the hosts) so that they are assigned the lowest addresses within the subnets they belong to. This is however not a limitation: we decided to proceed in this way to make finding errors in the addressing process easier for us. This method is also capable of catching and handling exceptions that might arise when trying to request addresses from a subnet the address manager has no notice of. This greatly simplifies troubleshooting mistakes and errors.

The reset_subnet() Method

1. **Parameters:**
   (a) `self - addr_manager`: The actual address manager instance.

(b) `subn - string`: The subnet whose associated addresses we want to reset.

2. **Returns:** Nothing.
3. **Description:** The need for this method arose when we implemented functionality allowing nodes and subnets to move around the network. The method will just overwrite the lowest free address for a given subnet with the subnet's network address plus one, effectively resetting addressing.

The assign_new_ip() Method

1. **Parameters:**
   (a) `self - addr_manager`: The actual address manager instance.
   (b) `node - d_node_inst`: The node we will assign an *IP* address to.
   (c) `subnet - string`: The subnet for which we want to assign an *IP* address.
2. **Returns:** Nothing.
3. **Description:** In order to reuse revoked *IPs* we need to assign freed addresses before moving on and assigning the first free address. When assigning a new *IP* to moved nodes, the method will check whether the `list` of freed *IPs* for the destination subnet (specified through the `subnet` parameter) is empty or not. If it is not, it will assign the first freed *IP* by `pop()`ping [57] it from the appropriate `list`. It will otherwise request a new address.

C.2 The graph_interpreter Module

Graph_to_virt_net.py

This file contains the source code defining the class in charge of interpreting a graph generated through the *networkx* module and invoking the necessary methods from the *virt_net* module to instantiate it. This class also contains the definitions for the methods implementing features such as the movement of nodes in an operational virtual network.

An instance of the `graph_interpreter` class will exist as long as the virtual network is online. It is needed before any virtual network elements are instantiated and it will be taken down together with the virtual network.

Imported Libraries

1. `os`: This module enables the execution of commands through a *sh* shell through the `os.system()` method. One can check the shell being spawned is indeed *sh* by running `os.system("echo $0")` or by querying [55].

2. `networkx`: This module provides several graph-related functionalities such as routing algorithms which are required by some of the methods.

3. `copy`: This modules allows us to create independent copies of "deep" `dictionaries`. This provides independent instances with the same contents instead of a shallow copy whose values are references to the **same** variables. Given *python* does not expose the *C* `pointer` type this idea can be regarded as awkward, but in *C's* terms we can state that this module will allow us to dereference all the `pointers` within the structures we are to copy so that the resulting variable contains references to a totally different memory region.

4. `net`: This module provides access to the `d_net` class, which acts as a gateway for accessing the facilities offered by the *virt_net* module.

Global Variables None.

The graph_interpreter Class

Class Attributes

1. `self.node_to_instance - dictionary/string/dictionary/string/X_inst`: This `dictionary` contains a `dictionary` for each and every type of network-aware machine. Each sub-`dictionary` contains all the instances of said type. Thus, one can access **every** instance through this `dictionary`. We decided to design the class in this way instead of accessing instances through the `d_net` instance to shorten several of the class' functions. Given this attribute was being referenced many times throughout the class we believe this decision improved the code's readability.

2. `self.net_instance - d_net_inst`: This attribute contains the network instance we are working with. This is the entry point for all the methods offered by the *virt_net* module, so we will make use of this attribute over and over again.

3. `self.connectivity_view - networkx_graph_inst`: This graph captures the logical connections that can be established according to the rules defined in our firewalls. In the absence of firewalls this graph would be **equivalent** to the one we are interpreting to bring the network up. The existence of this graph made reconfiguring firewalls dynamically much easier: this logical topology **will not** change even if the physical one does, so it serves as a reference that must always be respected.

