ANDRÉ DE OLIVEIRA ATALAIA
Bachelor in Computer Science and Engineering

SYMPHONY: A SCALABLE EMULATOR FOR DISTRIBUTED SYSTEMS

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
June, 2023

DEPARTMENT OF
COMPUTER SCIENCE

SYMPHONY: A SCALABLE EMULATOR FOR DISTRIBUTED SYSTEMS

ANDRÉ DE OLIVEIRA ATALAIA
Bachelor in Computer Science and Engineering

Adviser: João Leitão
Associate Professor, NOVA University Lisbon

Examination Committee

Chair: Pedro Abílio Duarte de Medeiros
Associate Professor, NOVA University Lisbon

Rapporteur: João Nuno de Oliveira e Silva
Assistant Professor, Instituto Superior Técnico

Adviser: João Leitão
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING
SPECIALIZATION IN SPECIALITY NAME

NOVA University Lisbon
June, 2023

Symphony: A Scalable Emulator for Distributed Systems

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.10) [23].

Acknowledgements

I would like to express my gratitude to the few selected people who helped me during the most challenging times in the course of this thesis. First, I would like to thank my advisor, João Leitão, for his wisdom that helped shape the ideas that birthed this work.

I also want to extend my deep thanks to two colleagues, Pedro Ákos Costa and Pedro Fouto, who served as a constant source of inspiration, motivation, and guidance throughout the course of this work.

I am also deeply indebted to the Department of Informatics of the Nova University of Lisbon for providing me with the resources, facilities, and opportunities necessary to pursue my academic goals.

Finally, I would like to thank my family and girlfriend for their unwavering love, support, and encouragement. Their belief in me, even during the most challenging times, has been a source of strength and inspiration.

This work was partially supported by the project NG-Storage (contract PTDC-CCI-INF/32038/2017), NOVA LINCS (grant UID/CEC/04516/2013), and the European Commission through the TARDIS Project (grant agreement no. 101093006).

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, with users having 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 for 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 introduce a tool that assists in defining and executing experiments for large scale distributed systems, using container technology and network emulation. We present 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), to ensuring that every application node receive their corresponding events. Lastly, we evaluated our tool by conducting experiments and comparing it against state-of-the-art tools, evaluating their performance when emulating large distributed systems.

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

Resumo

Os 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, introduzimos uma ferramenta que assiste na definição e execução de experiências para sistemas distribuidos de larga escala usando a tecnologia de *containers* e emulação de rede. Introduzimos uma arquitectura onde existem um nó gerente por máquina física, sendo este nó responsável por todos os nós aplicacionais naquela máquina física. Esta responsabilidade vai de receber informação relevante para a experiência (i.e. topologia de rede, propriedades das ligações, e eventos dinamicos) de modo a assegurar que cada aplicação recebe os seus correspectivos eventos. Por ultimo, realizámos experiências e comparámos a nossa ferramenta com outras existentes, avaliando a sua performance ao emular grandes sistemas distribuidos.

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

Contents

List of Figures

List of Algorithms

Acronyms

DI Cluster	Nova Department of Informatics Cluster *(pp. 56, 57)*
Linux tc	Linux Traffic Control *(pp. iv, v, x, 2, 11–13, 23, 28, 30)*

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 increasingly strict. In this context, it becomes essential to reliably measure key performance indicators, such as latency or throughput, in a realistic execution environment.

Creating realistic experimental conditions, when addressing large-scale distributed system, such as peer-to-peer systems, where thousands of processes are executing on 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, while the network conditions, will in most tools, not fully capture the intrinsic characteristics of 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 defined and 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 the complexity of setting up Linux tc.

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 [12]. 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 downright 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. Experiments that require the use of multiple machines can be costly when conducted on physical hardware. However, one way to address this issue is by using container technology to reduce the overall cost.

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. To extract reliable metrics, it is necessary to control the environment in which the experiment is executed, which includes controlling the underlying network. Emulation techniques can be used to control the underlying network and create a controlled environment.

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 our work tackles this lack of adequate solutions.

1.3 Contributions

The main contributions of this dissertation are the following:

- A new tool, named Symphony, that simplifies the planning and execution of experiments for large-scale distributed systems, taking into advantage container technology and realistic network emulation.

- An experimental evaluation of Symphony that shows the benefits of Symphony when comparing against a state of the art network emulator, both in their capability of emulating a single link to how they scale against larger distributed systems.

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 [30]. 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 containers orchestrators options. 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 we present Symphony. Starting by stating its requirements, followed by the system architecture. After that, we present the implementation of Symphony, explaining how each component is implemented and discussing the implementation details. Lastly, we present a possible service architecture implementation, which will be used in the following experiments.

- **Chapter 4** is where we document how to use and configure the work done in this dissertation. We start by documenting each experiment configuration file parameter, stating its meaning and its possible values. Followed by an explanation on how to configure the service to have it executing on a cluster of physical machines.Then we state how an user can interact with the service and best practices on how to do it. Lastly, we present and explain a few examples of experiment configuration files.

- **Chapter 5** is where we evaluate the work done in this dissertation. We start by presenting the evaluation goals. Followed by the experiment use cases that will be used to evaluate our work. Then we present the environment where all uses cases where executed. Lastly, we present and discuss each use case experimental results and their corresponding configuration.

- **Chapter 6** is where we present the conclusion of the work done in this dissertation and present the future work.

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 work done 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 done 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 [9]:

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) [7] 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 [37], 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 be dequeued into the hardware device, but this can depend on the existing qdisc. The most popular classfull qdisc are the Hierarchy Token Bucket and the Hierarchical Fair Service Curve.

