NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

DAVID MANUEL MARQUES ANTUNES

BSc in Computer Science

TOWARDS GENERIC AND SCALABLE NETWORK EMULATION

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

TOWARDS GENERIC AND SCALABLE NETWORK EMULATION

DAVID MANUEL MARQUES ANTUNES

BSc in Computer Science

Adviser: João Carlos Antunes Leitão
Assistant Professor, NOVA University Lisbon

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2023

Abstract

With the surge of microservice architectures, distributed systems and the usage of cloud in recent years, it has become increasingly relevant the importance of the network infrastructure. On the operation of distributed systems, proper network management in such systems is critical to ensure the correct operation of the system and providing quick response times for clients. Such systems become complex to integrate new features, which may lead to system failures and service unavailability. To avoid such mishap, a testing environment is required so new features will not disrupt the correct operation of the system.

Solutions capable of providing physical testbeds become unfeasible when referring to distributed systems composed of hundreds to thousands of nodes. Due to recent innovations and the adoption of Cloud, the deployment of distributed systems throughout the world has been simplified. Experimenters can turn to these platforms to create a proper testing environment, but unfortunately, such complex systems require the creation of complex testing infrastructures, deploying multiple machines throughout the world, and as such incur in high costs while being time-consuming. To avoid such costs, developers can turn to emulation or simulation to build less complex testing environments while maintaining low operating costs. The challenge is to find an adequate solution requiring less computing power but still captures the complexity of real execution environments and produce accurate results when compared to physical testbeds, particularly in what relates to the network conditions.

In this work, we want to explore the creation of a network emulator which can effectively be used by developers to test large scale distributed systems while achieving accurate results. Such a system allows experimenters to more accurately test and debug their distributed solutions and prepare for possible unknown scenarios. Testing and debugging a network includes multiple aspects from testing network properties, protocol properties, emulating datacenter networks, or emulating physical hardware. Our solution distinguishes from previous works by providing a real-time configuration of the system to allow the experimenter to modify in real-time the network properties while the experiment is running. Therefore, tools such as XDP, eBPF, TC and Netfilter can be explored as building

blocks for this solution as well as Docker and Kubernetes to build a scalable system.

Keywords: Distributed Systems, Emulation, Networking

Resumo

Com o aparecimento de arquiteturas micro-servico, sistemas distribuídos e a utilização da nuvem em anos recentes, tornou-se cada vez mais aparente a importância das infra-estruturas de rede. Com a operação de sistemas distribuídos, alcançar a devida gestão da rede nestes sistemas é critico para garantir a correta operação do sistema e providenciar respostas rápidas aos clientes. A adição de novas funcionalidades nestes sistemas torna-se complexo, originando falhas ou indisponibilidade do sistema. De modo a evitar tal infortúnio, a criação de um ambiente de testes é necessário para garantir que novas funcionalidades não perturbe o correto funcionamento do sistema.

Soluções capazes de providenciar infraestrutura física de testes são muitas vezes inviáveis tendo em conta sistemas distribuídos constituídos por centenas senão milhares de nós. Devido a inovações recentes e a adoção da nuvem, foi simplificada a implementação de sistemas distribuídos dispersos pelo mundo. Investigadores podem recorrer a estas plataformas para criar um ambiente de teste, mas infelizmente poderão incorrer em altos custos e também têm de lidar com a complexidade da sua configuração. De modo a evitar esses custos, programadores podem recorrer à emulação ou simulação para criar ambientes de teste menos complexos e manter os custos de operação baixos. O desafio é encontrar uma solução adequada que necessite de menos capacidade de computação, mas que continue a representar a complexidade de uma rede física e conseguir produzir resultados concretos quando comparado com uma rede real, particularmente as suas condições de funcionamento.

Neste trabalho, queremos explorar a criação de um emulador de rede capaz de ser utilizado por investigadores e engenheiros para poderem testar sistemas distribuídos de grande escala e obter resultados relevantes. Tal sistema irá permitir aos investigadores testar mais precisamente e depurar as suas soluções distribuídas e a sua preparação para cenários desconhecidos, nomeadamente em relação a condições de rede. O teste e depuração de uma rede incluem diversos aspetos, tais como as suas propriedades, as propriedades de protocolos de comunicação, emulação de redes de centros de dados, ou emular equipamento físico de rede. A nossa solução distingue-se de trabalhos prévios por permitir a configuração e modificação em tempo real das propriedades de rede enquanto

a experiência é executada. Portanto, a utilização de ferramentas como o XDP, eBPF, TC e Netfilter vão ser exploradas como elementos-base para a construção da nossa solução como também a utilização de Docker e Kubernetes para construir uma solução genérica de fácil utilização.

Palavras-chave: Sistems Distribuídos, Emulação, Rede

Contents

List of Figures

1 Introduction

Due to many recent technological innovations, more and more systems are becoming increasingly complex due to the number of internal application components distributed throughout different regions, which require inter-cooperation and utilize coordination mechanisms to ensure the correct orchestration of operations. The increase in scale and complexity makes testing and debugging novel applications troublesome, since testing a particular component requires the presence of other components in the architecture, introducing their own bugs while including hard to predict behaviors in the presence of different network anomalies. Faced with such challenges, developers have to turn to the creation of dedicated testing infrastructure to be able to test their applications before having real world impact. Engineers can turn to testbeds such as Emulab [1] or PlanetLab [2] which offer a realistic deployment scenario. However, such platforms limit both the scalability of experiments and the control of the experiment over the network.

With the increased popularity of Cloud solutions [3], the ability to achieve global coverage can be done in mere seconds. Cloud providers such as Azure [4], Google [5], and AWS [6] provides services that allow users to be capable of spinning up machines in any part of the world quickly. Researchers and practitioners can use these services to obtain machines from various parts of the world, but that leads to high costs that researchers might not be able to sustain. This, in turn, impairs the deployment of network topologies composed of hundreds to thousands of nodes around the world and leads experimenters to search for solutions such as emulation or simulation.

Having a testing ground leads to the ability to observe the application behavior when faced with different deployment scenarios and network conditions. In the case of evaluating the behavior of a distributed system under different network conditions, the testing can range from a simple modification of the network interface [7] to fully fledged Data-Center Networks [8]. There are many emulators and simulators designed throughout the years, but each focus on a specific network deployment type or properties, namely network devices [8], link properties [7, 9, 10] or hardware configurations[11].

The current state-of-the-art is the network emulator Kollaps [9], which utilizes recent technologies to emulate the end-to-end network properties but does not provide the ability

of configuring the network behavior in real time while also presenting some scalability issues, which we will further discuss in the following chapters.

In this work, we plan to address the aforementioned limitations by creating a network emulator capable of emulating the end-to-end properties of a network such as bandwidth, latency, jitter, packet loss while also being able to reconfigure the network in real-time during the execution of an experiment. Contrary to current approaches, we plan to explore and design where we redirect traffic to a centralized component to apply traffic shaping, while ensuring scalability and accurate results.

1.1 Objective

In this work, we plan to implement a scalable network emulator capable of accurately emulating a network topology consisted of thousands of network devices with the ability of changing its network properties in real-time while allowing the execution of real unmodified application code resorting to containerization technology, thus resulting in more accurate results when observing the behavior of the application. Achieving this high level goal comes with its challenges, namely the implementation of a network abstraction capable of emulating network properties fast enough to ensure accurate behavior with the network properties defined by the researcher. Another challenge is the ability to modify the network properties initially defined with new behaviors without disrupting the natural flow of execution. Finally, the challenge of ensuring lightweight computation, promoting better scalability when emulating a network topology with thousands of nodes, while load balancing the emulation execution to multiple machines to avoid exhausting the resources of a particular machine.

1.2 Expected Contributions

The expected contributions that we expect to produce from this work can be summarized as follows:

- emulating end-to-end properties of a network such as bandwidth limits, latency, jitter, packet loss and path congestion of complex and heterogeneous networks;
- running arbitrary and real unmodified application code by leveraging containerization technologies;
- enabling a testbed with the ability to change network properties at runtime during experiments;
- scaling into multiple machines to surpass the hardware limitations of a single machine.

1.3 Document Structure

The remainder of this document is organized as follows:

Chapter 2 will first cover some basic network concepts and present the current limitations of existing trivial solutions for testing and evaluating complex distributed systems. Such limitations will lead us to a discussion of alternative techniques such as emulation and simulation, where we discuss their differences and their relevance in a network context. Next, we present existing tools and techniques capable of modifying network properties and their applicability in this work. Finally, we discuss related projects which have tried to simulate or emulate certain network infrastructures, discussing the main employed mechanisms and their limitations.

Chapter 3 will describe how the previous literature relates to the objective of this work, and how we will approach the aforementioned challenges. Next, we will describe how the evaluation of the implemented emulator will be done to validate its design as well as to measure the accuracy of results obtained from real distributed applications being tested using our emulator. Finally, we present the planned work schedule for the following months.

2

Related Work

In this chapter, fundamental concepts will be presented throughout the sections that will build the foundations or are related with the solution to be proposed in Chapter 3. We will analyze current solutions and their limitations, present existing tools capable of manipulating network properties, and discuss related work.

The structure of this chapter will be the following: Section 2.1 introduces basic network concepts; Section 2.2 reflects current solutions capable of deploying large network topologies and their limitations; Section 2.3 introduces the meaning of simulation and emulation and discusses their application in a network context; Section 2.4 presents some network management and manipulation tools related to this work; Section 2.5 introduces and reviews concrete network simulators; Finally, Section 2.6 presents and discusses concrete network emulators.

2.1 Structure of a Network

Before delving further into subjects related to networking, we first introduce some basic network concepts, to help in the comprehension of the remainder of this document. As such, we can then ask ourselves, what constitutes a network? A network is a set of machines connected to each other capable of communicating through the exchange of messages. Of course, a network is not as simple as previously described, and evolves complex mechanisms. Between machines, there must exist a physical connection, known as data links, providing the flow of information. Since having a physical connection to every machine is impossible, specialized hardware devices capable of interconnecting multiple systems such as routers, switches, and hubs are employed in the network. This approach will lead to the creation of large networks requiring special communication to interconnect and coordinate, resulting on the Internet we know as of today.

In order to properly distinguish between different network mechanisms, researchers resort to the OSI Model to provide a clear distinction of network components. According to the OSI Model, a network can be described in seven layers, as shown in Figure 2.1.

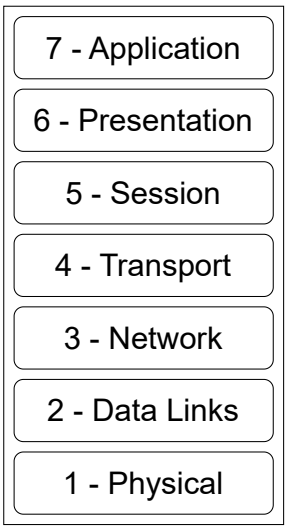

Figure 2.1: Seven Layers of Networking according to the OSI Model

The first layer represents the physical connections between devices capable of transmitting information; the second layer corresponds to direct network connections between two devices communicating through Ethernet frames; the third layer is the exchange of information using network packets, without requiring a direct connection to the destination; the fourth layer handles the flow of information sent between two endpoints; the remaining fifth, sixth, and seventh layers are the structures required to have a proper functioning application.

The primary focus of this work will be on the bottom three layers of the OSI Model, regarding the understanding, deployment, and network management of network devices.

Throughout this document, routers and switches belonging to a network will be referred to as network devices; the physical connection between two network devices as a link or data link; the physical arrangement of network devices and data links as a network topology; computers or running applications in the network as hosts or end hosts.