4. `self.addr_manager - addr_manager_inst`: A reference to **the same** `addr_manager_inst` that is an attribute of the `d_net` class. We are just referencing it though a class attribute to shorten several code lines so as to increase the code's readability.

The Constructor

1. **Parameters:**
   (a) `self - graph_interpreter_inst`: The actual graph interpreter instance.
2. **Returns:** Nothing.
3. **Description:** This method will check wether the user running the program is *root* through the `os.geteuid()` statement. If the user running the program is indeed *root*, execution will continua and several attributes including `self.net_instance` will be initialized. Otherwise the program will exit after informing the user the program is to be run by *root*.

The get_subnets() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
2. **Returns:** A `list/strings` of the currently active subnets given as the subnets' network address together with the subnet mask in *CIDR* notation.
3. **Description:** Not applicable.

The get_nodes() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
2. **Returns:** A `list/strings` containing the names of the currently active nodes.
3. **Description:** Not applicable.

The get_routers() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
2. **Returns:** A `list/strings` containing the names of the currently active routers.
3. **Description:** Not applicable.

The get_n_n_r() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
2. **Returns:** A `list/strings` containing the names of the currently active nodes and routers.
3. **Description:** This method was written to avoid having to call both `get_nodes()` and `get_routers()` back-to-back in several methods in an effort, once more, to shorten the code lines and improve the code's readability to the greatest extent.

The get_cnx_graph() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
2. **Returns:** A `networkx_graph_inst` representing the current logical topology.
3. **Description:** This method is just returning the `self.connectivity_view` attribute "behind the scenes".

The create_n_connect_node() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
   (b) `node - string`: Name of the node we are to create and connect or just connect to a bridge.
   (c) `bridge - string`: Name of the bridge we are to connect the node to.
2. **Returns:** Nothing.
3. **Description:** This method is called whenever we want to connect a node to a bridge. The method will check whether the node exists already: if it does, it will just be connected and if it does not, it will be created beforehand. All these nodes will be added to the `self.connectivity_view` attribute as well: this ensures no bridges will be present on that graph as they are transparent to us at the "logical connection" level. Due to how the graph describing the physical topology is translated and executed, all the bridges are guaranteed to exist before this method is run. This subtlety frees this method from having to instantiate bridges as well.

The create_n_connect_router() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
   (b) `router - string`: Name of the router we are to create and connect or just connect to a bridge.
   (c) `bridge - string`: Name of the bridge we are to connect the router to.
2. **Returns:** Nothing.
3. **Description:** This method is entirely analogous to the `create_n_connect_node()` method. It will just instantiate and connect routers instead of nodes.

The found_bridge() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
   (b) `b_name - string`: Name of the bridge that will be created, if appropriate.
   (c) `subnet - string`: subnet address in *CIDR* format specifying the subnet supported by said bridge.
2. **Returns:** Nothing.
3. **Description:** This method will check whether the bridge specified by the `b_name` parameter exists or not. If it does, it will just exit. If if does not, it will create both the bridge and the associated subnet.

The _instantiate_routes() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
   (b) `found_routes - dictionary/string/dictionary/string/list/string`: A `dictionary` containing the shortest path from **node** to each subnet. This dictionary is generated by means of the `networkx. shortest_path()` function within the `translate_n_execute_graph()` method we will describe below.
2. **Returns:** Nothing.
3. **Description:** This method will parse the `found_routes` parameter and instantiate the appropriate routes at every node and router. The anatomy of the `found_routes` parameter is described on listing C.1.

```
{
    'dest_subnet': {
        'source_node_a': [
            'source_node_a',
            'gateway_x',
            'dest_subnet_bridge'
        ],
        'source_node_b': [
            'source_node_b',
            'gateway_y',
            'dest_subnet_bridge'
        ],
        # There is one such entry for each node in the graph (routers,
            # bridges and hosts). Each of these entries has possibly
            # more hops, but we are only interested in the first one
            # as that is the gateway we need to configure.
    }
    # There is one such entry for each subnet.
}
```

Listing C.1: Data Structure Containing All the Network's Routes.