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:

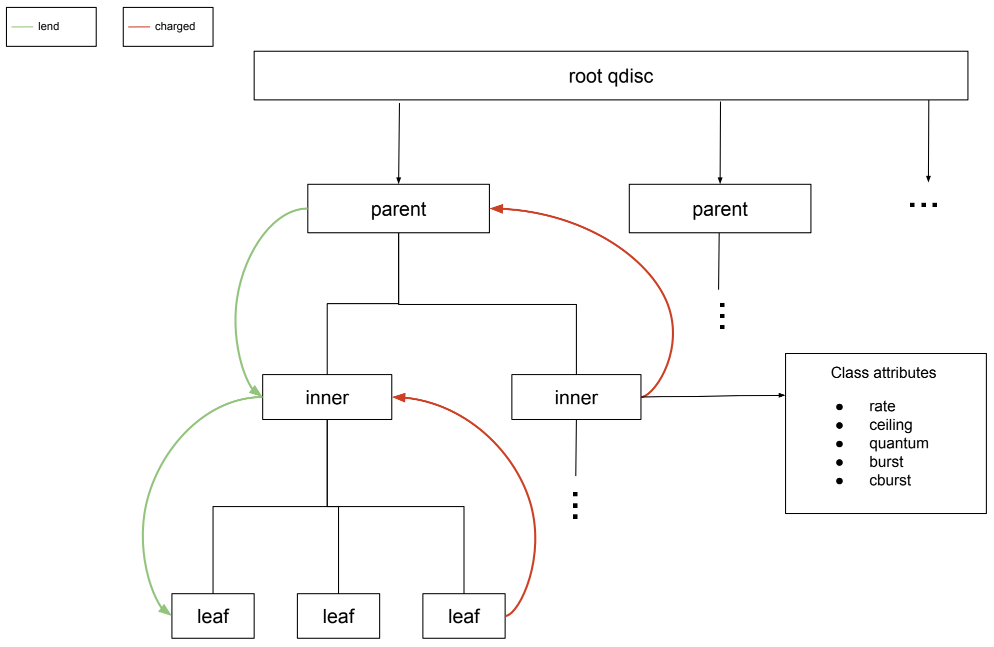


Figure 2.1: Flow of borrowed tokens in an HTB qdisc, adapted from [37].

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.

Hierarchical Fair Service Curve (HFSC) uses the network packing scheduling algorithm [33], to guarantee precise bandwidth and delay allocation for its leaf classes and to fairly allocate excess bandwidth within the class hierarchy minimizing the gap within the service curve and the service provided. In order to understand the HFSC qdisc, first we need to understand service curves. A service curve is a non decreasing function that refers to some time unit and dictates the amount of service at each specific point of time. These curves must allow taking into account idling periods and evaluating based on earlier active periods [24].

The HFSC is devised into three criterion: realtime criterion, linksharing criterion, and the upper limit criterion. Realtime criterion ignores class hierarchy to guarantee precise bandwidth and delay allocation. It uses the packets real time and eligible time, in order to send the packet with the shortest deadline that has a real time higher than the eligible time.

Linksharing criterion is responsible for distributing bandwidth across the class hierarchy. This criterion does not compare real time with virtual time, it only uses the virtual times of all active sub classes in order to schedule the one with the lowest virtual time. Although this approach works, it cannot guarantee fairness with non linear service curves.

Upperlimit criterion is an extension of linksharing criterion that only permits sending if packets real time is lower than fit time. This means that it limits the bandwidth available for all its children while updating each of their virtual time in order to guarantee fairness of bandwidth allocation.

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 [39]. 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.

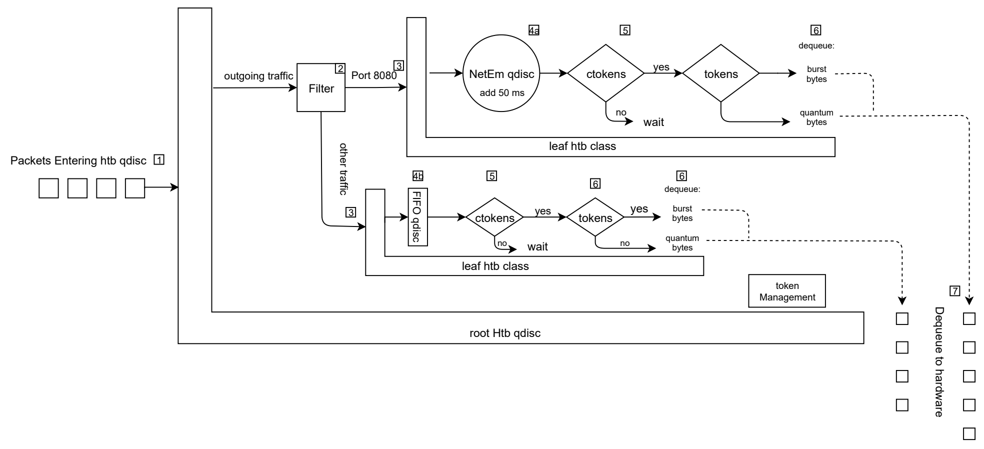


Figure 2.2: Example of shaping with Linux tc, adapted from [37].

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.

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 [29]. 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 [34]. 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 [28], 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 [10] 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 [27], 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 [19] 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.

