ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

Corso di Laurea in Informatica

ImagiNet: a network simulator based on VDE

Supervisor:
Chiar.mo Prof.
Renzo Davoli

Presented by:
Samuele Musiani

I Sessione
Anno accademico 2024/2025

We can lift ourselves out of ignorance, we can find ourselves as creatures of excellence and intelligence and skill. We can be free! We can learn to fly!

Richard Bach, Jonathan Livingston Seagull

Sommario

I software di simulazione di rete permettono agli utenti di condurre esperimenti su topologie di rete senza dover dipendere da hardware fisico e costoso. Questi strumenti consentono agli studenti di esercitarsi e comprendere meglio le reti di calcolatori, consentendo al contempo agli insegnanti di creare laboratori complementari per la valutazione degli studenti durante il corso. In questa tesi presentiamo ImagiNet, un simulatore di rete che si distingue dagli altri per la sua semplicità e le sue funzionalità. È progettato per essere moderno e leggero, consentendo la simulazione di scenari complessi su hardware di uso comune. Il programma è specificamente progettato per assistere gli studenti durante il loro percorso di studi e gli insegnanti durante i loro corsi universitari. ImagiNet ha già dimostrato il suo successo con un piccolo gruppo di studenti, sebbene nulla ne impedisca il suo utilizzo in classi universitarie di grandi dimensioni.

Abstract

Network simulation software allows users to carry out experiments on network topologies without depending on physical and expensive hardware. These tools enable students to practice and better understand computer networks, while enabling teachers to create complementary laboratories for evaluating students during a course. In this thesis, we introduce ImagiNet, a network simulator that distinguishes itself from others through its simplicity and capabilities. It is designed to be modern and lightweight, enabling the simulation of complex scenarios on commodity hardware. The system is specifically designed to assist students during their learning journey and teachers during their university courses. ImagiNet has already proven successful with a small group of students, though nothing prevents it from being used in large university-scale classes.

Table of contents

Introduction

Computer networks are a fundamental part of everyone's life as almost all devices used daily, from smartphones to computers, are connected through the Internet. To most people, the Internet appears as a vast and nearly mystical entity that just works, or just doesn't if the WiFi signal is not strong enough. But for computer science students, who have spent their nights studying textbooks and memorising protocols, the Internet is just an enormous number of computers that have agreed upon a set of rules to be able to communicate with one another. What once seemed mysterious now inspires admiration: the magic is still there, but is no longer steamed by ignorance but by charm and admiration. There is something fascinating about the fact that the entire world has agreed to a set of rules dictated by the need to exchange information as quickly as possible.

Internet and computer networks are playing and will play a fundamental role in the future, and it's essential that they keep working as long as needed. System administrators are the unseen guardians that everyone depends on, though most remain unaware of their presence. They are the little-known and invisible heroes who keep the Internet floating.

As with everything, a person needs to understand the basics before diving head-first into the myriad of protocols and rules that a network administrator needs to know. A great number of textbooks on these topics exist [1], [2], and readers who want to understand this thesis must have a good background knowledge of them. But knowledge without practice is the same as describing the elements of a food without actually testing it. To address this gap, theory and practical experience often have to be carried out simultaneously. Students who are learning how to program will only be able to truly understand it by writing code themselves and actually running it on a computer. The same is valid for computer network students: learning and memorising all the protocols is somewhat important, but observing the protocols in action will bring value and a deeper understanding of every actor involved.

In the past, when computers were not as powerful as they are today, the only way to practice with a network scenario was to have or rent physical hardware and experiment with it. This was impractical and often expensive, resulting in few people having the opportunity to apply what they were learning. Today, computers have become powerful enough to virtualise every network device and simulate network scenarios without the need to physically buy or rent expensive network hardware. As a result, a special type of software referred to as *network simulators* has quickly bloomed. This type of software allows users to have a fully functional and realistic network topology to experiment with inside the software itself. Practising computer networks while studying has become easily accessible, and students are now able to learn by experimenting with network protocols and network device configuration.

Virtual Square [3] has created the Virtual Distributed Ethernet (VDE) project, which can be used to connect virtual entities in an Ethernet-compliant way. This project has been described more in depth in Chapter 2.1.3. VDE is a very powerful tool, but can be difficult to manage, especially with a large topology simulation. ImagiNet is built on top of VDE to make complex configuration scenarios simple and easy to manage. Topologies can now be defined in a text file and started with just one command. ImagiNet will keep track of the state of the simulation and will be able to manage it through its entire duration.

ImagiNet was born with the intent to create a simpler way to configure and manage VDE topologies, though it became almost immediate that its capabilities were reaching way beyond a simple configuration tool. ImagiNet was becoming a unique network simulator that distinguishes itself from all the others. After this realisation, the development direction of ImagiNet has quickly changed.

ImagiNet is now an easy-to-use command-line tool capable of simulating network topologies effortlessly. It is designed for students and as a complementary tool for network course laboratories in universities. This thesis covers all relevant ImagiNet aspects and offers examples on how to use it for network laboratories. At the end of it, the reader should understand how ImagiNet place itself among all the other network simulation software and how powerful and useful it can be as a complement to a network course.

1.1. State of the art

There are a lot of Network simulation or emulation software with each of them having unique characteristics. This section is primarily focused on the most used ones and what makes them different from one another.

1.1.1. GNS3

GNS3 is an open-source cross-platform graphical network simulator first released in 2007 [4], [5], [6]. It's used to emulate network devices from various vendors (Cisco, Juniper, Mikrotik, Arista, etc.), and it's primarily used by students who need hands-on experience while studying for network certifications (e.g. CCNA) [7].

GNS3 is designed with a client-server approach where the server is a background process that is responsible for orchestrating all the simulations, while the graphical user interface (GUI) is the client that allows the user to interact in a friendly way with the server. The majority of the time, the server is hosted on the same machine as the client, but it is possible to host the server remotely. The only way to use GNS3 is with a GUI; it's not possible to use it without it.

Simple PCs that can execute simple commands like `ping`, `traceroute` or `netcat` are simulated using VPCs. These PCs provide a basic node to test connectivity, but are not meant to be a service runner. Nodes interconnect with uBridge[1], Dynamips is used for IOS[2] routers and switches, and QEMU and Docker are used for other types of devices [8], [9]. It must be noted that GNS3 can emulate various devices from different vendors, but the firmware images must be provided by the user. Most vendors do not have free and publicly available images that can be downloaded.

1.1.2. Packet Tracer

Packet Tracer is a network simulation tool developed by Cisco, where the user can practice networking and cybersecurity skills in a virtual lab [10]. It's primarily used for testing and practising for Cisco certifications (e.g., CCNA, CCNP, etc.), as it includes all the necessary Cisco firmware images for the devices that need to be studied during the preparation for a certification. As the images of Cisco devices are not openly distributed, this could be one of the only ways to test a Cisco device in a virtual environment.

[1]uBridge is developed by the GNS3 team: https://github.com/GNS3/ubridge

[2]Internetworking Operating System is a family of proprietary network operating systems developed by Cisco.

Packet Tracer is closed-source and not entirely free to use. There is a free version available, but it comes with limited functionality, one major limitation being the number of devices that can be included in a simulation, which makes it not very practical. The program has a graphical user interface and cannot be used without it. Since it is closed-source, there is no available information on how virtualisation is implemented or how PCs, switches, and routers are interconnected. Additionally, it does not support integration with external programs like QEMU or Docker, nor does it allow distributing the simulation load across one or more additional computers.

1.1.3. EVE-NG

EVE-NG is a network emulation platform designed for professionals and students to design, test, and simulate complex network topologies in a virtualised environment [11]. Unlike Packet Tracer and more similar to GNS3, EVE-NG supports images from multiple vendors. This makes it useful not just for Cisco certifications, but also for training scenarios where multi-vendor devices are interconnected. However, like GNS3, users must supply their own operating system (OS) images, as EVE-NG does not include them.

EVE-NG is closed-source and offers a Community edition and a Professional Edition. The Community edition is free to use, but is limited in the number of features. Both editions run in a web browser using an HTML5 interface, and it is not possible to use them without a graphical interface. Unlike Packet Tracer, it supports integration with other tools like QEMU and Docker and allows labs to be distributed across multiple servers. EVE-NG offers a more realistic lab environment than Packet Tracer, but it has a steeper learning curve and higher system requirements.

1.1.4. Boson NetSim

Boson NetSim is a closed-source, non-free network simulation tool designed specifically to help students prepare for Cisco certifications. In particular, there are three prices for CCNA, ENCOR and ENARSI certification [12].

The majority of labs are already configured, but customisation is possible. For device and network simulation, Boson's proprietary Virtual Packet Technology is used [13]. Cisco device images are already provided, but are limited and not all devices are available. The only version available is accessible via a website, which means that the simulation is not done on the user's machine, but on some proprietary server. While this can be useful as it can simulate topologies on a generic device that has a web browser, there is no possibility to integrate NetSim with any third-party tool like Docker or QEMU.

1.1.5. Kathará

Kathará is an open-source container-based network emulation tool for testing and learning [14]. The development of the project is based in Italy, with significant contributions coming from individuals linked with the Università degli Studi Roma Tre. It is also used in multiple universities to teach networking courses [15].

Kathará requires root privileges as it is heavily dependent on Docker, where every node in the network is a container. As Docker is not ideal for configuring network scenarios based on its configuration files and how it is managed, one can think of Kathará as a wrapper around it. It is a command-line tool and does not require a graphical interface.

It is based on three concepts: *device*, *collision domain* and *network scenario* [16]:

- A **device** is a virtual entity that represents and acts like a real device. It could be anything from a router, a simple PC or a full Domain Name System (DNS) server. Every device has its own disk and can be limited in the amount of resources (i.g, CPU and RAM) it can use. It is possible to pre-configure these devices with a configuration file that specifies what commands need to be executed at startup. In Listing 1 is reported an example of a file is reported that assigns an IP to the `eth0` interface and starts the Apache server.

  ```sh
  ip address add 10.0.0.1/24 dev eth0
  ip link set dev eth0 mtu 1450

  /etc/init.d/apache2 start
  ```

  Listing 1: A sample startup configuration file for a Kathará node [17]

- A **collision domain** is a layer 2 local area network (LAN) that interconnects devices. It acts like a hub because it does not filter packets based on MAC address like a switch would do. It is common to associate a single letter with collision domains in Kathará.

- **Network scenarios** are a representation of a network topology. In particular, it is a directory that contains multiple configuration files that specify which devices are present, in which collision domain they are attached and all the configuration needed for the devices. The Kathará command line interface (CLI) will then read this directory and start the topology. Listing 2 shows an example of which files are inside the directory. The `*.startup` files are configurations for devices, while the `lab.conf` file is the main topology configuration in which collision domains are specified. Listing 3 shows the content of the file `lab.conf`.

```device
lab.conf  pc1.startup  pc2.startup  r1.startup
r2.startup
```

Listing 2: Directory structure of a simple Kathará Network scenario [18].

```kathara
pc1[0]="A"

r1[0]="A"
r1[1]="B"

pc2[0]="C"

r2[0]="C"
r2[1]="B"
```

Listing 3: A configuration example for Kathará in which four devices are specified. We can see that `pc1` has only the `0` interface connected to the collision domain `A`. Instead, `r1` has the `0` interface attached to the collision domain `A` and the interface `1` attached to the collision domain `B` [19].

A *Docker Network Plugin* is a piece of software that defines and provides connectivity and communications between Docker containers [20]. The *Docker bridge* is the default driver for establishing private container networks, but it is not a proper layer 2 broadcast domain or a Linux bridge, as it requires an IP subnet assigned to it. To overcome the limitations of the default Docker's network plugins and create a real layer 2 broadcast domain, Kathará developed a new plugin [16].

This plugin is based on VDE and for every collision domains will create a hub to attach the containers [21]. *VDE-2* (Chapter 2.1.3.1) is used to create a hub, therefore a `vde_switch`[3] process is started with the flag `-x`. To attach a container to the switch, a TAP interface is created inside the container with `vde_tap` and it's attached to the switch as shown in Figure 1. It must be noted that the switch process is not running inside the containers but in a separate namespace that is not shared with the host. In fact, this works even though *VDE-2* is not installed on the physical machine, but the *pid* and *crl* sockets of the switch are still present under `/tmp/katharanp/kt-<NET ID>`.

[3]It is important to note that, at the time of writing, the switch is initialised with an excessively large number of ports. This leads to a significant performance degradation, as the *VDE-2* switch does not efficiently handle a high port count when the hub flag is enabled. As a result, Kathará experiences substantial performance issues under these conditions.

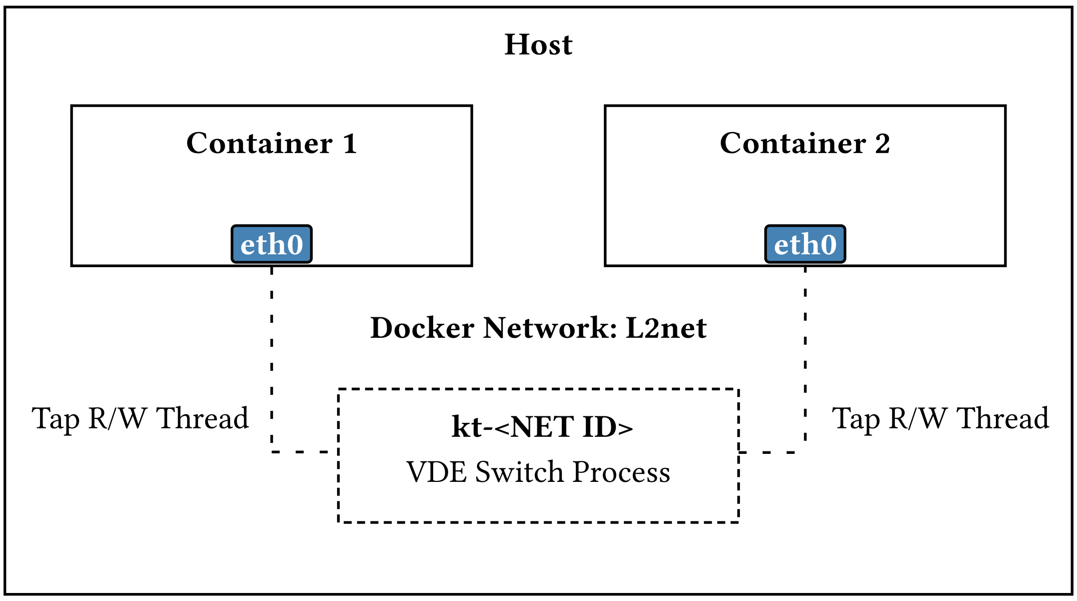

Figure 1: Kathará Docker Network Plugin diagram.

Docker and its network plugins are written in the Go programming language, while *VDE-2* is written i C. To be able to use VDE with Go, a feature called *Cgo* is used that lets Go packages call C code [22].

Kathará does not allow to configure the VDE switch as it is intended to be used only as a hub. This means that VLANs can't be configured or must be tagged inside the containers (but this leaves out the possibility of having untagged or access ports). It also does not provide the possibility to attach multiple switches together to test spanning tree protocol (STP) topologies.

Although users can provide their own container image, Kathará has a more open approach, and it is not intended for studying specific vendor hardware or software; therefore, it is not the best tool for studying for network certifications. Meanwhile, it is versatile and powerful for advanced routing and protocol testing. Inside the official lab repository, a lot of examples can be found about inter-domain and intra-domain routing, datacenter routing, openflow and even a full lab dedicated to P4 [23].

Kathará does not officially support integration with eternal tools like QEMU or other VDE connections. Also, the VDE switch used to create collision domains is started inside a different namespace, so it is impossible to connect to it from the host using a normal VDE connection. But it is possible to connect a collision domain to an interface on the host.

1.1.6. Containerlab

Containerlab is a CLI tool that provides orchestration and management for container-based networking labs [24]. It is built on top of Docker and requires root privileges in order to be used. Similar to Kathará, it uses configuration files to describe topologies in a declarative manner. These configuration files are in the YAML (YAML Ain't Markup Language) format [25]. Although Containerlab is primarily designed for containerised environments, it also supports the integration of virtual machines through Linux bridges or Open vSwitch (OVS) bridges [26]. A wide range of container images from various vendors are supported [27].

For container interconnections, Containerlab defines two types of networks [28]:

- The **management** network is based on Docker bridges. It connects all containers to the host and assigns both IPv4 and IPv6 addresses to each management interface.
- For inter-container peer-to-peer links, which are not part of the management network, a connection is created using Linux veth interfaces. These are virtual Ethernet devices that simulate a point-to-point connection between two endpoints, which may reside in different network namespaces.

Based on the available lab examples on the official website, Containerlab is targeted toward advanced networking studies. Numerous labs focus on dynamic routing protocols such as Open Shortest Path First (OSPF) and Border Gateway Protocol (BGP) in wide area network (WAN) topologies, as well as datacenter scenarios involving technologies like VXLAN (Virtual eXtensible Local Area Network) [29], [30].

1.1.7. Vnx

VNX is a general-purpose open-source virtualisation tool designed to help build virtual network testbeds automatically. It allows the definition and automatic deployment of network scenarios made of virtual machines of different types: Linux, Windows, FreeBSD, Dynamips routers or LXC containers. Based on a user-defined topology, it interconnects all the machines and also connects them to the host network [31].

VNX is significantly more complex than other tools. Its installation is non-trivial, and it requires a substantial amount of software to function. Network topologies are defined using XML (Extensible Markup Language) files, which can be difficult to read and manage. Running VNX requires root privileges, as well as a filesystem for each virtual device involved. It operates entirely from the CLI and does not rely on a GUI. As it is designed primarily for testbed environments, VNX provides a mature and structured way to define and execute commands within each device compared to other software.