The translate_n_execute_graph() Method

1. **Parameters:**

   (a) `self - d_net_inst:` The actual network instance.

   (b) `graph - networkx_graph_inst:` The graph representing the virtual topology we are to instantiate.

   (c) `fw_enable - boolean - optional:` A flag indicating whether the method should activate the firewall configuration specified through attributes of the nodes representing routers. This parameter proved useful when we were debugging the *ICS* topology seen on section 4.3.

2. **Returns:** An `int` indicating whether the virtual network could be instantiated (0) or if there was some kind of error (−1).

3. **Description:** This method relies on most of the above to correctly parse the graph provided as a parameter and end up with a completely functional virtual network. We can regard it as a "dispatcher" method in the sense that it coordinates many others to achieve a complex goal.

The move_subnet() Method

1. **Parameters:**

   (a) `self - d_net_inst`: The actual network instance.
   (b) `graph - networkx_graph_inst`: The graph representing the virtual topology that is currently instantiated.
   (c) `og_subnet - string`: The subnet we are to move, specified in *CIDR* format.
   (d) `dest_subnet - string`: The subnet the one we are moving should be attached to, specified in *CIDR* format.

2. **Returns:** Nothing.

3. **Description:** This method will connect the subnet specified through the `og_subnet` parameter to the one identified by the `dest_subnet` parameter. Doing so usually incurs in many subtleties that need to be managed so as not to disrupt the rest of the network. This translates into this method being noticeably longer than the other we have described up to now.

The move_node() Method

1. **Parameters:**
   (a) `self - d_net_inst`: The actual network instance.
   (b) `graph - networkx_graph_inst`: The graph representing the virtual topology that is currently instantiated.
   (c) `node - string`: The name of the node to displace.
   (d) `dest_subnet - string`: The subnet the node we are moving should be attached to, specified in *CIDR* format.
   (e) `pure_movement - boolean - optional`: This flag controls whether the changes triggered by the movement should be made effective in the internal network representation of the *virt_net* module. This parameter has been previously used for debugging this method and will often be `True`.

2. **Returns:** Nothing.

3. **Description:** This method will move the specified node to the provided subnet whilst handling all the intricacies that might arise when doing so. A prime example of these would be reconfiguring firewalls "on the fly" if the affected node had any restrictions in terms of the connections it was allowed to establish. Like the method before, this one is rather large.

C.3 The net_ctrl Module

Net_ctrl.py

This file defines the `launch_net()` function that will offer the facilities we described above. Just like with other *CLIs*, the user will be granted a set of commands that let him or her interact with the virtual network. Thee have already been described in section 4.1.2.

Imported Libraries

1. `networkx`: This module provides several graph-related functionalities such as offering graphical representations of graphs.
2. `matplotlib`: The *networkx* module relies on *matpolotlib* to generate graphical representations of graphs.
3. `datetime`: This module allows using timestamps on the abscissa axis of plots generated with *matplotlib*.

Global Variables None.

The launch_net() Function

1. **Parameters:**
   (a) `graph - networkx_graph_inst`: The graph representing the network topology to build.
   (b) `fw_on - boolean - optional`: A flag controlling whether the firewalls on routers should be activated (`True`) or not (`False`). It is `True` by default.
   (c) `report_mode - boolean - optional`: A flag controlling whether the function should be run in *report mode* (`False`) or *report mode* (`True`). It is `False` by default.
2. **Returns:** Nothing.
3. **Description:** After correctly bringing the virtual network represented by the *graph* parameter up, this function will enter an infinite loop serving as the *CLI's* main loop. When in the infinite loop, the function will wait for user input and parse it. It will then try to execute the invoked command, printing a message in case of error. This loop can be exited through commands such as `exit` or `quit`. This will also cause the dismantling of the entire virtual network. In other words, this

is **the function** keeping the network alive, as it is the one that will instantiate the `graph_interpreter` class which will in turn bring up all the other network elements. As always, this function can be modified and rewritten to suit other user's needs. As long as it instantiates the `graph_interpreter` class everything will "look the same" to auxiliary modules such as *graph_interpreter* and *virt_net*.