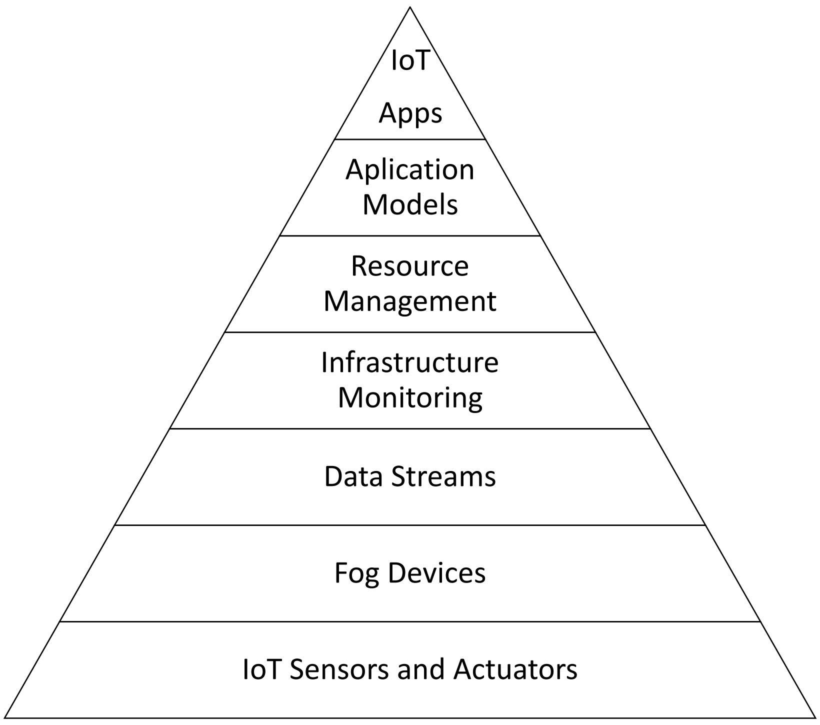


Figure 2.3: iFogSim's architecture divided into components, adapted from [19].

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 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 [36], **NS-3** [31] 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 [32]. 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.

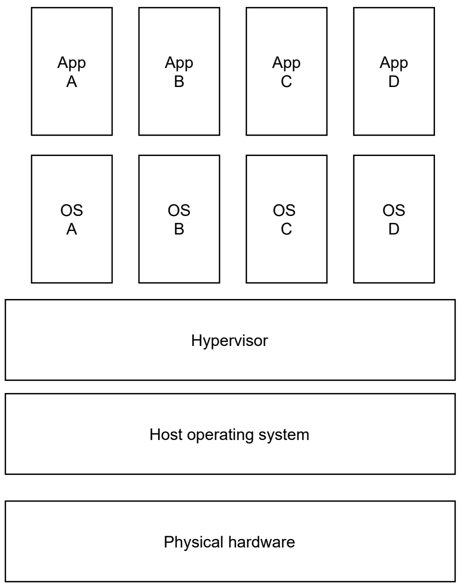


Figure 2.4: Virtual machines deployment, adapted from [8].

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 [38]. 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 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 opted 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).

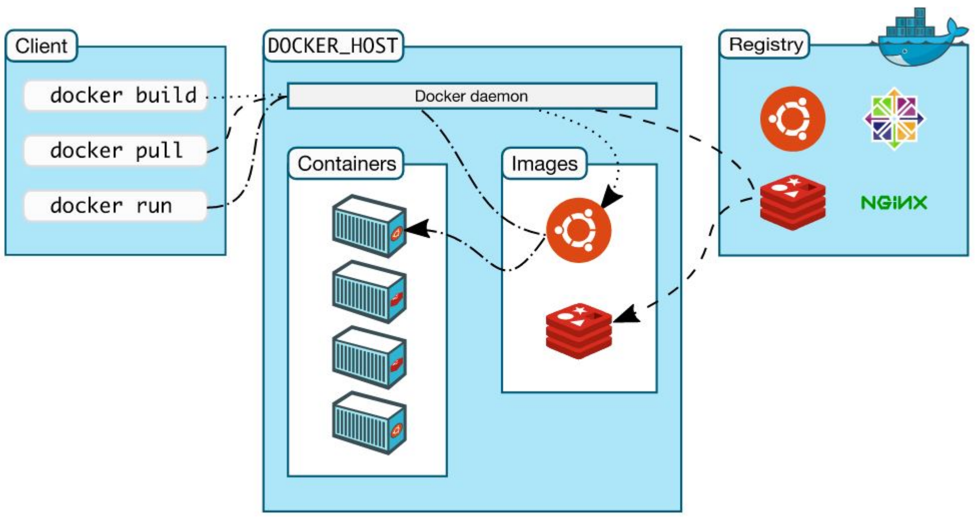


Figure 2.5: Docker architecture, taken from [13].

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.

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.

Kubernetes is a platform that offers automated deployment, scaling, and operations of application containers [21]. 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.