VNX uses *libvirt* to interact with the virtualisation capabilities of the host. It also integrates Dynamips virtualisation to allow limited emulation of Cisco and Juniper routers. It is also possible to integrate Open vSwitch with support for VLAN configurations, inter-switch connections and software-defined networking (SDN) configurations.

The VNX website is partially outdated, with many links no longer functional at the time of writing. As stated before, the tool is not particularly easy to install or manage, so VNX may not be the most suitable choice for studying networking, especially for users with limited background knowledge. However, it can be a powerful tool for more advanced use cases and complex testbed environments.

1.1.8. Marionnet

Marionnet is a virtual network emulator born in April 2005. It provides a graphical user interface that allows users to define, configure, and deploy simple computer networks [32]. The platform is built from the ground up to utilise User-Mode Linux (UML) for virtualisation and VDE for the network infrastructure.

Marionnet is specifically designed for teaching computer networking in universities. It does not require root privileges, making it safe and suitable for deployment on shared lab machines without compromising system integrity or security. It operates entirely within a GNU/Linux ecosystem, without relying on proprietary software or vendor-specific equipment. Its simplicity compared to other network emulation tools makes it especially well-suited for beginners. Students can experiment with basic topologies and develop a solid understanding of fundamental networking concepts.

As it is based on VDE, it is the first network emulator that provides a real vendor-neutral interface for the core switch. This allows students to learn concepts like VLANs or spanning tree without the complexity of vendor-specific commands.

Marionnet is based on the *VDE-2* (see Chapter 2.1.3.1) and does not fully leverage newer features introduced in *vdeplug4* (see Chapter 2.1.3.2). According to its official website, bug reports on Launchpad and the state of the latest release, development appears to have stagnated, and the project is likely in a maintenance or inactive state. This lack of ongoing support has resulted in compatibility issues with recent Debian-based distributions. Currently, Marionnet can be installed or used by downloading the latest available virtual machine image from the official website. However, this image is based on Ubuntu 16.04, a release that is more than nine years old and exhibits various stability and security issues.

1.2. Objective

With ImagiNet, we aim to provide a simple to use, open-source, vendor-neutral network simulator to test, teach and learn about computer networks. This project is inspired by Marionnet (Chapter 1.1.8), but it leverages new features of VDE that are part of *vdeplug4*. It is completely redesigned to be a lot smaller and performant, while providing more features and having a more user-friendly interface.

ImagiNet is intended to be used as a tool to learn and teach about simple and intermediate networking topics. It is perfect for universities as it does not require any privileges and can be installed in laboratories where users do not have root privileges on the computers. It can also be used remotely as it does not require a graphical user interface and can scale in order to be used across multiple machines.

1.2.1. Structure of the document

In Chapter 2, we cover the necessary background knowledge required to understand the subjects of all chapters. We also describe in detail all the design choices behind ImagiNet and how they relate to other software simulation tools. This chapter also includes a section entirely dedicated to the performance evaluation of ImagiNet and the simulations carried out with it.

In Chapter 3, we propose multiple laboratories and exercises that demonstrate how networking topics can be taught with ImagiNet. All laboratories are comprehensively commented and guided, serving as a starting point for hands-on examples of the most important concepts covered in a network course.

Finally, in Chapter 4, we present a comprehensive list of possible features to improve ImagiNet's capabilities and interface. The final section provides closing remarks for this work, reflecting on the achievements and future potential of the project.

Implementation

This chapter covers the implementation of ImagiNet, the design choices and a performance evaluation of the simulations created. By the end of this chapter, the reader should have a clear understanding of the design choices behind ImagiNet, the top-level implementation details and how it differs from other network simulation tools.

2.1. Background

This section provides the reader with the necessary background knowledge to understand the subjects addressed in the following chapter.

2.1.1. VMs

A Virtual Machine (VM) is a virtualization of a computer system. It is based on computer architectures and provides the functionality of a physical computer. By virtualizing the physical hardware, a real operating system can be run, by fully isolating it from the host where the virtual machine is running.

QEMU [33] and VirtualBox [34] are the most famous implementations of virtual machines. User-Mode Linux is another implementation that works only on Linux [35], [36].

2.1.2. Linux namespaces

Namespaces are a feature of the Linux kernel that partitions a set of resources in order to make different processes view different resources than others. The type of namespaces present at the time of writing are: mount (mnt), process id (pid), network (net), inter-process communication (ipc), hostname (uts), user id (user), cgroup namespace (for resource control isolation) and time namespace. Each of them isolates a single type of resource. For example, the network namespace is used to isolate network devices, IPv4 and IPv6 protocol stacks, IP routing tables and firewall rules. Meanwhile, the mount namespace is used to control mount points and is utilized by `vdens` (see Chapter 2.1.3.3) to provide a different DNS server than the one configured on the host machine.

Namespaces are used to implement containers in Linux. While containers might seem similar to virtual machines at first glance, an important difference is that processes within a namespace still share the same kernel as the rest of the operating system. This means that the performance are greater and the overhead is much lower, but the isolation is weaker: if there's a vulnerability in the kernel, the entire system could be compromised, not just the container.

2.1.2.1. Docker

Docker is a set of platform-as-a-service (PaaS) products to deliver software in lightweight containers. This tool allows to automate the deployment of applications in containers so that applications can work efficiently in isolation.

Containers are isolated from one another and bundle their own software, libraries and configuration files. They can communicate with each other through Docker bridges that are somewhat similar to IP switches. Because all of the containers share the services of a single operating system kernel, they use fewer resources than virtual machines.

As noted in the first chapter, a lot of network virtualization programs use Docker to simulate hosts. This allows for a great topology generation with a lot of different software and services that do not consume a large amount of resources like VMs. Docker works well within itself, but it's not the most flexible tool to be integrated with VDE or other external services and it also requires root privileges[4]. We did not think that integrating ImagiNet with Docker was necessary and that it would not bring significant value in the project. It is although possible, as ImagiNet is based on VDE, to integrate it manually with any external tools.

2.1.3. VDE

VDE is an acronym that stands for *Virtual Distributed Internet*. It is a virtual network that is Ethernet compliant and can run on different physical hosts, making it distributed [37]. The software is written in C and it offers a library that can be used to integrate VDE in other applications, but it is mostly a set of command

[4]There exists a rootless version of Docker but has more limitations.

line tools that can be used to create switches, cables, tap interfaces and tunnels between physical hosts.

VDE operates entirely in user space, without requiring any privileges. The only privilege required is if the user wants to create a tap interface. This is not a limitation of VDE, but rather a security feature of Linux. The fact that no privilege is required to use VDE makes it a perfect tool for ImagiNet to create the underlying networking. This is because it can be used in an environment where the user does not have root privileges, which is a common scenario in school and university laboratories.

VDE is integrated with several virtualization tools like QEMU [33], KVM [38], User-Mode Linux [35], [36] and VirtualBox [34]. It can also be used inside Linux namespaces (see Chapter 2.1.3.3), making it well-suited for interconnecting several virtual machines to create a virtual laboratory for learning and testing. It also must be noted that some networking TCP/IP stacks like *lwipv6* and *picotcp* [39] have sockets for VDE plugs, as well as namespace-implemented TCP/IP stacks like *libvdestack* [40] and educational emulators like *umps3* [41].

The initial VDE version is called *VDE-2* (see Chapter 2.1.3.1) and is the official VDE version supported by most of the virtualization platforms. Another version of some parts of VDE has been developed and is called *vdeplug4* (see Chapter 2.1.3.2). This last version does not cover all *VDE-2* functionalities but offers a re-implementation of some of them in order to make them more efficient. It also brings new plugins for new types of connections (like PTP or VXVDE). This last version is not directly integrated into *VDE-2* because it uses some features that are specific to GNU/Linux and can't be ported to other operating systems like MacOS, meanwhile, *VDE-2* must stay compatible with them.

2.1.3.1. VDE-2

VDE-2 [42] is the first and main implementation of VDE. It is all written in the C programming language [43] and provides multiple command line programs. ImagiNet only uses a subset of these tools as some of them are replaced by *vdeplug4* (see Chapter 2.1.3.2) or are just not implemented in the current version of the program. The following is a list of what parts of *VDE-2* are used by ImagiNet and how they work:

- **vde_switch**: This is an implementation of a managed switch. A switch is a physical (or in this case virtual) device that has multiple interfaces and is able to forward Ethernet frames between them. It differs from a hub because it forwards frames based on the destination MAC (Media Access Control) address.

A managed switch allows an administrator to modify the configuration like set VLANs (Virtual Local Area Network) or spanning tree options. If started normally, the VDE switch has a CLI that allows to configure multiple options or just print some internal structures like the MAC table. It is also possible to configure the VDE switch to act as a hub.

Unlike the majority of network simulations presented in Chapter 1.1, the VDE switch provides a simple and vendor-agnostic management interface. This is a great approach to learn how to manage a switch in a simple manner, without being tied to complex, vendor-specific commands. Some network simulation software relies on powerful tools like the OVS (Open vSwitch) for switch functionality. While OVS is a feature-rich virtual switch, its interface can be quite complex, especially for beginners without prior experience. A more in-depth analysis of the interface of this switch is provided in Chapter 3.2.1.

- **vdeterm**: Multiple VDE devices can be started in *daemon mode*, which detaches the process from the current terminal and allows it to run in the background. When a VDE switch is running in the background, `vdeterm` can be used to access his CLI. `vdeterm` also offers a better CLI experience with features like autocompletion and command history, which are not built into the managed devices themselves to keep the rest of the code simple and efficient. To allow `vdeterm` to manage a device, that device must expose a management socket, to which `vdeterm` can connect to.

- **wirefilter**: The `wirefilter` utility can be thought of as a managed cable. It acts just like a regular network cable, but it provides a dedicated CLI to allow the user to dynamically change its behaviour. This is incredibly useful for simulating real-world network conditions by adjusting parameters such as delay, noise, and packet drop percentage. It can simulate lossy and long-distance connections, where high latency and packet loss become significant factors. It's also a great tool for testing the resilience of new network protocols (like TCP) that need to be robust against packet loss. It's important to note that for standard, unmanaged cables in ImagiNet, the `vde_plug` utility from *vdeplug4* (Chapter 2.1.3.2) is used. The `wirefilter` cable is only used when a connection is specifically flagged with the `wirefilter` flag.

2.1.3.2. vdeplug4

vdeplug4 [44] is a more recent version of the `vde_plug` and `dpipe` utilities originally found in *VDE-2*. It is backwards compatible with *VDE-2*, but this new version leverages some specific Linux features that would break compatibility with other operating systems if implemented in the older *VDE-2*. This new version of

VDE brings the possibility to use plugins to be as modular and flexible as possible. Thanks to the *libvdeplug4* is also possible to write new plugins that integrate with the VDE ecosystem.

vdeplug4 comes with several pre-written plugins that provide implementations for a hub, an unmanaged switch, and various types of connections. These new hub and switch plugins are lighter than their *VDE-2* counterparts. However, because they are unmanaged, they do not offer any configuration options and at the moment are not used by ImagiNet. In regard to the various types of connections, the following is a list of the ones currently used by ImagiNet:

- **vde**: This type of connection allows the `vde_plug` utility to be backwards compatible with *VDE-2*. It is used by ImagiNet to connect to VDE switches. This plugin requires that some device is already listening to the specified endpoint. This means that it is not possible to start a cable without first starting the switch.

- **ptp**: This plugin is used to create a peer-to-peer connection between two devices. It is possible to start listening for a connection from both devices and if one of them disconnects, the other starts listening to the same endpoint. This is useful as there is not a predefined order in which two devices need to be started, unlike for the *vde* connection type.

- **vxvde**: This plugin implements VXVDE [45]. This is a VXLAN replacement that can be used to distribute and create VDE connections over a real network. This can be used to distribute the virtual laboratory on multiple physical machines. A laboratory is proposed in Chapter 3.5.1.

The utilities provided by *vdeplug4* are used by ImagiNet in the following manner:

- **vde_plug**: This is used to connect two endpoints together. ImagiNet uses this utility to create and simulate regular cables (that do not involve `wirefilter`). To connect single hosts the *ptp* plugin is used and the *vde* plugin is used to connect to *VDE-2* switches.

- **dpipe**: From the man page: “`dpipe` is a general tool to run two commands diverting the standard output of the first command into the standard input of the second and vice versa”. This utility is used by ImagiNet to create cables with `wirefilter`.

It must be noted that *VDE-2* also provides a version of both `vde_plug` and `dpipe`. It is important to install *vdeplug4* after *VDE-2* to override these versions with the new ones.

2.1.3.3. vdens

`vdens` [46] is a program written in C that creates a Linux namespace with its own independent networking stack that is compatible with VDE. It does not require being root on the system, and the user acquires all the management capabilities on the networking stack of the namespace. This grants the user with permissions to to create, remove and configure interfaces that are all connected to one or more VDE external connections.

ImagiNet uses `vdens` to simulate a lightweight network node that can be used to test connectivity with the same utilities that are installed in the host machine (like `ping`, `traceroute`, etc.). It is also possible to use the namespace as a router, enabling the *ip forwarding* kernel parameter.

`vdens` differ from a classic container, like the ones managed by Docker, because it is substantially simpler and tries to create only a network namespace. This means that all files and programs from the Linux host are still accessible and usable. This does not provide isolation from the host, but the code required to run `vdens` is much simpler and shorter than a classic container implementation like LXC or Docker. The isolation is not strictly a security risk because `vdens`, especially in the context of ImagiNet, is meant to be used only as a test tool where only safe code and safe utilities are executed. If some external and untrusted code has to be tested, a Virtual Machine is highly suggested.

2.1.3.4. libvdeslirp

Slirp, which originally was designed to provide PPP/SLIP connections over terminal lines, has evolved into a versatile TCP-IP emulator. It is now widely used by virtual machine hypervisors to provide virtual networking without requiring any host configuration or privileged access. Major virtualization platforms like QEMU, VirtualBox, and User-Mode Linux all utilize it for this purpose.

The QEMU development team separated their Slirp implementation into a standalone library called *libslirp*. To bring this functionality to VDE networks, the *libvdeslirp* [47] library was written as a wrap of the *libslirp* code.

By installing *libvdeslirp*, the user can utilize the *slirp* plugin with *vdeplug4*. This allows virtual hosts to connect to the Internet without needing any special privileges. It is especially useful for virtual machines where new tools must often be installed.

2.2. Declarative configuration

In the “Iron Age” of computer science, systems were tied to physical hardware, making infrastructure provisioning and maintenance a manual and time-consuming task. Now, in the “Cloud Age”, systems are decoupled from physical hardware. Automated software handles routine tasks, freeing up system administrators. Changes can be made rapidly, allowing for faster and more reliable change management.

Infrastructure as Code (IaC) applies software development practices to IT infrastructure. It ensures that system provisioning and configuration are consistent and repeatable by defining changes in code and deploying them automatically with built-in validation. This methodology allows organizations to leverage powerful software tools like version control systems (VCS), automated testing, and deployment orchestration to manage their infrastructure. Today, IaC is a universal practice in cloud environments, making it a standard for any organization that manages cloud infrastructure [48]. A notable tool that exemplifies this approach for major cloud providers is OpenTofu [49].

As principles and practices of IaC can be applied to infrastructure whether it runs on the cloud or is virtualized systems, ImagiNet is designed with this philosophy in mind.

To define a network topology, ImagiNet uses a single plain-text file that specifies all the devices and their connections. This approach allows users to effortlessly and reliably rebuild any topology on any machine configured to run ImagiNet. It also makes it simple to create, destroy, or replace devices and connections by modifying just a few lines of code, all without needing to interact directly with the ImagiNet interface. This approach stands in contrast to tools like GNS3 (Chapter 1.1.1) and Marionnet (Chapter 1.1.8), which require users to manually create and save configurations within the program’s graphical interface. ImagiNet is more aligned with tools like Kathará (Chapter 1.1.5), but instead of defining a its own custom file format, ImagiNet leverages the widely-used and human-readable YAML format.

YAML (YAML Ain’t Markup Language) is a widely used, human-friendly, data serialization language [25]. It was chosen for its simplicity and widespread adoption, as well as the availability of pre-written parsers for most programming languages. This last point allows ImagiNet to check for syntax errors without the need to develop custom code for a unique custom language.

```yaml
switch:
  - name: sw1

namespace:
  - name: ns1
    interfaces:
      - name: eth0

  - name: ns2
    interfaces:
      - name: eth0

cable:
  - name: conn1
    endpoint_a:
      name: ns1
      port: eth0
    endpoint_b:
      name: sw1

  - name: conn2
    endpoint_a:
      name: ns2
      port: eth0
    endpoint_b:
      name: sw1
      port: "10"
```

Listing 4: An example of a topology file for ImagiNet written in YAML. A switch called *sw1* is defined. Two namespaces (*ns1* and *ns2*) with one interface each (called `eth0`) are connected with two cables (respectively *conn1* and *conn2*) to the switch.

An example of a configuration file is provided in Listing 4. In this example, a switch is defined under the *switch* array. Two namespaces are present and, for each one, only an interface is declared. To connect two devices, a cable must be written in its appropriate section. A cable is defined by a name and its two endpoints. In this example, the cable *conn1* connects the `eth0` interface of the first namespace (*ns1*) to the switch. If a port is not specified, the first free port is chosen by ImagiNet. For the second cable (*conn2*), the `eth0` interface of the second namespace (*ns2*) is connected to the 10th port of the switch.