Bibliography

[1] *Docker container software*, https://www.docker.com, Accessed: 14/06/2021.

[2] M. Litvak, *ip - show / manipulate routing, network devices, interfaces and tunnels*, https://man7.org/linux/man-pages/man8/ip.8.html, The iproute2 Project, http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2, 2011.

[3] M. Kerrisk, *namespaces - overview of Linux namespaces*, 5.11, https://man7.org/linux/man-pages/man7/namespaces.7.html, The Linux man-pages Project, https://www.kernel.org/doc/man-pages/, 2021.

[4] *Python programming language*, https://www.python.org, Accessed: 15/06/2021.

[5] *AnyLogic simulation software*, https://www.anylogic.com, Accessed: 14/06/2021.

[6] *VirtualBox virtualisation software*, https://www.virtualbox.org, Accessed: 14/06/2021.

[7] *VMWare virtualisation software*, https://www.vmware.com, Accessed: 14/06/2021.

[8] *Kubernetes orchestration software*, https://kubernetes.io, Accessed: 14/06/2021.

[9] *Image portraying a vm-oriented architecture*, https://www.docker.com/sites/default/files/d8/2018-11/container-vm-whatcontainer_2.png, Accessed: 14/06/2021.

[10] *Docker's documentation portal*, https://www.docker.com/resources/what-container, Accessed: 14/06/2021.

[11] *The bocker project*, https://github.com/p8952/bocker, Accessed: 14/06/2021.

[12] *Image portraying a container-oriented architecture*, https://www.docker.com/sites/default/files/d8/2018-11/docker-containerized-appliction-blue-border_2.png, Accessed: 14/06/2021.

[13] *Kubernetes documentation portal*, https://kubernetes.io/docs/home/, Accessed: 14/06/2021.

[14] *Image portraying the structure of the Linux kernel*, https://upload.wikimedia.org/wikipedia/commons/5/5b/Linux_kernel_map.png, Accessed: 14/06/2021.

[15] *WireShark Ethernet documentation*, https://gitlab.com/wireshark/wireshark/-/wikis/Ethernet, Accessed: 14/06/2021.

[16] M. Kerrisk, *network_namespaces - overview of Linux network namespaces*, 5.11, https://man7.org/linux/man-pages/man7/network_namespaces.7.html, The Linux man-pages Project, https://www.kernel.org/doc/man-pages/, 2019.

[17] *net-tools deprecation proposal*, https://lists.debian.org/debian-devel/2009/03/msg00780.html, Accessed: 14/06/2021.

[18] *Comparison between net-tools and iproute2*, https://dougvitale.wordpress.com/2011/12/21/deprecated-linux-networking-commands-and-their-replacements/, Accessed: 15/06/2021.

[19] *Linux kernel bridges*, https://wiki.linuxfoundation.org/networking/bridge, Accessed: 15/06/2021.

[20] M. Kerrisk, *capabilities - overview of Linux capabilities*, 5.11, https://man7.org/linux/man-pages/man7/capabilities.7.html, The Linux man-pages Project, https://www.kernel.org/doc/man-pages/, 2021.

[21] M. Kerrisk, *veth - Virtual Ethernet Device*, 5.11, https://man7.org/linux/man-pages/man4/veth.4.html, The Linux man-pages Project, https://www.kernel.org/doc/man-pages/, 2021.

[22] E. W. Biederman and N. Dichtel, *ip-netns - process network namespace management*, https://www.man7.org/linux/man-pages/man8/ip-netns.8.html, The iproute2 Project, http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2, 2013.