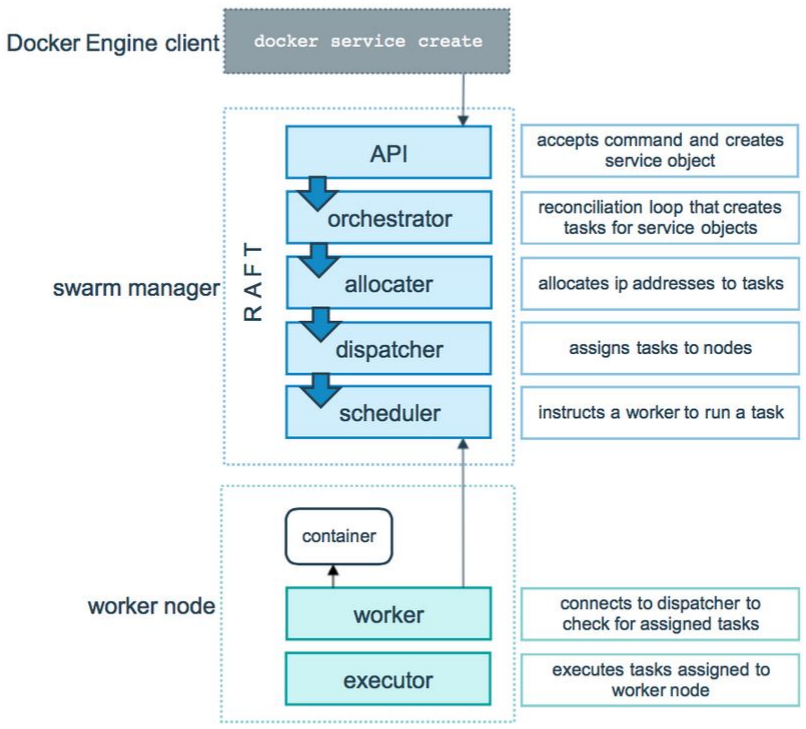

Docker Engine client
docker service create
swarm manager
RAFT
API
accepts command and creates service object
orchestrator
reconciliation loop that creates tasks for service objects
allocater
allocates ip addresses to tasks
dispatcher
assigns tasks to nodes
scheduler
instructs a worker to run a task
worker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

Figure 2.6: Flow of service create request in docker swarm, taken from [13].

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 [18]. 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 syntax, and should identify all services, links, bridges, and dynamic elements. 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. It is due to note that links can only have as a source or destination a previously named bridge or service. If a bi-directional link is described then two uni-directional links will be create with the same 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 affect 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 [26]. 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 try to distributed unused shares among competitor links.

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 htb qdisc will always accept the incoming packets, while the buffer from network devices will drop incoming packets when full. The qdisc capable of dropping packets is netem qdisc. To address these congestion limitations Kollaps will observe all requests of 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 Linux traffic control layer 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 run time 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.

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 in writing topology files for peer-to-peer experiments. This limitation is due to the fact that it does not allows to describe multiple copies of containers within a service and describe the links for those containers. This results in systems where it has thousands of the same application with similar configuration, the user is obliged to describe each peer as a service, resulting in extra lines of boilerplate configuration.

In the same topic of peer-to-peer experiments, Kollaps as a maximum of links that can be defined in an experiment. This maximum value is 65535 links, resulting in being unable to emulate large scale systems that require an higher link number than 65535. If we make no assumptions on how the network is composed and have each peer have a link to every peer, then the maximum number can be surpassed with a system of 257 peers.

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 [40] 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 [22] 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 [20] 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 [41] 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 [35]. 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 Linux 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 [6]. This tool was born at Netflix as to oblige its engineers to make their code resilient to individual service failures [11]. 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 done 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 done in this thesis.

In the following chapter, we present our architecture and it's requirements, followed by two possible adaptations.

3 Symphony

In this chapter we present our contribution: A framework capable of managing and executing experiments with a realistic emulated environment, Symphony. Symphony requirements, and how its various components interact with each other. Symphony is a network emulation system agnostic of application that offers network emulation on a cluster of physical machines, through the usage of docker containers, docker swarm and Linux tc.

We detail how each component of Symphony is implemented, further diving into the implementation challenges, and explaining why we chose each solution.

Lastly, we present one possible architecture implementation of symphony with a service architecture on a cluster of physical machines.

3.1 Requirements

As mentioned in Chapter 1, distributed systems have become a pillar of our daily lives, therefore the need to reliably tests those systems has increased. Testing Distributed systems can be done in three ways: simulation, emulation, and physical deployment. However if we want to test a distributed system in a realistic environment, emulation becomes the most suitable option because physical deployment has a great financial cost and simulation does not offer accurate results since the logic being executed is not exactly the same as in a real deployment.

Emulation, more precisely, network emulation has existed for years, but as distributed systems got bigger and bigger there is a need for more scalable network emulation. For that we propose Symphony, a scalable emulation tool that enables users to customize their experiment environments to achieve reliable results.

Our solution needs to comply with the following requirements:

Emulation scalability: As distributed systems have increase in size, so our solution needs to be able to scale for large distributed systems.

Ease of use: As users should be able to create experiments, to easily express their desired experiment environments without writing hundreds of configuration lines.

Produce reliable results: Researchers need to be able to trust the provided results, so they can be analyzed without questioning if they are correct or not.

Abstract of application: The network emulator should view the application as a black box. This means to be abstract of all application details, such as programming language, architecture and so on.

Realism: As emulation should provide an high degree of realism for results to be comparable with an actual deployment, and provide an insight of the application behaviour when deployed in the real world.

3.2 System Architecture

The system architecture, as can be seen in figure 3.1, revolves around four main components: Symphony Server, Symphony Manager, Symphony Worker, and Symphony Client.

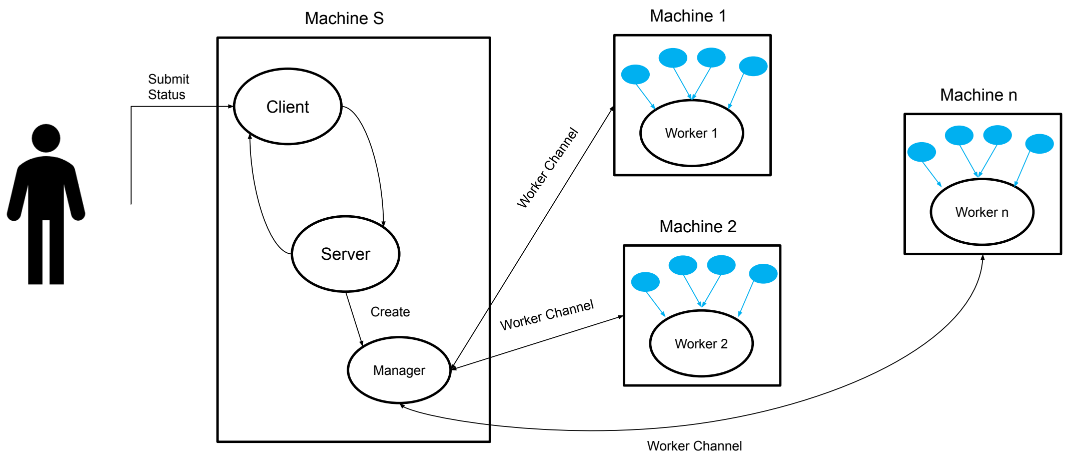


Figure 3.1: Symphony components architecture.

The Symphony-Server component responsibilities are: communication with other servers components if they are available; to receive request from clients, and to launch workers or managers, as seen in figure 3.1. The client component purpose is to send requests to servers to launch experiments or check on their status to the server component.

An experiment is modelled as a single manager and multiple workers, with their responsibilities being: The manager is responsible for parsing the experiment, generating all experiment data, prepare the emulation module, distribute containers across workers. The manager also does the job of coordinating the workers between each step of executing an experiment.

Workers receive their experiment data from their manager, start and run their corresponding containers, apply the emulation module and generate timers for experiment actions. Workers require two modules to execute experiments, a docker interface and a emulation module. This worker architecture has the following structure 3.2:

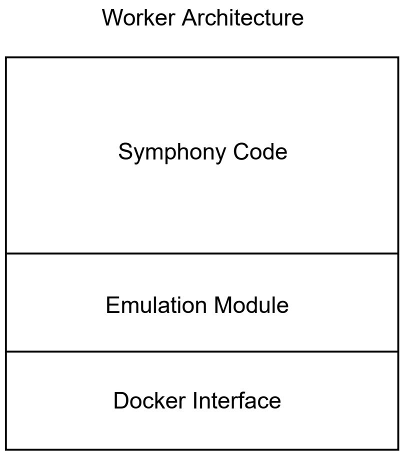


Figure 3.2: Worker component architecture.

Emulation module

The emulation module is responsible to apply the network emulation to each containerized application. This process must be agnostic of application, to achieve this two options were theorized: either spawn another container attached to the same network namespace or have a master process that jumps from each application network namespace applying the correct network emulation.

The chosen option was to attach another container to the same network namespace. The discussion that led to this choice can be seen later at 5.4.1.

Docker Interface

The Docker interface allows the worker component to communicate with the Docker daemon running on the local machine. This interface needs to offer all the available Docker commands and return an output indicating whether the command was executed successfully or failed. The output should also include any error messages, so the user can recognize their mistakes when using this Docker interface.

When choosing which framework to utilize on this interface we faced two options: either use the java process interface and wrap all docker daemon commands, or use the docker-java API [14] to build this interface.

We decided to go with the java process interface and wrap all docker daemon commands because it allowed for a more customization and its learning curve was less stepper than the docker-java API. We also took into consideration the performance point of view of both options and presume that the docker-java API might have a slight edge over the process interface, but this edge is insignificant when considering the time that would need to be invested to develop using the docker-java API.

Communication layer

The communication layer is present in all Symphony components as they use it to manage and establish TCP connections between each other and to send messages through those connections. To implement this communication layer, we use the Babel framework [16] which allows the creation of channels where components can send messages and manage the open connections. Additionally, Babel provides a timer feature, which is essential for components to delay certain executions by a certain amount of time.

3.3 Implementation

In this section, we present how each component of Symphony is implemented, further diving into the implementation challenges, and explaining why we choose each solution.

3.3.1 Symphony-server

Symphony server is a component that handles client TCP connections, manager TCP connections, receives commands to start experiments, launches manager and worker components, and maintains a membership of other active servers. This component can be modelled as seen in algorithm 1.

As we can see from the algorithm 1, when initializing this component it receives the membership of the system, and tries to establish connections to all other machines in its membership. We implemented this method with a timeout to retry establishing the connection, so the membership will be eventually complete. After receiving a client connection with its respective command, it first checks if the command is to submit an experience so we can launch a manager with the experiment data, or else handle other client command such as *stop* and *ls*.

The server component receives a "start workers"request message from the manager containing the address of the destination workers, which are the same as the destination servers. Then when a server receives a start worker it will launch the corresponding worker process.

The last job of the server component is to launch manager and worker components. This is implemented using "java run time exec"to execute a launch script that initializes a manager or worker in a separated process. The separated process is a essential part because both the manager or worker component will exit after the experiment is complete or if some fatal error occurred during the setup of the experiment, so in order to preserve the process in which the server is running they must execute in a separate process.

Algorithm 1: Symphony Server

```
Upon Init (membership) do:
   Trigger setupMembership ( membership);

Upon Receive (ClientConnection, command, data) do:
   If command = submitExperience then
      Call launchManager (data);
   Else
      Call handleClientCommand(command)

Upon Receive (ServerConnection, from ) do:
   If $from \notin$ membership then
      membership $\longleftarrow$ membership $\cup \{from\}$;
      Trigger Send (ServerConnection, from);

Upon Receive (StartWorkersRequest, manager, workersAddresses ) do:
   Foreach $host \in workersAddresses$ do
      If self = host then
         Call (StartWorker, manager,self);
      Else
         Trigger Send (StartWorker, manager,host);

Upon Receive (StartWorker,manager, host ) do:
      Call launchWorker (manager);

Upon Receive (ExperienceError,error, from ) do:
   If clientConnection is up then
      Trigger Send (ExperienceError,error,client);
```

3.3.2 Experiment

In this section we will discuss on the experiment architecture previously mentioned in section 3.2 is implemented. This architecture revolves around the manager and worker components and how they interact in order to execute an experiment. These components were implemented to be independent, meaning that they would operate if the rest of the components were offline, that said they only rely on those components to start an experiment. Both the experiment components are shown in diagram 3.3:

3.3.2.1 Manager

Symphony manager component is responsible for parsing the user input, setup the experiment and manage all experiment worker components. The user input is received as a YAML file, or a default YAML file and a override YAML file, that contains all the

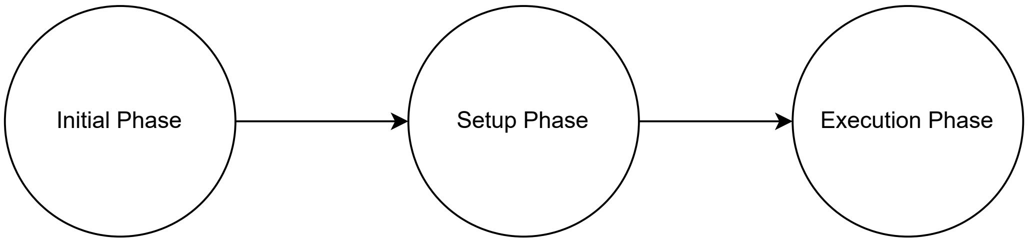


Figure 3.3: Diagram of manager and worker component phases.

experiment data.

Initial Phase

Following the diagram, the first step is when the component is initialized. This initial step contains two major operations: the reading of the experiment configuration file and the generation of each application container name and IP address. The process of reading the YAML file is by opening each file and mapping each value to a java class using the snakeyaml constructor, then all fields will be mapped to their respective value and if their value is absent, it will be represented as null. To implement the default YAML file that is overridden by another YAML file, we use a sequence input stream that takes as the first input the default YAML file, and second the override YAML file. This operation of reading the default input stream first and then the override input stream works because when mapping a YAML file to a java class, snakeyaml creates a data structure of containing each key and their respective value. Since there cannot be duplicate keys, when the overridden file is read, it updates the value of all its containing keys. Using this method allows for users to not repeat the same configuration across different experiments, and instead use the same default file, with different overridden files.

The name and IP address generation process is determined by the order in which groups are presented in the configuration file. In broad terms, a group is a collection of applications that execute on specific physical machines. We will discuss how to create groups and their respective parameters in detail later 4.1.1.8. When a group is selected for the generation process, it will generate each name by concatenating node, the group name and the order of its given IP address, for example if a group start address is 192.85.7.0 "node-groupname-1"will relate to the IP address of 192.85.7.1 . The IP address are given sequentially for each application within a group, starting at the groups start address. If no start address is defined, the groups IP addresses will start at the lowest IP address still available within the provided subnet.

Setup phase

This phase of the manager component starts with a request to the local Symphony server with the addresses of all experiment workers. After all workers have established a connection with the manager it starts the second part of the setup phase, as we can see from the figure 3.4.

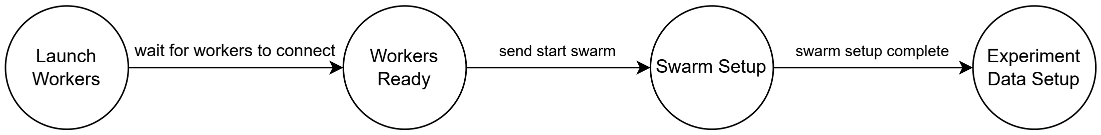


Figure 3.4: Diagram of setup phase in the manager component.

This second part of the setup phase is to save all images used in the experiment so they can be loaded by workers if needed, and to initialize and join the docker swarm. These application docker images will be loaded onto the a directory across all machines so workers can load an image if necessary. Initializing the docker swarm is a task for the first worker, who will initialize the swarm and become its leader, then the manager will await for an response from the leader worker containing the token to join the swarm and broadcast that token to the other workers. Once all workers have joined the swarm part three of the setup phase will commence.

This third part of the setup phase is when the manager sends all of the experiment data to the workers. This phase starts by the manager sending each worker their respective data, which means sending each worker all the experiment information that they must execute and their respective actions. This process is done group by group, distributing the applications among the groups machines in a round-robin manner. After completing the previous task, we will have a set of applications per worker and we will send each worker their respective data. The final task of this phase is the distribution and parsing of all experiment actions per worker. The actions are parsed and grouped by worker, which means that if an action is to be executed in a application a worker is running, then the action needs to be sent to that worker. Once all the actions are grouped, they are sent to their respective workers, and the setup phase is complete after all workers acknowledge their actions.

Execution phase

This phase of the manager component starts when all workers have acknowledge receiving their experiment data and actions. When that state is achieved, the manager will send to all workers a message to start spawning the application containers and apply the network emulation rules.

When all workers signal the manager that they have completed applying the network emulation rules, the manager will send a message to all workers to start executing the applications. This waiting barrier is used to try to mitigate different execution time between different application within an experiment.

After receiving an acknowledge that all workers have finished launching their application, the manager will send all workers a message to start their experiment timers. Then the manager will await for all workers to acknowledge that they finished executing the experiment to terminate its process.

3.3.2.2 Worker

Symphony worker component receives experiment data from the manager and starts each application container and their respective emulation module. The worker is responsible for creating timers for all its respective actions. Finally, the worker will report any errors that occurred during setup or when executing the application to his manager.

Initial Phase

This phase of the worker component occurs after it is initialized by the server component. This phase is just to establish a connection with its experiment manager. After all workers have established their connection with the manager the next phase will start.

Setup phase

This phase of the worker component happens when the manager sends a message to the first worker to create the docker swarm. The first worker will then create the docker swarm and be its leader, re transmitting the swarm token to its manager, which will then broadcast said token to the other workers. After all workers have acknowledge to join the docker swarm, the next step of the setup phase begins.

The next step of the setup phase is to transmit the respective experiment data to each worker. When a worker receives the experiment data, it will store that data and await for the start setup message. When the start setup message arrives, the worker will loop though all the received docker images, and search if they are locally setup using the docker inspect image command, if they do not exist locally the worker will read the corresponding image from the shared directory and load such image. When launching the application containers, to avoid them starting to run the application their image entry-points are switched to an empty shell. When the worker has finished launching all application containers the emulation step begins.

The emulation step of the setup phase involves launching a emulation volume where the emulations rules are located and an extra container attached to each application. This container uses an image, that has installed Linux traffic control and scripts to apply all network emulation rules, and will be attached to the application network namespace. When each emulation container as been launched, they will execute each specific network emulation rules, acknowledging the worker when they finish executing. After all emulation containers finish applying the emulation rules, the setup phase is concluded and the execution phase begins.

Execution phase

This phase of the worker component starts when the worker finishes applying the emulation rules and waits for the manager to send a message to start executing the applications. When the message to start the execution of applications arrives, the worker will start executing all their respective applications.

The execution of applications first involves checking if there are any start actions. If they exist, all application targets of those actions will start executing at the given target time. If those start actions do not exist for those applications on the worker, they will start with the start command at the target time of 0. The worker will signal the manager when the execution of applications is complete. This concludes the first step of the execution phase

The second step of the execution phase starts when the worker receives a start experiment timer message. This barrier was implemented to try to mitigate different execution times of applications and force all workers to start the experiment timer at the same time. When a worker starts its experiment timer, it will also start all action timers starting from the start actions list, ending at the end actions list.

The last step of the execution phase is when the experiment timer expires, signalling that the experiment has terminated. When the experiment timer expires, the worker will message its manager with an "experiment done"message, in order to signal that it has finished the experiment. When the connection to the manager drops, the worker will execute the cleanup routine. This cleanup routine involves deleting and removing all experiment containers and leaving the docker swarm.

3.3.3 Emulation module

In this section we discuss the implementation of the emulation module. The emulation module is responsible for creating and applying network emulation on each application, it does that by spawning an extra container attached to the application network namespace, where it can apply the network emulation.

To achieve the desired network emulation we use the Linux traffic control, previously presented in 2.2.3. We leverage Linux traffic control qdisc, more specific the htb to apply bandwidth restriction and netem to apply latency and other desired network properties.

3.3.3.1 Rule Generation

The emulation module obligations is to generate each application network emulation rules so workers can apply them. These rules are extracted from the network properties matrix, where each position represents the value of latency or bandwidth from application X to application Y. The rule generation is divided into two scenarios: peer-to-peer and server-client. In the peer-to-peer scenario, applications can communicate with all their peers, while in the server-client scenario, client applications will only communicate with server applications. The peer to peer scenario is handled by creating a thread to compute each application rules files, meaning that each thread is responsible to create the files containing the emulation rules for a single application.

The server-client scenario can be seen as an optimization of the peer to peer scenario, as we do not need to generate the emulation rules for client to client interactions. This assumption simplifies reading the input, if the following condition is true:

$$2 * c * s + s^2 < m^2 \tag{3.1}$$

In the equation above, the letters have the following meaning: c as the number of clients, s as the number of servers, and m is the number of servers plus the number of clients. This can be simplified into the following equation:

$$s^2 < c^2 + s^2 \tag{3.2}$$

This equation translates that reading twice the matrix of size c*s and reading the server matrix is cheaper than reading the matrix of size m. This means that if the number of clients is greater than zero, the server-client scenario results in reading fewer values than the peer to peer scenario.

3.3.3.2 Qdiscs

Symphony network properties are modelled as a matrix, either latency or bandwidth, where each position represents the value of latency or bandwidth from application X to application Y. To implement the bandwidth matrix we used the following hierarchy, seen in figure 3.5.