ImagiNet imposes some constraints that can't be encoded in the YAML language and must be verified by the application logic. Some of them are:

- Device names must be unique in all the topology
- Interfaces must only have one connection attached to them.
- The switch can't run out of ports
- If an ip is specified under a namespace interface it must be valid
- If a gateway is specified under a namespace interface it must be valid and inside the ip subnet
- If a configuration file for a switch or a cable is provided, the file must be readable

Having only a simple plain text file, allows ImagiNet users to define new topologies without having to rely on the ImagiNet interface itself. To create a new topology only a simple text editor is needed. It is very easy to share topologies with other persons or students, and also this format is perfect for version control systems like Git [50].

2.3. Interface

A command-line interface (CLI) allows users to interact with a software using text commands, in contrast to a graphical user interface (GUI) which relies on visual elements. Since CLIs run directly from a terminal, they do not require a graphical environment. While some users might perceive CLIs as more complex, they offer a superior combination of simplicity and portability compared to GUIs. Creating a user-friendly and portable GUI is a significantly more demanding development task than building a CLI. Furthermore, CLIs tend to have a longer lifespan, whereas GUIs can quickly become outdated.

ImagiNet was specifically designed as a CLI tool to maximize its portability across various desktop environments. This design choice also ensures it can be used on computers that lack a graphical interface entirely. For a detailed overview of ImagiNet's current CLI interface, the reader can refer to the resource provided in Listing 5.

The ImagiNet CLI has several groups of commands:

- **Topology**: This sets of commands are used to create, destroy and manage topologies. This set includes:

```
Create and manage VDE topologies

Usage: imaginet [OPTIONS] [COMMAND]

Commands:
  add     Add a device to the current topology
  attach  Attach to a device in the topology
  create  Create a topology from a yaml configuration
  clear   Stop and delete the current topology
  dump    Dump current raw configuration
  exec    Execute a command in a device
  import  Import a topology from a raw configuration file
          (generated with dump)
  rm      Remove a device from the topology
  start   Start devices in the current topology
  status  Status of running topology
  stop    Stop devices in the current topology
  help    Print this message or the help of the given subcommand(s)

Options:
  -b, --base-dir <BASE_DIR>  Base directory for all imaginet files
  -t, --terminal <TERMINAL>  Terminal to open when starting or
                             attaching to a device
  -c, --config <CONFIG>      Path to global configuration file
  -v, --verbose...           Verbosity level. Can be used multiple
                             times for increased verbosity
  -h, --help                 Print help
  -V, --version              Print version
```

Listing 5: ImagiNet CLI

- `create`: This command initiates the creation of a network topology based on a specified topology file. If no file is provided an empty topology is created. This command is typically the first step, used to establish a network scenario that can then be interacted with and managed.

- `clear`: At the time of writing, ImagiNet is able to manage only one topology at a time. If a topology was already created and another has to be created, this command will remove all the ImagiNet files and clear the current topology. It also attempts to stop any devices that are still running.

    - `add`: After creating a topology from a file, is possible to add a device without the need to modify the topology file and without the need to stop and recreate the entire topology from scratch.

    - `rm`: This command is the opposite of the `add` and will remove a device from the current active topology.

    - `dump`: When devices are added or removed using the `add` or `rm` commands, the active topology no longer matches the original topology file. To save the current modified topology to a file, the `dump` command can be used. This command prints the internal representation of the topology in a format that can be imported later using the `import` command. It's important to note that ImagiNet's internal representation for the active topology differs from the format used in the initial topology files for the `create` command. As of now, the `dump` command cannot generate a file compatible with the original topology file format.

    - `import`: This command takes the output generated by the `dump` command as input and uses it to recreate a saved topology

- **Management**: This set of commands are used to manage the running topology. This set includes:

    - `start`: This command accepts a list of device names to be started. If no devices are specified, the entire topology will be started. When starting namespaces, a new terminal window is launched, which requires a graphical desktop environment. However, if no graphical environment is available, the `--inline` flag can be used to start a namespace directly in the current terminal. This command is also idempotent, meaning it will not attempt to start a device that is already running.

    - `stop`: This command attempts to stop a specified device. It is an idempotent command, meaning it does nothing if a device is already stopped.

    - `status`: This command prints a brief status of the topology. The status includes which devices are defined and their state. There is a verbose flag that allows to print of all the configuration options of the devices.

- **Interaction**: This set of commands are used to interact with the current devices in the topology. This set includes:

    - `exec`: This command is used to execute a command with arguments inside a device.

- `attach`: For some devices, like switches and cables with `wirefilter`, the CLI interface is not automatically opened by ImagiNet. To interact with them the user must attach to them with this command. It is also possible to open multiple terminals inside the same namespace.

2.4. Rust

Since all the VDE tools are written in C, to be consistent, it would have been possible to use the same language for ImagiNet. However, a more modern, high-level language was chosen instead, because ImagiNet doesn't require a large number of system calls or deep interaction with the core system, while it requires a lot of parsing that is more challenging in low-level languages like C. Python was considered for this project, but its lack of a type system and its interpreted nature were considered unsuitable for the tool's requirements.

The Rust programming language [51] was chosen for writing ImagiNet. It offers a powerful type system and is a compiled language, which provides significant performance benefits and allows for more error checks at compile time rather than at runtime. It also has pattern-matching capabilities, which are ideal for handling and parsing command-line arguments. Another convincing feature of Rust is its memory safety. The language is designed to prevent an entire class of bugs and vulnerabilities related to memory management at compile time, such as buffer overflows and use-after-free errors [52].

Rust's type system allows for the definition of custom data structures that internally represent different types of devices, complete with their own methods. A code snippet from the `src/vde/namespace.rs` file in the ImagiNet repository is reported in [53]. In this code, a `Namespace` structure is defined, containing a name, a vector of interfaces, and a vector of configuration commands. The `NSInterface` type has only the name field that is mandatory, meanwhile, the `ip` field is defined using the `Option` enum. This allows the value to be either `Some(value)`, indicating that a value is present, or `None`, indicating its absence. For example, when an interface's IP address is not specified, its `ip` field is set to `None`. If the interface is configured with the IP address `192.168.1.1`, the value

[5]This notation is not really valid in Rust since the `"..."` encodes a static string while in the `ip` definition there is a `String` type.

```rust
pub struct Namespace {
    name: String,
    interfaces: Vec<NSInterface>,
    config: Vec<String>,
}

pub struct NSInterface {
    name: String,
    ip: Option<String>,
    gateway: Option<String>,
}
```

Listing 6: A snip of the Namespace and Namespace interface definition types used in ImagiNet

would be `Some("192.168.1.1")`[5]. This eliminates the need for `null` values and makes the presence or absence of data explicit in the type system.

Rust also has a `Result<...>` type that can be `Ok(..)` or `Err(...)`. This allows functions to return only a result type that encodes both the correct value or an error with details. The caller can use pattern matching on the result and handle all the errors that have occurred. We believe this explicit error-handling approach is superior to the exception-based models found in languages like Python.

To interact with VDE and start a topology, ImagiNet does not rely on any VDE library directly, but it just executes the VDE utility commands as external programs. This design choice allows a new version of a VDE utility to be released and immediately integrated into ImagiNet without the need to recompile or update the code.

2.5. Stateful without a service

ImagiNet needs to be stateful in the topology that it manages as subsequent commands have to operate on the same active topology. For example, if a user starts a device, a subsequent call to the status command should display the device as active.

To keep track of the current state of the application, the majority of the network simulation tools rely on themselves being active throughout the duration of the simulation. Some of them, instead of being always active, interact with a daemon that keeps track of the state of the simulation in the background. For example, GNS3 relies on its daemon, while Kathará does not have its daemon but relies on Docker, which in turn relies on the Docker daemon to keep track of the container's lifetime.

ImagiNet was designed with a new approach in which the state of the topology is written on the filesystem of the host and no daemon of any sort is involved. By default, ImagiNet has a working directory located in `/tmp/imnet/` where it writes the topology file that describes the internal representation of the current active topology. Then, for each device, a new directory is created where the necessary sockets and pid files are placed. For each call to the ImagiNet CLI, the topology file is parsed and for the status of each device, the process pointed by the pid file is checked. Every VDE component (except for `vdens`) can be run as a background process and can generate a pid file that holds the process pid. This is used by ImagiNet because a device is considered active if the process associated with the device pid is running. For `vdens` a custom initialization script is implemented that is used to configure the namespace and write the pid of the process.

This design approach allows ImagiNet to avoid running continuously and consuming resources. Since topology management is only a small part of the overall simulation process, the performance overhead of parsing the topology file is negligible.

2.6. Keeping it simple

ImagiNet was designed to be as simple as possible while still being capable of simulating complex topologies. The namespaces are based on `vdens` (Chapter 2.1.3.3) and although they provide a very lightweight PC simulation to test connectivity, simple routing and firewall rules, they can restrict the ability to run multiple instances of the same program or daemon. Furthermore, a user may be unable to run a specific program because the namespace does not provide root privileges (despite the user inside the namespace having all the capabilities necessary to manage the network configuration). In fact, some applications simply check if the user is root rather than checking for the necessary capabilities.

For more complex scenarios where a fully isolated host is required, a virtual machine (VM) must be used. We decided not to integrate VM management directly into ImagiNet to avoid increasing the project's complexity. Virtual machines can be created using various hypervisors and involve a significant number of configuration options that are outside the scope of ImagiNet's core purpose. VMs can also be created manually using an Infrastructure as Code (IaC) tool like Vagrant [54], which aligns with ImagiNet's design philosophy.

To enable users to integrate virtual machines (VMs) with ImagiNet, a new type of connection was implemented that exposes an endpoint to connect external tools that are compatible with VDE. This type of connection is called "open" and means that ImagiNet does not care what is attached to it and delegates this responsibility to the user. An example of an open connection is in Listing 7, where a simple namespace is connected with a cable to an *open* endpoint.

The path of the open endpoint can't be safely determined before running ImagiNet because it is based on the base working directory that can be configured. This means that after writing a topology with an open connection, the topology must be created into ImagiNet with the `create` command and then the endpoint of the connection can be printed with `status -v` as shown in Listing 8.

The `endpoint_b` is the open endpoint and it exposes a *ptp* connection. This is a plugin found in *vdeplug4* (Chapter 2.1.3.2) that allows the creation of efficient peer-to-peer connections between two devices. We now present a simple way to attach a QEMU VM to the endpoint. Other tools can be used as long as they are compatible with *vdeplug4*. In Listing 9 is reported a QEMU command to start a

```yaml
namespace:
  - name: ns1
    interfaces:
      - name: eth0

cable:
  - name: conn1
    endpoint_a:
      name: ns1
      port: eth0
    endpoint_b:
      name: open1
      open: true
```

Listing 7: Example of a topology file with an open connection

```
Cables:
- conn1 inactive
  endpoint_a: PTP ./ns1/eth0 eth0
  endpoint_b: PTP /tmp/imnet/opn/open1
  wirefilter: false
```

Listing 8: Example of an *open* connection cable listed with the `status` command and the verbose (`-v`) flag.

```bash
qemu-system-x86_64 -enable-kvm -cdrom alpine.iso \
    -monitor stdio -device e1000,netdev=vde0 \
    -netdev vde,id=vde0,sock=ptp:///tmp/imnet/opn/open1
```

Listing 9: Qemu command to start a VM based on Alpine attached to a PTP VDE endpoint

simple VM based on an image of Alpine Linux [55] that has an interface attached to the open connection `endpoint_b`.

We have presented the open connections as a way to attach a VM to the current topology managed by ImagiNet, but as it is a simple VDE endpoint the user can leverage it in any way he wants. In fact, not all features of VDE are present in ImagiNet and in this way, some of them can be integrated. This allows for maximum flexibility and integration with third-party tools, unlike most of the network simulation software presented in Chapter 1.1.

2.7. Real word experience

Some network simulation software sacrifices realism for the sake of user convenience, which we believe can be confusing for students new to network administration. For instance, GNS3 (Chapter 1.1.1) allows the user to place Wireshark in the middle of a cable between two devices. In reality, Wireshark is almost always placed on one of the end hosts. Similarly, Kathará (Chapter 1.1.5) doesn't treat cables as separate entities, preventing users from simulating a cable fault by simply "destroying" it.

In contrast, ImagiNet was developed to mimic the design and management of real-world networks. Each device operates as a standalone entity, independent of others. This means the user can, for example, keep a namespace running while rebooting a switch. Moreover, all cables are treated as independent entities, allowing the user to activate or deactivate them to simulate faults without needing to modify any other devices in the topology.

2.8. Performance

The following sections will present some benchmarks that can help the reader understand how ImagiNet performs and how the created VDE topology performs.

All tests were done on an `Intel(R) Core(TM) i5-8365U CPU @ 1.60GHz` with `16GB` of RAM.

2.8.1. ImagiNet performance

Aside from starting the actual devices, which is a part not strictly related to how ImagiNet performs, the most challenging task is to parse the topology file provided and create a topology. During the parsing of the topology, a large number of checks need to be performed to make sure that multiple endpoints are not used more than once or that devices (like switches) do not run out of ports. These checks are mandatory because if, for example, in the topology, a switch with 8 ports has 9 devices attached, when starting all the devices, the 9th one will report an error that is cryptic to the user. ImagiNet tries to avoid errors by first checking that the topology can be safely created.

We don't expect users to simulate massive topologies with ImagiNet. The complexity of managing numerous devices grows significantly, and most educational objectives can be achieved with smaller, more manageable networks. However, for testing purposes, we decided to create a progressively larger topology to test ImagiNet parsing performance. To create a very large topology we wrote a script that is reported in Appendix A. The topology generated by the script is in two parts: the first part is a block that contains multiples switches with multiple namespaces attached to them and a router that is connected to all the switches. The block can be viewed in the Figure 2 figure. The second part of the generated topology connects every router from every block to all the other routers in a mesh topology as shown in figure Figure 3.

Every switch in the block has its own subnet, defined as `10.{n}.0.0/16` where `n` i the number of the switch in the block. The router has an interface for all

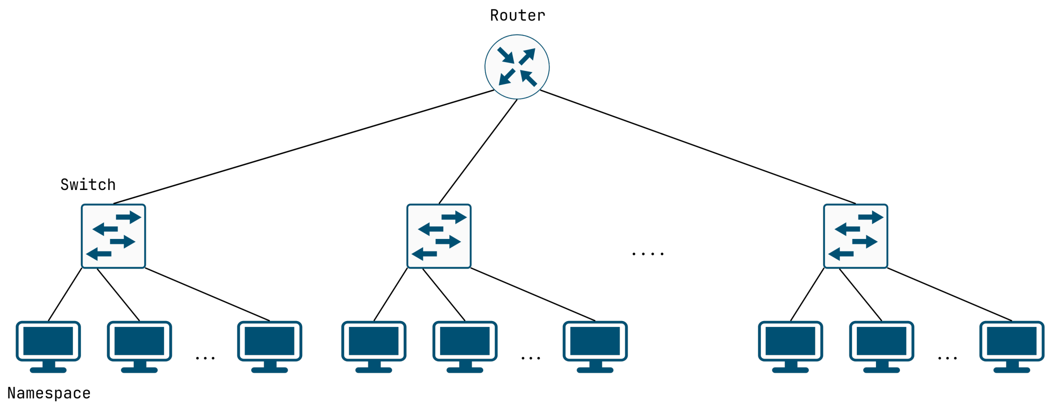

Figure 2: Block diagram generated

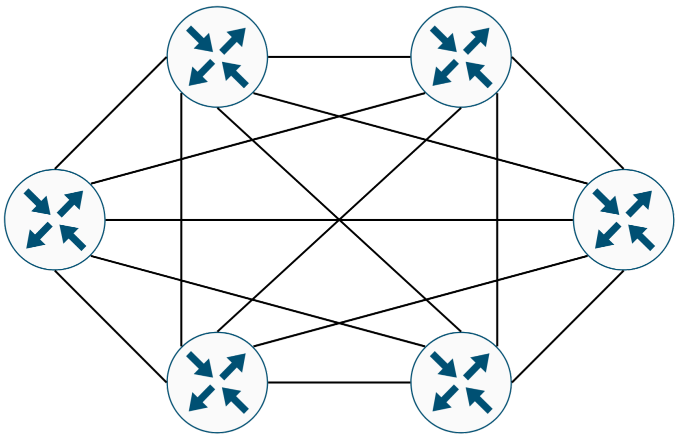

Figure 3: Routers mesh topology

the switches and other `m` interfaces for the connection to other routers in other blocks.

To evaluate ImagiNet's parsing performance, four different topologies were used to conduct tests. The script used to generate the topology has three parameters that can be customised to control the size of the network. The first one is the *number of namespaces per switch*. This factor dictates how many namespaces are generated for every switch in a block. The second parameter is the *number of switches* per block. The third one is the *number of blocks*. The parameters used for all the tests are reported in Table 1. The total number of devices in the generated topologies is reported in Table 2.

After generating the topology files and placing them in `/tmp/topo{n}.yml`, `hyperfine` [56] was used to evaluate performance. The command in Listing 10

Topology	**Namespaces per switch**	**Switches per block**	**Blocks**
`topo0.yaml`	4	4	4
`topo1.yaml`	8	8	8
`topo2.yaml`	16	16	16
`topo3.yaml`	40	16	20

Table 1: Parameters used to generate test topologies

Topology	**Namespaces**	**Switches**	**Cables**
`topo0.yaml`	68	16	86
`topo1.yaml`	520	64	604
`topo2.yaml`	4112	256	4472
`topo3.yaml`	12820	320	13310

Table 2: Total number of devices per topology

will first remove the ImagiNet working directory[6] and then run the `create` command for the topology file.

```bash
hyperfine -N --warmup 3 --parameter-scan t 0 3 \
  --prepare 'rm -rf /tmp/imnet' 'imaginet create /tmp/topo{t}.yaml'
```

Listing 10: Command used to test ImagiNet parsing

Table 3 reports the results of the benchmarks. ImagiNet, even for extremely large topologies, performs really well. The smallest topology test has an amount of 170 objects and takes under 5 milliseconds to complete the parsing and creation of the topology. This means that the running time of ImagiNet on normal topologies is negligible and the user should not be impacted by it in any way.