[23] M. Kerrisk, *proc - process information pseudo-filesystem*, 5.11, https://man7.org/linux/man-pages/man5/proc.5.html, The Linux man-pages Project, https://www.kernel.org/doc/man-pages/, 2021.

[24] H. Eychenne, *iptables/ip6tables — administration tool for IPv4/IPv6 packet filtering and NAT*, 1.8.4, https://man7.org/linux/man-pages/man8/iptables.8.html, iptables Project, http://www.netfilter.org/, 2021.

[25] *WireShark protocol analyser*, https://www.wireshark.org, Accessed: 15/06/2021.

[26] *Thread discussing the interaction between kernel bridges and iptables*, http://patchwork.ozlabs.org/project/netdev/patch/1246379267.3749.42.camel@blaa/, Accessed: 15/06/2021.

[27] *Overview of the Docker Engine*, https://www.docker.com/products/container-runtime, Accessed: 15/06/2021.

[28] *Docker Hub image repository*, https://hub.docker.com, Accessed: 15/06/2021.

[29] *Depiction of a container's lifecycle*, https://miro.medium.com/max/1129/1*vca4e-SjpzSL5H401p4LCg.png, Accessed: 15/06/2021.

[30] R. Moskowitz, D. Karrenberg, Y. Rekhter, E. Lear, and G. J. de Groot, *Address Allocation for Private Internets*, RFC 1918, 1996. DOI: 10.17487/RFC1918. [Online]. Available: \url{https://rfc-editor.org/rfc/rfc1918.txt}.

[31] *The NetworkX python package*, https://networkx.org, Accessed: 15/06/2021.

[32] *pickle — Python object serialization*, https://docs.python.org/3/library/pickle.html, Accessed: 15/06/2021.

[33] *Bipartite Graphs*, https://en.wikipedia.org/wiki/Bipartite_graph, Accessed: 15/06/2021.

[34] *The Python import system*, https://docs.python.org/3/reference/import.html, Accessed: 15/06/2021.

[35] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms," in, Second. Cambridge, MA: MIT Press, 2001, ch. 24.3, ISBN: 0262032937 9780262032933 0070131511 9780070131514 0262531968 9780262531962.

[36] T. Li, C. Feng, and C. Hankin, "Improving ICS cyber resilience through optimal diversification of network resources," *CoRR*, vol. abs/1811.00142, 2018. arXiv: 1811.00142. [Online]. Available: http://arxiv.org/abs/1811.00142.

[37] *sshpass source code*, https://sourceforge.net/projects/sshpass/files/sshpass/1.08/, Accessed: 15/06/2021.

[38] I. Marsa-Maestre, J. M. Gimenez-Guzman, D. Orden, E. de la Hoz, and M. Klein, "React: Reactive resilience for critical infrastructures using graph-coloring techniques," *Journal of Network and Computer Applications*, vol. 145, p. 102 402, 2019, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2019.07.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804519302279.

[39] J. Moy, *OSPF Version 2*, RFC 2328, 1998. DOI: 10.17487/RFC2328. [Online]. Available: https://rfc-editor.org/rfc/rfc2328.txt.

[40] V. Jacobson, C. Leres, and S. McCanne, *tcpdump - dump traffic on a network*, https://man7.org/linux/man-pages/man8/tcpdump.8.html, The tcpdump Project, http://www.tcpdump.org/, 2020.

[41] *The Python Package Index*, https://pypi.org, Accessed: 16/06/2021.

[42] *Docker installation documentation*, https://docs.docker.com/engine/install/ubuntu/, Accessed: 16/06/2021.

[43] M. Kerrisk, *execve - execute program*, 5.11, https://www.man7.org/linux/man-pages/man2/execve.2.html, The Linux man-pages Project, https://www.kernel.org/doc/man-pages/, 2021.