As previously mentioned, a small network can be composed of a large amount of network devices, becoming impossible for researchers to build their own physical testbeds, performing manual configuration for a particular network topology and execute experiments. Alternative approaches researchers can resort to mitigate such impossibility will be discussed in the following sections.

2.2 Cloud and Physical Testbeds

Complex systems running in a single machine quickly encounter performance issues, namely the high utilization of a particular component causes resource starvation of the remaining system components, degrading operation throughput. As a countermeasure, developers can distribute the system into multiple machines, in order to evenly distribute the system load. These approach leads to a more complex system, due to requiring

coordination mechanisms to ensure the correct operation of the system [12]. On the other hand, this complexity leads to a more scalable solution.

To research these distributed systems, a local physical testbed with a couple of machines is feasible, cheap, and simple to set up. Challenges start to emerge when said systems start to be spread throughout multiple geolocated regions and interoperability between regions is necessary. In other words, testing and evaluating distributed systems is particularly challenging when they become large scale distributed systems.

Researchers can resort to research testbeds, i.e., Emulab [1] and PlanetLab [2], involving hundreds of machines deployed across multiple geolocated regions. These research testbeds revolve around a community of researchers, resulting in multiple projects being tested in those infrastructures, and thus could be difficult to request for a specific network topology and the requester has the responsibility of configuring the machines to provide an adequate testing environment to test and validate large distributed systems.

A possible alternative to those testbeds is resorting to cloud infrastructure. Cloud providers such as Google [5], AWS [6] and Microsoft Azure [4] are amongst the most popular cloud providers currently. These providers contain data-centers spread across the world [13] and provide several tools and services that make deploying multiple machines across multiple data-centers. Unfortunately, such functionality has an associated cost and creating a testbed composed of hundreds of machines across the world results in a high cost of operation, becoming unfeasible to maintain in a monetary sense.

Now we are left to wonder if previous solutions are not feasible to test distributed systems, what can researchers turn to when they and need to test or evaluate a distributed system that operates on hundreds, if not thousands, of machines scattered through regions across the world, while being cost friendly? Researchers can resort to techniques such as simulation [10, 14] and emulation [7, 9, 15] as a cost-effective approach while also obtaining accurate results when comparing to physical testbeds. These approaches will be discussed in later sections of this chapter.

2.3 Simulation and Emulation in a network context

2.3.1 Simulation

In Computer Science, experimenters and researchers resort to simulation to simulate the expected behavior of a particular component, a set of components or an entire system.

Models are built in order to test said component and to predict its behavior when given a particular set of inputs [16]. The model is devised according to the behavior the researcher plans to observe. The creation of an accurate model built in a simulation environment allows engineers the use of real hardware, enabling the possibility of observing its functionality without needing a full implementation of the system [17, 18].

In simulation, every behavior of the model is configured, and as such, experimenters can have a deeper understanding of the model's execution since they can observe every

value present. One of those values is the passage of time. By altering the passage of time in the simulation, experimenters are capable of simulating in slower hardware or the slower execution of a complex model comprised of many components, its correct execution.

Simulated models run in isolated mode, without interacting with real world events. Furthermore, the execution of the model in a simulated fashion could lead to inaccurate results when compared to its implementation in a real testing environment [19] due to external anomalies not accounted for in the simulated model.

2.3.2 Emulation

Another approach previously mentioned in Section 2.2 when researching is to resort to emulation [20]. Emulation, as in simulation, requires the construction of a new model.

Emulation tries to design a model closely resembling to its physical counterpart. The positive outcome of this procedure is the ability to do live testing, enabling experimenters to test an emulated model with real and verified systems, leading to more accurate results when compared to a physical testbed due to unpredictable behavior from external input [21]. On the other hand, the downside of executing the experimentation inside an emulated environment comes its real time execution, restricting the effectiveness of the emulation due to the physical limitations of the machine. In [9], while evaluating their scalable network emulator by comparing with other network emulators to measure accuracy, limitations of other network emulators when emulating large network topologies were shown.

2.3.3 Network Context

In networking, all kinds of systems can be tested. Systems can range from a simple simulation of a communication protocol like TCP [22], emulate network devices [11], or emulating large network topologies [9].

To create testing environments to obtain accurate results without being computationally expensive, we arrive at a system which meshes the concept of emulation with simulation, as discussed by the authors in [20]. According to them, network emulation is the union between the usage of simulation and live environments. The correct approach to solve the problem is by defining simpler but accurate network models, capable of interacting with real technology. Throughout the years, there have emerged network simulation tools such as NS-3 [10] or OMNeT++ [14], but ultimately are limited to the execution of the network model. Section 2.5 elaborates further on how such simulators operate.

Work has also been done to bring simulators closer to a live testing environment. An example of such a system is Dockemu [23] who uses Docker [24] to build a containerized environment to allow the experimenter to run arbitrary code while also modifying the Docker network configuration to allow the connection of containers with the simulation running in NS-3 [10]. Since there is a mix with real applications, the simulated model from NS-3 is executed in real-time.

For network emulation as previously stated before and also mentioned by the authors of [20], the construction of simple network models suited to interact with real world events is necessary. This approach lets the researcher obtain accurate results, while simulating various components in the network, reducing the required computational resources by avoiding the full emulation of every component. These are the approaches of network emulators such as DummyNet [7], NetEm [25], Mininet [26], Maxinet [8], and Kollaps [9], to name a few. Section 2.6 provides more details on the operation of the mentioned emulators.

2.4 Network Management and Manipulation Tools

Before diving into the discussion of existing approaches to network simulators and emulators, we use this section to present existing tools capable of modifying certain aspects of the network, as well as solve existing challenges we may face in the implementation of a possible solution.

In the following sections, we begin describing eXpress Data Path (XDP), a high-performance packet processing framework in Section 2.4.1; Section 2.4.2 presents extended Berkeley Packet Filter (eBPF), an instruction set and virtualized environment capable of executing in XDP; Section 2.4.3 presents Linux Traffic Control (TC), a powerful network tool capable of modifying the behavior of network interfaces; Section 2.4.4 introduces Netfilter, a network tool capable of applying packet shaping to incoming packets; Section 2.4.5 discusses a different approach to manage a network, through the usage of Software Defined Networks (SDN); Section 2.4.6 discusses the approach of using Network Function Virtualization (NFV); Section 2.4.7 discusses different approaches to integrating application code without modifying it; Section 2.4.8, addresses scalability tools for the proposed solution; Finally, in Section 2.4.9 reflects how these tools and techniques are helpful in contributing to the development of the proposed solution.

2.4.1 eXpress Data Path

The eXpress Data Path [27], also known as XDP, is a high-performance packet processing framework operating at the kernel level.

XDP programs execute with a limited subset of kernel functions in order to provide safety of execution, since code is running in kernel-space with the highest privilege.

Using XDP enables the user to integrate code to work alongside the network stack, utilizing other network functionalities such as the routing table, and enabling the modification of it on the fly. Since XDP programs run directly on the device driver, the host application is agnostic to changes made by XDP. Due to such operating approach, XDP programs avoid the overhead of context switching between user-space and kernel-space restrictions.

Since XDP runs on the device driver, XDP is the first component to interact with a given packet. As such, it enables its modification without the Linux kernel or a given application noticing.

For XDP programs to run, they require a hook from the network device. A hook executes code for every packet that arrives at the network device. The hook enables XDP programs to run directly on the network device. XDP has access directly to the contents of the received packet and can then modify it. XDP uses eBPF (see section 2.4.2) instruction set and virtualized environment to safely run code in the kernel.

2.4.2 Extended Berkeley Packet Filter

The Extended Berkeley Packet Filter [28] (eBPF) is the successor of the Berkeley Packet Filter [29] (BPF). Its functionality is provided by an instruction set and an execution environment inside the Linux Kernel. It allows the modification of packet processing in network devices. An eBPF program can be compiled from restricted C language and then executed by XDP (see Section 2.4.1).

Compared to BPF, eBPF comes with additional instructions, increased available registers while also increasing the available memory. eBPF's new instructions allows making function calls, providing the capability of modifying storage, called maps. Maps are key-values stores capable of storing user-defined data. Processes in user-space can create multiple maps, and they can both be accessed by other processes and eBPF programs, which are running in the kernel. ePBF programs keep a reference counter for each map as a mechanism to ensure that if no program points to a particular map, it can to free its memory.

A user-space program has limited capabilities and does not have access to hardware, and as such requires the usage of system calls to be able to access the kernel to perform tasks such as sending a network packet. A program running in kernel-space has access to every instruction available, as well as access to network devices. As such, the flexibility of sharing maps allows the sharing of data between user-space programs and eBPF programs running in the kernel.

Another useful functionality of eBPF programs is the ability to call other eBPF programs, via tail calls [30], by reutilizing existing kernel memory to run the next ePBF program. Such flexibility allows the developer to design simpler programs.

eBPF uses a verifier to ensure integrity and security when executing the program in the kernel. It ensures the termination of the execution by verifying for the existence of loops through exploiting tail calls.

Several solutions have resorted to eBPF to reimplement existing tools. The authors in [31] utilize eBPF to reimplement existing mechanisms provided by NetEm [25]; implementation of Network Function Virtualization (see Section 2.4.6) in PolyCube [32], and in network monitoring [33] and observability [34, 35].

Unfortunately, eBPF programs come with limitations. As previously mentioned, eBPF only has access to a subset number of C language library functions. Another limitation is its limited stack space, which maximum size is 512 bytes. Another restriction is the impossibility of executing loops, allowing for more complex programs to be defined.

2.4.3 Linux Traffic Control

The Linux Traffic Control [36–38] (TC) is a powerful network tool available on Linux. It is capable of altering the default configurations of network devices to change how arriving packets are processed. TC is capable of doing shaping, scheduling, policing, and dropping incoming network traffic. It does so by utilizing what they call *qdiscs*, which is short for "queuing disciplines". Queues are the backbone of TC, since they store incoming packets into a memory buffer, waiting for packets to be processed.

Queues are important because they allow for several actions. One of them is applying bandwidth limits, which requires an algorithm to be applied to the queue. The user can then define which algorithm to be applied to the queue, with configurable parameters. For example, TC provides a First-In-First-Out (FIFO) or a Token Bucket Filter (TBF) algorithms, which have different behaviors. FIFO, as the name suggests, will process packets in the same order as they are enqueued and dequeued. The algorithm allows the limitation of the queue's size, dropping packets as soon as the queue is full. TBF is an algorithm that generates tokens at a pre-defined speed, thus restricting the number of packets processed. When a token is available, it will then process a bucket of packets. The availability of tokens is useful to restrict the bandwidth limit of the interface.

Packet scheduling allows the reorganization of packets for output, which is quite useful when it is required for network traffic that contains different priorities. Such technique allows the application of Quality of Service mechanisms.

Packet classification provides the means to mark packets in order to redirect them to other queuing disciplines. The classification can occur during the arrival, routing or transmitting of a packet.

Policing operates akin to that of an if statement. For example, traffic limiting can also be done through traffic policing, since the condition can be a configurable upper limit on the current bandwidth. If this policy is violated, packets are dropped, else the enqueue occurs.

Although TC is great for traffic processing, it also presents some downsides. TC has a complicated syntax, resulting in undesirable complexity when configuring proper mechanisms. TC policies also need to be configured in a small window of execution to ensure the configured constraints are satisfied. TC performance is influenced by the operating systems' tasks (degrading it if is running heavier tasks).

2.4.4 Netfilter

Going up in the network stack of Linux, we find Netfilter [39, 40], a packet manipulation framework, allowing the execution of callback functions to configured network events. Programs such as arptables [41], iptables [42] and ip6tables [43] and recently nftables [44–46] utilize Netfilter to manipulate packets in their respective network layer. Arptables manipulates packets at the Second Layer of the OSI Model (see Section 2.1), handling Ethernet Address Resolution Protocol [47] (ARP) communication while iptables and ip6tables manages IPv4 [48] and IPv6 [49] communication, respectively.

Netfilter allows the aforementioned programs to process packets, apply filters and in the case of iptables, forward packets, and do Network Address Translation (NAT) by utilizing the available network hooks. Those hooks are executed after the packets have passed through the TC layer of the Linux Kernel. This implementation permits hiding network devices behind a particular machine, thus reducing the number of public IPs [50].

Similarly to TC, iptables and nftables also suffers from the same limitations, scalability. With the addition of new rules, its processing overhead stops being negligible [51].

2.4.5 Software Define Networking

In small networks composed by a reduced number of network devices, the manual configuration of the network can be relatively simple. The problem surges when there is a large increase in the amount of network devices present, since manual configuration cannot be easily handled without significant risk of network configuration errors. A way to mitigate such problem and being able to tackle large quantities of network devices, is to rely on Software Defined Networking (SDN).

According to the authors in [52] and the authors of an extensive survey on Software Defined Networking approaches [53], this solution came to fruition since network engineers were stuck with the provided software given by manufacturers. By enabling a more flexible configuration in network devices, it is possible to separate the set of network operations into two groups, the control-plane and the data-plane. The control-plane is related with the configuration of network devices to handle forwarding mechanisms of traffic to other devices, while the data-plane involves the management of network rules to decide where the traffic should flow. With this definition, we can observe the simple architecture of an SDN in Figure 2.2.

The separation of the network in these two layers results in the ability to centralize the configuration of the network to the controllers, which are aware of the entire network topology. This knowledge permits the management of forwarding, filtering and prioritization of rules and behavior for traffic while also being aware of the optimal paths [54] to specific network devices.

Since the controller has the responsibility of managing the network, the network devices have the responsibility of applying traffic rules. With this design, the devices do not have knowledge of their particular actions. As such, when encountering a packet with

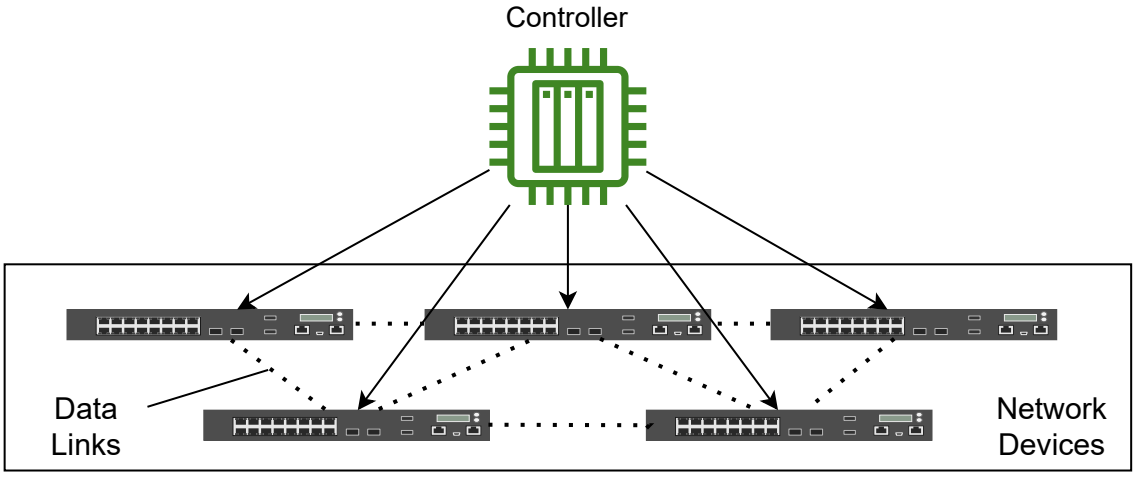

Figure 2.2: Typical software defined network architecture

a destination not present in the flow table, they request information to the controller. The controller will reply to the device with the respective action they will execute, populating their flow tables. Each device contains their respective flow tables dictating the action they execute depending on the traffic received. The Figure 2.3 shows the constitution of an SDN device.

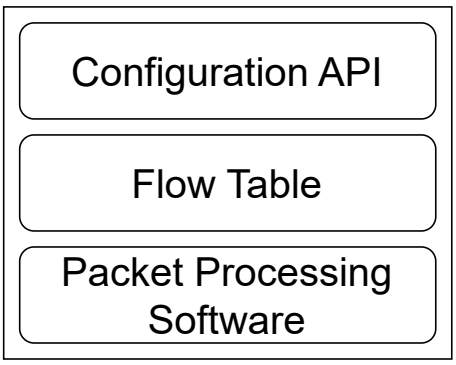

Figure 2.3: SDN device architecture

The protocol defining the communication of the controller and the network device is named OpenFlow [55] and allows the remote configuration of the flow table of a network device. The protocol also permits the configuration of many network mechanisms, permitting the separation of traffic into private networks by implementing VLANs [56] and fast packet forwarding with MPLS [57]. The open vSwitch [58] is a virtualized switch implementing the OpenFlow protocol for its configuration. According to [59], the vSwitch contains many features such as packet classification utilizing the tuple search packet classifier adapted to be cache friendly. Network emulators such as Mininet [26] and Maxinet [8] leverage this approach to emulate a virtualized network environment, utilizing the open vSwitch implementation and many other controllers to configure the

network devices.

2.4.6 Network Function Virtualization

While Software Defined Networks manage the control-plane of the network, defining the flow of traffic and interconnecting different machines, Network Functions Virtualization [60, 61] (NFV) works on the data-plane level and, as the name implies, it applies functions to the type of traffic received. This approach also appeared for the same reasons as Software Defined Networking. It works by abstracting dedicated machines for a specific task by virtualizing those operations and, as such, using more general purpose hardware to support a simpler configuration of network applications, while providing greater flexibility and scalability.

Simple examples NFVs in use include the implementation of firewalls, DHCP servers, and traffic encryption. Polycube [32] is a framework utilizing eBPF (see Section 2.4.2) with the purpose of creating such NFVs, with already implemented solutions. The past approach would require the physical installation of a load-balancer and proper configuration. With the usage of NFVs, the network device is kept as simple as possible, configuration wise, and only requires the necessary code to be able to run NFVs. As such, these devices will rely heavily on techniques such as containerization and Virtualization (see Section 2.4.7).

2.4.7 Virtualization and Containerization

As previously mentioned in Section 2.2, creating physical testbeds with a high number of components results in a strenuous task for its installation and configuration. An alternative approach is the usage of virtualization. With Virtualization, the developer gains the ability to virtualize and aggregate multiple physical devices in a single physical machine, by running software capable of isolating the hardware resources into isolated virtual machines. Virtual machines are composed of virtualized operating systems, mostly known as Guest OSes, which are run in specialized software, named hypervisors. Figure 2.4 shows the architecture distinction between two types of hypervisors.

Type 1 hypervisors run virtual machines on top of the hardware, providing better access to hardware than Type 2 hypervisors, where there is the operating system layer between the hypervisor and the hardware [62]. As such, Type 2 hypervisors have worse performance than their Type 1 counterpart, since the hypervisor needs to pass requests to the operating system to access hardware resources. On the other hand, Type 1 hypervisors require support from the underlying hardware while Type 2 is better supported in more commodity hardware since it is dependent on the underlying operating system, since hardware manufacturers tend to provide better support for. As mentioned previously, both hypervisors are capable of abstracting a physical network by virtualizing the network devices.

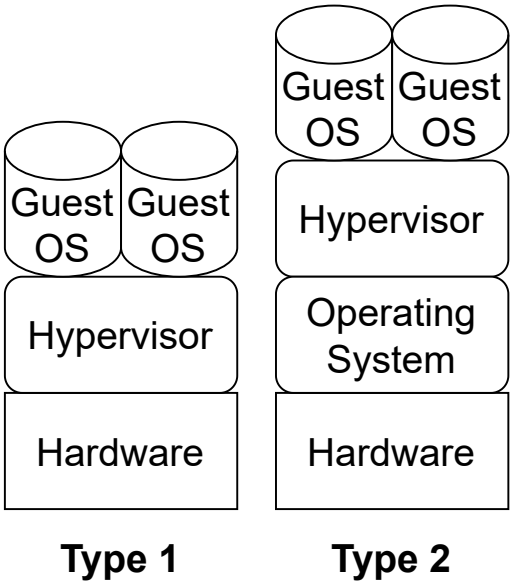

Figure 2.4: Types of a Virtual Machine Manager's architecture

Another alternative to virtualization is containerization. Containerization works similarly to a Type 2 hypervisor, but instead of having a hypervisor layer, contains a container engine. As opposed to virtualization, containerization does not virtualize a complete operating system but instead isolates applications, enabling the utilization of underlying resources from the host operating system. Tools such as Docker [24] or Linux LXC [63] utilizes cgroups [64] and Kernel namespaces [65] to isolate a process in the system, limiting the visibility of other processes running in the system while being provided with their own environment such as their own network configuration or filesystem.

Both approaches present mechanisms capable of creating a virtualized network without requiring its physical implementation. Both approaches present advantages and disadvantages, with the most notorious being performance, as demonstrated in [66–68], where containerized solutions presented lower resource when compared to Virtual Machines. In [69], the utilization of an application, namely the NoSQL database Cassandra [70], demonstrated higher throughput of transactions when running in a containerized environment as opposed to a virtualized environment.

2.4.8 Scalability

In Section 2.4.7, we addressed technologies capable of abstracting network devices into a single machine by resorting to virtualization or containerization. With the network topologies consisting of hundreds of network devices, we rapidly reach the hardware limitations of a physical machine. To surpass such restrictions, we must consider scalable solutions which will help us in providing scalable mechanisms to our proposed solution.

As mentioned in Section 2.4.7, containerization provides a lightweight environment

capable of running application code without requiring changes on the underlying environment to accommodate the application to be executed on. Containerization tools such as Docker [24] or Linux LXC [63] provide the environment to do so, enabling the abstraction of network devices. The problem arises when we reach hardware limitations. The emerging challenge then becomes the introduction of new machines, leading to the spreading of network emulation into multiple machines. The goal is the implementation of distributed mechanisms capable of coordinating between machines, ensuring the correct operation of the emulated network topology.

By itself, the aforementioned tools are not capable of coordinating between separate machines, and therefore we require the utilization of container orchestration tools such as Kubernetes [71] or Docker Swarm [72]. These tools provide the capability of interconnecting multiple Docker environments present in separate machines, allowing seamless communication between containers in different machines.

The aforementioned tools also contain extra mechanisms focused on application management, that do not interest us from a networking point of view, and could possibly introduce overhead. Examples of these mechanisms constitute the management of the availability of the system by providing replication mechanisms and load balancing between replicas [73, 74].

2.4.9 Discussion

Throughout the previous sections, we have presented multiple tools capable of manipulating network information exchange between the hosts. Tools such as XDP and eBPF, capable of high performance packet processing even before the packet is processed by the Linux Kernel, while TC and Netfilter are capable of classifying and filtering packets according to the destination received. These tools will be helpful as building blocks in the development of the proposed solution, since they provide the capability of configuring the emulated network according to the given network topology designed by an experimenter. Unfortunately, these tools come with their own limitations. For example, eBPF programs are restricted to a subset of functions as well as a limited memory stack space, corresponding to 512 bytes, meaning the executed programs need to be small. In TC and Netfilter, such tools present scalability problems when configured with many rules. In TC, the usage of many algorithms to manage bandwidth limits for each possible destination will lead to overhead in the execution of our network emulator.

On a higher layer of network abstraction, other approaches to network emulation such as SDNs, that utilize virtualization to deploy a virtualized network, while NFVs, work on the data-plane, are capable of implementing packet modification mechanisms, providing the structure to complement a virtualized network to be similar to a real one without requiring specialized network hardware. Unfortunately, such approach will not be taken since utilizing SDNs and NFVs requires the full implementation of network devices, which leads to scalability issues when emulating large networks.

Finally, we presented two different approaches into abstracting the operating system, allowing the execution of unmodified application code while also integrating with existing environments. We presented virtualization, capable of aggregating multiple network devices into a single physical machine. It does so by completely virtualizing the operating system, leading to waste of computational power, and degrading performance. Another approach is containerization, which does not fully virtualize the operating system but instead isolates the application itself, allowing the sharing of operating system resources. This approach is the most useful to us since it provides a lightweight environment, compared to virtualization, to run unmodified applications. Such approach leads us resorting to container orchestration tools capable of connecting containerized environments between multiple machines, surpassing the physical limitations of a single physical machine, while providing seamless communication between containers in separate environments and management of large deployments.

2.5 Network Simulators

As mentioned previously in Section 2.2, due to the limitations of utilizing research testbeds or utilizing cloud infrastructure to create a dedicated testing environment, researchers resort to emulation or simulation to create a network model capable of abstracting real network topologies into more simplified models. Throughout this section, we will present related network simulators, their objectives, and limitations. We will describe OMNeT++ in Section 2.5.1; 2.5.2 present NS-3; 2.5.3 describes Peersim; Finally, we will discuss their contributions and relevance to this work.

2.5.1 OMNeT++

OMNeT++ [14, 75, 76] positions itself as a discrete event simulator, meaning the simulation executes through events corresponding to state changes. It also follows an object-oriented framework approach, which facilitates the experimenter to build their own simulation according to their needs. OMNeT++ has a modular approach, enabling the creation of *simple modules*, written in C++, which can be joined to make *compound modules*. Modules communicate through the usage of *messages*, which can be internally inside a compound module between simple modules or externally to between compound modules. Messages are sent through *gates*, which are the interfaces of a module. *Links* serve to create a connection between different modules, thus being able to control network properties.

OMNeT++ uses its own declarative language called NED to define the structure of the simulation model.

OMNeT++ has been used to model various domains such as wired and wireless communication networks [77, 78], network switching algorithms [79], and simulating wireless topologies [80].

2.5.2 NS-3

Similar to OMNeT++, NS-3 [10, 76] defines network simulation through code. It uses the C++ language for the creation of simulation models. While OMNeT++ was developed has a network simulation framework, NS-3 has focused on the creation of more accurate models, providing easier debugging methods for the created models. Due to its discrete event architecture, developed models must be programmed to process their respective events, advancing the simulation time based on the events that are processed. Since NS-3 is model based, existing models can provide a quicker development cycle when prototyping new systems [18], since users can utilize already implemented models made from other developers.

NS-3 as been proven as useful to accurately define a model and study its behavior, and ensure the implemented model correctly implements the designed system, such as simulating OpenFlow switches [81], simulating routing protocols in datacenter networks [82] or validating wireless networks [17].

2.5.3 PeerSim

PeerSim [83–85] is a java based scalable network simulator with the sole purpose of simulating peer-to-peer networks, by providing two different simulation engines, cycle-based, where protocols are executed according to a given order or event-based, advancing the simulation time based on the events created. Utilizing a simple configuration file, PeerSim can create a random network topology consisting of thousands of nodes, effectively permitting the testing of large scale peer-to-peer networks. Due to the nature of the mentioned engines, PeerSim is capable of such feature since it does not require the events of the network to be processed in a small window to provide accurate results.

To achieve its scalable nature, it resorts to a simple architecture to reduce its execution footprint, by employing a simple architecture consisting of the simulation engine, the configuration manager and the network management logic. It maintains an array structure to store each individual peer, and each individual peer contains the number of connected peers. Due to the modularity provided by PeerSim, the peer structure can be extended with load balancing capabilities, aggregation of statistics, and observability of behavior in the network.

PeerSim has been used to build and validate network protocols such as Hyparview [86] which is a membership protocol, and Plumtree [87] which is a gossip based protocol.

Unfortunately, due to its modular nature, the experimenters are responsible for implementing their own protocols onto PeerSim, and any meaningful metrics the experimenters are interested in measuring. Since PeerSim is Java based, researchers are restricted to implementing their solutions in Java. Due to its focus on simulating peer-to-peer networks, the simulator abstracts the lower layers of the network, and as such does not take into account network devices such as routers and switches belonging to a network topology.

2.5.4 Discussion

Even though thesimulators described previously are capable of configuring a network topology with the required settings for the researcher, and efforts have been made to allow OMNeT++ and NS-3 to receive inputs outside the simulation as well as being able to run on multiple host machines [88], they can not help us in solving our problem since both approaches require the definition of network device models, and as such when running the simulation, scalability issues would arise from simulating complete behavior. PeerSim [83], on the other hand, was designed for testing peer-to-peer networks, with scalability as a core feature. As such, their abstraction of the network does not simulate the behavior of network devices that constitutes the network topology, and applying existing network properties such as bandwidth limits, jitter, latency or packet loss.

Another downside of the simulation is the lack of consideration for unpredictable events such as system overhead, leading to inaccurate results [19]. As such, we conclude the usage of such tools and or even the creation of a new network simulator is not the correct approach since we want to be as close to a native implementation of a network topology configured by the researcher while obtaining accurate results. As explained previously, the results obtained from simulated models do not correspond to the reality expected, and, in particular, they do not exercise real implementations which can further distance results produced from these tools to those that will be obtained by a real implementation.

2.6 Network Emulators

As was previously mentioned, network simulators, although capable of simulating the desired properties, ultimately suffer from various limitations, and we can conclude such approach is not the most effective approach in abstracting the network. Another approach would be network emulation.

In the following sections, related work regarding network emulators is presented. Section 2.6.1 presents DummyNet, a simple network emulator capable of testing communication protocols; Section 2.6.2 presents ModelNet, a complex network emulator capable of abstracting the network and integrating unmodified code into the experiment; Section 2.6.3, describes CrystalNet, a network emulator capable of emulating network device configurations; Section 2.6.4 we describe Mininet, a network emulator capable of emulating a network through the usage of SDNs, and Maxinet which corresponds to the scalable version of Mininet; Section 2.6.5 discusses Kollaps, a decentralized scalable network emulator which provides a possible approach to solving the problem we presented previously; finally, a discussion about the mentioned network emulators, and how they relate to the objectives of this work.

2.6.1 DummyNet

DummyNet [7] was created with the objective of emulating a network with routers and switches, where each has different network properties. With the objective to avoid the deployment of real routers and switches on a real testbed, avoiding high costs on acquiring real hardware, by of generating multiple network scenarios and allowing reproducibility and accurate results. DummyNet builds a simulation of the network with the intuition of testing various network properties as well as different queuing policies. DummyNet achieves such feat by intercepting the traffic passed between two network layers by introducing pipes between them. Network packets sent from the higher layer will be put into these pipes before being transferred to the lower layer. By introducing this middle step, DummyNet can utilize these pipes to manipulate the following aspects of the communication between two endpoints, such as queuing policies, queue reordering and queue sizes, bandwidth, latency, and packet loss by simulating various components between two peers. To simulate such behaviors, DummyNet relies on a timer to implement those limitations and a mathematical model to compute the appropriate delay to be applied. By analogy, each pipe can represent a different network component and DummyNet can apply modifications to the traffic that passes through that pipe to simulate the passage of the packet through that network component, effectively simulating the effects on traffic that the network component would introduce. Such flexibility enables the experimenter to quickly define multiple network topologies.

Since DummyNet intersects packets between layers, it works with any protocol. As such, it allows the testing of applications without having to make modifications to accommodate the emulator. As an example, an experimenter can observe the behavior of the TCP protocol in multiple settings.

Unfortunately, DummyNet has some limitations. It relies on a timer to implement bandwidth and latency, being restricted to the polling of time from the system and thus limiting the ability to apply bandwidth and latency properties. Therefore, the packet stays in queue for longer than originally intended. Another limitation is the fact DummyNet only runs on a single host, limiting scalability since it is restricted to the existing hardware resources of a single machine, limiting experimentation with large network topologies. Other tools such as Linux traffic control (see Section 2.4.3) employ similar properties as DummyNet.

2.6.2 ModelNet

ModelNet [15] tries to provide a solution to emulate large networks while also shaping traffic to include bandwidth constraints, latency, packet loss, and queuing disciplines.

Links are built using DummyNet's (see Section 2.6.1) base implementation of pipes, with improved accuracy and support for multi-hop and multicore emulation.

ModelNet was built with a distributed approach, separating the executing applications from the running emulation. ModelNet names servers that are connected to a dedicated

server cluster where the tested application is executed as edges nodes. The deployment of an experiment in ModelNet goes through five phases:

Creation regards the translation of the network topology defined by the experimenter to a graph where its edges correspond to links and *nodes* represent clients, *stubs* which are internal networks or *transits*, which connect stubs. The definition is converted to a graph modeling language where the researcher can add attributes such as packet loss rates, bandwidth limits latency and much more;

Distillation manages the conversion of the graph into actual network configurations, while simplifying the network topology through a breadth-first search to generate successive frontier sets. This results in finding interior sets which do not belong to the first frontier set, and thus the interior links are replaced by a full mesh interconnecting interior nodes. This ensures the packets pass between $(2 * walk - in) + 1$ pipes rather than the full network, saving on computing cycles;

Assignment applies the created pipe configurations to the respective node in the cluster, according to the new generated network topology;

Binding assigns which servers are edge nodes and deploys the application;

Run starts the applications.

During execution, the usage of pipes enables the emulation of path congestion since the pipe is shared between multiple flows as well as packet drops, resulted from the queue size being reached. Packets are processed every system tick, where ModelNet will parse each queue to dequeue packets that have surpassed their delayed time.

Due to its architecture and design choices, ModelNet has some limitations. For example, the creation of infrastructure and its configuration can be difficult, since ModelNet separates the emulation to some servers and the application execution to other machines. Such separation leads to the risk of saturating the link connecting to the edge nodes to the emulation nodes, in cases where there are hundreds of applications instances running. Due to its queuing processing implementation, the network constraints can be broken since packets are processed according to the system clock, leading to inaccurate results. Another limitation is the fact the network during execution is static, as such there is no possibility to turn on or turn off data links.

Recent technologies, namely container orchestration such as Kubernetes [71], would simplify the management of the servers and deployment of experiments.

2.6.3 CrystalNet

CrystalNet [11] was created to improve network reliability by emulating the control-plane of the network. The control-plane corresponds to the components which configure the

network. it allows for testing heterogeneous network device configurations to ensure their correct execution before being used in production environment.

Some errors CrystalNet tries to mitigate are software bugs such as the incorrect functioning of a particular protocol; an unexpected behavior or change in format; configuration bugs such as the incorrect parameterization of a network device or human errors such as the miss-configuration of a particular link. It does so by employing the use of virtual Machines or containers to run network devices, not requiring the creation of physical testbeds. Since the devices are run in a virtualized environment, CrystalNet can emulate the network as if it was a physical network, promoting transparency since the virtualized devices can run production configurations, thus ensuring a more accurate environment for testing.

Due to the design choices taken by CrystalNet, it is highly focused on the control-plane and testing of network devices and as such does not provide support to run applications while also not being able to change network properties such as bandwidth and latency during an experiment.

2.6.4 Mininet and Maxinet

Mininet [26] enables researchers and experimenters to experiment with Software Defined networks and OpenFlow [55](see Section 2.4.5). A Python API is exposed to allow the creation of custom experiments. The usage of OpenFlow allows Mininet to configure switches according to the defined network, using a controller. When the switch receives a packet with an unknown destination in its flow table, it asks the controller to populate its flow entry with the correct operation. Mininet uses Linux virtualization mechanisms [63] to run network devices with their own configuration, isolating various components, thus being able to virtualize the given network.

Due to its design, Mininet is restricted to only one machine and as such does not scale. Maxinet [8] addresses this limitation by running multiple Mininet instances, connecting them through Generic Routing Encapsulation (GRE) tunnels [89], which consist of encapsulating a packet inside another data packet to simplify communication between two machines, enabling the communication between two mininet instances.

Although Mininet does a great job emulating a network topology, we are only interested in a simpler approach since we want to ensure the correct transit of a packet and apply traffic shaping, without emulating complete network devices.

2.6.5 Kollaps

Kollaps [9, 90] is a decentralized network emulator with the objective to emulate large-scale distributed systems while ensuring desired network properties.

Kollaps enables the description of the network topology in a file, converting the described network to rules to be applied by TC [38] (see Section 2.4.3) during experimentation.

To avoid simulating routers and switches, Kollaps rearranges the network using a technique called *network collapsing*, aggregating network properties of multiple links into a single one. This approach reduces the computation power required to find a routing path to the destination while maintaining end-to-end properties. The network rearrangement occurs at the beginning of the experiment, reducing computing costs by calculating the shortest routing path to be utilized during the experiment.

Kollaps leverages TC to apply end-to-end link properties between hosts such as bandwidth limitations, latency, packet loss and jitter. Since Kollaps simplifies the network at the end hosts, requires real-time modification of the TC rules. Such work is done by an emulation core, which keeps the TC rules consistent with the network topology and updates the configured properties to emulate path congestion. In order to do so, the emulation core observes the bandwidth usage, and share it with other emulation cores to apply a fair share of bandwidth for connections in the same routing path.

With the presence of an emulation core in each application container running, Kollaps provides the definition of dynamic behavior by configuring it in the network topology description. When the experiment is running, the emulation core is responsible to apply the dynamic behavior defined by the practitioner. As mentioned previously, the dynamic behavior is also preprocessed before the start of the experiment.

Kollaps leverages technologies such as Docker [24] and Kubernetes [71] to simplify the network model, since Docker enables the definition of a network, simulating containers as individual machines with their individual properties such as their own network configurations. Using Docker allows any application to run in a containerized environment, thus allowing any arbitrary code to be run in the configured network topology without requiring to modify applications in the configured environment made by the experimenter. Leveraging the usage of Kubernetes allows Kollaps to become scalable to multiple machines while abstracting the complexity of managing communication between multiple machines.

As described previously, Kollaps contains the necessary structure to be capable of emulating large network topologies, but due to the technique employed to collapse the network, loses the ability to dynamically modify a network during experiment. The authors try to mitigate this limitation by computing the network properties of the dynamic changes defined in the network topology description before the execution of the experiment. The avoidance of recomputing network properties during runtime means each emulation core will have to contain every dynamic behavior of the network topology. Although for simple networks the storage of extra link properties can be negligible, the problem arises with very dynamic network topologies, leading to scalability issues, due to every emulation core storing that information. Since the emulation core is responsible for applying the network constraints defined by the practitioner, it requires the coordination between multiple containers to compute at runtime the bandwidth available in the routing path, and in very large networks with lots of network devices and container applications, such coordination can become a bottleneck when considering large distributed systems,

capable of generating a modest quantity of application traffic.

Another limitation due to network collapsing, is the lack of support for multipath routing due to applying the shortest path routing algorithm to compute the shortest path between end hosts and computing the appropriate network restrictions.

2.6.6 Discussion

Throughout the previous sections, we presented multiple network emulators and discussed their objective and implementation. The objective of this work is the proper emulation of a large network topology consisting of hundreds to thousands of nodes. As such, we argue that the presented network emulators fail to solve such problem, but provide possible approaches to solving some of our challenges.

In DummyNet [7], the emulator is designed to test communication protocols without affecting the application. Such approach leads us to utilize similar tools such as TC to be capable of modifying the network behavior and introduce bandwidth limitations, latency, jitter, and packet loss.

In ModelNet [15], the network emulator demonstrates the capability of emulating a network topology. Such capability comes from the usage of graphs to simplify the network topology and aggregate network properties, simplifying execution. Unfortunately, its architecture poses some scalability issues, but as such, prompts us to search for tools such as Docker [24] and Kubernetes [71] to solve scalability issues and surpass the limitations of a single machine.

In CrystalNet [11], their focus is on testing configuration on network devices. By emulating the control-plane of the network, they were able to validate the behavior of the network devices when subjected to inputs from the emulated network. Since their focus is the control-plane, they do not provide support to run applications and test their behavior. They do demonstrate the usage of virtualization to run unmodified version of the network devices.

In Mininet [26], their focus is the emulation of SDNs to emulate the network topology of a data-center. Such approach lead us to consider the usage of SDNs and NFVs to implement the network topology of our emulator, but since the full emulation of network devices poses scalability issues, they are discarded.

Kollaps [9], similarly to ModelNet, is capable of emulating a network topology. It utilizes recent technologies such as Docker and Kubernetes to employ unmodified application code into the emulated network. They utilize a ModelNet like approach by also preprocessing a network topology defined in a file, and translate to a simpler network model, by collapsing the properties, and applying them at the end hosts. Such approach provides better scalability since it avoids running full network devices, but is limited in flexibility, since it becomes impossible to introduce dynamic behavior during experiment which as not been configured previously. Such approach guides us to search for an alternative solution capable of collapsing the network properties and minimizing

computation overhead, while being flexible enough to be capable of configuring it during experimentation without jeopardizing the accuracy of the emulator.

2.7 Summary

In this chapter, we have delved into many subjects related to networking, such as existing tools and techniques, and related work. We first began by explaining simple network concepts such as the definition of a network, its constitution and how it is divided according to the OSI Model and why it is relevant to our work. Then we approached possible solutions to deploy a desired network topology by resorting to research testbeds or the Cloud and explaining why such an approach is not feasible, leading us to find other solutions such as Emulation and Simulation. We explained the concept of Emulation and Simulation and its relevancy in a network context. After that, we described existing tools capable of modifying network properties such as XDP and eBPF, TC, and Netfilter. In a higher abstraction of the network, we presented SDNs and NFVs which represent the virtualization of physical devices present in a network topology and finally discuss approaches to run unmodified application code. We covered the concept of virtualization and containerization, finishing the section by addressing scalability techniques and reflecting on the benefits of the mentioned tools and techniques to our proposed solution. After that we discussed existing network simulators and their limitations, followed by the existing network emulators and their relation with our goal.

In conclusion, the tools discussed and the various approaches taken from existing network emulators, provide us with a strong foundation on how to tackle some problems and challenges that may arise during the development of the network emulator.

In the following chapter, we will reflect on our work in the following months. We describe a possible approach to proposed contributions set in the Chapter 1 while mentioning challenges we may face during development as well as the evaluation required to validate our solution. We close the chapter by providing a schedule delimiting the tasks needed to achieve our solution.

3

Future Work

As presented in Chapter 1, we plan to address the problem of emulating large scale systems operating over large network topologies without the need for dedicated machines. Our work consists in creating a network emulator capable of abstracting a real network topology to its most essential components. In our proposed solution, we expect to provide the following contributions: emulating end-to-end properties of a network such as bandwidth limits, latency, jitter, packet loss and path congestion of complex and heterogeneous networks; running arbitrary and real unmodified application code by leveraging containerization technologies; enabling a testbed with the ability to change network properties at runtime during experiments; scaling the component of the emulator that manages the network emulator into multiple machines to surpass the hardware limitations of a single machine.

In the following sections, we will begin discussing our planned approach of implementing the network emulator leveraging on some presented tools in Chapter 2 and a possible approach to achieve each identified contribution mentioned previously (Section 3.1); Section 3.2 discusses possible challenges we will face in the development of the emulator; Section 3.3 presents the methodology we will use to ensure the expected solution provides accurate results and emulates correctly a large network topology; Finally, in Section 3.4, we provide a development schedule for the following months.

3.1 Proposed Solution

In Chapter 2, we have presented tools and techniques to help in developing the proposed network emulator. Such tools have been shown to be capable of manipulating network properties. Tools such as TC and eBPF have been used in the design and implementation of other network emulators, but such emulators were designed to solve other concrete problems and are not capable of emulating large network topologies while providing real-time network reconfiguration. Figure 3.1 illustrates an initial architecture of our network emulator.

As depicted in the Figure 3.1, at a high level, we will be employing containerization

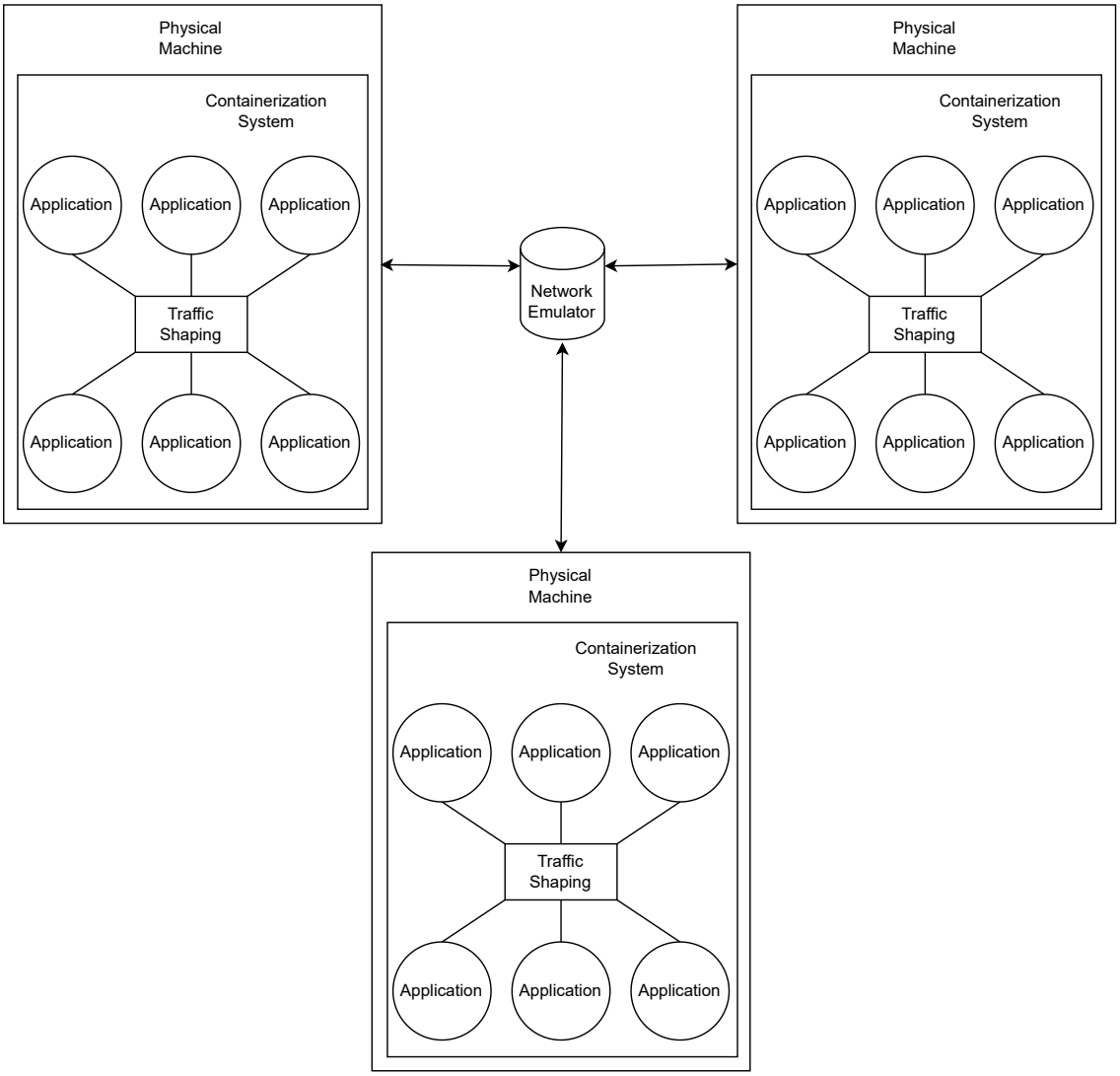

Figure 3.1: Possible architecture of the Network Emulator

orchestration tools such as Kubernetes [71] or Docker Swarm [72] for container orchestration capabilities between multiple physical machines. These machines will be responsible for operating large scale systems on an emulated network. For each machine, besides executing application code, a traffic shaping component will be present, responsible for applying traffic shaping rules to packets in the network to ensure the configured properties are respected. At a central point, a component responsible for the deployment and execution of large distributed systems, inter-machine communication, and configuring on a machine-to-machine basis the traffic shaping component to provide the adequate network properties. Such architecture poses some challenges, and in the following sections we identified possible challenges we might face and discuss a possible approach to solve them. We also identify other challenges which we will tackle during the implementation phase.

3.1.1 Integration of application code

One of our expected contributions is the ability to run application code in the emulator without requiring the modification of the source code or application logic. As such, the approach we will be using is the utilization of containerization (see Section 2.4.7), since it provides a lightweight environment and provides application isolation in the operating system. The execution of the application in a containerized environment provides the network emulator the capability to modify container properties, such as its network capabilities, in a way that is transparent to the application itself. Such flexibility will provide the management and manipulation of the traffic going to the container and apply the network model defined by the experimenter.

3.1.2 End-to-End Network Properties

To simulate end-to-end network properties, we plan to utilize tools such as TC, eBPF, and Netfilter since it is currently unclear the precise limitations of these tools, we will develop the emulator in a modular way and experiment will all those alternatives. Furthermore, we also plan to devise a solution that combines these tools in a clever way, i.e., to achieve different goals in the emulation. Since Docker provides the capability to manipulate network properties without interfering with the application, we can employ TC to configure the network interface of each individual container to configure network properties such as bandwidth limits, jitter, packet loss, and latency. eBPF on the other hand, allows the ability to process packets received and, as such, providing extra flexibility in communication management, since we can process packets before they reach the application. Another use of eBPF is the registration of packet metrics, which will provide data to validate the accuracy of the experimentation. Utilizing Netfilter will allow the application of forwarding rules to the traffic passed through the containers.

3.1.3 Scalability

Since one of our objectives is to emulate large distributed networks and systems, scalability is an essential requirement. As such, our approach to scalability leads us to reducing computing overhead. To achieve this, we plan the deployment and centralization of the network traffic in a single (logical) component allowing resource sharing when applying traffic shaping as opposed to the approach used in Kollaps [9], which utilizes an emulation core in each container to ensure the network properties are configured. We argue the overhead will be minimized in our approach because the shaping component will manage the network limitations without coordinating with another shaping component, and instead, contact the central component to be forwarded to the correct traffic shaping component present in a different machine. By utilizing a central component aware of the network topology, enables and optimizes distribution of load and traffic rules between machines, restricting an individual machine to be configured accordingly with a subset of

network devices. Therefore, each machine will have a reduced number of traffic rules to be applied, improving performance, and allowing a more scalable environment. Regarding inter-machine traffic, the shaping component can forward traffic destined to another machine through the central component, thus redirecting to the appropriate machine, avoiding complete awareness of the network topology in each individual machine.

To promote scalability, the description of the network topology can contain an extra field, capable of providing the network emulator an extra insight by providing an affinity attribute to network devices leading to the network emulator deploying them in the same machine, thus improving performance by avoiding inter-machine communication, and avoiding delegating of traffic to the central component.

3.2 Challenges

Previously, we have presented possible challenges which we will tackle during the execution of our planned work. Unfortunately, there are challenges we will face during the development of the network emulator, which depend on other aspects, which we now briefly discuss.

3.2.1 Configuring a Network Topology

The first challenge we will face regards the configuration of the network in our emulator. To do so, we will need to provide the definition of network devices present in the network and describe their behavior and limitations. Then, according to the defined behavior we will need to translate those configurations into correct configurations in other tools such as eBPF and TC, ensuring the network properties are satisfied. Initially, when the emulated network is empty or has a lower number of devices, such translation may be simple to execute, but the problem arises when there are many network devices, leading to the modification of many containers, which could lead to unoptimized implementations of network properties. A possible approach to the configuration of a network topology would be similar to ModelNet [15] and Kollaps [9], converting a network topology described in a markup language such as YAML [91] or XML [92] and creating a graph of the network. Executing the shortest path algorithm, such as Djisktra's algorithm [93], to compute the shortest path between two end hosts and apply the corresponding network properties. With the shortest paths calculated, the network emulator, according to the given network topology, assigns clusters of network devices who it thinks are closely related, and deploys them in a machine, prepopulating the traffic shaping component with the computed network properties.

3.2.2 Managing the correct state between machines

Another challenge we foresee is the coordination of our network emulator when running on multiple machines and ensuring an accurate emulated network. Our network emulator

will need to be capable of coordinating between different machines to ensure the correct operation of the network, and be aware of changes introduced by the experimenter. This is due to the traffic shaping component needing to be configured according to the existing application containers running on different machines, since it needs to apply different traffic shaping rules for different (pairs of) communicating application instances on different containers. Since the central component contains the full network topology to properly assign each machine with their own traffic shaping rules, we can leverage such data structure to additionally store the subset of network devices each machine is emulating. By utilizing this centralized approach, individual machines can request for additional traffic shaping rules when handling communications with another machine. The central component, when receiving such request can decide on two possible outcomes, one where it provides network properties to apply or to set the forwarding of the communication to the central component, in order to redirect it to another traffic shaping component. To be able to configure an emulated network and test large distributed systems, the experimenters are required to communicate with the central component of our network emulator, who will then issue traffic shaping rules to the machines. Since the machines cannot reconfigure themselves, the traffic shaping component will always execute traffic shaping rules provided by the central component. The problem arises when introducing severe tempering to the emulated network. To accommodate for such eventuality, each traffic shaping component will store temporarily affected packets, and after the convergence of the network to a correct state, apply the new traffic shaping rules.

3.2.3 Dynamic behavior

The last challenge we foresee is the implementation of dynamic behavior in the network topology. As opposed to Kollaps [9], which utilizes a static approach by computing beforehand every behavior in the network before execution, limiting the capability of modifying the experimentation when executing, we plan to be able to modify the network in real time and be capable of introducing random events in the network. This is useful to support more interactive debugging of systems by the experimenters. The difficulty of this operation comes due to the overhead introduced when modifying the network, leading the network emulator to recompute new paths and modifying the network topology without influencing the flow of information.

As a starting point, we identify two main approaches to providing dynamic behavior to a network topology, the modification of existing network properties or the addition, removal or unavailability of a particular network device. In the case of reconfiguring existing network properties, due to the nature of our approach in maintaining a smaller subset of known network properties in each individual machine, there is the advantage of recomputing in a localized zone the new values, without requiring intervention in the whole system. In regard to the addition, removal or temporary unavailability of a network device, a different approach is required. Joining a new network device leads the central

component to recompute the appropriate network properties, and insert them into the machine whose existing number of connections and network devices affected, with the objective of reducing inter-machine communication. The removal of a network device would be a simple operation since the network device is already inserted in the network, only requiring the central component to recompute new values, and apply the new traffic shaping rules to the affected machine. The temporary unavailability of a network device or a connection is a simple operation and can be done locally in a machine simply by finding the next possible path in the network.

3.3 Evaluation

To evaluate our proposed solution, we plan on asserting the performance of our network emulator in three main axes. The first axis is to evaluate the accuracy of our network emulator by registering the time it takes from one packet to reach its destination, and then validate it according to the network topology defined.

The second axis focuses on comparing with different solutions such as Mininet [26], Maxinet [8], and Kollaps [9] and utilizing the same network topology to compare the accuracy of the mentioned emulators against ours. We will also measure the accuracy of our network emulator against the other emulators when employing a network topology with different sizes.

Finally, we plan to utilize existing distributed systems available such as Cassandra [70] and demonstrate the accuracy of our emulation by running workloads and modifying the network during the experiment. Due to its ring like topology and utilization of consensus algorithms, it is a great candidate to test our proposed solution in the best conditions possible, since the central component can aggregate Cassandra instances belonging to the same ring and deploy them in a single machine, and due to consensus generate local traffic. Cassandra contains replication strategies, namely "Datacenter Aware"replication strategy resulting in communication between different Cassandra rings. With this particular network topology, we can evaluate the impact of traffic shaping locally and inter-machine traffic shaping. By introducing dynamic behavior in our emulated network, we evaluate the impact in performance, namely throughput and latency during experiment.

On the other hand, we plan to resort to The InterPlanetary File System [94, 95] (IPFS) to demonstrate the accuracy of our emulation in its worst conditions. IPFS is a peer-to-peer distributed file system with the objective of creating a shared file system between connected devices. Due to its peer-to-peer nature, the access of content in the network can involve any device connected. Therefore, the traffic generated in a IPFS network is completely arbitrary, leading to the impossibility of organizing pairs of IPFS nodes into a single machine, therefore negating the benefits of utilizing local traffic shaping, increasing the usage of the central component to forward traffic to the correct machine.