Topology	**Mean** (ms)	**Min** (ms)	**Max** (ms)	**Relative**
`topo0.yaml`	4.6 ± 0.8	3.0	5.8	1.00
`topo1.yaml`	18.7 ± 0.5	17.9	19.6	4.10 ± 0.71
`topo2.yaml`	286.1 ± 6.8	273.2	292.0	62.79 ± 10.83
`topo3.yaml`	2482.4 ± 65.4	2353.2	2577.9	544.67 ± 94.15

Table 3: Benchmark summaries

[6]This is done because if `create` finds an already created working directory it will stop. If the `--force` flag is passed to it, ImagiNet will try to parse the old topology and stop all devices before creating the new one.

2.8.2. Topology performance

ImagiNet was not designed to create the most efficient topology. This design choice was made because efficient VDE topologies would impose some constraints on device interactions from the user's perspective. The most simple example of this is a namespace (`vdens`) attached to a switch (`vde_switch`). The most efficient way to connect the namespace to the switch would be to create a VDE connection directly. To do this a `vde_switch` must be started with `vde_switch -s /tmp/sw` and a namespace attached to it with `vdens vde:///tmp/sw`. In this case, the switch will open a socket and listen to incoming connections, the namespace will connect to the switch via the socket and they will start exchanging packets. The problem with this scenario arises when the switch must be restarted. If the switch is restarted the listening socket is dropped, and all connections previously established with other devices are dropped. From the namespace perspective, nothing happens, except that now all packets sent over the VDE interface do not reach any destination. If the switch is restarted, even with the same parameters, the namespace has no way to connect back to it but reboot. ImagiNet was designed to be as realistic as possible with how real world networks function, and in a real world network, a switch can be restarted without the need to restart all the devices attached to it. Furthermore, from the user's perspective, the only two processes involved were the switch and the namespace. No cables are directly involved because the cable is coupled with the namespace. As a result, users cannot simulate a cable fault by simply turning it off.

To address this limitation, ImagiNet makes extensive use of the *ptp* connections available in *vdeplug4* (Chapter 2.1.3.2). In the scenario described, ImagiNet would start the switch and the namespace, but it would create a *ptp* endpoint for the interface. Then, it would launch a separate process for the cable, connecting the *ptp* endpoint to the switch. This design allows the switch to rebooted without restating the namespace. Only the cable needs to be restarted, and since the cable holds no configuration, this can be done easily. This design choice allows for a more realistic network simulation, but it brings a small cost as now more processes are involved. In the previous, more efficient scenario, only two processes were used; now, three are involved, as there is a dedicated process for the cable. This also adds more hops for packets as it travels between more sockets. It's important to note, however, that this performance trade-off is solely due to the way cables are managed. The namespaces and switches are still configured in the most efficient way possible.

To evaluate the performance impact of this design choice, we tested three specific topologies. A fourth, medium-to-large topology was also included to assess performance in a network that would be difficult to manage without a tool like ImagiNet.

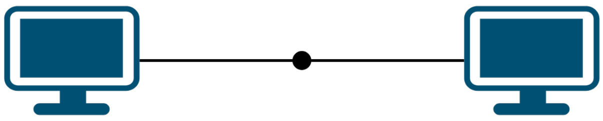
Figure 4: Efficient topology n.0

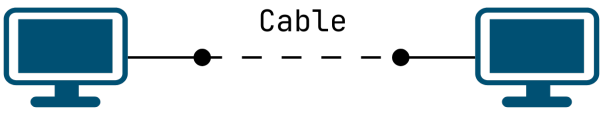

Figure 5: ImagiNet topology n.0

For all topologies, we measured the performance loss by measuring bandwidth using `iperf3` ([57]) and latency using `ping`.

The first topology is composed of two namespaces connected to each other. Appendix B.1 reports the commands used to generate the most efficient topology, while Appendix B.2 reports the ImagiNet topology file. In Figure 4 the efficient topology is presented, while in Figure 5 the ImagiNet topology is presented.

The second topology is composed of four namespaces connected sequentially as shown in Figure 6. The ImagiNet counterpart is shown in Figure 7. The first and the last namespaces are in different subnets and the middle ones are configured to route traffic between them. The commands used to generate the efficient topology and the ImagiNet topology file can be found in Appendix B.3 and Appendix B.4.

The third topology is composed of three switches and three namespaces as shown in Figure 8. VLANs are configured in the middle switch to separate traffic between the other two switches. All the commands used to generate the topology can be found in Appendix B.5, while the ImagiNet topology file can be found in Appendix B.6

In Table 4 are reported the `iperf3` tests across the three different topologies. In the last column is reported the performance loss in percentage. In Table 5 are reported the `ping` tests.

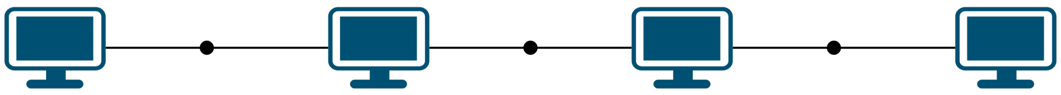
Figure 6: Efficient topology n.1

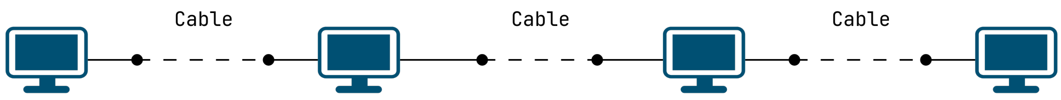

Figure 7: ImagiNet topology n.1

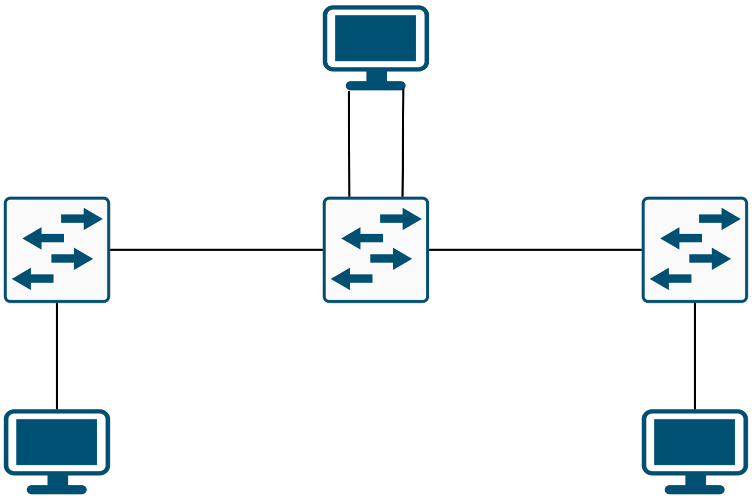
Figure 8: Topology n.2

Topology	**Efficient** (Mbit/s)	**ImagiNet** (Mbit/s)	**Loss**
`Topology 0`	2900	2340	19.31
`Topology 1`	1460	1190	18.49
`Topology 2`	1170	806	30.77

Table 4: Iperf3 benchmarks

Topology	**Avg e.** (ms)	**Mdev e.** (ms)	**Avg I.** (ms)	**Mdev I.**
`Topology 0`	0.389	0.099	0.815	0.095
`Topology 1`	1.065	0.188	1.760	0.314
`Topology 2`	1.946	0.133	2.585	0.711

Table 5: `ping` benchmarks. *Avg* stands for *average*, *Mdev* stands for *mean deviation*, *e.* stands for *efficient* and *I.* stands for *ImagiNet*

A last topology is presented to report performance across a medium-sized topology that is relatively large to build by hand. The topology is illustrated in Figure 9 and the ImagiNet file can be found in Appendix B.7. To test this topology `iperf3` was run simultaneously from the two namespaces on the left to the two namespaces on the right. The average bandwidth reported was `412 Mbit/s` for the first namespaces and `416 Mbit/s` for the second ones.

While other tools may offer superior performance, we believe the topologies generated by ImagiNet are more than efficient enough for nearly all of its intended applications. The simulation carried out by VDE, which occurs entirely in user space, is not the most performant method: tools that leverage the kernel network stack for simulation achieve better results. However, this user-space approach is a core feature of ImagiNet, as it does not require any special user privileges to

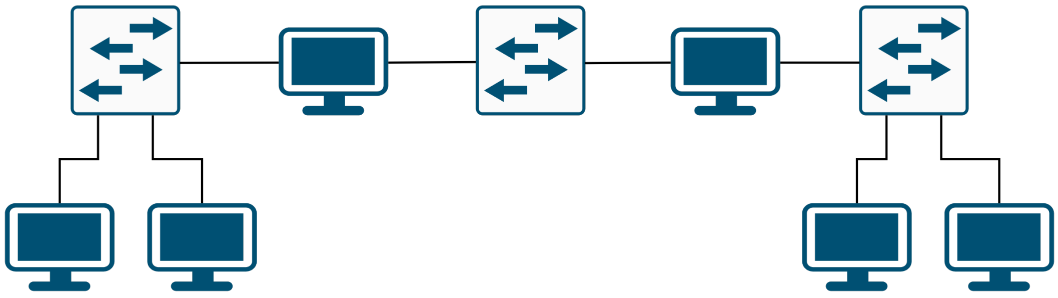

Figure 9: Topology n.3

run. This is a crucial benefit for a learning tool. The performance difference does not significantly impact its educational purpose, and in fact, the results for many performance tests are still quite positive.

Learning by doing

The learning by doing teaching method is a hands-on, task-oriented process that places learners as active participants in real-world experiences. It's based on the idea that students learn best by turning experiences into knowledge [58]. A computer network course can benefit from this approach by utilising network laboratories, where students will have direct experience with network scenarios and problem troubleshooting. Using a virtualisation technology to perform these laboratories can be cost and time-saving. As the laboratories are all simulated and no actual networking hardware is required, the maintenance of real equipment is removed, and also the restoration of the initial state of the laboratory can be simply carried out by a reset of the simulation program. From the pedagogical point of view, student interactions with a virtualised networking laboratory are practically the same as the interactions they would have with a real network. Furthermore, by parameterising the settings of the virtual network scenario, each student can be assigned a unique, personalised network that is different from their peers'. This method is effective for class sizes ranging from 100 to 400 students. Experiments on teaching computer networks using this approach have reported very positive results. The majority of students who participated felt they learned more through these exercises than they would have with more traditional methods [59], [60].

The following chapter presents a series of laboratories and exercises designed to teach a subset of networking topics for a university computer network course. At the end, the reader should have a clear understanding of how ImagiNet can be used by students and teachers for hands-on network teaching.

3.1. Networking basics

In theory, a computer networking course should be independent of a specific vendor's equipment or operating system. However, to complete the following laboratory exercises, students will need to use a small subset of commands specific to GNU/Linux and VDE. We believe the comprehensive benefits of these labs outweigh the minimal effort required to learn these commands. Furthermore, while the Linux commands will be universally useful in any professional context, the VDE commands are vendor-neutral, and their logic can be easily translated to other vendors' equipment.

3.1.1. Lab 0

This laboratory exercise is designed to familiarise students with essential commands for managing network interfaces and IP addresses on GNU/Linux systems using the *iproute2* [61] package. The laboratory utilises the same network topology employed in the performance evaluation chapter, with a graphical representation provided in Figure 5. To ensure a clean learning environment, the ImagiNet topology file is presented without any pre-existing configuration, as shown in Appendix C.1.

Students participating in this laboratory should possess fundamental knowledge of: Internet Protocol (IP) addressing principles, Network masking concepts and CIDR (Classless Inter-Domain Routing) notation for representing IP addresses and subnet masks.

At the end of this laboratory, students will have gained a practical understanding of:

- The role of IP addresses and subnet masks in network communication.
- ICMP (Internet Control Message Protocol) [62] functionality and the specific *Echo Request*, *Echo Reply* packet types.
- ARP (Address Resolution Protocol) [63] mechanisms and their role in Layer 2/ Layer 3 communication.
- TCP (Transmission Control Protocol) [64] connection establishment through the three-way handshake process.
- Analysis of network protocols using Wireshark [65].

This laboratory is organised into three distinct parts, each focusing on specific concepts while utilising the same underlying topology. In the first part, students will learn to configure network interfaces with appropriate IP addresses and conduct basic connectivity testing using the `ping` utility. In the second part, students will be familiarised with Wireshark for packet analysis, and they will experience the IP-to-MAC address resolution process through ARP. The final part focuses on testing TCP connectivity using `netcat` and analysing TCP protocol behaviour through Wireshark.

3.1.1.1. Interface configuration and connetivity test

To begin the laboratory exercise, students must first download the topology file referenced in Appendix C.1. The students must then proceed to create and start the topology using ImagiNet.

After the topology is started, two terminal windows will be visible, each corresponding to a distinct namespace within the topology. To distinguish the two terminals, students can observe the hostname displayed in each terminal's shell prompt. An example of the shell prompt is shown in Listing 11.

```bash
user@ns1:~$
```

Listing 11: Example of a shell prompt for a namespace

Students are now able to print the current status of the namespace interfaces. To do that, the first command from the *iproute2* packet should be introduced: `ip address`. An example of an output of this command is reported in Listing 12. In this output, there is a large amount of data, but only the important information is pointed out:

- There are two interfaces: the loopback interface (`lo`) is a virtual interface that lets the machine talk to itself. The `eth0` interface is the actual "physical" network interface that allows communication with other network devices attached to it.
- The `lo` interface has the loopback IP address configured: `127.0.0.1/8` for IPv4 and `::1/128` for IPv6. The `eth0` interface does not have any IP address configured yet.
- The `eth0` interface has a MAC address of `22:d2:46:1e:39:2e`, while the loopback interface has a special MAC address.

Since the theoretical aspects of networking should be covered by the course professor and multiple textbooks, no explanations of networking concepts are provided in these laboratory exercises. This and the following labs will focus solely on the practical implementation and observation of selected networking topics. If any topic is not well understood, we refer students to two textbooks that comprehensively cover the relevant theory [1], [2].

```bash
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host proto kernel_lo
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group
default qlen 1000
    link/ether 22:d2:46:1e:39:2e brd ff:ff:ff:ff:ff:ff
```

Listing 12: Output of the `ip address` command on an unconfigured namespace

Students should now try to configure the interface in both namespaces with two IPs that are on the same subnet. To add an IP address to an interface, the command `ip address add <ip> dev <interface>` can be used. It must be noted that the `<ip>` parameter has to be in CIDR notation, otherwise the netmask will automatically be `/32`. After picking two IPs (in this example, `10.0.0.1/24` and `10.0.0.2/24` are chosen) and after adding them to the appropriate interface, students can verify their operations by displaying again the interface's status with the `ip address` command. A partial example is reported in Listing 13, where the `10.0.0.1/24` IP was assigned to the `eth0` interface with the `ip address add 10.0.0.1/24 dev eth0` command.

```bash
2: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
group default qlen 1000
    link/ether 22:d2:46:1e:39:2e brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/24 scope global eth0
       valid_lft forever preferred_lft forever
```

Listing 13: Output of the `ip address` where `lo` is omitted. The `eth0` interface has an IP address

The last step that students must carry out to enable communication between the two namespaces is to bring both interfaces into the *UP* state. To do this, the command `ip link set <interface> up` can be used. For both namespaces, students should run `ip link set eth0 up`. It must be noted that, for namespaces, the interface never becomes *UP*, but changes its state to UNKNOWN. Students can now verify that the interface has changed state with the `ip address` command, as shown in Listing 14. Students should also notice that now `eth0` has an IPv6 link-local address too.

```bash
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
fq_codel state UNKNOWN group default qlen 1000
    link/ether 22:d2:46:1e:39:2e brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::20d2:46ff:fe1e:392e/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
```

Listing 14: Output of the `ip address` where `lo` is omitted. The `eth0` interfaces is now *UP*.

Students are now able to test the connectivity between the two namespaces with the `ping` tool. This tool allows sending ICMP Echo requests to an IP address specified as an argument and tracking ICMP Echo replies. If students execute the command `ping 10.0.0.2` on the first namespace, an output similar to Listing 15 should be printed. It must be noted that the `ping` command does not terminate by default. The `Ctrl+C` keyboard combination must be entered to stop it. The `64 bytes ...` lines mean that the other namespace is replying to the echo request.

```bash
user@ns1:~$ ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.774 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.389 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.679 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.580 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.610 ms
^C
--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4088ms
rtt min/avg/max/mdev = 0.389/0.606/0.774/0.127 ms
```

Listing 15: Example of the `ping` command

Students can now experiment by pinging unassigned IP addresses, attempting to ping the IP address of the other namespace after bringing down its interface using `ip link set eth0 down`, or deleting the second namespace IP address and assigning an IP address to it that is not within the same subnet as the first. To delete an IP address, the command `ip address delete <ip> dev <interface>` can be used. To delete all IP addresses on an interface, the `ip address flush dev <interface>` can be used.

3.1.1.2. Wireshark and ARP

The second part of this laboratory focuses on packet capture, analysis, and ARP. Observing the actual packets flowing through the wire and examining their contents is one of the most effective ways to understand how networking protocols operate. Starting with this lab, there will be a strong emphasis on observing which packets are generated and which are received during each operation.

Assuming students are working with a pre-configured topology where the two namespaces can successfully `ping` each other, they can begin capturing packets using Wireshark. Wireshark is a free and open-source packet analyser available on most operating systems. It requires a graphical user interface (GUI) to operate. If a GUI is not available, the terminal-based alternative, `tshark`, can be used instead. Official Wireshark documentation can be found on its website [65].