[44] B. Fox and C. Ramey, *bash - GNU Bourne-Again SHell*, https://man7.org/linux/man-pages/man1/bash.1.html, The bash Project, http://www.gnu.org/software/bash/, 2020.

[45] A. G. Morgan, *setcap - set file capabilities*, https://man7.org/linux/man-pages/man8/setcap.8.html, The libcap Project, https://git.kernel.org/pub/scm/libs/libcap/libcap.git/, 2020.

[46] A. G. Morgan, *getcap - examine file capabilities*, https://man7.org/linux/man-pages/man8/getcap.8.html, The libcap Project, https://git.kernel.org/pub/scm/libs/libcap/libcap.git/, 2020.

[47] A. G. Morgan, *capsh - capability shell wrapper*, https://man7.org/linux/man-pages/man1/capsh.1.html, The libcap Project, https://git.kernel.org/pub/scm/libs/libcap/libcap.git/, 2020.

[48] *iproute2's source code*, https://github.com/shemminger/iproute2, Accessed: 17/06/2021.

[49] A. G. Morgan and T. Kukuk, *PAM, pam - Pluggable Authentication Modules for Linux*, https://man7.org/linux/man-pages/man8/PAM.8.html, The linux-pam Project, http://www.linux-pam.org/, 2016.

[50] A. G. Morgan, *pam_cap - PAM module to set inheritable capabilities*, http://manpages.ubuntu.com/manpages/xenial/man8/pam_cap.8.html, The linux-pam Project, http://www.linux-pam.org/.

[51] C. B. et al., *usermod - modify a user account*, 4.8.1, https://www.man7.org/linux/man-pages/man8/usermod.8.html, The shadow-utils Project, https://github.com/shadow-maint/shadow, 2021.

[52] L. Torvalds and J. C. Hamano, *git - the stupid content tracker*, 2.31.1.163.ga65ce7, https://man7.org/linux/man-pages/man1/git.1.html, The git Project, https://github.com/shadow-maint/shadow, 2021.

[53] S. Chacon and B. Straub, *Pro Git*. Apress, 2021, https://git-scm.com/book/en/v2. Accessed: 18/06/2021.

[54] *Introductory git talk*, https://github.com/pcolladosoto/wacky_stuff/blob/master/internship_docs/Git_talk.pdf, Accessed: 17/06/2021.

[55] M. Kerrisk, *system - execute a shell command*, 5.11, https://man7.org/linux/man-pages/man3/system.3.html, The Linux man-pages Project, https://www.kernel.org/doc/man-pages/, 2021.

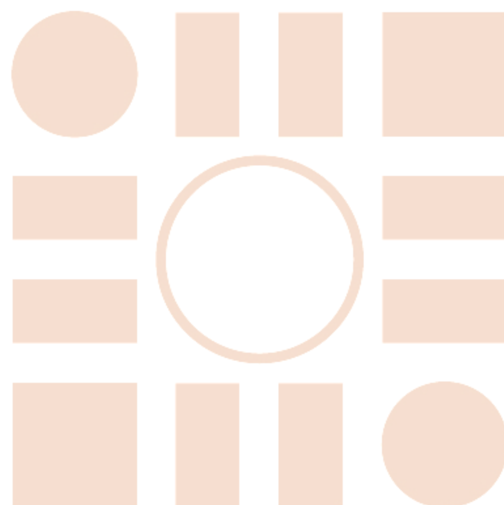
[56] *Classless Inter-Domain Routing (CIDR)*, https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing, Accessed: 15/06/2021.

[57] *Python datastructures tutorial*, https://docs.python.org/3/tutorial/datastructures.html, Accessed: 15/06/2021.

[58] *Python built-in functions*, https://docs.python.org/3/library/functions.html, Accessed: 15/06/2021.

[59] *sys — System-specific parameters and functions*, https://docs.python.org/3/library/sys.html#sys.maxsize, Accessed: 15/06/2021.

Universidad de Alcalá
Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá