André de Oliveira Atalaia

Bachelor in Computer Science and Enginnering

Support for Scalable Emulation of Distributed Systems

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

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

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

July, 2021

Abstract

Distributed Systems are becoming increasingly prevalent in our daily lives and their relevance has never been so prominent in everyday activities. This naturally leads systems to increase in complexity and dimension, and users to have much higher expectations about the performance and correctness of these systems. This creates a significant pressure for conducting systematic and detailed performance assessments of novel proposals through experimentation.

Conducting experiments on distributed systems can be challenging, since one has to launch multiple processes and retain control over them. While you can manually run and control experiments in small distributed systems, such possibility becomes daunting when large-scale distributed systems such as peer-to-peer systems or recent proposes in the context of Web 3.0. To be able to extract key performance metrics, which depend heavily on the underlying network, and since the networks where these systems operate are many times unstable, this can lead to inconclusive results. Moreover, when considering systems operating at a global scale, it is extremely hard to have access to competitive resources in many locations, to achieve realistic conditions. To emulate the underlying network realistically we can use linux tc to shape links between the end hosts to exhibit target properties. To do this for large systems with thousands of nodes is however, very complex.

In this work, we propose a tool that assists in defining and executing experiments for large scale distributed systems, using container technology and network emulation. We propose an architecture where a manager node exists per physical machine, and this manager is responsible for all the application nodes in that physical machine. This responsibility goes from receiving the experiment information (network topology, link properties, dynamic events, such as node failures), to ensuring that every application node receive their corresponding events. Lastly, we plan to evaluate our tool by conducting users tests, and comparing it to state-of-the-art tools. This work is being conducted in collaboration with Protocol Labs, as a means to test their DHT in a realistic environment.

Keywords: Large Scale Distributed systems evaluation, network emulation, container technology

Resumo

Sistemas distribuídos estão cada vez mais presentes no nosso dia-a-dia, e a sua importância nunca esteve tão em destaque nas actividades do dia-a-dia. Naturalmente, isto leva a que os sistemas aumentem a sua complexidade e dimensão, e que os seus utilizadores tenham maiores expectativas sobre o desempenho e exatidão destes sistemas. Isto cria uma pressão significativa para a reliazação de avaliações sistemáticas e detalhadas do desempenho de novas propostas através de experimentação.

A realização de experiências em sistemas pode ser desafiante, uma vez que se tem de lançar múltiplos processos e manter o control sobre estes. Enquanto se pode executar e controlar experiências manualmente em sistemas distribuídos pequenos, tal possibilidade torna-se impossível quando em comparação com sistemas distribuídos de larga escala, tais como sistemas entre-pares (do Inglês *peer-to-peer*, P2P) ou propostas recentes no contexto da Web 3.0. A extração de métricas de desempenho dependem fortemente da rede subjacente, e uma vez que estas redes onde os sistemas operam são muitas vezes instavéis, isto pode levar a resultados inconclusivos. No entanto, quando consideramos sistemas que operam numa escala global, é extremamente difícil ter acesso a recursos competitivos em muitas localizações, para conseguir alcançar condições realistas. Para emular realisticamente a rede subjacente, podemos usar o linux tc para moldar ligações entre os *end hosts* para exibir as propriedades alvo. Para o fazer para grandes sistemas com milhares de nós é, no entanto, muito complexo.

Nesta dissertação, propomos uma ferramenta que auxilia na definição e execução de experiências para sistemas distribuídos de larga escala, utilizando tecnologia de *containers* e emulação de rede. Propomos uma arquitectura onde existe um nó gestor por máquina física, nó este que é responsável por todos os nós de aplicação nessa mesma máquina. Esta responsabilidade vai desde da receção da informação da experiência (topologia de rede, propriedades de ligação, eventos dinâmicos, tais como falhas nos nós), até assegurar que cada nó de aplicação receba os seus eventos correspondentes. Por último, planeamos avaliar a nossa ferramenta através da realização de testes com utilizadores, e da comparação com ferramentas de última geração. Este trabalho está a ser conduzido em colaboração com a Protocol Labs, como forma de testar a sua tabela de dispersão distribuída num ambiente realista.

Palavras-chave: Avaliação de sistemas distribuídos em grande escala, emulação de rede, tecnologia de *containers*

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

Distributed systems have become essential in our daily lives. In broad terms, a distributed system can be defined as a group of processes, many times executing on different machines, that exchange information through a network as to cooperate to achieve a given goal. Due to their increasing relevance, their dimension and complexity have also being increasing, which leads to significant challenges when testing and validating their correctness.

Correctness can be demonstrated by formal methods, enabling to show that under a given set of assumption regarding networks and faults (among others) the system is able to continue to operate and produce correct results. Another aspect that is also relevant is the attained performance, since the expectations, from users are also becoming more and more strict. In this context, one wants to reliably measure key performance indicators, such as latency or throughput, in a realistic execution environment.

Creating realistic conditions, when addressing large-scale distributed system, such as peer-to-peer systems, where with thousands of processes executing in machines scattered throughout the world, with variable network conditions is not an easy task as this environment is hard to create (i.e. access thousands of machines scattered throughout the world). Unfortunately the performance of such systems is many times highly dependent of the real network conditions, and scale.

One way to circumvent this challenge is to rely on simulation tools. However, these might (easily) produce results that are not accurate since the logic being executed is not exactly the same as in the real application (simulated code usually simplifies the lag of the simulated systems for scalability properties), while the network conditions, will in most tools, not fully capture the real execution environment.

An interesting alternative to this is to rely on emulation, where the real system code is executed in an emulated network, meaning under a network that is controlled by software to exhibit similar properties to the real network where one expects the system to operate in reality. However, when considering large-scale systems, with thousands or more processes running on different machines, emulating such complex network is hard and error-prone. Moreover, coordinating the execution of a large number of processes over the emulated network is also daunting, specially when considering that multiple physical machines might be required.

In the case of our work, an emulated network is achieved by controlling the links connecting each node. Controlling such links is achieved by using linux traffic control to shape outgoing traffic at the host to the desired properties. While this task may seem simple, it runs into many obstacles such as scalability issues with the amount of filtering and qdiscs.

1.2 Motivation

Distributed systems are often used to provide services online. These online services must be reliable and scalable, as increase in latency or lack of reliability can have impact on customers trust [11]. In order to estimate if changes in the system can have an impact on these properties, experiments must be reliably conducted. Experiments in small sized distributed systems can be executed manually, where the person conducting the experiment (practitioner) controls each node individually by hand. While this may be tedious, but possible in small distributed systems, it becomes down right impossible to manually launch and control each node in a large-scale distributed system. Running experiments on a large scale distributed system is a hard task, since there are hundreds or thousands of processes concurrently being executed. These experiments on real hardware have a high cost, so to minimize this cost container technology can be used.

Extracting metrics from experiments in large-scale distributed systems can also prove to be a hard task, as they can heavily depend on the system execution environment. Controlling the environment where the experiment is run, in this case controlling the underlying network with the usage of emulation techniques.

While there are many emulators that provide a controllable network model, in a large scale scenario, scalability issues may arise. These emulators differ on their architecture as some model the whole network including the network devices, such as routers, switches, access points, etc. Other emulators only enforce the network properties at each host using tools, such as Linux traffic control to shape outgoing traffic.

Finally, these emulators must allow to control the flow of execution within the experiment and to dynamically change network properties, so the level of realism can be similar to that of an actual deployment. There is however, no tool that provides all of these features, being scalable and generic (i.e. independent of the type of distributed system being tested). In this ware we tackle this lack of adequate solutions.

1.3 Expected Contributions

In this work we expect to achieve the following:

- Design a tool that simplifies the planning and execution of experiments for large-scale distributed systems, taking into advantage container technology and realistic network emulation.
- Evaluate and compare our proposed tool to the state-of-the-art network emulators.

1.4 Research Context

The work conducted in this thesis emerged from a collaboration of the NOVA School of Science and Technology and Protocol Labs [23]. Protocol Labs is a company that specializes in research and development of large-scale decentralized distributed systems. Some examples of their work include: Filecoin, a cryptocurrency based on a distributed storage systems designed to store humanity's information; IPFS, a distributed system for file sharing; libp2p, a framework for developing distributed and decentralized applications; TestGround, a platform for testing, benchmarking and simulating distributed and P2P systems.

This work comes in the context of the research project of a multi level DHT evaluation in real systems, where this tool will be leveraged to assist in the planning and execution experiments so that relevant metrics can be extracted for evaluation without having to pay the cost of deploying a real system which would be unfeasible in the context of IPFS. This work also stems from the large experience in the computer science department on experimental assessment of distributed systems, as these experiments have a high cost in both human effort and time.

1.5 Document organization

The continuation of this document is organized in the following manner:

- **Chapter 2** is where related work is discussed. It starts by detailing what is a distributed system and how to evaluate them. Then we move on to network models where we define concepts on network models. We look into a few simulators and discuss their viability for this work. Furthermore, we compare virtualization and containerization and discuss a few containers orchestrators. We look into emulation and analyze emulators and discuss their limitations. Finally, we look into other tools for testing used in the software industry.
- **Chapter 3** is where an initial approach is presented and a schedule for future work is provided.

Chapter 2
Related Work

In this chapter we first detail how to evaluate distributed systems, where we present different methods of evaluation and discuss some relevant metrics to be measured when conducting an evaluation of a distributed system (Section 2.1).

We then detail a few concepts on network models, and present techniques that enable to emulate different networks topologies as well as control key properties of (individual) network links (Section 2.2).

Next we address simulation as a means to materialize a network model, and then we analyze simulators to contextualize the ware that we plan to conduct in this thesis with these tools (Section 2.3).

In the following section, we look into virtualization and containerization as techniques to provide an isolated execution environment, providing their definitions and presenting the most popular container orchestrators currently available (Section 2.4).

We discuss existing wares, that similarly to this one, focus on emulation, providing an overview of how they operate and their limitations (Section 2.5).

Lastly, we discuss other tools used in the industry for testing distributed systems and discuss how these tools relate to the work to be conducted in this dissertation (Section 2.6).

2.1 Evaluating Distributed Systems

Distributed systems, by definition, are composed by a group of processes that are connected through a network, where they cooperate to achieve a certain goal. In the distributed system environment there are some common assumptions that must be made in order to reason about the state, operation, and evolution of the distributed system, being the main classes of assumption divided in the following groups [8]:

Timing assumptions captures how time is perceived by a system, and can be either synchronous or asynchronous. An asynchronous system does not make any timing assumptions about processes and links, but it can measure the passage of time with the transmission and delivery of messages, with a happens-before relation between messages. On the other hand, a synchronous system imposes an upper bound on computation and communication, this means that the system limits the time of executing steps or message transmission to always less than the upper bound.

Internal state model which commonly perceives every process is a deterministic automaton that owns a set of states, an initial state, and each state has input and output, as variables that act has a gate to send or receive information to the exterior.

Fault model is a model that dictates how to perceive process failures. The existing fault models are: crash fault, omission fault, fail stop and Byzantine:

The crash fault model imposes that once a process fails it stops message transmission.

The omission fault model dictates that on a process fail it will omit the transmission of an arbitrary number of messages.

Fail stop model acts as the crash fault model, but additionally it will notify all the other process of its failure.

Byzantine model states that on a process fail it can have an arbitrary behavior.

Network model captures the behavior exhibited by the network. Some existing network models are: fair loss links, stubborn links, perfect links, logged perfect links and authenticated perfect links. Fair loss links does not guarantee every message will be delivered, but it provides the fair loss property: If a message is sent an infinite number of times, it will be delivered an infinite number of times.

Stubborn links operate over fair loss links and send every message an infinite number of times, causing it to be delivered an infinite number of times. This model is used as the base for the remaining models.

Perfect links provide reliable delivery and no duplication, which states that if a message is sent, it will be eventually delivered only once.

Logged perfect links also guarantees reliable delivery and no duplication, but it logs message delivery so that if a process crash and reboots, it does not deliver the same message twice.

Authenticated perfect links offers the above-mentioned properties but replaces the no creation property for authenticity property. The authenticity property dictates that if a message is delivered and its sender p is correct, then the message was sent by p. This is ensured by having an authenticate function before sending the message that verifies its sender and the message contents.

When evaluating distributed systems there are two main venues that can be pursued and that are complementary among them: a formal evaluation or measuring key system performance indicative metrics. A formal evaluation consists of proving the correctness of the system under a given assumption model, but this evaluation does not provide insight on the system performance. Measuring system performance metrics, according to their relevance to end users, like latency or throughput, is a way to determine a distributed system adequacy for real usage.

To measure attributes like average latency or throughput, since they can highly depend on the underlying network and the network might exhibit an unstable behavior during experiments, the results can be inconclusive [3]. In order to have conclusive evaluations of distributed systems there is a clear need to control the network layer. This can be achieved using either emulation or simulation to create a controllable network over which the target system can be evaluated.

2.2 Network Models

In this section we will further improve on the definitions of network models referred in Section 2.1, and provide further relevant concepts about networks that are relevant to our ware. Then we continue by explaining how to control the network layer, and finally we look into tools that shape the network considering a set of desired properties.

2.2.1 Concepts

The network models are assumptions on how each network link between two processes behaves, and they are usually essential to prove the correctness of distributed protocols and distributed system formally.

A real distributed system deployment environment is composed of (potentially) thousands of links whose combination forms a network, over which the system operates. In these networks, a link is materialized at the physical layer by (potentially shared) hardware, hence network hardware failures can result in multiple links affected. These link failures can create network partitions, which are when link failures segment (usually temporarily) the network so that a node **A** cannot communicate with node **B** (or more generally become unreachable by another set of nodes).

The most popular networks are the local area network (LAN) and the Internet. LAN is an individual network that covers a small location and is owned by individuals, its characteristics are its ease of resource sharing within the network, high data transfer rates, high fidelity, low latency, low error rate, and a high homogeneity of devices connected to it. Unlike a LAN, the Internet is a global network that has varying latencies, is error-prone, highly volatile, and prone to form network partitions exhibiting a high heterogeneity of devices connected to it.

2.2.2 Control the Network

In order to guarantee a stable network across multiple experiments there is a need to manage and control it. Controlling the network depends on the type of network itself, so we will analyze how to control each network type:

Software defined networks (SDN) [6] are virtual networks that separate the control and data plane, offering more control and flexibility. By separating control and data planes, it separates the decisions of network elements (routers, switches, and access points) from the data plane, which is where all application data is sent. Resulting in all decisions relevant to the network topology, routing, and network properties being managed centrally at the control plane. The control plane in SDNs is usually managed by a centralized component that has visibility over the whole network and dispatches messages to setup and maintain the desired network topology by controlling individual network hardware components.

Data centers use virtual networks to guarantee network resources to tenants [4]. Many data centers operate in a pay-as-you-go basis for allocation of computation for tenants. This allocation is ensured using virtual machines that isolate machine resources from other tenants, however all tenants share the intra-cloud network. This sharing of the intra-cloud network can lead to unpredictable performance, so data centers use a virtual network to mitigate that risk. These virtual networks can be divided into virtual clusters and virtual oversubscribed clusters, but both utilize a management plane and data plane. The management plane receives tenants request and allocates resources, while maintaining each bandwidth share across the physical network. The data plane limits each VM bandwidth rate at the end-host hypervisor, and therefore, there is usually no need for reservations at datacenter switches. Using both these methods (maintaining each bandwidth share and limiting each VM bandwidth at the end-host hypervisor) virtual networks are able to allocate and isolate network resources from different tenants.

Finally, there are networks, like the Internet, that are built without using software management like SDN and data center virtual networks. The Internet is built on top of network devices in a decentralized manner. In order to control these networks, we must control the links connecting each node, for that we use tools that can shape those links at the host to exhibit target properties. We will examine such tools in detail bellow.

2.2.3 Linux Traffic Control

Linux traffic control (linux tc) is a program that enables manipulation of traffic control elements and monitoring of the bandwidth usage of active connections [1]. It offers an interface to apply filtering of data and manipulation of network properties through the use of queuing disciplines. This combination of filtering and queuing disciplines is what shapes traffic at the end host, as to enforce limitations on both latency and bandwidth (notice however that the maximum bandwidth and minimum latency depend on the underlying hardware).

2.2.3.1 Queuing Disciplines

Queuing disciplines [29], or qdisc, are queues that handle how packets are sent to the hardware network device. When packets are sent to the network from an application, they are first enqueued into the designated qdisc then dequeued and sent to the hardware network device, enabling the qdisc to store traffic when the hardware cannot. These queues can also have mechanisms to control network properties (delay packets, limit bandwidth, drop packets), which we will analyze further ahead. Qdiscs are divided into classfull or classless qdiscs and have the following interface: enqueue, dequeue, requeue, drop, init, change, reset, destroy, and dump.

Classless qdisc do the most basic shaping of traffic, such as delaying, reordering and dropping packets. Some classless qdisc are:

- Fifo is a qdisc that has a fifo queue that packets transverse, and it's the simplest qdisc.
- Token Bucket Filter (TBF) is a qdisc allows traffic to go through until it reaches a threshold. It is implemented as a virtual bucket that has multiple tickets, when a packet is enqueued it tries to acquire a ticket. If there is none available it will wait for a certain amount of time, otherwise it gets a ticket and passes through. If packets arrive more frequently than tickets, eventually packets will get dropped.
- Stochastic Fairness Queuing (SFQ) is a qdisc that reorders its queue, resulting in each packet having a probability of being sent.
- Network Emulator (netem) is a qdisc that allows manipulation of network properties, such as add delay, drop packets and more.
- Heavy-Hitter Filter (hhf) is a qdisc that targets heavy hitters, meaning it tries to locate heavy flows and allocate them to separate queues with less priority, so that this traffic does not affect critical traffic.
- Random Early Detection (red) is a qdisc that detects when a flow surpasses a threshold of bandwidth, and it starts randomly dropping packets to simulate physical congestion.
- Fair Queue Scheduler (fq) observes TCP pacing and scales the number of flows within the qdisc.
- CoDel (codel) and Fair Queuing CoDel (fq codel) are both qdiscs that use queues that utilize the active queue management algorithm (AQM). This algorithm was devised to address the shortcoming of red and its variants. The Fair Queuing CoDel combines both fair queue and the codel models, to give a fair share bandwidth to all flows using the queue. This is the default qdisc since systemd 217 [2].

Classfull qdisc can contain additional qdiscs and filters attached to it. Packets when enqueued will follow the rules of the qdisc and either be processed at the root and then dequeued, or sent to a class within the qdisc. All traffic that does not get classified within the qdisc can just dequeued into the hardware device, but this can depend on the existing qdisc. The most popular classfull qdisc is the Hierarchy Token Bucket.

Hierarchy Token Bucket (HTB) uses tokens and buckets to control traffic, more precisely it is widely used to restraint bandwidth usage. The htb qdisc architecture is represented in Figure 2.1, all its classes have the same attributes and work in the following manner:

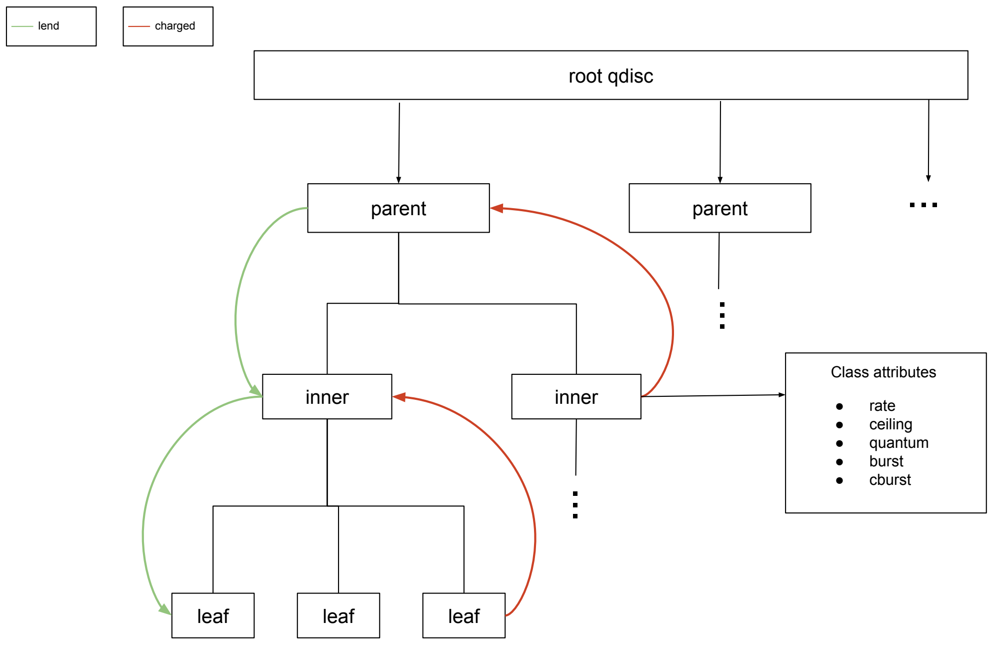

Figure 2.1: Flow of borrowed tokens in an HTB qdisc, adapted from [29].

1. Packets are enqueued at the root qdisc.

2. The root qdisc classifies packets and sends them to its parent class, and it repeats until the packets reach a leaf class.

3. If its state is lower than the rate attribute, the leaf classes will dequeue packets while there are tokens left and other classes will lend tokens to their children.

4. If its state is between the rate and ceil attribute, the leaf classes will try to borrow tokens from their parent and dequeue packets correlated to the amount of tokens borrowed. The other classes will try to borrow tokens from their parent and lend to their children in quantum increments.

5. If its state is higher than the ceil, the leaf classes do not dequeue packets, delaying them to achieve the proposed rate. The other classes will not borrow or lend any tokens.

6. After acquiring their respective tokens, packets transverse the structure until they reach the root qdisc and are dequeued to the hardware device.

This architecture only applies shaping in the leaf nodes, meaning that the sum of rates of leaf classes should not be greater than the ceil of their parent. This rule exists so that leaf classes do not ignore the parents rate, which will happen if their sum is bigger than their parents rate.

2.2.3.2 Filters

Filters are one of the key components of the linux tc, they can be attached to classfull qdiscs, allowing classification of packets when they enter the root qdisc and determine to which subclass the packet must go. One of filters main component is its classifier, with the most common being the u-32 filter [31]. This filter allows matching of bit fields in the packet, since using bits and masks is not an easy task, it offers two modes:

- The first mode being a filter that assigns packets to a destination based on packets attributes. This mode can be used to chain filters to form a tree-like structure.

- The second mode is to serve hashtables, meaning that a filter would select a hashtable and select an attribute (selector). This selector would be used to compute a hash and use the computed hash as a key look up for its next filter or destination.

The use of hashtables within u32 follows this pattern: Create a hashtable by giving it a divisor and an ID, create or link filters to such hashtable and define how the hash should be computed, and finally add filters to the buckets in the hash table. To simplify this last step a sample hash can be given, so the filter will be added in the sample bucket.

The other Filter component is a policer, which classifies traffic in a binary manner and sends it to the corresponding destination. This can be particularly useful to limit traffic income as to limit one's bandwidth consumption.

2.2.3.3 Example

Here we present and discuss an example of shaping outgoing traffic with linux tc, which will combine all topics discussed previously: queuing disciplines and filtering. In this example we have a root htb qdisc that has two leaf htb classes, one that contains a netem qdisc and the other a Fifo qdisc, as illustrated by Figure 2.2.

1. All incoming traffic enters through the root htb qdisc, and it is redirected into his filter.

2. This filter distinguishes traffic into traffic going to port 8080 and other traffic.

3. The filter will redirect traffic to its respective leaf htb class.

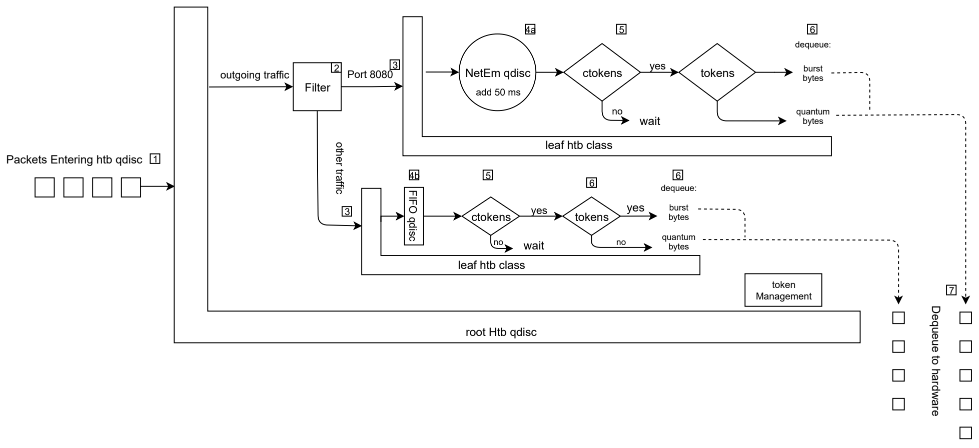

Figure 2.2: Example of shaping with linux tc, adapted from [29].

4. After being enqueued in their leaf htb class, they are first enqueued in the corresponding classless qdisc. In **a**, the netem qdisc will add a delay in traffic of 50ms, in **b** there is a fifo qdisc which corresponds to a fifo queue that traffic must go through.

5. If there are no ctokens, wait for ctokens to arrive.

6. If there are no tokens, dequeue in quantum bytes, otherwise dequeue in burst bytes.

7. Packets exit through the root qdisc into the network hardware device.

2.2.3.4 Discussion

We presented an example of how to use linux tc, and as we can see a simple example of adding 50 ms of latency to traffic on port 8080, and to limit outgoing traffic bandwidth can result in a complex diagram. Complexity of linux tc can increase drastically when increasing the number of links (connections). The increase in the number of connections, will result in an increase of filtering to redirect traffic to the correct htb class.

2.3 Simulation

In the computer science research community there is a need to validate and test its results of research frequently, as well as compare such research aspects with state-of-the-art solutions. There are a few different and complementary ways to validate one's research such as: an analytical approach, or an experiment approach where a prototype is executed under realistic conditions. The analytical approach is when a mathematical model of the system is analyzed, as it can be possible to infer how good is a solution if the model is simple, but in the case of distributed systems many aspects may need to be simplified or even discarded to apply such techniques. Another approach is to run the actual system (or prototype), this introduces a lot of issues as setting up, accessing and coordinating nodes, applying protocol changes in each node during the experiment among other practical issues [22]. Simulation comes as a way to create a simplified model of the actual system, creating a repeatable and controllable environment without paying the cost of setting up and executing the actual system.

2.3.1 Simulators

Simulators can be considered from different perspectives, as their primary focus can differ. Since our focus is on distributed systems, potentially large scale ones, we will analyze their design taxonomy, more precisely their simulation engine and modelling framework [26]. A simulator engine can be either cycle-based, which is an engine that executes a step in regular time intervals, or event-based, which is an engine that executes a step each time an event is consumed. The simulators that we discuss in this document are the following:

PeerSim [21], a java P2P simulator, has an architecture that supports both cycle-based or event-based simulation, and it focuses on modularity and configuration while maintaining scalability. Its network is modeled as a list of nodes, where each node has a list of protocols and these nodes can be altered via initializers or monitored via both observers and controllers. PeerSim takes advantage of the Java reflection mechanism to setup the experimental environment from a configuration file, which defines the whole experiment and can include overwrites to default implementations of some modules or key components of the simulator. This simulator also offers:

- Several tools to operate over graphs as it treats overlay networks as graphs and provides special initializers and observers for those networks.

- The possibility to add external modules to the configuration file. These external modules can add additional features to the simulation, that otherwise would need to be fully programmed.

- Addition and configuration of a transport Layer and mechanisms to generate churn, as it can be configured to use trace-based datasets. It is to note that the transport layer is supported only in event-based simulations (since in cycle-based simulator protocols evolve in a constant time-step).

CloudSim [9] is a cloud computing event-based simulator that offers modelling, simulating and experimentation focused on cloud infrastructures.

It has a multi layer design that consists of User code, CloudSim, and the CloudSim core simulation engine. The user code layer is where we specify the simulation configurations (number of machines, resources of machines, applications, VMs, . . .), this layer can be extended to allow: the generation of different workloads/configurations, to model the cloud availability and do tests with custom configuration, allowing modification of provisioning techniques for clouds and federations. The CloudSim layer is divided into multiple sublayers, such as user interface structures, VM services, cloud Services, cloud resources and network. We will focus on the network sublayer in CloudSim. This sublayer is composed of message delay calculation and network topology and uses a conceptual network abstraction, resulting in network entities (routers, switchers) not being simulated.

The message delay calculation is performed by leveraging on a latency matrix, where each entry represents a delay that a message will have when it is sent from entity **A** to entity **B** over the network. Network topology description is loaded and stored using BRITE [20], which uses its own BRITE file format. Each node stored in the BRITE file represents a single entity from a variety of entities (hosts, data centers, cloud brokers, . . .).

All messages sent by CloudSim entities are processed by a network topology object that appends the latency to each message and sends it to the event management engine, where it will wait to be delivered only after the simulator advances on the amount of simulated time equal to that latency before delivering the message to its destination. Using this external file method allows for different experiments to re-use the same network topology file.

iFogSim [14] is a simulator designed to provide a reliable and controlled IoT environment, so properties like latency, energy consumption, network congestion and other operational costs can be easily measured. iFogSim's architecture, as seen in Figure 2.3, is divided in seven main components: IoT applications, application models, resource management, infrastructure monitoring, data streams, fog devices, and IoT sensors and actuators.

The bottom component is the IoT sensors and actuators, they emit chunks of data for applications, have different geographical locations, and have the ability to simulate a wide variety of sensors, as they can be configured for data-emission characteristics, intermission time and size of data chunks. This component does not deal with network issues that happen due to collision on the wireless medium among colocated devices, so to solve this issue they propose to abstract this aspect into high level properties such as latency or bandwidth, or build a model that mimics such network issues and plug it into iFogSim.

The next component that models the fog devices, they are elements that can host the

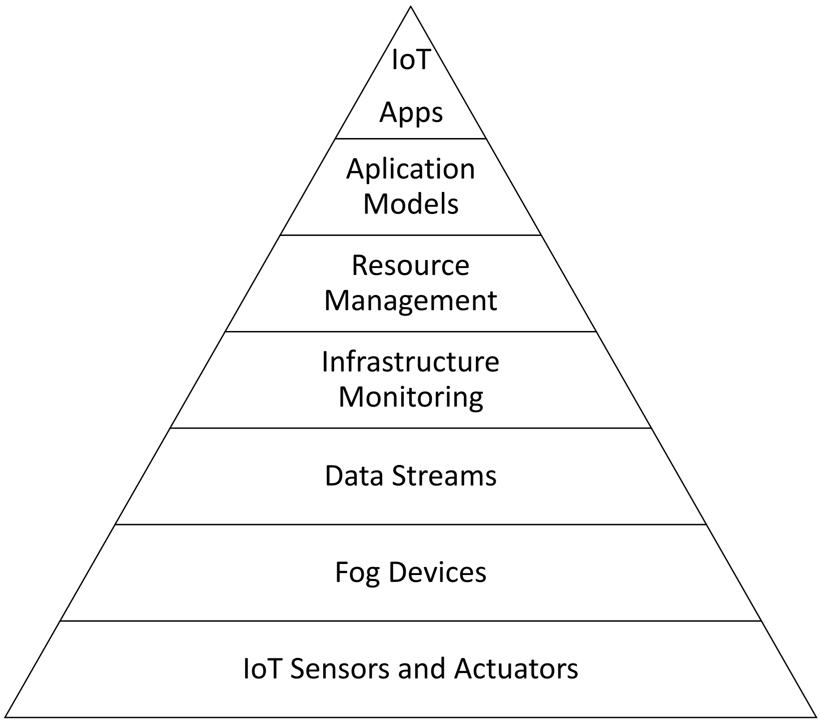

Figure 2.3: iFogSim's architecture divided into components, adapted from [14].

application modules, serve as gateways for sensors to connect to the network. This component includes a large variety of devices ranging from edge devices to cloud resources, so they are all arranged in a hierarchical topology with only direct communication between parent and child devices. This communication method invalidates all communication between same hierarchical level devices, such as smartphone to smartphone for instance.

The following component is Data streams which are sequences of values that come from the lower components. These values can range from raw data coming from sensors or retransmitted data coming from fog devices. These fog devices also generate special data streams that are resource use details, which are meant for the monitoring component to consume.

The monitoring component fetches and stores resource usage data from devices. These devices are sensors or actuators, and this component supplies this data to the resource management.

The resource management component is a core component that receives monitoring data and manages resources, so that application requirements are met and there is no waste in resources. This uses a scheduler that tracks device availability in order to pick a host for the application module and reserve device resources for such module. This can go from a simple allocation of application modules to fog devices or to a complex allocation such as being able to support migration of components and dynamically change each device available computational resources.

Lastly there is the application models component and IoT applications, which materialize the applications designed to run in fog environments. More precisely, applications are developed based on the distributed data flow model (DDF), which consists of a model to represent applications in the form of directed graphs, with the vertices being applications modules and directed edges representing the flow of data between modules. Currently, two models for IoT applications are supported: Sense-process-actuate and Stream-processing. Sense-process-actuate model dictates that information is harvested by sensors and transmitted as data streams, which are consumed by applications executing in fog devices that send commands to actuators. Stream-processing model says that there is a network of application modules executing on fog devices that continuously consume data streams produced from sensors. This latter model can be viewed as a subcategory of the first model.

NS-2 [28], **NS-3** [24] are discrete event-based network simulators that enables simulation on wireless or local, IP or non-IP, networks.

NS offers the following models to simulate a network, nodes represent an end-system; network devices represent devices that connect nodes to communication channels; communication channels is where data is transferred between nodes; communication protocols are protocols that receive network packets and are organized in a protocol stack, protocol headers are specialized data structures contained within network packets that is often associated with a specific network protocol (containing control data); finally, we have network packets which is the basic unit of information that can be sent between nodes.

The simulation of a network using NS can be summarized in 3 steps:

1. Create the network topology using the network elements described above.
2. Create the data demand, which is an application, or several applications, that receive and send data to the network.
3. Run the simulation until it runs out of events in the main loop or a configured stop time is reached.

One of the drawbacks of using NS2 or NS3 is their high learning curve. This high learning curve is due to both NS being very low level and requiring a lot of time for new users to design and achieve the target network.

2.3.2 Discussion

Simulators provide a controllable and repeatable environment at a low cost. While this is beneficial for earlier stages of the development, for activities such as validating insights or debugging and verifying the correctness of new solutions, these simulated models do not offer enough realism when compared to the real system operation. Since these tools cannot offer enough realism, they cannot be used to predict all behaviors that will be exhibited by a distributed protocol or system in a real setting. Another problem with these tools is that most of them do not offer the same programming interface as the real system, making the transition from simulated environment to a real system much harder.

Validating or evaluating full-fledged solutions in a full realistic setting, may benefit from emulation. Emulation is an interesting candidate as it offers an execution environment very similar to the reality, in terms of behavior and programming interface, however this comes at a higher computational cost.

2.4 Virtualization and Containerization

In this section we will describe virtualization and containerization, as they are techniques that provide an isolated environment where application code can be executed. Then we discuss a technology for orchestrating multiple isolated environment.

2.4.1 Virtual Machines

Virtual machines (VMs) are similar to actual physical machines that execute an operating system that uses a set of virtual resources to create an isolated environment where applications can execute [25]. They can be materialized as either a system VMs or process VMs, being that process VMs are dedicated to a single process and are instantiated when the process is created and removed when the process is terminated. We will focus on system VMs, as there might be a need to have multiple processes inside each VM. They are deployed in a hypervisor environment, which provides each virtual machine with their virtual resources and manages them, usually having multiple VMs in a physical machine.

2.4.2 Containerization

Containerization is similar to virtualization in a sense that it provides an isolated environment for an application to run. Unlike virtual machines that use a hypervisor to manage virtualized hardware, containers run in the user space of the operating system kernel. To give a definition on containerization, would be that it utilizes operating system virtualization to provide isolation between containers [30]. While this allows that multiple containers to run on isolated user spaces and at a lower cost than virtualization, it comes at the cost that it is dependent on the operating system. This dependency means that operating systems that do not share any similarity cannot be run in containers, for example you cannot run a Microsoft Windows container on an Ubuntu server.

One of the major differences between VMs and containers, is that in VMs virtualize the operating system resulting in not being OS dependent, as seen in Figure 2.4. Since

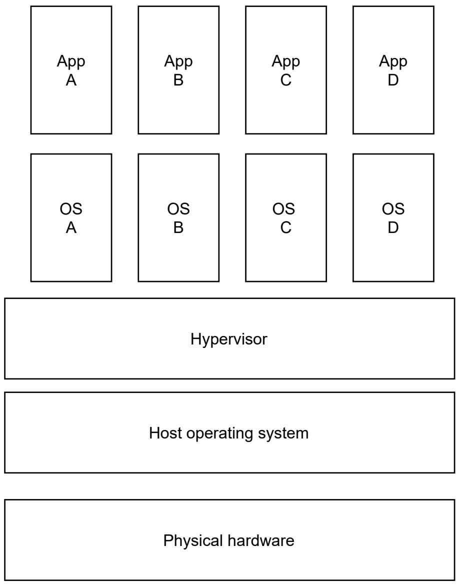

Figure 2.4: Virtual machines deployment, adapted from [7].

VMs virtualize the whole system, they are more computationally expensive than containers. Such computational cost when experimenting with hundreds or thousands of applications instances (i.e, processes) can be a significant limitation. Since, our experiments are homogenous regarding the operating system, this OS dependency is irrelevant in the context of our ware and for the target systems, that we aim of evaluating. For this reason we will opt to use containers over virtual machines in this work.

We will be using docker as the container technology of this work for its wide usage within the community. We discuss docker and the docker ecosystem in detail in the following.

2.4.2.1 Docker

Docker is a platform that allows to build and execute applications in containers. It's architecture, as illustrated in Figure 2.5, is a client-server architecture. The client can make request to the docker daemon, which manages docker services. Docker registry is a storage for docker images, it can be public, like the docker hub, or privately managed (e.g. by organizations).

Within Docker ecosystem there are several relevant entities: docker images, containers, networks, volumes, plugins. We will focus our attention on docker images, containers and networks, which are the entities most relevant to pursuing the goals of this ware.

Docker images are read-only templates to create docker containers. They can be build using a docker file with the commands necessary to create and run the image. When creating an image, docker creates a layer for each instruction and only builds layers that have changed, resulting in lightweight, small and fast creation of images.

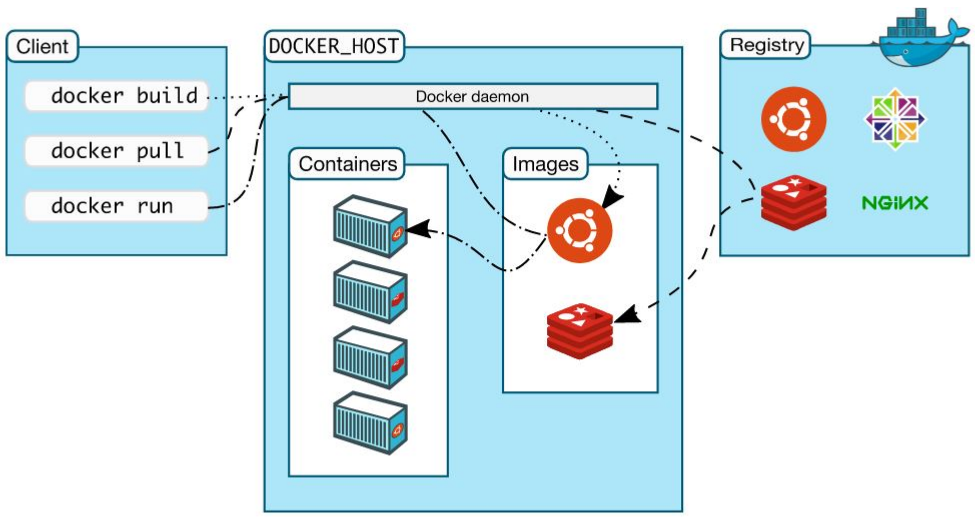

Figure 2.5: Docker architecture, taken from [12].

Docker containers are instances of images, they can be created, started, stopped, moved or deleted using the Docker API or CLI. Docker containers be attached to one or more docker networks, they can also be attached to storage and the level of isolation of its resources from other containers and from the host can be fine-tuned. All contents within a container that are not saved into persistent storage are deleted after the container is removed.

Docker networks provide an abstraction of the network that interconnects multiple containers, they are pluggable using drivers. The existing drives are: bridge, host, overlay, macvlan, none, third party.

Bridge network provides is a link which passes traffic between network segments and can be either software or hardware. In case of docker it uses a software bridge, that provides communication between connected containers on the same host and isolation from containers not connected to the network or that are connected to different networks. This is the default network driver if it is not specified, but we can create user-defined bridges that offer a few more capabilities, such as: automatic DNS resolution between containers, additional control on removing and adding containers into the network. Notice that the configuration of network only affects containers connected to that network.

Overlay network driver connects multiple daemon host through a distributed network. It provides communication between swarm services or standalone containers, as it removes the need for OS-level routing. This network can only be used in a docker swarm because it requires the ingress network, created by docker swarm on creation or joining a swarm. We discuss Docker swarm bellow.

2.4.2.2 Container Orchestrators

Docker Swarm is service that allows creation and management of docker clusters. These docker nodes have two possible functions: manager or worker, the manager's job is to maintain membership and the desired state of the swarm, while the worker's job is to execute the tasks assigned to them and then report the task conclusion to the managers.

A service is the task unit with which users interact with the swarm, it is the task the workers must execute. When creating a service the user must provide a container image. A service when created will reach a manager node for it to assign tasks to the worker nodes, once such task is assigned it can succeed or fail. The number of tasks is related to the number of replicas set in the swarm configuration and the number of replicas given in the service command, although if the service is global it will run on every available worker in the cluster. The service creation flow is illustrated in Figure 2.6.

Load balancing within the swarm is done with the ingress load balancing. This load balancer receives outside requests for the container published port, then it redirects those request to a running instance of the service.

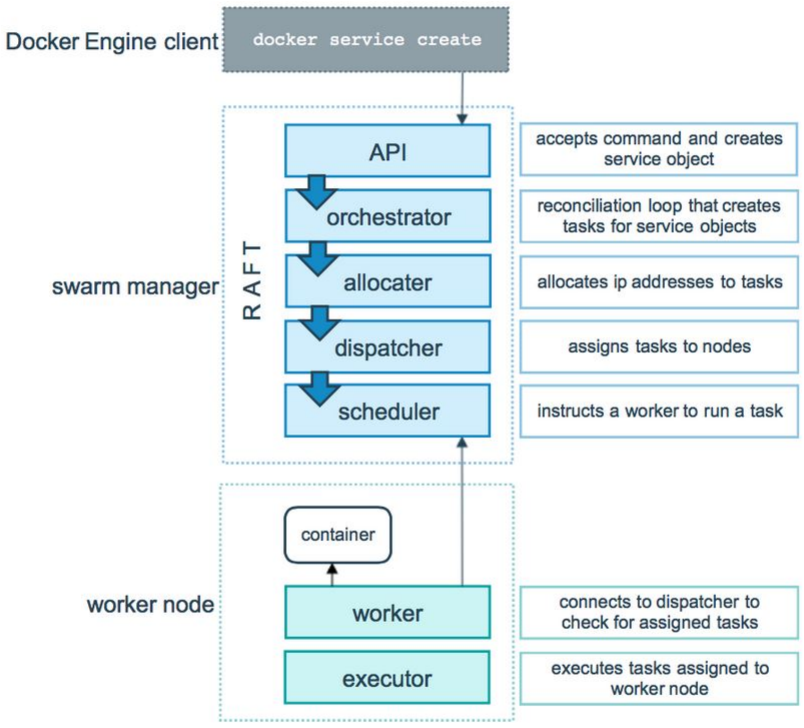
Docker Engine client
docker service create
swarm manager
RAFT
API
orchestrator
allocater
dispatcher
scheduler
accepts command and creates service object
reconciliation loop that creates tasks for service objects
allocates ip addresses to tasks
assigns tasks to nodes
instructs a worker to run a task
worker node
container
worker
executor
connects to dispatcher to check for assigned tasks
executes tasks assigned to worker node

Figure 2.6: Flow of service create request in docker swarm, taken from [12].

Kubernetes is a platform that offers automated deployment, scaling, and operations of application containers [16]. Kubernetes has a variety of entities to achieve those properties, including: pods, service, volumes, namespaces, controllers, and a control plane. We will focus on pods as they are the entities on which applications are executed.

Pods run single or multiple containers to confine an application, while providing them storage resources and a network IP. If multiple pods run the same application, they are grouped and managed by a single controller. Networking inside the pod utilizes the same namespace, meaning that containers can communicate with localhost, however outside the pod they must share their network resources. Volumes inside a pod are shared among all containers, and they are used for persisting data in case of a container crash.

Kubernetes enables specification of network plugins to control how pods communicate with other entities, those plugins can either be a container network interface (CNI) or Kubenet.

Kubernetes has a scheduler (Kube-Scheduler) giving pods, without a node attributed, to the best fitting physical node and operates over the control plane of Kubernetes.

Discussion

Both these tools provide container orchestration, but seem to have different focuses. Kubernetes is more sophisticated than Docker Swarm, it allows other type of containers but is more complex to deploy and manage those containers. Swarm, on the other hand, only works with docker containers and allows for a lower level management of containers.

2.5 Emulation

Network emulation fills the gap between real world deployment and simulation, as simulation serves the main purpose of protocol validation and testing, but it does not provide enough realism for the case of large-scale systems. As for real world deployment it is too labor intensive, requires a lot of resources and can be unstable within experiments, lacking the necessary control for the experiments. Network emulation provides a controllable virtual network for the real system to be executed, allowing to reach conclusions about its behavior in several relevant network conditions, enabling the recreation of those situations, easily. It also provides a smooth transition between code that executes on an emulated and code that executes on a real deployment, as there is no difference in accessing the emulated network or a real one.

2.5.1 Kollaps

Kollaps is a decentralized network emulator that is agnostic of the application language and transport protocol [13]. Kollaps architecture can be dissected into some main components, being: the emulation manager, deployment generator, tc abstraction layer, and a dashboard.

The **deployment generator** receives a network topology and maps it to a deployment plan. The network topology is written in XML or in YAML syntax, and should identify all services, links, bridges, and dynamic elements (such as link removals and additions, changing latency). Services correlate to a set of containers sharing the same image and can give parameters values to each container once deployed. The bridges map to networking devices that have unique names and are arbitrarily connected to achieve complex topologies through the use of links. Links can be uni-directional or bi-directional, with obligatory attributes for source, destination, network properties, and the container network to attach to. They are all uni-directional, so when a bi-directional link is described, it will be mapped to two uni-directional links in opposite directions with a mirror of the link properties. Kollaps supports dynamic events in the same format as the network topology, these events can change network properties, removal or insertion of links, bridges, and services.

The **emulation manager** is the component responsible for enforcing the emulated network properties to each container per physical machine. Since it does not emulate the internal state of the network, the topology is described at the end hosts. To achieve that, the topology description is parsed into a graph structure, then it computes the shortest path between each pair of containers. This shortest path can contain several links, where each link contributes to the path network properties. Network properties (such as latency, packet loss, and jitter) can be computed via the links' physical properties (sum or multiplication), but the bandwidth requires additional attention since it is not only restricted by the physical property of each link. Bandwidth also depends on all actives flows in the same path resulting in bandwidth being computed at runtime to avoid over allocation. To not congest the links, a fair allocation mechanism must be adapted to allocate bandwidth among competitor links. In a real deployment, this allocation mechanism does not exist because routers and switches store the excess packets to compensate the congestion until the point where buffers overflow and packets are dropped. Transport protocols that ignore the packet loss, like UDP do not suffer from this issue, but TCP has a congestion mechanism to allow a fair bandwidth allocation for each link.

Kollaps instead of modelling the network elements, adopts a model to compute a fair share of bandwidth, the RTT-Aware-Min-Max model [19]. This model was inspired by TCP Reno and gives a share of bandwidth proportional to the round-trip time, however it does not assure that all bandwidth will be allocated, since the link can utilize less bandwidth than its attributed share and to leverage that the emulation manager will increase the share of other links proportional to their original shares.

This congestion model produces unreliable results while the allocated bandwidth is more than the maximum capacity (bandwidth overflow) because of the interactions between linux tc, congestion algorithms of TCP, and Linux's TCP Small Queues. Linux tc queues and buffers from network devices have different actions when in presence of overflow: the buffer from network devices when full will drop incoming packets, but the linux tc qdisc, more precisely the htb qdisc, when full will back-pressure the application instead of dropping packets, this is due to packet loss being done in netem and not in the htb qdiscs. There is also the Linux's TCP Small Queues [18], they reduce the number of packets in qdiscs and device queues in order to reduce the RTT, mitigating the buffer bloat. They work by listening to the amount of data being enqueued and when it goes over a threshold, the socket will stop further packets from being enqueued. To address these congestion limitations Kollaps will observe the requested bandwidth and when it surpasses the maximum available it will contact netem to drop packets per flow proportional to the amount above the maximum available bandwidth.

The emulation manager will spawn an emulation core assigned to the network namespace of each container. The emulation core is responsible for updating the emulation model, enforcing topology constraints through the TCAL and the dynamic events. This design allows support of any containerized application, while reducing computational and network overhead since the emulation cores share data via shared memory. This allows for the emulation manager to aggregate all data from emulation cores and exchange it with other emulation managers, making it scale with the number of physical machines instead of the number of containers. The dynamic events are pre-computed to improve accuracy of emulation of large graph topologies, as computing such events at runtime would require several seconds resulting in degrading accuracy.

The **TC abstraction layer** (TCAL) uses linux traffic control (linux tc) to manipulate network properties and retrieve bandwidth of active connections. Kollaps uses two types of qdiscs the htb qdisc and netem qdisc, for each destination it creates a htb qdisc that enforces bandwidth for all flows to such destination, then creates a netem qdisc, attached to its corresponding htb qdisc, to apply network properties. Netem qdiscs have an u32 universal 32bit filter where traffic is matched, this filter is a two-level hashtable with key being the destination IP address of packets and them it redirects the packets to their corresponding netem qdisc. This two level design is to circumvent the limitations of u32, which provides an array of 256 positions instead of real hashing. TCAL maps the third octet of the IP address to the first level and the fourth octet to the second level, so constant lookup times can be achieved while avoiding collisions. Traffic is directed first to the netem qdisc to apply all network properties desired then dequeued from netem and enqueued to the parent htb. TCAL structures are queried and updated during an experiment, more precisely to retrieve bandwidth and apply the dynamic properties. To minimize these calls it uses netlink sockets that communicate directly with the kernel, avoiding spawning a new tc process.

The **dashboard** is a web interface that allows monitoring of the experiments. In this dashboard the user can visualize a graph based representation of the emulated topology, dynamic events, ongoing traffic, and status of services.

2.5.1.1 Limitations

Kollaps has a key limitation that is the fact that it does not allows to describe the containers IP in the network topology, facilitating experiments as the IPs do not change with each experiment. Another limitation that we can identify is that when modelling large network topologies, since Kollaps does not allow the reference of network topology files within another topology file, it can lead to extremely long, repetitive and error-prone network descriptions.

2.5.2 ModelNet

ModelNet [32] is a network emulation environment where applications run on edge nodes and all network traffic goes through a physical host that enforces all desired network properties. These core nodes apply queuing disciplines to packets, when the queue buffers starts overflowing packets will be dropped.

Modelnet execution can be summarized in five phases:

The **create phase** receives as input the topology and formats it to graph modelling language (GML).

The **distillation phase** takes as input the topology in GML format, and creates a pipe topology that models the same network. This phase can also simplify the network albeit at the cost of decreasing the emulation accuracy. This simplification comes with the collapsing of the network topology, a more collapsed topology would have the lowest per packet overhead but also a lower accuracy as link contention among competitive flows is not emulated.

The **assignment phase** receives as input the pipe topology and maps pieces of such topology to each core node (i.e., nodes that execute the application components). This assignment is a NP problem as it depends on many variables: jitter, latency, bandwidth, routing. The employed solution on ModelNet is a greedy approach where each core node randomly selects nodes from the topology and selects links from the same node in a round-robin manner.

The **binding phase** attributes virtual nodes (VN) to edge nodes and installs the application while creating the necessary scripts to facilitate emulation. This phase assigns multiple VNs to each edge node and attributes each edge node to a single core node. In this phase the core nodes need to generate the shortest path between VNs and save it in a routing matrix. While this approach may lead to memory issues, it can be circumvented by using hash based cache of routes or using hierarchical lookups. Finally, this phase also configures each edge node with a designated IP address.

The **run phase** executes the application in each VN. This phase consists of automated scripts that execute the application in every VN. The application must use their bound IP address (given in the binding phase) instead of their physical machine IP address, in order to use the emulated network.

Core nodes emulate the received traffic in the following manner: They intercept all packets based on their IP address. When a packet is matched it is introduced into the ModelNet kernel module that begins with looking up the route for his source and destination. Then it creates an ID that points to the packet and schedules this ID to the appropriate pipes. These pipes are shared among all core nodes and have a max amount of queuing, this enables ModelNet to emulate congestion and packet drops. The pipe scheduling utilizes a heap of pipes sorted by the earliest exit time for the first packed. This scheduler executes once every clock tick and runs at the highest kernel priority. Each run goes through the heap of pipes looking for delayed (later than real time) packets, its first ID is removed and the packet is scheduled for deliver or enqueued to the next pipe, relative to the packet's destination. Next the scheduler calculates the new deadline for all packets dequeued earlier and reinserts the new pipes into the heap, sorted by the new deadline. To support multicore nodes a table that stores pipe ownership is created at the binding phase, so it can be looked up during run time.

This scheduler maintains emulation accuracy even in high cpu saturation due to the second operation having a higher level than the first. This means when cpu is overloaded incoming packets are dropped instead of losing emulation accuracy.

2.5.2.1 Limitations

ModelNet has a few limitations since VNs may not get their proper share of bandwidth due to simplifying the network in the distillation phase. When there is a need to scale the emulated network, you need to scale the internal network bandwidth of core nodes, scaling this physical network which can result in a bottleneck of scalability. Finally, Modelnet performance decreases with a high percentage of cross core traffic, and such it might not benefit from additional core nodes.

2.5.3 MiniNet

MiniNet [17] is a single machine network emulator, capable of running up to thousands of hosts in a single node. MiniNet achieves network emulation by utilizing lightweight virtualization built into the linux OS, more precisely by running processes inside network namespaces and using virtual Ethernet pairs. It emulates the following network devices:

- Links as a virtual Ethernet pair.
- Hosts as processes running inside each network namespace, they have their own virtual Ethernet interface and a pipe to the parent MiniNet process. This parent process is used to send commands and monitor child processes.
- Switches as it uses a software, OpenFlow, that provides switches with same semantics as the hardware switches, both user-space and kernel-space are supported.
- Controllers can be instantiated from inside or outside the system, as the controller must have IP-level connectivity with the switches within the system.

MiniNet-Hifi

MiniNet-Hifi [15] is a centralized container based network emulator that tries to solve the fidelity limitation of MiniNet (see Section 2.5.3). This limitation in MiniNet is due to all resources being multiplexed in the linux scheduler, which does not guarantee the fairness property.

MiniNet-Hifi solves performance fidelity using performance isolation, using control groups (cgroups) and CPU bandwidth limits. It groups processes in different cgroups and applies a maximum cpu quotas to each cgroup, ensuring that cpu time is fairly shared among all cgroups. This technique along with monitoring of performance fidelity allows MiniNet-Hifi to achieve resource isolation, provisioning, monitoring, and performance fidelity.

MaxiNet

MaxiNet [33] is an extension of MiniNet (see Section 2.5.3) that allows deployment of a cluster of workers. This cluster deployment is achieved by using GRE tunnels to connect nodes on different workers, these workers must be connected under the same switch to successfully create a tunnel, resulting in a locality constraint over its deployments.

2.5.3.1 Limitations

MiniNet and MiniNet-Hifi suffer from the same key limitation as they're both a single machine emulator, meaning that it cannot scale to larger network topologies.

MaxiNet, being an extension of MiniNet that solves the single machine key limitation, introduces the limitation that deployments must have locality in order to create the GRE tunnels (all machines in the same local network).

2.6 Other tools for testing

In this section we analyze tools used for testing software in the industry. These tools are offered in the context of facilitating the testing of distributed algorithms and providing stable environment for testing. Both of these tools are publicly available at their respective website.

2.6.1 TestGround

TestGround is a tool created by Protocol Labs for testing, benchmarking and simulating distributed and P2P systems, that enables the creation of distributed test plans [27]. We will analyze some parts of TestGround architecture, more precisely: how test plans are made and their properties, how it manages synchronization in distributed workloads, how it manages the network layer and shapes traffic in the network.

2.6.1.1 Test Plan

In TestGround a test plan is a collection of test cases that test a specific system or part of a system. They are built in a similar way to a unit test with a local API, as puppeteering and exposing an external API are not necessary. TestGround treats every test plan separately and each test plan must oblige with these factors:

- **Execution:** expose a single point of entry.

- **Input:** utilize a standardized runtime environment.
- **Output:** records results into a JSON schema and any additional output goes to a predetermined configuration path.

A test plan must have a manifest file as its root. This manifest file is written in TOML and must have: the given test name, which builders and runners to execute, the test cases desired with all its parameters and their respective information. When executing a test plan, a test run is created with a unique ID that is used for output collection and differentiation between outputs of different executions.

2.6.1.2 Synchronization

Synchronization is key for distributed workloads, where every node must know which action to perform on a given time. The solution found in the design of TestGround is to use a distributed coordination model. This is done by tests plans using an API that calls a synchronization store that offers distributed synchronization mechanisms. This API offers a REDIS client that provides synchronization mechanisms: signals, barriers and, publishing/subscribing mechanisms.

2.6.1.3 Network

TestGround network is split into two networks: control and data network. The control network is only used by TestGround services, while the test instances use the data network. This enables scenarios of network harsh conditions, without affecting the internal services.

Sidecar is a TestGround process that starts and configures both networks, it must be run in priviliged-mode and listens from request via the synchronization service. This process only runs on Docker or Kubernetes environments. It is responsible for initializing the network interface, applying network configuration requests via NetLink and doing garbage collection on inactive entries in the sync service.

2.6.1.4 Traffic Shaping

Traffic shaping in TestGround is the ability of a test instance to change its own network properties, such as IP address, latency or bandwidth. To change its network properties an instance must use a network client to communicate with the sidecar, and such client must send a network configuration containing all the desired properties. If you want to change your own IP address, you must generate a unique ID to avoid conflicts, you can use your unique ID or if you do not have one it can be requested from the sync service. After having your own ID sequence, you retrieve the test subnet from the runenv and change your IP, this is only supported in IPv4 addresses. After sending your network configuration to the sidecar, it will apply it and send a signal after confirming the changes.

2.6.1.5 Limitations

TestGround's limitations exist mostly on the network it emulates. It's network is shaped using linux TC and configures a TC tree consisted of the HTB Qdisc, followed by the HTB class, followed by the Netem QDisc. Since only one queue is supported at the moment, only one value of latency, bandwidth, is applied for every node in the network, resulting in a highly homogenous and unrealistic environment.

2.6.2 Chaos Monkeys

Chaos monkeys is a toolset that appeared in the context of chaos engineering, as to ensure that individual service fails will randomly happen in production environment such that developers code must be resilient to such failures to provide a service with high availability [5]. This tool was born at Netflix as to oblige its engineers to make their code resilient to individual service failures [10]. It is configured using a TOML file, with information on how the chaos monkey should terminate containers, how to access the MySQL database, how to access the spinnaker interface, and if it allows dynamic configuration. Chaos monkeys require a MySQL database to save their daily termination schedule and to carry out a minimum time between terminations. Another important aspect of chaos monkeys is that it uses two groups of containers, the eligible for termination or not. Lastly, it logs the amount of errors occurred for later analysis, and it checks if there is an outage present before terminating a container, as to not cripple the system during unplanned outages.

2.6.3 Discussion

These corporate tools are designed mainly for internal use or are mostly focused on the development, design, and test of their products. The work proposed in this thesis complements such tools, in a sense that it provides an emulated environment with a high level of control and realism for researchers to validate and test their solutions, which does not depend on the type of solution being tested.

2.7 Discussion

In this chapter we presented how to evaluate distributed systems, where we discussed different methods of evaluation and some relevant metrics to be measured when conducting evaluations of distributed systems. Then, we provided important concepts about relevant networks to our ware and how to control each network, followed by an analysis of a tool to shape the network to a set of desired properties. Next, we looked into simulation as a means to materialize a network model, followed by the analysis of some simulators. We moved on to providing an isolated execution environment, where containerization and virtualization were discussed. We further developed containerization by providing definitions on Docker and the most popular container orchestrators available. Furthermore, we discussed existing wares that focus on emulation, providing insight on those wares and their limitations. Lastly, we discussed other tools used by the industry to test distributed systems and how they relate to the work proposed in this thesis.

In the following we present our proposed solution and present our calendar of the work to be done.

Chapter 3

Proposed Work

In this chapter, we first detail a proposed solution, where we discuss the requirements for such solution and an architecture that can serve as a basis to provide all the relevant requirements, along with a proposed prototype (Section 3.1).

Next, we address on how to evaluate our solutions, starting by comparing with the sate-of-the-art tools, and presenting the parameters with which we will compare those tools to our own (Section 3.2).

Lastly, we present a work plan schedule to be followed in the future (Section 3.3).

3.1 Proposed Solution

Our proposed solution, as mentioned in Section 1, is to devise and implement a tool that assists in the planning and execution of experiments of large-scale distributed systems, using containers and network emulation. The requisites for our solution are its ease of use, ability to scale experiments for large distributed systems, and high emulation accuracy.

To address this challenges, we propose the following architecture: A manager node per physical machine that communicates with other manager nodes to start and coordinate the experiment, that receives the information necessary to start an experiment (network topology, dynamic events). This manager node, when starting the experiments would create containers with an application image that would extend the base image required to run the emulation scripts (linux tc scripts). The application node would be controlled by its locally deployed manager, in the case of dynamic experiment events, like link failures, these are also applied by the local manager. This architecture can be seen in Figure 3.1.

This architecture allows us to have a prototype that runs in every physical machine allocated to the experiment. We plan to have our prototype receive an experiment file

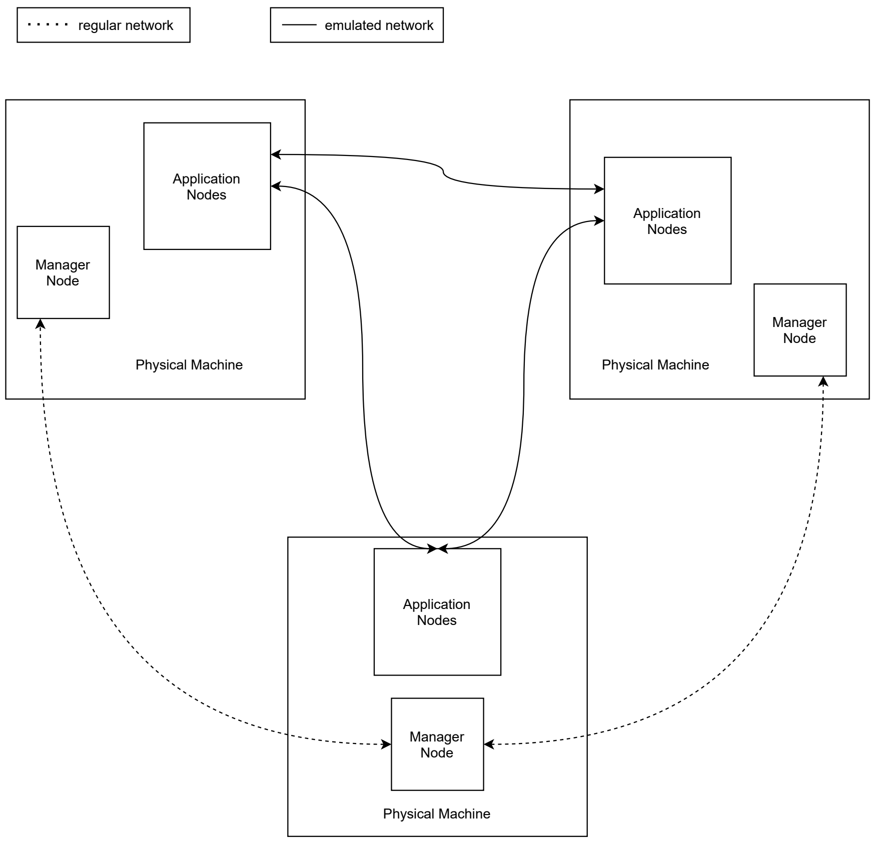

Figure 3.1: The proposed solution architecture.

describing every step in the experiment. More precisely, this file would describe the target network topology (including each link target properties, latency matrix, etc), a script on what events must occur, the time duration of the experiment, a description on which application nodes image to launch, and directory in which the experiment logs will be stored. After that, the prototype would parse the experiment file and send information to the coordinators in other physical machines, in order to balance the number of nodes in a single physical machine. When every coordinator has received the experiment information, they can start the experiment. During the experiments, the application nodes would send logs to and receive dynamic events through their respective local coordinator. After the experiment is complete, the coordinators terminate all application nodes and collect the remaining logs.

3.2 Evaluation Plan

To evaluate this work we will do a comparison between our work and the state-of-the-art tools for planning and executing experiments, do a comparison between our tool's emulations capabilities and the state-of-the-art emulators (e.g. Kollaps 2.5.1, MiniNet

2.5.3, etc.). The work will be evaluated in the following topics:

- Number of nodes supported, as to measure the capability of running experiments in large scale distributed systems.
- Volume of metadata, to measure scalability of our solution prototype with the increase of applications nodes and physical machines.
- Emulation accuracy, latency, bandwidth. To validate that the target topology is being correctly enforced.
- Emulation accuracy in different flow environment, mostly short living flows vs long living flows. To see if the longevity of flows has an impact on the accuracy of emulation.
- Usability of the proposed solution.
- Execution time.

To test the usability of our prototype, we propose to have users, with at least a bachelor's in computer science, execute a specific experiment for a distributed system and measure the time it takes for them to complete the task. We will compare those time results, with users of the same academic background, but instead of using the proposed solution prototype to build and execute the experiment, they will have to construct the scripts necessary to run the same experiment.

3.3 Work Plan

In this section we propose the following schedule for the planning of our work, as seen in Figure 3.2. This plan has a duration of 7 months, and is divided in the following tasks:

- **Preliminary Solution** is a solution that can receive a network topology, creating the corresponding containers, while generating the linux tc scripts for each container, creating the emulated network. After the mentioned solution is implemented, we will validate the emulation capabilities with iPerf3 clients and evaluate it through testing of a distributed hash table application.
- **Final Solution** is to iterate on the previous solution, trying to optimize the tool. This solution is also validated through iPerf3 clients and evaluated through a DHT application.
- **Final Evaluation** is to extract metrics on the state-of-the-art tools and emulators to compare against the work described above, and conduct users tests to ascertain the usability of our ware.

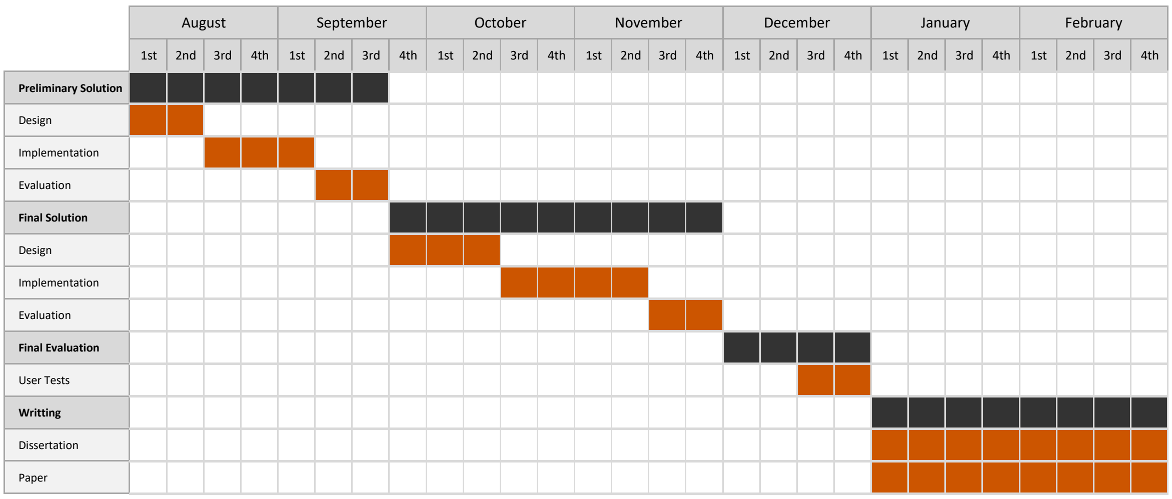

Figure 3.2: Gantt chart of the proposed work schedule.

- **Writing** is divided into two phases. First is writing the dissertation of this work, second it's writing the resulting paper.

Bibliography

[1] W. Almesberger et al. *Linux network traffic control—implementation overview*. 1999 (cit. on p. 8).

[2] *ArchLinux: Advanced traffic control*. https://wiki.archlinux.org/title/Advanced_traffic_control, note=Accessed: June. 2021 (cit. on p. 9).

[3] P. Bailis, K. Kingsbury, and J. Networks. "An informal survey of real-world communications failures". In: (), p. 13 (cit. on p. 7).

[4] H. Ballani et al. "Towards Predictable Datacenter Networks". In: *SIGCOMM Comput. Commun. Rev.* 41.4 (Aug. 2011), pp. 242–253. issn: 0146-4833. doi: 10.1145/2043164.2018465. url: https://doi.org/10.1145/2043164.2018465 (cit. on p. 8).

[5] A. Basiri et al. "Chaos Engineering". In: *IEEE Software* 33.3 (2016), pp. 35–41. doi: 10.1109/MS.2016.60 (cit. on p. 28).

[6] K. Benzekki, A. El Fergougui, and A. Elbelrhiti Elalaoui. "Software-defined networking (SDN): a survey". In: *Security and Communication Networks* 9.18 (2016), pp. 5803–5833. doi: https://doi.org/10.1002/sec.1737. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.1737. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1737 (cit. on p. 8).

[7] D. Bernstein. "Containers and Cloud: From LXC to Docker to Kubernetes". In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84. doi: 10.1109/MCC.2014.51 (cit. on p. 18).

[8] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. 2nd. Springer Publishing Company, Incorporated, 2011, pp. 1–40. isbn: 3642152597 (cit. on p. 5).

[9] R. N. Calheiros et al. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms". In: *Software: Practice and Experience* 41.1 (2011), pp. 23–50. doi: https://doi.org/10.1002/spe.995. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.995. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.995 (cit. on p. 14).

[10] *ChaosMonkeys documentation: ChaosMonkeys Overview.* https://netflix.github.io/chaosmonkey/, note=Accessed: June. 2021 (cit. on p. 28).

[11] G. DeCandia et al. "Dynamo: Amazon's Highly Available Key-Value Store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: https://doi.org/10.1145/1323293.1294281 (cit. on p. 2).

[12] *Docker documentation: Docker Overview.* https://docs.docker.com/get-started/overview/, note=Accessed: June. 2021 (cit. on pp. 19, 20).

[13] P. Gouveia et al. "Kollaps: Decentralized and Dynamic Topology Emulation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387540. URL: https://doi.org/10.1145/3342195.3387540 (cit. on p. 21).

[14] H. Gupta et al. "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments". In: *Software: Practice and Experience* 47.9 (2017), pp. 1275–1296. DOI: https://doi.org/10.1002/spe.2509. eprint: https://www.onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2509. URL: https://www.onlinelibrary.wiley.com/doi/abs/10.1002/spe.2509 (cit. on pp. 14, 15).

[15] N. Handigol et al. "Reproducible Network Experiments Using Container-Based Emulation". In: *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. CoNEXT '12. Nice, France: Association for Computing Machinery, 2012, pp. 253–264. ISBN: 9781450317757. DOI: 10.1145/2413176.2413206. URL: https://doi.org/10.1145/2413176.2413206 (cit. on p. 25).

[16] *Kubernetes documentation: What is Kubernetes.* https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, note=Accessed: June. 2021 (cit. on p. 20).

[17] B. Lantz, B. Heller, and N. McKeown. "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Monterey, California: Association for Computing Machinery, 2010. ISBN: 9781450304092. DOI: 10.1145/1868447.1868466. URL: https://doi.org/10.1145/1868447.1868466 (cit. on p. 25).

[18] *Linux TCP Small Queues.* https://lwn.net/Articles/507065/. Accessed: May. 2021 (cit. on p. 22).

[19] L. Massoulie and J. Roberts. "Bandwidth sharing: objectives and algorithms". In: *IEEE/ACM Transactions on Networking* 10.3 (2002), pp. 320–328. DOI: 10.1109/TNET.2002.1012364 (cit. on p. 22).

[20] A. Medina et al. “BRITE: an approach to universal topology generation”. In: *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 2001, pp. 346–353. DOI: 10.1109/MASCOT.2001.948886 (cit. on p. 14).

[21] A. Montresor and M. Jelasity. “PeerSim: A scalable P2P simulator”. In: *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*. IEEE. 2009, pp. 99–100 (cit. on p. 13).

[22] S. Naicken et al. “A survey of peer-to-peer network simulators”. In: *Proceedings of The Seventh Annual Postgraduate Symposium, Liverpool, UK*. Vol. 2. 2006 (cit. on p. 13).

[23] *Protocol Labs website*. https://protocol.ai/about/, note=Accessed: July. 2021 (cit. on p. 3).

[24] G. F. Riley and T. R. Henderson. “The ns-3 Network Simulator”. In: *Modeling and Tools for Network Simulation*. Ed. by K. Wehrle, M. Güneş, and J. Gross. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 15–34. ISBN: 978-3-642-12331-3. DOI: 10.1007/978-3-642-12331-3_2. URL: https://doi.org/10.1007/978-3-642-12331-3_2 (cit. on p. 16).

[25] J. Smith and R. Nair. “The architecture of virtual machines”. In: *Computer* 38.5 (2005), pp. 32–38. DOI: 10.1109/MC.2005.173 (cit. on p. 17).

[26] A. Sulistio, C. S. Yeo, and R. Buyya. “A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools”. In: *Software: Practice and Experience* 34.7 (2004), pp. 653–673. DOI: https://doi.org/10.1002/spe.585. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.585. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.585 (cit. on p. 13).

[27] *TestGround documentation: TestGround Overview*. https://docs.testground.ai/, note=Accessed: June. 2021 (cit. on p. 26).

[28] *The Network Simulator ns-2*. http://nsnam.sourceforge.net/wiki/index.php/User_Information, note=Accessed: June. 2021 (cit. on p. 16).

[29] *Traffic Control How To*. https://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html, note=Accessed: June. 2021 (cit. on pp. 8, 10, 12).

[30] J. Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014 (cit. on p. 17).

[31] *Universal 32bit Filter*. https://man7.org/linux/man-pages/man8/tc-u32.8.html, note=Accessed: May. 2021 (cit. on p. 11).

[32] A. Vahdat et al. “Scalability and Accuracy in a Large-Scale Network Emulator”. In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2003), pp. 271–284. ISSN: 0163-5980. DOI: 10.1145/844128.844154. URL: https://doi.org/10.1145/844128.844154 (cit. on p. 24).

[33] 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 p. 26).