To launch Wireshark within the first namespace, students can run the following ImagiNet command: `imaginet exec ns1 wireshark`. Once Wireshark is open, students should select the appropriate network interface (`eth0`) to begin capturing packets.

While Wireshark is actively capturing, students should initiate a `ping` from one namespace to the other. This allows them to observe the packets flowing through the interface in real time. ICMP packets will appear in the capture: the namespace sending the ping will generate ICMP Echo Request packets, and the receiving namespace will respond with ICMP Echo Reply packets.

This provides students with a clear view of all the protocols involved in the communication: from the Ethernet frame to the IP header and finally to the ICMP packet. They should examine the Ethernet frame to identify the MAC addresses of both namespaces and the IP header to see the source and destination IP addresses.

Students may also notice the presence of ARP packets during communication. They should analyse these, especially the destination MAC address in the Ethernet frame of the ARP requests, which should be a broadcast address. In the corresponding ARP reply, students can observe the resolved MAC address returned by the target device.

To manually generate an ARP request, students can use the arping tool. For example, to resolve the IP address `10.0.0.2` to its corresponding MAC address, they can run `arping 10.0.0.2`.

Each namespace maintains an ARP table, which stores IP-to-MAC address mappings. To view the ARP table, students can use the `ip neighbour` command. It is also possible to flush the ARP table of an interface with the `ip neighbour flush dev <interface>` command.

3.1.1.3. TCP basics

One of the most important protocols used on the Internet is the Transmission Control Protocol (TCP). TCP enables the reliable transmission and reception of data, ensuring that information is delivered accurately and can withstand issues such as packet loss or network errors. While TCP is a large and complex protocol, some of its fundamental behaviours can be observed in this laboratory.

Assuming students are using the same pre-configured topology, where the two namespaces can successfully `ping` each other, they can initiate a TCP connection using the `netcat` utility. On one namespace, students can start a "server" that listens on a port with the following command: `nc -vlkp <port>`. On the other namespace, they can connect to this server using: `nc -v <ip> <port>`. Once the connection is established, students can test it by typing some text into one terminal and pressing `Enter`. The text should appear on the other end, confirming that the connection is working properly.

If these actions were performed while Wireshark was actively capturing traffic, students can now analyse the entire TCP stream. They should observe the typical three-way handshake that initiates a TCP connection: SYN, SYN-ACK, ACK. Following this, a series of PUSH-ACK and ACK packets are exchanged as data is transmitted. If the connection was properly closed, students should also see FIN-ACK packets marking the termination of the session. Additionally, the actual text exchanged during the connection should be visible within the captured TCP packets.

3.1.2. Lab 1

The second laboratory uses the same topology as the first one, except this time the cable is simulated using `wirefilter`. As the ImagiNet topology file is almost identical, we only provide how to modify the cable in Appendix C.2

Wirefilter is used to simulate network perturbations on a wire. The delay can be modified, the noise, the packet drop percentage and much more. To manage all the `wirefilter` options, students must attach to its CLI. To do that, the following ImagiNet command can be used: `imaginet attach conn1`

We assume that the topology, although it is a different lab, is already configured to allow the namespaces to ping each other. If the teacher does not want the students to reconfigure IP addresses on both interfaces, the topology file for ImagiNet can be changed, and IPs can be added for each namespace. In this way, when the lab is started, everything is already configured. A practical example of how this can be achieved is already provided in Appendix B.2.

Students, to start familiarising with `wirefilter`, should ping one namespace from the other and start experimenting with some configuration commands like the `delay` and `dup`. Students can increase the delay with `delay <ms>` and the duplication percentage with `dup <percentage>`. On the output of the `ping` command, something like Listing 16 should be visible.

```bash
user@ns1:~$ ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.256 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.815 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=401 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=401 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=402 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, +1 duplicates, 0% packet loss,
time 3053ms
rtt min/avg/max/mdev = 0.256/201.085/401.927/200.549 ms
```

Listing 16: Example of the `ping` command when using `wirefilter`

3.1.2.1. TCP and UDP on lossy connections

TCP is substantially different from UDP (User Datagram Protocol) as a transport protocol because it ensures that all data arrives at the destination without loss. To demonstrate this difference, two scripts have been provided: one for the client and one for the server, referenced in Appendix C.3 and Appendix C.4. These scripts are designed to simulate a simple communication system: the server listens for TCP or UDP connections, and the client connects to it. The client sends the number `1` every second. Both the client and server maintain a counter that starts at zero and increments each time a `1` is sent or received.

Students should run the server in one namespace and the client in the other. It is recommended to begin with a TCP connection and then test the same setup using UDP. Initially, `wirefilter` has all parameters set to default values, and both client and server counters should remain synchronised. Next, students can use the `loss <percentage>` command in `wirefilter` to introduce artificial packet loss. For TCP, even if the server's counter temporarily stops increasing due to dropped packets, it will eventually catch up and resynchronize with the client once the loss is removed. This behaviour can be observed by introducing a packet loss rate of 50% to 80%, then reducing it back to 0%.

In contrast, for UDP connections, any dropped packets are permanently lost. As a result, the server will not be able to keep its counter in sync with the client's. Even a relatively low packet loss rate of 5% to 10% is enough to demonstrate this behaviour.

3.1.2.2. TCP performance

Another important set of experiments students can perform involves analysing TCP bandwidth in relation to packet loss and network delay between two endpoints. To measure TCP performance, the use of the `iperf3` [57] utility is recommended.

To start an `iperf3` server on one namespace, students should use the `iperf3 -s` command. To start the client from another namespace, students should use the `iperf3 -c <ip>` command. To extend the duration of the test, the `-t <seconds>` option can be added.

Both delay and packet loss significantly impact TCP performance. To observe this, students should start an `iperf3` session and modify these parameters using `wirefilter`. We recommend using small values for delay and loss, as even minor changes can drastically reduce throughput, sometimes down to 0 Bytes/s.

An example of this behaviour is shown in Listing 17, where setting the delay to just 1 ms results in a noticeable performance drop.

```bash
user@ns1:~$ iperf3 -c 10.0.0.1
Connecting to host 10.0.0.1, port 5201
[  5] local 10.0.0.2 port 58648 connected to 10.0.0.1 port 5201
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec   119 MBytes   995 Mbits/sec
[  5]   1.00-2.00   sec   152 MBytes  1.28 Gbits/sec
[  5]   2.00-3.00   sec  81.4 MBytes   682 Mbits/sec
[  5]   3.00-4.00   sec  68.1 MBytes   571 Mbits/sec
[  5]   4.00-5.00   sec  85.2 MBytes   715 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-5.00   sec   506 MBytes   848 Mbits/sec
[  5]   0.00-5.00   sec   504 MBytes   845 Mbits/sec
```

Listing 17: Example of an `iperf3` test when `delay` change on `wirefilter`

3.2. Switching and VLANs

In this section, a new type of network device is introduced: the switch. Within ImagiNet, switch simulation is handled using the VDE switch. A more detailed overview of its command-line interface can be found in Chapter 3.2.1.

The following labs will help students understand how a switch operates, how it differs from a hub, and how to configure and manage VLANs to separate layer 2 broadcast domains.

3.2.1. VDE switch interface

The VDE switch offers a clear, vendor-neutral interface that is perfect for learning how to manage switches and configure VLANs. Unlike vendor-specific interfaces, which involve proprietary command sets and much steeper learning curves, the VDE switch offers a simplified and plain approach, focused more on the core switch functionalities, making it especially well-suited for educational environments.

A subset of commonly used commands is shown in Listing 18. One of the most useful features of the VDE switch is the ability to inspect the internal MAC address table, which reveals how the switch associates MAC addresses with specific ports. This functionality helps clarify how switches learn and forward Ethernet frames as opposed to hubs. The `hash/print` command displays the entire hash table, as reported in Listing 19, while the `hash/find` command can locate the port associated with a specific MAC address.

Managing VLANs is also straightforward. New VLANs are created using the `vlan/create <vid>` command. Ports can be assigned to VLANs in untagged mode with `port/setvlan <port> <vid>`, and to configure tagged VLAN membership, the `vlan/addport <vid> <port>` command is used. VLANs can also be listed and removed through corresponding commands.

Although the command-line interface differs from those found in commercial devices, the VDE switch has all the essential capabilities for managing Layer 2 broadcast domains.

3.2.2. Lab 2

This laboratory is designed to understand how switch operates, how they differ from hubs, and how Ethernet frames are forwarded. The topology file for this laboratory is available at Appendix C.5, and it defines three namespaces that are connected to a single switch.

```
COMMAND PATH       SYNTAX          HELP
------------       --------------  ------------
help               [arg]           Help (limited to arg when
specified)
logout                             logout from this mgmt terminal
shutdown                           shutdown of the switch
showinfo                           show switch version and info
hash               ============    HASH TABLE MENU
hash/showinfo                      show hash info
hash/setexpire     N               change hash entries expire time
hash/print                         print the hash table
hash/find          MAC [VLAN]      MAC lookup
port               ============    PORT STATUS MENU
port/showinfo                      show port info
port/sethub        0/1             1=HUB 0=switch
port/setvlan       N VLAN          set port VLAN (untagged)
port/print         [N]             print the port/endpoint table
port/allprint      [N]             print the port/endpoint table
                                   (including inactive port)
vlan               ============    VLAN MANAGEMENT MENU
vlan/create        N               create the VLAN with tag N
vlan/remove        N               remove the VLAN with tag N
vlan/addport       N PORT          add port to the vlan N
                                   (tagged)
vlan/delport       N PORT          delete port from the vlan N
                                   (tagged)
vlan/print         [N]             print the list of defined vlan
vlan/allprint      [N]             print the list of defined vlan
                                   (including inactive port)
```

Listing 18: Subset of VDE switch commands

```
vde[/tmp/imnet/sw1/mgmt]: hash/print

Hash: 0005 Addr: a6:88:7e:28:22:8e VLAN 00 to port: 02  age 10 secs
Hash: 0019 Addr: 42:d3:fd:ed:a4:a1 VLAN 00 to port: 01  age 14 secs
Hash: 0073 Addr: 3a:9a:66:a1:66:67 VLAN 00 to port: 10  age 08 secs
Success
```

Listing 19: Switch MAC hastable

Students should begin by launching the topology and configuring all namespaces to be on the same IP subnet. Once configured, it should become immediately evident that all namespaces can communicate with each other, demonstrating the basic functionality of the switch.

Next, Wireshark should be started in one of the namespaces. Then, a `ping` command should be issued from the second namespace to the third. Since a switch only forwards broadcast and unknown unicast frames to all ports, Wireshark should display only ARP requests. Neither the ICMP Echo packets nor the ARP replies for the other namespaces should be visible, as they are forwarded directly to the correct destination ports.

Students should then connect to the switch CLI using `imaginet attach sw1` and change the switch behaviour to simulate a hub with the `port/sethub 1` command. In this mode, all frames are forwarded to every port, except the one they were received on. As a result, Wireshark should now display the full ICMP traffic, including Echo requests and replies. Reverting the switch to normal operation can be done using `port/sethub 0`.

At this point, students should examine the switch's MAC address table using `hash/print`. The entries should reflect the correct mapping of MAC addresses to switch ports, based on which port namespaces are attached.

An extended setup involving multiple interconnected switches can also be simulated. When one switch is connected to another, the port linking them will show multiple MAC addresses in its table, corresponding to devices reachable through the remote switch. This demonstrates that a single port can learn and store multiple MAC address associations.

3.2.3. Lab 3

This laboratory is designed to help students understand how VLANs work, how to configure them on a switch, and how packets are encapsulated with the IEEE 802.1Q VLAN tag [66]. The topology used in this lab builds on the previous one, with modifications detailed at Appendix C.6. It defines four namespaces connected to a single switch.

3.2.3.1. VLAN configuration and isolation

Students must configure the switch to isolate the first and second namespaces from the third and fourth. Two VLANs should be chosen (we chose VLAN `10` and VLAN `20` in this example) and assigned as untagged VLANs to the appropriate switch ports (also known as access ports).

The topology file connects the first namespace to the first port of the switch, the second namespace to the second and so on. To create the VLANs on the switch, the `vlan/create <vlan>` command has to be used, while to assign an untagged VLAN to a port, the `port/setvlan <port> <vlan>` command has to be used. Students can verify the current status of VLANs with the `vlan/print` command.

To verify the correct VLAN configuration, namespaces in the same VLAN should be able to communicate, while namespaces in different VLANs should not be able to reach each other. It is highly recommended to run Wireshark on two namespaces from different VLANs to ensure that the traffic is truly isolated and packets differ between the two captures.

3.2.3.2. 802.1Q tag

In the second part of this lab, students will capture packets that include an 802.1Q tag. To dynamically add a new namespace to the running topology, the following command can be used: `imaginet add namespace ns5 -- --iface eth0`. To connect this namespace to port 5 of the switch, the command `imaginet add cable --port-a eth0 --port-b 5 conn5 ns5 sw1` can be used. Alternatively, students may choose to modify the topology file and recreate the full topology, but this requires reconfiguring all namespaces and the switch.

Once the new namespace is connected, students should start it and bring the `eth0` interface into the *UP* state. Then they should configure the switch to send both VLANs as tagged on port number 5 with the `vlan/addport <vlan> <port>` command. Students should now open Wireshark on the new namespace and listen for broadcast packets from other namespaces (these can be manually generated using the `arping` tool). Students should observe that captured packets now include the 802.1Q VLAN tag, and the VLAN ID (VID) in the tag should match the VLANs configured on the switch.

3.2.3.3. More advance scenarios

More advanced scenarios can be designed by introducing multiple switches and configuring trunk ports (ports that carry traffic for multiple tagged VLANs). These setups allow for more complex assignments where students can explore inter-switch VLAN communication and more advanced network segmentation.

Additionally, misconfiguring VLANs can be used to simulate cybersecurity vulnerabilities, such as VLAN hopping attacks. These scenarios provide useful opportunities for students to understand the importance of correct VLAN configurations.

3.3. Routing

This section presents a series of laboratories and exercises designed to help students understand how routing works. Routing is a fundamental aspect of the Internet, enabling packets to traverse multiple and distinct networks to reach their destination.

There are three main types of routing:

- **Connected** routes are automatically created when a router has multiple interfaces, each assigned to a different network. These are the simplest to configure. Students only need to assign the correct IP addresses to the interfaces and enable *ip forwarding* between them.
- **Static** routes are manually defined and specify how to reach a particular network via a specific next-hop router. These routes are often used in controlled environments where network paths do not change frequently.
- **Dynamic** routes are established using dynamic routing protocols such as Border Gateway Protocol (BGP) or Open Shortest Path First (OSPF). These protocols automatically adjust routes based on network topology changes. Dynamic routing requires more advanced networking knowledge, and no labs are included for it.

In the following laboratories, students will learn how to configure connected and static routes, and will also integrate these routing concepts with VLANs covered in the previous labs.

3.3.1. Lab 4

This laboratory is designed to help students understand and become familiar with connected routes. The topology for this lab is shown in Figure 10, and the corresponding ImagiNet file is provided at Appendix C.7. In this setup, the router is implemented as a regular namespace. However, since it performs packet routing between networks, it is represented differently in the diagrams to make its role clearer.

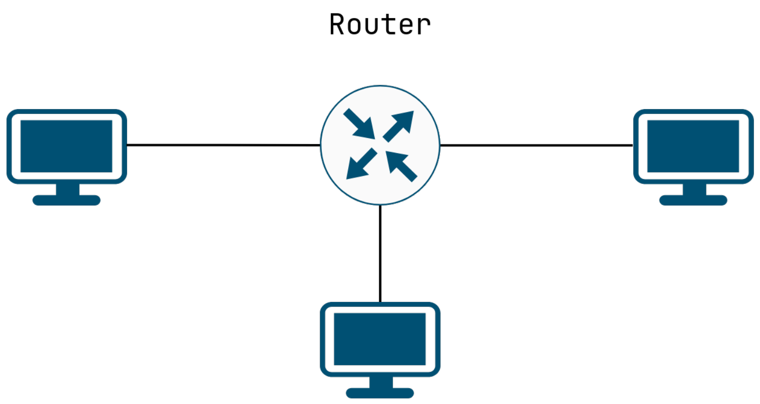

Figure 10: Lab 4 topology

Students should be able to configure each namespace's interface with an IP address in a different subnet and set a default gateway for each one. The default gateway can be configured using the `ip route add default via <ip> dev <interface>` command. For example, if the first namespace belongs to the `10.0.0.0/24` subnet, its gateway could be `10.0.0.254`, which can be configured with `ip route add default via 10.0.0.254 dev eth0`.

On the router namespace, students must assign IP addresses to each interface, using the same IPs previously set as gateways in the corresponding namespaces. Once this configuration is complete, each namespace should be able to ping the router and its interfaces. However, communication between namespaces will still not be possible, because *ip forwarding* is not yet enabled.

To allow packets to be forwarded between interfaces on the router, students must enable IP forwarding using the following command: `sysctl -w net.ipv4.ip_forward=1`. Once IP forwarding is enabled, all namespaces should be able to communicate with each other across the router.

It is highly recommended that students use Wireshark to capture packets both in a namespace and on the router. This will allow them to observe how the Ethernet frame is modified at each hop.

3.3.2. Lab 5

As the number of networks increases, assigning a dedicated physical interface for each network on a router quickly becomes impractical. To address this limitation, routers are often configured to handle multiple VLANs on a single physical interface. VLAN tags are used to keep traffic from different networks logically separated. This setup is commonly known as a "router on a stick" configuration.

In this laboratory, students will learn how a router with a single physical interface can route between multiple networks separated by VLANs. They will practice creating virtual sub-interfaces on a Linux host and configuring connected routes to enable inter-VLAN communication.

The topology used in this lab is illustrated in Figure 11, and the corresponding ImagiNet configuration file is available in Appendix C.8. It consists of three namespaces and a router, all connected to a single switch.

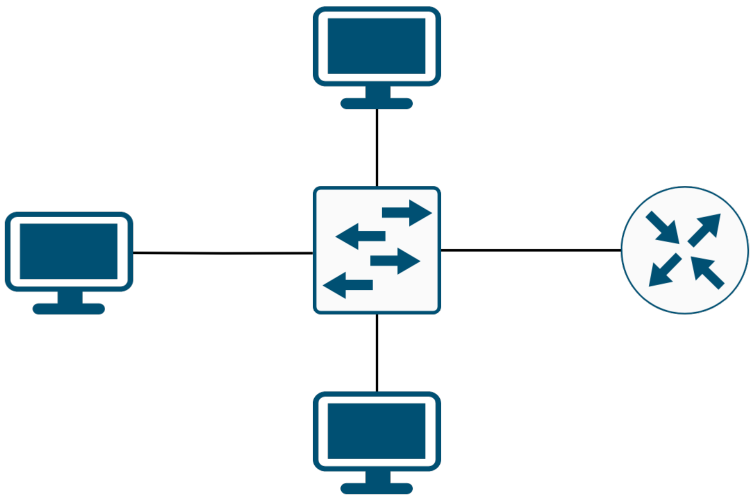
Figure 11: Lab 5 topology

Assuming students have completed and understood the previous labs, they should now be able to configure each namespace with an IP address and a corresponding gateway, placing each namespace in a different subnet. The switch must be configured with three distinct VLANs, assigning each namespace port as untagged and tagging all VLANs on the router's port.

To enable routing between VLANs over a single physical interface, students must create a virtual interface for each VLAN. This is done using the `ip link add link <interface> name <new interface> type vlan id <vid>` command. The `<interface>` parameter is the name of the interface on which the virtual interface will operate, while the `<new interface>` field is the name of the new interface. For example, to create a virtual interface named `eth0.10` for the VLAN `10` on physical interface `eht0`, the `ip link add link eth0.10 name eth0 type vlan id 10` command can be used. After creation, each virtual interface can be treated like a physical one. Since the configuration of connected routes has already been covered in previous labs, students should now be able to configure the router to enable communication between the namespaces.

As in previous labs, students are encouraged to capture packets using Wireshark on both the physical interface and the virtual interfaces of the router. They should observe that the physical interface receives packets tagged with the 802.1Q VLAN header, while the virtual interfaces receive only the packets belonging to their respective VLAN, with the VLAN tag already stripped by the kernel.

3.3.3. Lab 6

In this laboratory, students will learn what static routes are and how to configure them on a Linux host. The topology used in this lab is illustrated in Figure 12, and the corresponding ImagiNet configuration file is provided at Appendix C.9.

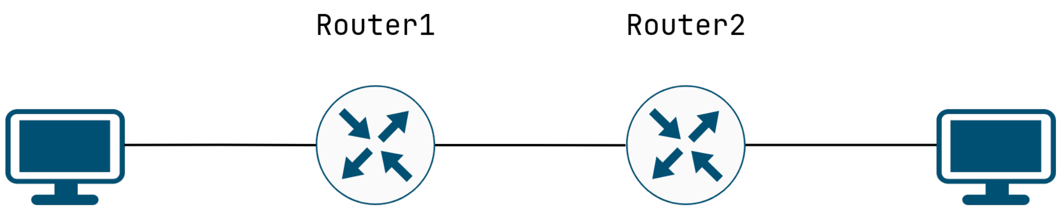

Figure 12: Lab 6 topology

At the end of this lab, students should be able to configure the network in a way that allows the namespaces to communicate with each other. The first task is to assign each namespace an IP address in a different subnet and configure the correct default gateway. These gateway IPs must correspond to the IPs assigned to the router interfaces connected to the namespaces.

Next, students need to configure the routers themselves. Each router interface should be placed in a distinct subnet, including the interface that connects the two routers together. With this configuration, each namespace should be able to reach its own gateway, but communication between the two namespaces will not yet be possible. This is because, when a packet arrives at the router, the router has no route indicating how to reach the destination network that sits behind the other router.

To enable full communication between the namespaces, static routes must be added to the routers. Each router needs to be told how to reach the subnet connected to the other one. This can be done using the `ip route add <subnet> via <next-hop> dev <interface>` command. For instance, if `Router2` has the IP address `172.16.0.2`, and the second namespace is in the `192.168.0.0/24` network, then `Router1` can be configured with the following command: `ip route add 192.168.0.0/24 via 172.16.0.2 dev eth1`.

To inspect the current routing table on a machine, students can use the `ip route` command. For verification and debugging purposes, the `traceroute` utility is useful, as it displays the path taken by packets to reach their destination. This helps confirm whether routing has been set up correctly or identify where in the network the packet path is failing. An example of a `traceroute` output is provided in Listing 20.

```
user@ns1 ~$ traceroute 192.168.0.1
traceroute to 192.168.0.1 (192.168.0.1), 30 hops max, 60 byte
packets
 1  _gateway (10.0.0.254)  0.716 ms  0.688 ms  0.675 ms
 2  172.16.0.2 (172.16.0.2)  1.358 ms  1.356 ms  1.353 ms
 3  192.168.0.1 (192.168.0.1)  2.228 ms  2.266 ms  2.258 ms
```

Listing 20: `traceroute` example

Students are encouraged to capture and analyse packets on both routers while the two namespaces communicate. This allows them to observe how packets are forwarded across different networks and how the Ethernet and IP headers are modified at each hop.

3.4. Firewalls

Firewalls are network devices that control the flow of traffic based on a set of predefined rules. These rules determine whether traffic should be allowed, dropped, or explicitly rejected. In addition to traffic filtering, firewalls often handle Network Address Translation (NAT), enabling communication between private and public networks.

While managing a firewall is typically a more advanced topic covered in system and network administration courses, a basic introduction to key firewall operations can still be valuable in more introductory settings. This lab is designed to provide a simple overview of essential concepts and commands related to firewall management.

In this lab, firewalls are simulated using Linux namespaces, which support full use of the `iptables` utility.

3.4.1. Lab 7

In the following laboratory, students will explore how Network Address Translation (NAT) functions and how to configure Destination NAT, also known as *port forwarding*. This concept is used for enabling access to internal services from external networks.

The network topology for this lab is shown in Figure 13, and the corresponding ImagiNet configuration file is available at Appendix C.10. The provided configuration is designed to let students concentrate specifically on the firewall and NAT aspects, as the rest of the topology is already set up. However, for those who wish to configure everything from scratch, it is entirely possible to remove the pre-configured settings and manually configure the entire network, including IP addresses, routes, and interfaces.

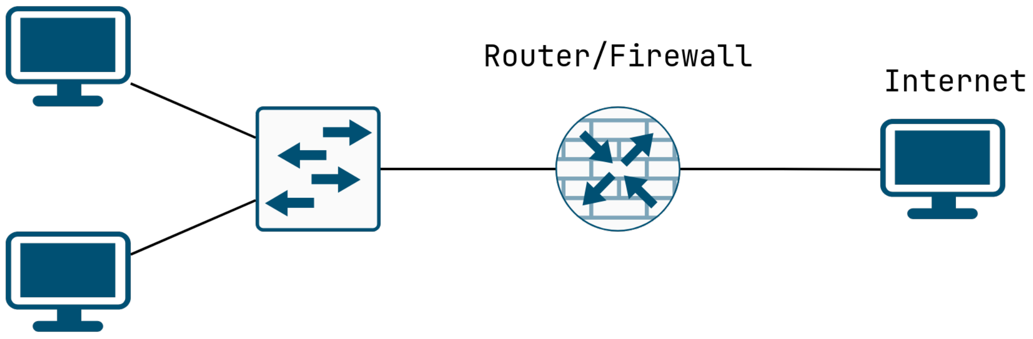

Figure 13: Lab 7 topology

The first part of this laboratory introduces a setup where a router also functions as a firewall, using NAT to allow two namespaces to communicate as they are part of a private network. While in medium-larger infrastructures firewalls are often standalone devices, in small networks it is common to combine routing and firewall functionality in a single system to simplify management and enforce internal traffic policies.

In the second part of this lab, students will configure Destination NAT to enable external hosts on the Internet to access one of the internal namespaces through a specific port. This setup is common in real-world scenarios where multiple internal servers with private IP addresses need to be reachable from the outside.

To configure packet filtering and NAT rules on Linux systems, several user-space tools are available. This lab uses the `iptables` utility, whose documentation is available on the official Netfilter website [67]. While `iptables` remains widely used, students may also explore more modern alternatives such as `nftables`, also maintained by Netfilter. For managing complex rule sets, especially as they grow in size and complexity, tools like Shorewall [68] are recommended to simplify firewall configuration and maintenance.

3.4.1.1. Source NAT

After creating and starting the topology, students should begin by configuring the basic policies for the INPUT, OUTPUT, and FORWARD chains using the `iptables` utility. These chains determine how the firewall handles packets that are received by the host, sent from the host, or forwarded through the host, respectively. To list the current set of rules, the command `iptables -L` can be used. The output should be similar to Listing 21. In this case, the three chains are all empty, end their policies are all ACCEPT, meaning that all traffic is currently allowed.

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
```

Listing 21: `iptables` chain example

A basic firewall that is situated at the boundary of the Internet should implement a deny-by-default policy. This means setting the default policies of at least the INPUT and FORWARD chains to DROP, so that only explicitly allowed traffic is permitted. To set a default policy to a chain, the `iptables -P <chain> <policy>` command can be used. In this example, to DROP everything by default in the INPUT and FORWARD chains, the commands `iptables -P INPUT DROP` and `iptables -P FORWARD DROP` can be used. Students can verify the success of these commands by using the `iptables -L` command.

Before changing the default policies, the router could communicate freely with both the internal namespaces and the Internet. However, at this stage, all communication should stop, except between the two namespaces connected via the switch. This behaviour might seem surprising. For example, if a student tries to ping the Internet namespace from the router, the packet is still sent because the OUTPUT chain policy remains set to ACCEPT. Wireshark would even show the response returning to the router. However, the response is dropped by the router itself because the INPUT chain now rejects all incoming packets, even those that are part of established connections. To allow returning traffic from already existing or new but related connections, students should add a rule to accept packets in the ESTABLISHED or RELATED states. This can be done with the following command[7]:

```
iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED \
    -j ACCEPT
```

After adding the rule to accept ESTABLISHED and RELATED connections, the router is able to receive replies from other devices in the network. However, students should notice that while the router is able to ping the Internet, the Internet namespace is not able to reach the router if communication starts from it. This behaviour is consistent with the current firewall configuration, as only the returned traffic from previously initiated connections is allowed. If a packet belongs to a new connection that has not originated from the router itself is dropped.

To allow internal namespaces to reach the Internet through the router, the FORWARD chain must be configured. This chain filters packets that traverse the router but are neither originated from nor destined to the router itself.

To permit forwarding of packets that arrive on the `eth0` interface (which is connected to the internal network) and leave via the `eth1` interface (the external/Internet network), the following rule should be added:

```
iptables -A FORWARD -i eth0 -o eth1 -j ACCEPT
```

With this rule, students will observe that packets, such as ICMP echo requests, can leave the internal namespaces, pass through the router, and reach the Internet namespace. However, the reply packets will not be generated. This is because the Internet namespace does not have a route back to the internal subnet. The default behaviour for Linux is to discard traffic that has an unknown destination. To test and confirm this, students can manually add a static route on the Internet namespace with the `ip route add 10.0.0.0/24 via 90.80.70.1 dev eth0` command. The Internet namespace will now attempt to return packets to the internal network. However, the firewall still blocks these return packets unless another rule is added to the FORWARD chain to permit return traffic for established connections. This can be done with:

```
iptables -A FORWARD -i eth1 -o eth0 -m conntrack \
```

[7]The backslash (\) in the command indicates a line continuation, but in practice it should be written on a single line.

```
    --ctstate RELATED,ESTABLISHED -j ACCEPT
```

At the current state of the firewall, replies from the Internet to the internal namespaces will be allowed, and full two-way communication that originates from the namespaces should be possible.

At this point in the lab, students should notice that the internal namespaces are able to send packets to the Internet, but the packets that arrive on the Internet namespaces have a private source address corresponding the the namespaces' subnet. As private IPs are not routable on the Internet, students should now configure the NAT.

In this case, NAT modifies packets that exit the external interface of the router and changes the source IPs to the router's own public IP, making the communication with the Internet correct and possible.

If previously a static route was configured on the Internet namespace, it should be removed with the `ip r del 10.0.0.0/24 via 90.80.70.1 dev eth0` command. Then, the router needs to *masquerade* the outgoing packets from the internal network when they are forwarded to the Internet. This is done by adding a rule to the *nat* table in the POSTROUTING chain:

```
iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE
```

Now everything is correctly configured, and namespaces should be able to ping the Internet namespace. The ICMP packets arriving on it should have the source address equal to the router's public address, instead of the namespace that generated the connection.

3.4.1.2. Destination NAT

In the possibility that a service running on the port `3000` on the first namespace that has the `10.0.0.1` IP address configured must be reachable from the Internet, a Destination NAT rule has to be added. This is also known as *port forwarding*: every connection to the router's public IP at that specific port will be redirected to the configured namespace. The following command can be used to add the destination NAT rule:

```
iptables -t nat -A PREROUTING -i eth1 -p tcp --dport 3000 \
    --to 10.0.0.1:3000 -j DNAT
```

At this point, every TCP connection that arrives on the `3000` ports has the destination IP rewritten as `10.0.0.1`. But, since the FORWARDING table is now responsible for letting packets pass through the router, a new rule must be added to allow external connections for the `3000` port. This can be done with the following command:

```
iptables -A FORWARD -i eth1 -o eth0 -p tcp -m tcp --dport 3000 -j
ACCEPT
```

Students should now verify that everything is configured properly with the *netcat* utility by listening for connections on the configured namespace while connecting from the Internet.

3.5. Advance scenarios

The laboratories presented so far highlight the networking aspects of each topic while using a minimal set of administration tools for learning and experimentation. This approach works well for computer network courses but may be too basic for network administration courses. ImagiNet can also support advanced network administration courses where multiple Linux hosts function as routers, firewalls and servers. While we have chosen not to include more complex scenarios, teachers who see potential value in using ImagiNet for their courses should experiment with custom topologies customised to their specific course needs.

3.5.1. Group laboratories

The final laboratory proposed differs from previous exercises by requiring students to work in groups rather than individually. This laboratory assumes students can work on computers connected to the same network. This can be accomplished by conducting the laboratory in a physical university computer lab where all machines share the same network connection. Alternatively, Virtual Private Networks (VPNs) can be used to connect student computers as if they were on the same local network. Instructors should choose the approach that best fits their requirements.

ImagiNet can use VXVDE (Chapter 2.1.3.2) to create a virtual switch that operates over a physical network. This capability allows multiple computers on the network to run ImagiNet and connect virtualised namespaces from different physical hosts to each other. Using this feature, the following laboratory creates a small topology for each student where a router is connected through VXVDE to a virtual switch that also connects other students' routers. The router portion of this topology is shown in Figure 14.

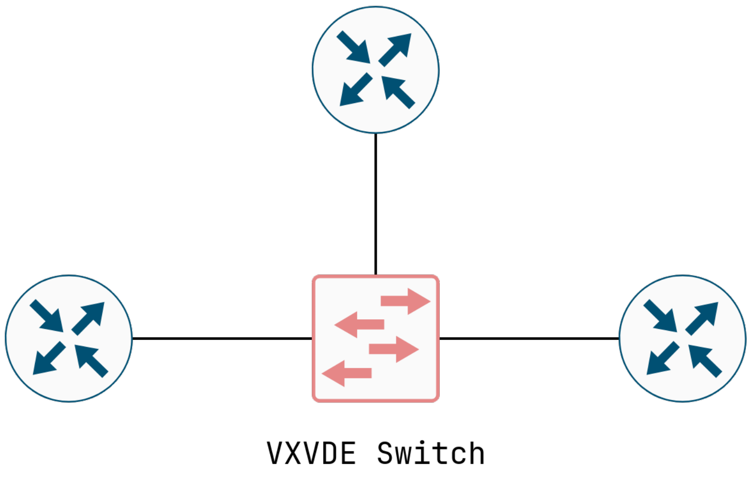

Figure 14: Router topology for the VXVDE laboratory

Each student's router can be connected to a local switch where multiple VLANs are configured for multiple namespaces. Students must first correctly configure the internal routing within their own topology. Then they need to collaborate with other students to decide which IP addresses are assigned to each router and determine what routes can be reached from each one. All the previously presented laboratories should provide the necessary knowledge to administer this topology. Students must now work together to troubleshoot problems and misconfigurations from both ends of the network. A firewall configuration can also be added to each router to prevent unauthorised traffic from other students. A partial example of an ImagiNet file for this topology can be found at Appendix C.11.

3.6. Teaching with ImagiNet

During its development, ImagiNet was used to teach network topics to a small group of students who participated in system administration laboratories. Some of the topologies and labs presented earlier are inspired by the topologies actually used to teach those topics. During these laboratories, students were actively supported by more advanced and experienced students who helped them better understand what they were simulating.

Overall, the experience was extremely positive as all students successfully learned the laboratory topics. Some students even continued working on more advanced exercises during non-university hours.

We believe that ImagiNet is an excellent tool for teaching and, for a small group of students, it has proven extremely valuable. While no actual laboratories with a large number of students have been tested, we are convinced this would be entirely possible and would bring significant value alongside theoretical lectures.

Conclusions

4.1. Improvements

In this section, several potential improvements are suggested. Since the project is released under the AGPL-3.0 License [69], we welcome users to freely share and modify the software to meet their specific needs. We would be glad to incorporate significant improvements into the official project repository.

4.1.1. IaC differential updates

As shown in Chapter 2.2, ImagiNet uses a declarative Infrastructure as Code (IaC) approach to define topologies. Given a topology file, ImagiNet creates the topology using the `create` command. After the topology is created, users must use the `add` and `rm` commands to modify it dynamically without stopping and deleting the running devices.

Modern IaC tools often allow users to reapply a configuration file that has already been applied. The tool then checks the differences with the current application state and applies only the changes, leaving unmodified components untouched.

ImagiNet could adopt a similar approach, allowing users to modify the original topology file and apply it to the current running topology. ImagiNet would calculate the differences and apply only the necessary changes, eliminating the need to stop and clear the current application state.

4.1.2. Missing VDE features and optimisations

Although ImagiNet covers many VDE features, it does not cover them all. For the educational purposes that were the primary focus during development, this is not problematic, but ImagiNet could also be considered a tool for configuring VDE scenarios where additional features are needed.

vdeplug4 (Chapter 2.1.3.2) provides a more lightweight implementation of switches and hubs compared to the current implementation based on *VDE-2* (Chapter 2.1.3.1). The main difference is that the *vdeplug4* switch is unmanaged, so it cannot be configured with VLANs or other parameters. For topologies that do not require managed switches, a flag could be implemented to distinguish between managed and unmanaged switches, allowing ImagiNet to choose the appropriate implementation.

ImagiNet currently includes a flag to convert a switch to a hub, but it still uses the *VDE-2* implementation. It would be possible to use *vdeplug4* instead, since a hub does not need a management interface, except for educational purposes, where dynamically switching between the two device types helps demonstrate differences in frame forwarding.

4.1.3. Didattic router and firewall

Network administration courses can benefit significantly from hands-on routing experience and Linux firewall usage. However, for network courses focused primarily on theoretical topics, firewalls can present a complex interface and command set that may overwhelm students at first.

The VDE project does not need a router and a firewall, but for educational purposes, it would be possible to develop custom user programs that function as simplified routers and firewalls. These programs would feature a reduced command set to help students during laboratories, eliminating the need to learn complex management tools and instead focusing on core networking concepts.

This approach would be particularly valuable for firewall components, where `iptables` or `nftables` are typically involved. A program with a simple interface, similar to the VDE switch, that can act as a transparent firewall would help students experiment with basic packet filtering rules and NAT configurations. These tools would help students connect theory with real-world practice by making it easier to understand fundamental concepts without being hindered by complicated syntax or setup.

4.1.4. Software packaging

Software packaging refers to the process of bundling software and its associated components into a distributable format that can be easily installed and managed on various computer systems. Packaging enables tracking of installed files and automatic software updates while providing users with a straightforward way to install programs.

Currently, to install ImagiNet, users must download and compile the source code themselves after resolving all necessary dependencies. This process can be time-consuming and may present barriers for users who are less familiar with compilation procedures and dependency resolution. It would be beneficial if ImagiNet were packaged for the major package managers of GNU/Linux distributions. This would allow users to install the software easily and resolve all dependencies automatically, significantly reducing the setup time and making ImagiNet more accessible to educators and students who want to focus on learning rather than software installation labours.

4.2. Final thoughts

In conclusions, this thesis has explored the design and implementation of a network simulator based on Virtual Distributed Ethernet (VDE). Although this project began with a different scope, it quickly became clear that ImagiNet would serve as an excellent simulation tool for educational purposes. From this moment of realization, every design choice was made toward creating a tool that is both powerful and simple enough to be used in university computer networking courses.

ImagiNet is not the first network simulator used as a teaching tool, and it certainly will not be the last. Multiple network simulation softwares exist, each with different features and characteristics. The overview in Chapter 1.1 should help readers understand how ImagiNet positions itself among other simulators, particularly when considering its implementation covered in Chapter 2. After reading Chapter 3, readers should have a wider and clearer understanding of the educational possibilities that ImagiNet offers.

This thesis presents only a small portion of all the possible network simulation topologies that are possible with ImagiNet. The simulation possibilities are nearly limitless, which can help teachers customize and adapt this tool to their specific course requirements. The flexibility of ImagiNet allows educators to create scenarios that precisely match their curriculum objectives, from basic networking concepts to more advanced network administration tasks.

ImagiNet has achieved its intended purpose and has proven successful in teaching a small group of students. While it has not yet been utilised in large courses involving hundreds of students, we believe this would be entirely feasible and beneficial. The architecture and design principles of ImagiNet support scalability, and we encourage educators to experiment with larger class sizes. Such accomplishments would provide valuable insights into the tool's effectiveness at scale and could further validate its potential as a comprehensive educational platform for network learning.

Appendices

A Topology generator

```python
from dataclasses import dataclass, field, asdict
from typing import List, Optional
import yaml

NS_PER_SW = 4
SW_PER_GROUP = 4
GROUPS = 4

@dataclass
class NSInterface:
    name: str
    ip: Optional[str] = None
    gateway: Optional[str] = None

@dataclass
class Namespace:
    name: str
    interfaces: List[NSInterface] = field(default_factory=list)

@dataclass
class Switch:
    name: str
    ports: int = 32

@dataclass
class Endpoint:
    name: str
    port: Optional[str] = None

@dataclass
class Cable:
    name: str
    endpoint_a: Endpoint
    endpoint_b: Endpoint
    wirefilter: Optional[bool] = None

@dataclass
class Topology:
    switch: List[Switch] = field(default_factory=list)
    namespace: List[Namespace] = field(default_factory=list)
    cable: List[Cable] = field(default_factory=list)

def generate_namespaces(g):
    nss = []
    for sw in range(SW_PER_GROUP):
        for ns in range(NS_PER_SW):
            name = f"ns{g:03}{sw:03}{ns:03}"
            ns = Namespace(name=name,interfaces=[NSInterface(
                    name="eth0",
                    ip=f"10.{sw}.{ns//256}.{ns%253 + 1}/16",
                    gateway=f"10.{sw}.{ns//256}.254")])
            nss.append(ns)

    return nss

def generate_switches(g):
    sws = []
    for sw in range(SW_PER_GROUP):
        name = f"sw{g:03}{sw:03}"
        sw = Switch(name=name, ports=max(32, NS_PER_SW + 1))
        sws.append(sw)
```

```python
    return sws


def generate_router(g):
    name = f"r{g:03}"
    router = Namespace(name=name)
    interfaces = []
    for i in range(SW_PER_GROUP):
        interfaces.append(NSInterface(name=f"int{i}",
                                      ip=f"10.{g}.{i}.254/16"))
    for i in range(GROUPS):
        if i == g:
            continue
        interfaces.append(NSInterface( name=f"ext{i}",))

    router.interfaces = interfaces
    return router


def generate_cables(g):
    cables = []
    for sw in range(SW_PER_GROUP):
        for ns in range(NS_PER_SW):
            name = f"c{g:03}{sw:03}{ns:03}"
            endpoint_a = Endpoint(name=f"ns{g:03}{sw:03}{ns:03}",
                                  port="eth0")
            endpoint_b = Endpoint(name=f"sw{g:03}{sw:03}")
            cable = Cable(name=name, endpoint_a=endpoint_a,
                          endpoint_b=endpoint_b)
            cables.append(cable)

        name = f"ci{g:03}{sw:03}"
        endpoint_a = Endpoint(name=f"sw{g:03}{sw:03}")
        endpoint_b = Endpoint(name=f"r{g:03}", port=f"int{sw}")
        cable = Cable(name=name, endpoint_a=endpoint_a,
        endpoint_b=endpoint_b)
        cables.append(cable)

    return cables
```

```python
def generate_router_cables(routers):
    cables = []
    for g in range(GROUPS):
        for i in range(GROUPS):
            if i <= g:
                continue
            name = f"cr{g:03}{i:03}"
            endpoint_a = Endpoint(name=f"r{g:03}", port=f"ext{i}")
            endpoint_b = Endpoint(name=f"r{i:03}", port=f"ext{g}")
            cable = Cable(name=name, endpoint_a=endpoint_a,
            endpoint_b=endpoint_b, wirefilter=False)
            cables.append(cable)

    return cables

def main():
    nss, sws, cs, routers = [], [], [], []
    for g in range(GROUPS):
        nss = nss + generate_namespaces(g)
        sws = sws + generate_switches(g)
        routers.append(generate_router(g))
        cs = cs + generate_cables(g)

    rcables = generate_router_cables(routers)

    topology = Topology(
        namespace = nss + routers,
        switch = sws,
        cable = cs + rcables,
    )

    yaml_str = yaml.dump(asdict(topology), sort_keys=False,
                         default_flow_style=False)
    print(yaml_str)

if __name__ == "__main__":
    main()
```

B Topology creation

B.1 Topology 0 efficient

```bash
#!/bin/bash
ip l set vde0 up
ip a a 10.0.0.1/24 dev vde0
bash
```

Listing 22: Script `/tmp/t0-ns0.sh`

```bash
#!/bin/bash
ip l set vde0 up
ip a a 10.0.0.2/24 dev vde0
bash
```

Listing 23: Script `/tmp/t0-ns1.sh`

```bash
# Terminal 0
vdens ptp:///tmp/t0 /tmp/t0-ns0.sh
# Terminal 1
vdens ptp:///tmp/t0 /tmp/t0-ns1.sh
```

Listing 24: Commands to start the topology

B.2 Topology 0 ImagiNet

```Yaml
namespace:
  - name: ns1
    interfaces:
      - name: eth0
        ip: 10.0.0.1/24

  - name: ns2
    interfaces:
      - name: eth0
        ip: 10.0.0.2/24
cable:
  - name: conn1
    endpoint_a:
      name: ns1
      port: eth0
    endpoint_b:
      name: ns2
      port: eth0
```

B.3 Topology 1 efficient

```bash
#!/bin/bash
ip a a 10.0.0.1/24 dev vde0
ip l set vde0 up
ip r a default via 10.0.0.254 dev vde0
bash
```

Listing 25: Script `/tmp/t1-ns0.sh`

```bash
#!/bin/bash
ip a a 10.0.0.254/24 dev vde0
ip l set vde0 up
ip a a 172.16.0.1/24 dev vde1
ip l set vde1 up
/usr/sbin/sysctl -w net.ipv4.ip_forward=1
ip r a 192.168.0.0/24 via 172.16.0.2 dev vde1
bash
```

Listing 26: Script `/tmp/t1-ns1.sh`

```bash
#!/bin/bash
ip a a 172.16.0.2/24 dev vde0
ip l set vde0 up
ip a a 192.168.0.254/24 dev vde1
ip l set vde1 up
/usr/sbin/sysctl -w net.ipv4.ip_forward=1
ip r a 10.0.0.0/24 via 172.16.0.1 dev vde0
bash
```

Listing 27: Script `/tmp/t1-ns2.sh`

```bash
#!/bin/bash
ip a a 192.168.0.1/24 dev vde0
ip l set vde0 up
ip r a default via 192.168.0.254 dev vde0
bash
```

Listing 28: Script `/tmp/t1-ns3.sh`

```bash
# Terminal 0
vdens ptp:///tmp/t0 /tmp/t1-ns0.sh
# Terminal 1
vdens --multi ptp:///tmp/t0 ptp:///tmp/t1 -- /tmp/t1-ns1.sh
# Terminal 2
vdens --multi ptp:///tmp/t1 ptp:///tmp/t2 -- /tmp/t1-ns2.sh
# Terminal 3
vdens ptp:///tmp/t2 /tmp/t1-ns3.sh
```

Listing 29: Commands to start the topology

B.4 Topology 1 ImagiNet

```yaml
namespace:
  - name: ns1
    interfaces:
      - name: eth0
        ip: 10.0.0.1/24
        gateway: 10.0.0.254

  - name: ns2
    interfaces:
      - name: eth0
        ip: 10.0.0.254/24
      - name: eth1
        ip: 172.16.0.1/24
    commands:
      - ip r a 192.168.0.0/24 via 172.16.0.2 dev eth1
      - /usr/sbin/sysctl -w net.ipv4.ip_forward=1

  - name: ns3
    interfaces:
      - name: eth0
        ip: 172.16.0.2/24
      - name: eth1
        ip: 192.168.0.254/24
    commands:
      - ip r a 10.0.0.0/24 via 172.16.0.1 dev eth0
      - /usr/sbin/sysctl -w net.ipv4.ip_forward=1

  - name: ns4
```

```yaml
    interfaces:
      - name: eth0
        ip: 192.168.0.1/24
        gateway: 192.168.0.254

cable:
  - name: conn1
    endpoint_a:
      name: ns1
      port: eth0
    endpoint_b:
      name: ns2
      port: eth0

  - name: conn2
    endpoint_a:
      name: ns2
      port: eth1
    endpoint_b:
      name: ns3
      port: eth0

  - name: conn3
    endpoint_a:
      name: ns3
      port: eth1
    endpoint_b:
      name: ns4
      port: eth0
```

B.5 Topology 2 efficient

The `/tmp/t2-ns0.sh` script is the same as `/tmp/t1-ns0.sh` in Listing 25.

The `/tmp/t2-ns1.sh` script is the same as `/tmp/t1-ns3.sh` in Listing 28.

```bash
#!/bin/bash
ip a a 10.0.0.254/24 dev vde0
ip l set vde0 up
ip a a 192.168.0.254/24 dev vde1
ip l set vde1 up
/usr/sbin/sysctl -w net.ipv4.ip_forward=1
bash
```

Listing 30: Script `/tmp/t2-ns2.sh`

```
vlan/create 10
vlan/create 20
port/create 1
port/create 2
port/create 3
port/create 4

port/setvlan 1 10
port/setvlan 2 20
port/setvlan 3 10
port/setvlan 4 20
```

Listing 31: The switch configuration file `/tmp/t2-sw.conf`

```bash
# Terminal 0
vde_switch -s /tmp/sw0 -d
vde_switch -s /tmp/sw1 -d
vde_switch -s /tmp/sw2 -f /tmp/t2-sw.conf -d
vdens vde:///tmp/sw0 /tmp/t2-ns0.sh
# Terminal 1
vdens vde:///tmp/sw1 /tmp/t2-ns1.sh
# Terminal 2
vdens --multi 'vde:///tmp/sw2[3]' 'vde:///tmp/sw2[4]' -- \
  /tmp/t2-ns2.sh
# Terminal 3
vde_plug 'vde:///tmp/sw0' 'vde:///tmp/sw2[1]'
# Terminal 4
vde_plug 'vde:///tmp/sw1' 'vde:///tmp/sw2[2]'
```

Listing 32: Commands to start the topology

B.6 Topology 2 ImagiNet

The switch configuration file `/tmp/t2-sw.conf` is in Listing 31.

```yaml
switch:
  - name: sw0
  - name: sw1
  - name: sw2
    config: /tmp/t2-sw.conf

namespace:
  - name: ns0
    interfaces:
      - name: eth0
        ip: 10.0.0.1/24
        gateway: 10.0.0.254

  - name: ns1
    interfaces:
      - name: eth0
        ip: 192.168.0.1/24
        gateway: 192.168.0.254

  - name: ns2
    interfaces:
      - name: eth0
        ip: 10.0.0.254/24

      - name: eth1
        ip: 192.168.0.254/24
    commands:
      - /usr/sbin/sysctl -w net.ipv4.ip_forward=1

cable:
  - name: conn0
    endpoint_a:
      name: ns0
      port: eth0
    endpoint_b:
      name: sw0

  - name: conn1
    endpoint_a:
      name: ns1
      port: eth0
    endpoint_b:
      name: sw1

  - name: conn2
    endpoint_a:
      name: sw0
    endpoint_b:
      name: sw2
      port: "1"

  - name: conn3
    endpoint_a:
      name: sw1
    endpoint_b:
      name: sw2
      port: "2"

  - name: conn4
    endpoint_a:
      name: ns2
      port: eth0
    endpoint_b:
      name: sw2
      port: "3"

  - name: conn5
    endpoint_a:
      name: ns2
      port: eth1
    endpoint_b:
      name: sw2
      port: "4"
```

B.7 Topology 3 ImagiNet

The switch configuraton file `t3-sw.conf` is the same as the `t2-sw.conf` configuration file in Listing 31.

```yaml
switch:
  - name: sw0
```

```
    config: t3-sw.conf
  - name: sw1
    config: t3-sw.conf
  - name: sw2

namespace:
  - name: ns0
    interfaces:
      - name: eth0
        ip: 10.0.0.1/24
        gateway: 10.0.0.254

  - name: ns1
    interfaces:
      - name: eth0
        ip: 10.0.1.1/24
        gateway: 10.0.1.254

  - name: r0
    interfaces:
      - name: eth0
        ip: 10.0.0.254/24
      - name: eth1
        ip: 10.0.1.254/24
      - name: eth2
        ip: 172.16.0.1/30
    commands:
      - /usr/sbin/sysctl -w net.ipv4.ip_forward=1
      - ip r a 192.168.0.0/24 via 172.16.0.2 dev eth2
      - ip r a 192.168.1.0/24 via 172.16.0.2 dev eth2

  - name: r1
    interfaces:
      - name: eth0
        ip: 192.168.0.254/24
      - name: eth1
        ip: 192.168.1.254/24
      - name: eth2
        ip: 172.16.0.2/30
    commands:
      - /usr/sbin/sysctl -w net.ipv4.ip_forward=1
      - ip r a 10.0.0.0/24 via 172.16.0.1 dev eth2
      - ip r a 10.0.1.0/24 via 172.16.0.1 dev eth2

  - name: ns2
    interfaces:
      - name: eth0
        ip: 192.168.0.1/24
        gateway: 192.168.0.254

  - name: ns3
    interfaces:
      - name: eth0
        ip: 192.168.1.1/24
        gateway: 192.168.1.254

cable:
  - name: conn0
    endpoint_a:
      name: ns0
      port: eth0
    endpoint_b:
      name: sw0
      port: "1"

  - name: conn1
    endpoint_a:
      name: ns1
      port: eth0
    endpoint_b:
      name: sw0
      port: "2"

  - name: conn2
    endpoint_a:
      name: sw0
      port: "10"
    endpoint_b:
      name: r0
      port: eth0
```

```
- name: conn3
  endpoint_a:
    name: sw0
    port: "11"
  endpoint_b:
    name: r0
    port: eth1

- name: conn4
  endpoint_a:
    name: r0
    port: eth2
  endpoint_b:
    name: sw2

- name: conn5
  endpoint_a:
    name: r1
    port: eth2
  endpoint_b:
    name: sw2

- name: conn6
  endpoint_a:
    name: r1
    port: eth0
  endpoint_b:
    name: sw1
    port: "10"

- name: conn7
  endpoint_a:
    name: r1
    port: eth1
  endpoint_b:
    name: sw1
    port: "11"

- name: conn8
```

```
    endpoint_a:
      name: ns2
      port: eth0
    endpoint_b:
      name: sw1
      port: "1"

  - name: conn9
    endpoint_a:
      name: ns3
      port: eth0
    endpoint_b:
      name: sw1
      port: "2"
```

C Laboratories

C.1 Lab 0

```yaml
namespace:
  - name: ns1
    interfaces:
      - name: eth0
  - name: ns2
    interfaces:
      - name: eth0
cable:
  - name: conn1
    endpoint_a:
      name: ns1
      port: eth0
    endpoint_b:
      name: ns2
      port: eth0
```

C.2 Lab 1

The topology is identical to the one defined for Lab 0 in Appendix C.1. The only difference is the cable that has `wirefilter` enabled.

```yaml
cable:
  - name: conn1
    endpoint_a:
      name: ns1
      port: eth0
    endpoint_b:
      name: ns2
      port: eth0
    wirefilter: true
```

C.3 TCP/UDP client script

```python
import socket
import time

def start_client(protocol, ip, port):
    count = 0
    proto = protocol.lower()
    match proto:
        case 'tcp':
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        case 'udp':
            s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        case _:
            print("Unsupported protocol. Use TCP or UDP.")
            exit(1)

    s.connect((ip, port))
    print(f"Connected to {proto} server at {ip}:{port}")
    while True:
        time.sleep(1)
        count += 1
        print(f"Count: {count}")
        s.send("1\n".encode())
```

```python
if __name__ == "__main__":
    proto = input("Enter protocol (TCP/UDP): ").strip()
    ip = input("Enter server IP address: ").strip()
    port = int(input("Enter server port number: ").strip())
    start_client(proto, ip, port)
```

C.4 TCP/UDP server script

```python
import socket

def start_server(protocol, port):
    count = 0
    proto = protocol.lower()
    # Create socket based on protocol
    if proto == 'tcp':
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.bind(('0.0.0.0', port))
        s.listen(1)
        print(f"TCP server listening on port {port}")
        conn, addr = s.accept()
        print(f"Connected by {addr}")
        recv_func = lambda: conn.recv(1024)
    elif proto == 'udp':
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.bind(('0.0.0.0', port))
        print(f"UDP server listening on port {port}")
        recv_func = lambda: s.recvfrom(1024)[0]
    else:
        print("Unsupported protocol. Use TCP or UDP.")
        exit(1)

    while True:
        data = recv_func()
        if not data and proto == 'tcp':  # TCP connection closed
            break
        # Count occurrences of '1' in the received data
        ones_received = data.decode().count('1')
        if ones_received > 0:
            count += ones_received
            print(f"Received {ones_received} ones, Total: {count}")
    if proto == 'tcp':
        conn.close()

if __name__ == "__main__":
    proto = input("Enter protocol (TCP/UDP): ").strip()
    port = int(input("Enter port number: ").strip())
    start_server(proto, port)
```

C.5 Lab 2

```yaml
switch:
  - name: sw1

namespace:
  - name: ns1
    interfaces:
      - name: eth0
  - name: ns2
    interfaces:
      - name: eth0
  - name: ns3
    interfaces:
      - name: eth0

cable:
  - name: conn1
    endpoint_a:
      name: ns1
      port: eth0
    endpoint_b:
      name: sw1
      port: "1"
  - name: conn2
    endpoint_a:
      name: ns2
      port: eth0
    endpoint_b:
      name: sw1
      port: "2"
  - name: conn3
    endpoint_a:
      name: ns3
      port: eth0
    endpoint_b:
      name: sw1
      port: "3"
```

C.6 Lab 3

This topology is based on the topology from Lab 2 (Appendix C.5), but with a fourth namespace and a new cable. Only the new namespace and cable are reported.

```Yaml
namespace:
  - name: ns4
    interfaces:
      - name: eth0

cable:
  - name: conn4
    endpoint_a:
      name: ns4
      port: eth0
    endpoint_b:
      name: sw1
      port: "4"
```

C.7 Lab 4

```Yaml
namespace:
  - name: ns1
    interfaces:
      - name: eth0
  - name: ns2
    interfaces:
      - name: eth0
  - name: ns3
    interfaces:
      - name: eth0
  - name: router
```

```
    interfaces:
      - name: eth0
      - name: eth1
      - name: eth2

cable:
  - name: conn1
    endpoint_a:
      name: ns1
      port: eth0
    endpoint_b:
      name: router
      port: eth0
  - name: conn2
    endpoint_a:
      name: ns2
      port: eth0
    endpoint_b:
      name: router
      port: eth1
  - name: conn3
    endpoint_a:
      name: ns3
      port: eth0
    endpoint_b:
      name: router
      port: eth2
```

C.8 Lab 5

Yaml

```yaml
switch:
  - name: sw1

namespace:
  - name: ns1
    interfaces:
      - name: eth0
  - name: ns2
    interfaces:
      - name: eth0
  - name: ns3
    interfaces:
      - name: eth0
  - name: router
    interfaces:
      - name: eth0

cable:
  - name: conn1
    endpoint_a:
      name: ns1
      port: eth0
    endpoint_b:
      name: sw1
      port: "1"
  - name: conn2
    endpoint_a:
      name: ns2
      port: eth0
    endpoint_b:
      name: sw1
      port: "2"
  - name: conn3
    endpoint_a:
      name: ns3
      port: eth0
    endpoint_b:
      name: sw1
      port: "3"
  - name: conn4
    endpoint_a:
      name: router
      port: eth0
    endpoint_b:
      name: sw1
      port: "10"
```

C.9 Lab 6

Yaml

```yaml
namespace:
  - name: ns1
    interfaces:
      - name: eth0
  - name: router1
    interfaces:
      - name: eth0
      - name: eth1
  - name: router2
    interfaces:
      - name: eth0
      - name: eth1
  - name: ns2
    interfaces:
      - name: eth0

cable:
  - name: conn1
    endpoint_a:
      name: ns1
      port: eth0
    endpoint_b:
      name: router1
      port: eth0
  - name: conn2
    endpoint_a:
      name: router1
      port: eth1
    endpoint_b:
      name: router2
      port: eth0
  - name: conn3
    endpoint_a:
      name: router2
      port: eth1
    endpoint_b:
      name: ns2
      port: eth0
```

C.10 Lab 7

```yaml
switch:
  - name: sw1

namespace:
  - name: ns1
    interfaces:
      - name: eth0
        ip: 10.0.0.1/24
        gateway: 10.0.0.254
  - name: ns2
    interfaces:
      - name: eth0
        ip: 10.0.0.2/24
        gateway: 10.0.0.254
  - name: router
    interfaces:
      - name: eth0
        ip: 10.0.0.254/24
      - name: eth1
        ip: 90.70.60.1/30
    commands:
      - /usr/sbin/sysctl -w net.ipv4.ip_forward=1
  - name: internet
    interfaces:
      - name: eth0
        ip: 90.70.60.2/30

cable:
  - name: conn1
    endpoint_a:
      name: ns1
      port: eth0
    endpoint_b:
      name: sw1
  - name: conn2
    endpoint_a:
      name: ns2
      port: eth0
    endpoint_b:
      name: sw1
  - name: conn3
    endpoint_a:
      name: router
      port: eth0
    endpoint_b:
      name: sw1
  - name: conn4
    endpoint_a:
      name: router
      port: eth1
    endpoint_b:
      name: internet
      port: eth0
```

C.11 Lab VXVDE

Yaml

```yaml
namespace:
  - name: router
    interfaces:
      - name: eth0

cable:
  - name: conn1
    endpoint_a:
      name: router
      port: eth0
    endpoint_b:
      name: vxvde1

vxvde:
  - name: vxvde1
    addr: 239.0.0.1
    port: 14789
```

Bibliography

[1] L. L. Peterson and B. S. Davie, *Computer networks: a systems approach.* Elsevier, 2007.

[2] J. Kurose and K. Ross, “Computer networks: A top down approach featuring the internet.” Pearson Addison Wesley, 2010.

[3] “VirtualSquare.” [Online]. Available: https://wiki.virtualsquare.org/#/

[4] “GNS3.” [Online]. Available: https://gns3.com/

[5] “GNS3 history.” [Online]. Available: https://gns3.wordpress.com/history/

[6] “GNS3 history.” [Online]. Available: https://rednectar.net/gns3-workbench/a-little-gns3-history/

[7] J. C. Neumann, *The Book of GNS3*, 1st ed. USA: No Starch Press, 2014.

[8] “vpcs.” [Online]. Available: https://github.com/GNS3/vpcs/

[9] “GNS3 README.” [Online]. Available: https://github.com/GNS3/gns3-server/blob/master/README.md

[10] Cisco, “Packet Tracer.” [Online]. Available: https://www.netacad.com/cisco-packet-tracer

[11] “Eve NG.” [Online]. Available: https://www.eve-ng.net/

[12] Boson, “NetSim.” [Online]. Available: https://www.boson.com/NetSim-13

[13] Boson, “NetSim user guide.” [Online]. Available: https://boson.com/Files/Support/NetSim-Online-User-Guide.pdf

[14] "Kathará." [Online]. Available: https://www.kathara.org/

[15] "Kathará Success Stories." [Online]. Available: https://www.kathara.org/stories.html

[16] M. Scazzariello, L. Ariemma, and T. Caiazzi, "Kathará: A Lightweight Network Emulation System," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–2. doi: 10.1109/NOMS47738.2020.9110351.

[17] "Kathará node startup configuration file." [Online]. Available: https://github.com/KatharaFramework/Kathara-Labs/blob/main/main-labs/data-center-routing/data-center-vxlan/kathara-lab_vxlan-base/s1.startup

[18] "Kathará lab directory example." [Online]. Available: https://github.com/KatharaFramework/Kathara-Labs/tree/main/exercises/02-add-one-router/lab

[19] "Kathará lab.conf file example." [Online]. Available: https://github.com/KatharaFramework/Kathara-Labs/tree/main/exercises/02-add-one-router/lab/lab.conf

[20] "Docker Network Plugins." [Online]. Available: https://docs.docker.com/engine/extend/plugins_network/

[21] "Kathará network plugin." [Online]. Available: https://github.com/KatharaFramework/NetworkPlugin/

[22] "CGo." [Online]. Available: https://go.dev/blog/cgo

[23] "Kathará lab directory example." [Online]. Available: https://github.com/KatharaFramework/Kathara-Labs/tree/main/main-labs

[24] "containerlab." [Online]. Available: https://containerlab.dev/

[25] "YAML." [Online]. Available: https://yaml.org/

[26] "containerlab VM." [Online]. Available: https://containerlab.dev/manual/vrnetlab/

[27] "containerlab Images." [Online]. Available: https://containerlab.dev/manual/kinds/

[28] "containerlab Network." [Online]. Available: https://containerlab.dev/manual/network/

[29] “containerlab Labs Examples.” [Online]. Available: https://containerlab.dev/lab-examples/lab-examples/

[30] M. Mahalingam *et al.*, “Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks.” [Online]. Available: https://www.rfc-editor.org/info/rfc7348

[31] “VNX.” [Online]. Available: https://web.dit.upm.es/vnxwiki/index.php/Main_Page

[32] “Marionnet.” [Online]. Available: https://www.marionnet.org/site/index.php/en/

[33] “QEMU.” [Online]. Available: https://www.qemu.org/index.html

[34] “VirtualBox.” [Online]. Available: https://www.virtualbox.org/

[35] J. D. Dike, “User-mode Linux,” in *Proceedings of the 2001 Ottawa Linux Symposium (OLS)*, Ottawa, 2001.

[36] J. D. Dike, “Making Linux Safe for Virtual Machines,” in *Proceedings of the 2002 Ottawa Linux Symposium (OLS)*, Ottawa, 2002.

[37] R. Davoli, “Vde: Virtual distributed ethernet,” in *First international conference on testbeds and research infrastructures for the development of networks and communities*, 2005, pp. 213–220.

[38] “kvm.” [Online]. Available: https://linux-kvm.org/

[39] “picotcp.” [Online]. Available: https://github.com/virtualsquare/picotcp

[40] “libvdestack.” [Online]. Available: https://github.com/rd235/libvdestack

[41] “umps3.” [Online]. Available: https://wiki.virtualsquare.org/#/education/umps

[42] “VDE-2.” [Online]. Available: https://github.com/virtualsquare/vde-2

[43] B. W. Kernighan and D. M. Ritchie, *The C programming language.* prentice-Hall, 1988.

[44] “vdeplug4.” [Online]. Available: https://github.com/rd235/vdeplug4

[45] R. Davoli and M. Goldweber, “VXVDE: A switch-free VXLAN replacement,” in *2015 IEEE Globecom Workshops (GC Wkshps)*, 2015, pp. 1–6.

[46] "vdens." [Online]. Available: https://github.com/rd235/vdens

[47] "libvdeslirp." [Online]. Available: https://github.com/virtualsquare/libvdeslirp

[48] K. Morris, *Infrastructure as code.* O'Reilly Media, 2020.

[49] "OpenTofu." [Online]. Available: https://opentofu.org/

[50] "Git." [Online]. Available: https://git-scm.com/

[51] "Rust." [Online]. Available: https://www.rust-lang.org/

[52] S. Klabnik and C. Nichols, *The Rust programming language.* No Starch Press, 2023.

[53] "ImagiNet Github repository." [Online]. Available: https://github.com/samuelemusiani/imaginet

[54] "Vagrant." [Online]. Available: https://developer.hashicorp.com/vagrant

[55] "Alpine Linux." [Online]. Available: https://www.alpinelinux.org/

[56] "Hyperfine." [Online]. Available: https://github.com/sharkdp/hyperfine

[57] "iperf3 website." [Online]. Available: https://iperf.fr/iperf-download.php

[58] T. H. M. and, "Experiential learning – a systematic review and revision of Kolb's model," *Interactive Learning Environments*, vol. 28, no. 8, pp. 1064–1077, 2020, doi: 10.1080/10494820.2019.1570279.

[59] L. Bellido, D. Fernández, E. Pastor, and J. Berrocal, "New strategies for learning computer networks," in *Proceedings of the 2012 IEEE Global Engineering Education Conference (EDUCON),* 2012, pp. 1–5. doi: 10.1109/EDUCON.2012.6201146.

[60] L. Bellido Triana, D. Fernández Cambronero, and E. Pastor Martín, "Towards a virtualized Internet for computer networking assignments," in *2013 IEEE Global Engineering Education Conference (EDUCON)*, IEEE, 2013, pp. 1009–1014. doi: 10.1109/EduCon.2013.6530231.

[61] "iproute2." [Online]. Available: https://wiki.linuxfoundation.org/networking/iproute2

[62] "Internet Control Message Protocol." [Online]. Available: https://www.rfc-editor.org/info/rfc792

[63] "An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware." [Online]. Available: https://www.rfc-editor.org/info/rfc826

[64] W. Eddy, "Transmission Control Protocol (TCP)." [Online]. Available: https://www.rfc-editor.org/info/rfc9293

[65] "Wireshark website." [Online]. Available: https://www.wireshark.org/

[66] "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks," *IEEE Std 802.1Q-2022 (Revision of IEEE Std 802.1Q-2018)*, vol. 0, no. , pp. 1–2163, 2022, doi: 10.1109/IEEESTD.2022.10004498.

[67] "Netfilter." [Online]. Available: https://www.netfilter.org/

[68] "Shorewall." [Online]. Available: https://shorewall.org/

[69] "AGPL-3." [Online]. Available: https://www.gnu.org/licenses/agpl-3.0

Acknowledgements

I would like to express my deepest gratitude to my family for their unconditional and immense support. They encouraged me to follow my dreams without any pressure or imposition. I am truly grateful for their presence and belief in my abilities.

I am grateful to my supervisor, Renzo Davoli, for his teaching and especially for supporting the wonderful project that ADMStaff is. During these university years, I had the freedom of experimentation that helped me discover my interests and understand what master's degree to pursue.

I deeply express my gratitude to Gio, for these 14 years and for all the amazing adventures and time spent together. You shaped the person that I have become and you have profoundly helped me uncover my creative half. You taught me that being unconventional is rare and something to be proud of.

I will forever be thankful to Mia, for being my harbour when I was barely afloat. You are the only person who is capable of comforting me when everything is collapsing.

I will always remember Omar, Lele and Lollo, for being there since the beginning. The moments spent laudly chatting over the most ephemeral things, and the days spent programming and hunting the most insidious bugs are impossible to forget.

I will never forget how joyful and bright the days were with Alice. All the hours spent underground without a ray of sunshine and the nights filled with live music. Your energy is endless.

I am obliged to Luca, for introducing me to ADM and believing in my abilities from the first year. You are an inspiration.