3.4 Development Cycle

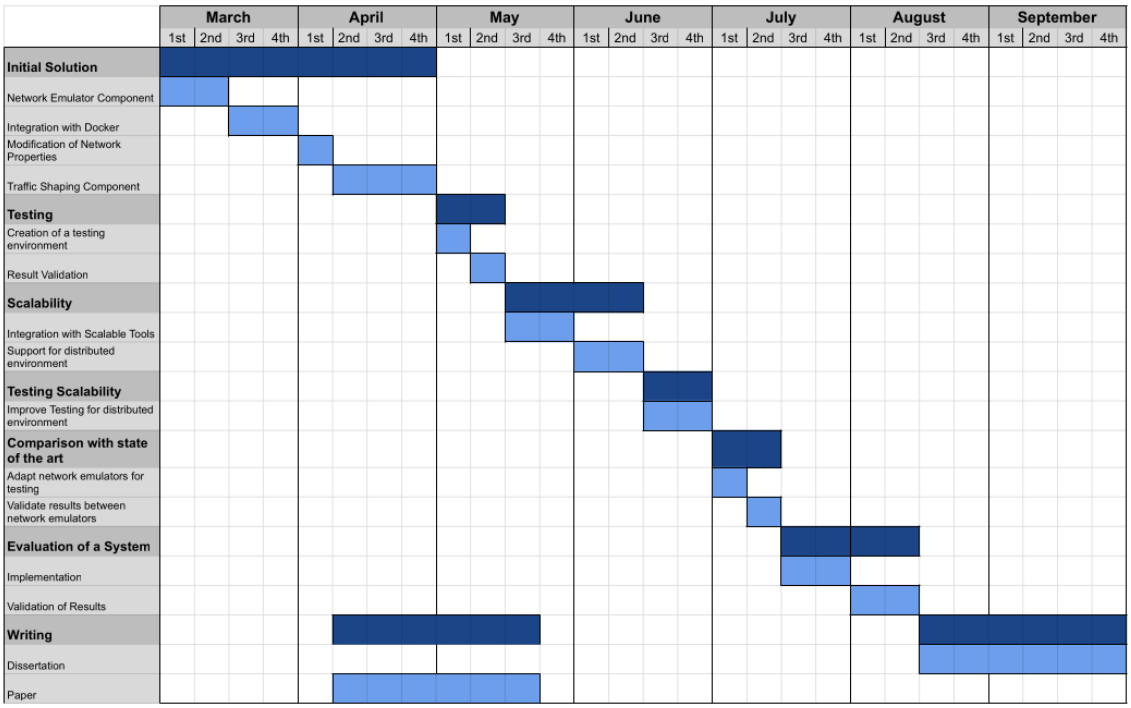

Figure 3.2: Gantt chart delimiting the expected work schedule

In Figure 3.2, we present the expected work schedule for the following months. We divide our work in seven main sections:

Initial Solution In this phase, we expect to implement the first iteration of the emulator, consisting of accepting a simple network topology and configuring the traffic shaping rules into the traffic shaping component to execute them in the emulated network;

Testing In this phase, we plan to implement the foundation for the extraction of data regarding the network. With this data, we plan to implement a validation mechanism capable of calculating the accuracy of our network according to the configured network topology;

Scalability In this phase, we plan on addressing the scalability of the proposed solution by implementing distributed mechanisms capable of allowing the network emulator to be scaled across different physical machines;

Testing Scalability In this phase, we plan to improve the testing environment to accommodate for the distributed network emulator and test its scalability with larger network topologies and maintain the accuracy of the network emulator;

Comparison with state of the art In this phase, we evaluate our network emulator by comparing network accuracy and scalability against other network emulators;

Evaluation of a System In this phase, we will use a distributed system use case to assert the accuracy of scalability of our network emulator particularly when dynamically modified the network model during experiment;

Writing In this phase, time will be allocated for the writing of the dissertation and a paper.

During the first two phases, we will also be writing a paper to be submitted to a conference.

Bibliography

[1] *Emulab*. Accessed Jan. 2023. URL: https://www.emulab.net/portal/frontpage.php (cit. on pp. 1, 6).

[2] *PlanetLab*. Accessed Jan. 2023. URL: https://planetlab.cs.princeton.edu/tutorial.html (cit. on pp. 1, 6).

[3] *Enterprises using cloud computing*. Accessed Fev. 2023. URL: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises#Enterprises.E2.80.99_dependence_on_cloud_computing (cit. on p. 1).

[4] *Cloud computing services: Microsoft Azure*. Accessed Jan. 2023. URL: https://azure.microsoft.com/ (cit. on pp. 1, 6).

[5] *Google Cloud Platform*. Accessed Jan. 2023. URL: https://cloud.google.com/ (cit. on pp. 1, 6).

[6] *cloud computing services - amazon web services*. Accessed Jan. 2023. URL: https://signin.aws.amazon.com/ (cit. on pp. 1, 6).

[7] L. Rizzo. "Dummynet: A Simple Approach to the Evaluation of Network Protocols". In: 27.1 (1997), 31–41. ISSN: 0146-4833. DOI: 10.1145/251007.251012. URL: https://doi.org/10.1145/251007.251012 (cit. on pp. 1, 6, 8, 19, 23).

[8] P. Wette et al. "MaxiNet: Distributed emulation of software-defined networks". In: *2014 IFIP Networking Conference*. 2014, pp. 1–9. DOI: 10.1109/IFIPNetworking.2014.6857078 (cit. on pp. 1, 8, 12, 21, 30).

[9] P. Gouveia et al. "Kollaps: Decentralized and Dynamic Topology Emulation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387540. URL: https://doi.org/10.1145/3342195.3387540 (cit. on pp. 1, 6–8, 21, 23, 27–30).

[10] Nsnam. *NS-3 Network Simulator*. URL: https://www.nsnam.org/ (cit. on pp. 1, 6, 7, 17).

[11] H. Liu et al. "CrystalNet: Faithfully Emulating Large Production Networks". In: *SOSP '17 Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 599–613. ISBN: 978-1-4503-5085-3. URL: https://www.microsoft.com/en-us/research/publication/crystalnet-faithfully-emulating-large-production-networks/ (cit. on pp. 1, 7, 20, 23).

[12] L. S. Vailshery. *Challenges surrounding SAAS apps worldwide 2021*. Accessed Fev. 2023. URL: https://www.statista.com/statistics/1267901/saas-apps-biggest-challenges-organizations-worldwide/ (cit. on p. 6).

[13] Accessed Jan. 2023. URL: https://infrastructuremap.microsoft.com/ (cit. on p. 6).

[14] A. Varga and R. Hornig. "An overview of the OMNeT++ simulation environment". In: 2008, p. 60. DOI: 10.1145/1416222.1416290 (cit. on pp. 6, 7, 16).

[15] A. Vahdat et al. "Scalability and Accuracy in a Large-Scale Network Emulator". In: *SIGOPS Oper. Syst. Rev.* 36.SI (2003), 271–284. ISSN: 0163-5980. DOI: 10.1145/844128.844154. URL: https://doi.org/10.1145/844128.844154 (cit. on pp. 6, 19, 23, 28).

[16] J. Banks et al. *Discrete-event System Simulation*. Prentice-Hall international series in industrial and systems engineering. Pearson Prentice Hall, 2005. ISBN: 9780131446793. URL: https://books.google.pt/books?id=CWZRAAAAMAAJ (cit. on p. 6).

[17] D. Dugaev and E. Siemens. "A Wireless Mesh Network NS-3 Simulation Model: Implementation and Performance Comparison With a Real Test-Bed". In: 2019. DOI: 10.13142/kt10002.01 (cit. on pp. 6, 17).

[18] G. Carneiro, H. Fontes, and M. Ricardo. "Fast prototyping of network protocols through ns-3 simulation model reuse". In: *Simulation Modelling Practice and Theory* 19 (2011-10), pp. 2063–2075. DOI: 10.1016/j.simpat.2011.06.002 (cit. on pp. 6, 17).

[19] R. Chertov, S. Fahmy, and N. B. Shroff. "Fidelity of Network Simulation and Emulation: A Case Study of TCP-Targeted Denial of Service Attacks". In: *ACM Trans. Model. Comput. Simul.* 19.1 (2009). ISSN: 1049-3301. DOI: 10.1145/1456645.1456649. URL: https://doi.org/10.1145/1456645.1456649 (cit. on pp. 7, 18).

[20] E. Lochin, T. Pérennou, and L. Dairaine. "When should I use network emulation?" In: *annals of telecommunications - annales des télécommunications* 67.5 (2012), pp. 247–255. ISSN: 1958-9395. DOI: 10.1007/s12243-011-0268-5. URL: https://doi.org/10.1007/s12243-011-0268-5 (cit. on pp. 7, 8).

[21] P. Bailis and K. Kingsbury. "The Network is Reliable: An Informal Survey of Real-World Communications Failures". In: *Queue* 12.7 (2014), 20–32. ISSN: 1542-7730. DOI: 10.1145/2639988.2655736. URL: https://doi.org/10.1145/2639988.2655736 (cit. on p. 7).

[22] J. Postel. *Transmission control protocol*. Accessed Jan. 2023. URL: https://www.rfc-editor.org/rfc/rfc793 (cit. on p. 7).

[23] M. A. To, M. Cano, and P. Biba. "DOCKEMU – A Network Emulation Tool". In: *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*. 2015, pp. 593–598. DOI: 10.1109/WAINA.2015.107 (cit. on p. 7).

[24] *Accelerated, containerized application development*. Accessed Jan. 2023. URL: https://www.docker.com/ (cit. on pp. 7, 14, 15, 22, 23).

[25] S. Hemminger. "Network emulation with NetEm". In: *Linux Conf Au* (2005) (cit. on pp. 8, 9).

[26] K. Kaur, J. Singh, and N. Ghumman. "Mininet as Software Defined Networking Testing Platform". In: 2014 (cit. on pp. 8, 12, 21, 23, 30).

[27] T. Høiland-Jørgensen et al. "The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel". In: CoNEXT '18. Association for Computing Machinery, 2018, 54–66. ISBN: 9781450360807. DOI: 10.1145/3281411.3281443. URL: https://doi.org/10.1145/3281411.3281443 (cit. on p. 8).

[28] M. A. M. Vieira et al. "Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications". In: 53.1 (2020). ISSN: 0360-0300. DOI: 10.1145/3371038. URL: https://doi.org/10.1145/3371038 (cit. on p. 9).

[29] S. McCanne and V. Jacobson. "The BSD Packet Filter: A New Architecture for User-Level Packet Capture". In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX'93. USENIX Association, 1993, p. 2 (cit. on p. 9).

[30] *BPF: Introduce bpf_tail_call() helper*. URL: https://lwn.net/Articles/645169/ (cit. on p. 9).

[31] S. Becker et al. *Network Emulation in Large-Scale Virtual Edge Testbeds: A Note of Caution and the Way Forward*. 2022. DOI: 10.48550/ARXIV.2208.05862. URL: https://arxiv.org/abs/2208.05862 (cit. on p. 9).

[32] S. Miano et al. "A Framework for eBPF-Based Network Functions in an Era of Microservices". In: *IEEE Transactions on Network and Service Management* 18.1 (2021), pp. 133–151. DOI: 10.1109/TNSM.2021.3055676 (cit. on pp. 9, 13).

[33] M. Abranches et al. "Efficient Network Monitoring Applications in the Kernel with eBPF and XDP". In: *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2021, pp. 28–34. DOI: 10.1109/NFV-SDN53031.2021.9665095 (cit. on p. 9).

[34] C. Liu et al. "A protocol-independent container network observability analysis system based on eBPF". In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. 2020, pp. 697–702. DOI: 10.1109/ICPADS51040.2020.00099 (cit. on p. 9).

[35] J. Levin. "ViperProbe: Using eBPF Metrics to Improve Microservice Observability". In: 2020 (cit. on p. 9).

[36] Accessed Jan. 2023. URL: https://almesberger.net/cv/papers/tcio8.pdf (cit. on p. 10).

[37] *1. introduction to linux traffic control*. Accessed Jan. 2023. URL: https://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html (cit. on p. 10).

[38] Accessed Jan. 2023. URL: https://man7.org/linux/man-pages/man8/tc.8.html (cit. on pp. 10, 21).

[39] *The netfilter.org project*. Accessed Jan. 2023. URL: https://www.netfilter.org/ (cit. on p. 11).

[40] A. Jones. *Netfilter and IPTables: A Structural Examination*. 2004 (cit. on p. 11).

[41] Accessed Jan. 2023. URL: https://man7.org/linux/man-pages/man8/arptables-nft.8.html (cit. on p. 11).

[42] Accessed Jan. 2023. URL: https://man7.org/linux/man-pages/man8/iptables.8.html (cit. on p. 11).

[43] Accessed Jan. 2023. URL: https://linux.die.net/man/8/ip6tables (cit. on p. 11).

[44] *NFTABLES: A new packet filtering engine*. Accessed Jan. 2023. URL: https://lwn.net/Articles/324989/ (cit. on p. 11).

[45] *The return of nftables*. Accessed Jan. 2023. URL: https://lwn.net/Articles/564095/ (cit. on p. 11).

[46] *Nftables Wiki page*. Accessed Jan. 2023. URL: https://wiki.nftables.org/wiki-nftables/index.php/Main_Page (cit. on p. 11).

[47] D. Plummer. *An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*. RFC 826 (Internet Standard). RFC. Updated by RFCs 5227, 5494. Fremont, CA, USA: RFC Editor, 1982-11. DOI: 10.17487/RFC0826. URL: https://www.rfc-editor.org/rfc/rfc826.txt (cit. on p. 11).

[48] J. Postel. *Internet Protocol*. RFC 791 (Internet Standard). RFC. Updated by RFCs 1349, 2474, 6864. Fremont, CA, USA: RFC Editor, 1981-09. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/rfc/rfc791.txt (cit. on p. 11).

[49] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Draft Standard). RFC. Obsoleted by RFC 8200, updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112. Fremont, CA, USA: RFC Editor, 1998-12. DOI: 10.17487/RFC2460. URL: https://www.rfc-editor.org/rfc/rfc2460.txt (cit. on p. 11).

[50] M. Boye. "Netfilter Connection Tracking and NAT Implementation". In: 2012 (cit. on p. 11).

[51] P. Sutter. *Benchmarking nftables*. Accessed Jan. 2023. URL: https://developers.redhat.com/blog/2017/04/11/benchmarking-nftables#the_first_test (cit. on p. 11).

[52] A. Tanenbaum, D. Wetherall, and N. Feamster. "Chapter 5 - The Network Layer". In: *Computer Networks, EBook, Global Edition*. 6th ed. Pearson Education, Limited, 2021, 435–441 (cit. on p. 11).

[53] D. Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. DOI: 10.1109/JPROC.2014.2371999 (cit. on p. 11).

[54] P. Goransson, T. Culver, and C. Black. "Chapter 2 Why SDN?" In: *Software defined networks: A comprehensive approach*. Morgan Kaufmann Publishers, 2017, 21–35 (cit. on p. 11).

[55] N. McKeown et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (2008), 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: https://doi.org/10.1145/1355734.1355746 (cit. on pp. 12, 21).

[56] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach*. 6th. 2019. ISBN: 0128103515. URL: https://book.systemsapproach.org/ (cit. on p. 12).

[57] E. Rosen, A. Viswanathan, and R. Callon. *Multiprotocol Label Switching Architecture*. RFC 3031 (Proposed Standard). RFC. Updated by RFCs 6178, 6790. Fremont, CA, USA: RFC Editor, 2001. DOI: 10.17487/RFC3031. URL: https://www.rfc-editor.org/rfc/rfc3031.txt (cit. on p. 12).

[58] *Production quality, Multilayer Open Virtual Switch*. Accessed Jan. 2023. URL: https://www.openvswitch.org/ (cit. on p. 12).

[59] B. Pfaff et al. "The Design and Implementation of Open VSwitch". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. USENIX Association, 2015, 117–130. ISBN: 9781931971218 (cit. on p. 12).

[60] K. Gray and T. D. Nadeau. "Chapter 1 - Network Function Virtualization". In: *Network Function Virtualization*. Ed. by K. Gray and T. D. Nadeau. Boston: Morgan Kaufmann, 2016, pp. 1–18. ISBN: 978-0-12-802119-4. DOI: https://doi.org/10.1016/B978-0-12-802119-4.00001-8. URL: https://www.sciencedirect.com/science/article/pii/B9780128021194000018 (cit. on p. 13).

[61] R. Mijumbi et al. "Network Function Virtualization: State-of-the-Art and Research Challenges". In: *IEEE Communications Surveys & Tutorials* 18.1 (2016), pp. 236–262. DOI: 10.1109/COMST.2015.2477041 (cit. on p. 13).

[62] M. Portnoy. *Virtualization Essentials*. 1st. USA: SYBEX Inc., 2012. ISBN: 1118176715 (cit. on p. 13).

[63] *Container and virtualization tools*. Accessed Jan. 2023. URL: https://linuxcontainers.org/ (cit. on pp. 14, 15, 21).

[64] *cgroups(7) — Linux manual page*. Accessed Jan. 2023. URL: https://man7.org/linux/man-pages/man7/cgroups.7.html (cit. on p. 14).

[65] *namespaces(7) — Linux manual page*. Accessed Jan. 2023. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html (cit. on p. 14).

[66] J "Berggren and J. Karlsson. ""Differences in performance between containerization & virtualization : With a focus on HTTP requests"". In: *"Dissertation"* () (cit. on p. 14).

[67] B. Bashari Rad, H. Bhatti, and M. Ahmadi. "An Introduction to Docker and Analysis of its Performance". In: *IJCSNS International Journal of Computer Science and Network Security* 173 (2017-03), p. 8 (cit. on p. 14).

[68] S. Soltesz et al. "Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors". In: *SIGOPS Oper. Syst. Rev.* 41.3 (2007), 275–287. ISSN: 0163-5980. DOI: 10.1145/1272998.1273025. URL: https://doi.org/10.1145/1272998.1273025 (cit. on p. 14).

[69] S. Shirinbab, L. Lundberg, and E. Casalicchio. "Performance evaluation of containers and virtual machines when running Cassandra workload concurrently". In: *Concurrency and Computation: Practice and Experience* 32.17 (2020), e5693. DOI: https://doi.org/10.1002/cpe.5693. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5693. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5693 (cit. on p. 14).

[70] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (2010), 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: https://doi.org/10.1145/1773912.1773922 (cit. on pp. 14, 30).

[71] *Production-grade container orchestration*. Accessed Jan. 2023. URL: https://kubernetes.io/ (cit. on pp. 15, 20, 22, 23, 26).

[72] *Swarm mode overview*. Accessed Fev. 2023. 2023. URL: https://docs.docker.com (cit. on pp. 15, 26).

[73] N. Marathe, A. Gandhi, and J. M. Shah. "Docker Swarm and Kubernetes in Cloud Computing Environment". In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. 2019, pp. 179–184. DOI: 10.1109/ICOEI.2019.8862654 (cit. on p. 15).

[74] I. M. A. Jawarneh et al. "Container Orchestration Engines: A Thorough Functional and Performance Comparison". In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8762053 (cit. on p. 15).

[75] A. Varga. *Simulation Manual - Version 6.0*. Accessed Jan. 2023. URL: https://doc.omnetpp.org/omnetpp/manual/index.html (cit. on p. 16).

[76] K. Wehrle, M. Güneş, and J. Gross, eds. *Modeling and Tools for Network Simulation*. Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-3-642-12331-3. URL: https://doi.org/10.1007%2F978-3-642-12331-3 (cit. on pp. 16, 17).

[77] F. Estevez et al. "Enabling Validation of IEEE 802.15.4 Performance through a New Dual-Radio Omnet++ Model". In: *Elektronika ir Elektrotechnika* 22 (2016). DOI: 10.5755/j01.eie.22.3.15321 (cit. on p. 16).

[78] M. Stoffers et al. *Enabling Distributed Simulation of OMNeT++ INET Models*. 2014. DOI: 10.48550/ARXIV.1409.0994. URL: https://arxiv.org/abs/1409.0994 (cit. on p. 16).

[79] J. Oladipo, M. C. duPlessis, and T. B. Gibbon. "Implementation and validation of an Omnet++ optical burst switching simulator". In: *2017 International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN)*. 2017, pp. 1–6. DOI: 10.23919/PEMWN.2017.8308024 (cit. on p. 16).

[80] L. Kamarudin et al. "Review and Modeling of Vegetation Propagation Model for Wireless Sensor Networks Using Omnet++". In: *Network Applications, Protocols and Services, International Conference on* 0 (2010-09), pp. 78–83. DOI: 10.1109/NETAPPS.2010.21 (cit. on p. 16).

[81] N. S. Prashanth. "OpenFlow Switching Performance using Network Simulator - 3". In: *Dissertation* (2016) (cit. on p. 17).

[82] L. Alberro et al. "Experimenting with Routing Protocols in the Data Center: An ns-3 Simulation Approach". In: *Future Internet* 14.10 (2022). ISSN: 1999-5903. DOI: 10.3390/fi14100292. URL: https://www.mdpi.com/1999-5903/14/10/292 (cit. on p. 17).

[83] A. Montresor and M. Jelasity. "PeerSim: A scalable P2P simulator". In: *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*. 2009, pp. 99–100. DOI: 10.1109/P2P.2009.5284506 (cit. on pp. 17, 18).

[84] I. Kazmi and S. F. Y. Bukhari. "PeerSim: An Efficient & Scalable Testbed for Heterogeneous Cluster-based P2P Network Protocols". In: *2011 UkSim 13th International Conference on Computer Modelling and Simulation*. 2011, pp. 420–425. DOI: 10.1109/UKSIM.2011.86 (cit. on p. 17).

[85] *PeerSim: A peer-to-peer simulator*. Accessed Fev. 2023. URL: https://peersim.sourceforge.net/ (cit. on p. 17).

[86] J. Leitão, J. Pereira, and L. Rodrigues. "HyParView: a membership protocol for reliable gossip-based broadcast". In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Edinburgh, UK, 2007, pp. 419–429 (cit. on p. 17).

[87] J. Leitão. "Gossip-Based Broadcast Protocols". MA thesis. Faculdade de Ciências da Universidade de Lisboa, 2007 (cit. on p. 17).

[88] A. Quereilhac et al. "Automating Ns-3 Experimentation in Multi-Host Scenarios". In: *Proceedings of the 2015 Workshop on Ns-3*. WNS3 '15. Association for Computing Machinery, 2015, 1–8. ISBN: 9781450333757. DOI: 10.1145/2756509.2756513. URL: https://doi.org/10.1145/2756509.2756513 (cit. on p. 18).

[89] Accessed Jan. 2023. URL: https://community.cloudflare.com/t/learning-center-what-is-gre-tunneling-how-gre-protocol-works/347586 (cit. on p. 21).

[90] S. Amaro. MA thesis. 2021. URL: https://fenix.tecnico.ulisboa.pt/cursos/meic-t/dissertacao/1972678479055152 (cit. on p. 21).

[91] *The official YAML web site*. Accessed Fev. 2023. URL: https://yaml.org/ (cit. on p. 28).

[92] *Extensible markup language (XML)*. Accessed Fev. 2023. URL: https://www.w3.org/XML/ (cit. on p. 28).

[93] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (1959), pp. 269–271. ISSN: 0945-3245. DOI: 10.1007/BF01386390. URL: https://doi.org/10.1007/BF01386390 (cit. on p. 28).

[94] J. Benet. "IPFS - Content Addressed, Versioned, P2P File System". In: (2014). DOI: 10.48550/ARXIV.1407.3561. URL: https://arxiv.org/abs/1407.3561 (cit. on p. 30).

[95] P. Á. Costa, J. Leitão, and Y. Psaras. "Studying the workload of a fully decentralized Web3 system: IPFS". In: (2022). DOI: 10.48550/ARXIV.2212.07375. URL: https://arxiv.org/abs/2212.07375 (cit. on p. 30).