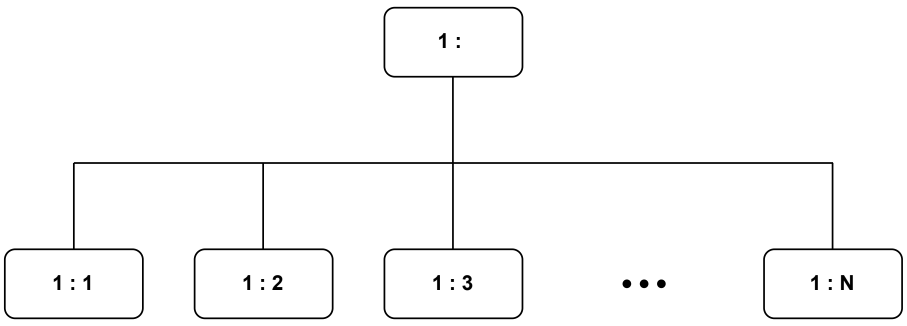


Figure 3.5: Htb classes hierarchy implemented.

This hierarchy applies the concepts of the htb qdisc, more precisely the top htb class applies the overall bandwidth restriction of the application, while the leaf classes apply the individual link to each application. For this hierarchy to work properly, the sum of the bandwidth rates of the leaf classes should not be greater than the ceiling of their parent's bandwidth rate. This is because the leaf classes only use their parent's bandwidth after overflowing their own bandwidth rate.

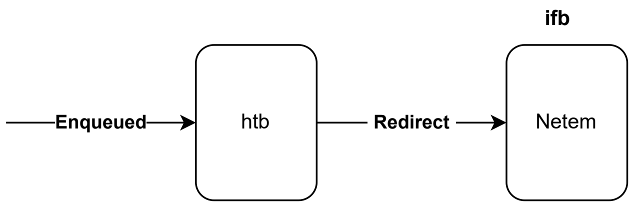


Figure 3.6: Flow of packets implemented.

The latency matrix is implemented, as a list of ifbs with each unique latency value, as we can see in figure 3.6. The network traffic after being dequeued from the htb class is then redirect into the correct ifb for its latency value to be applied.

3.3.3.3 Filtering

A vital part of the emulation module is how the filtering of each packed is executed, because we need to comply with scalability in order to emulate large distributed systems. To ensure that filtering each network packet has a constant cost despite the magnitude of the distributed system we utilized u32 [39] hash mechanism.

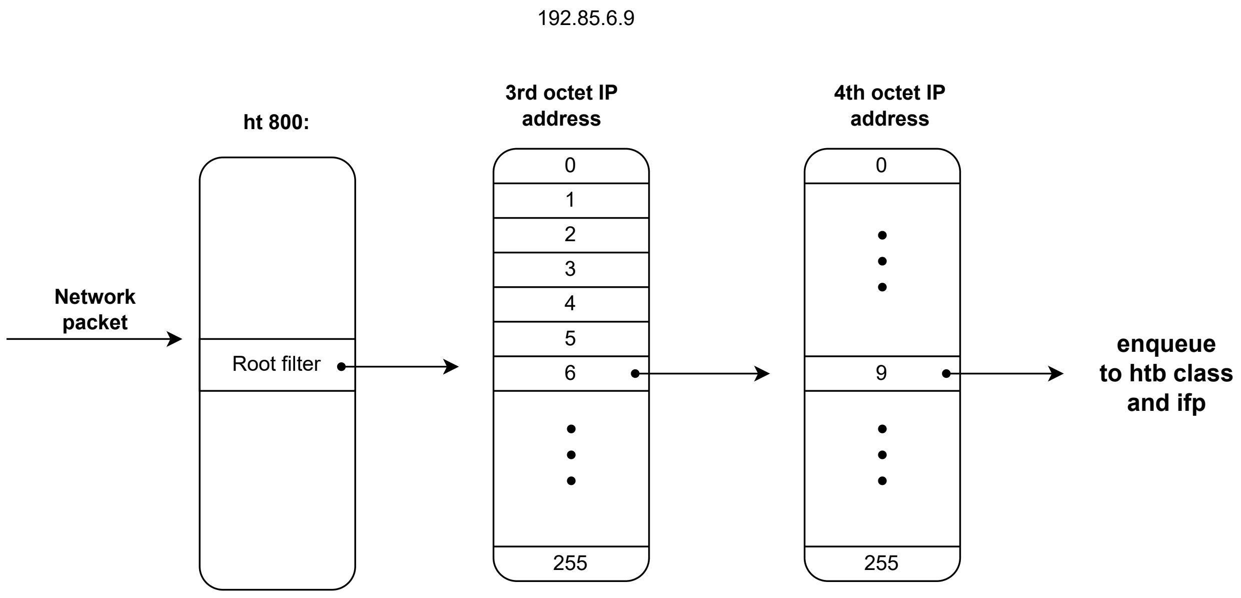


Figure 3.7: Flow of packet filtering, with the example of filtering the IP address 192.85.6.9

This hash mechanism creates a 256 entry table, so in order to achieve a constant cost, we map the third and forth octet of the IP address to a table entry and place a root filter to capture all traffic of the desired subnet, like in the example 3.7. In this example we have the IP address of 192.85.6.9, first it is capture by the root filter of subnet 192.85.0.0/16, then it hashes the third octet in order to find the exact table entry. The table entry of the third octect then redirects to the next table which hashes the fourth octet of the IP address, finding its exact entry and enqueuing in its corresponding htb class and redirecting to the correct ifb.

This method of filtering allows for a complexity of 4 steps to reach the correct destination, although it only supports an maximum of 65 thousand distinct IP address or a subnet of CIDR /16, we believe that adding an extra table of redirection would increase the number of tables exponentially and settled for a maximum of 65 thousand IP addresses.

3.3.4 Symphony-client

Symphony client is a component of the system architecture that serves two main purposes: to communicate with the Symphony server and to execute a high-level validation of the user input. The Symphony client is used to send commands to the Symphony server to start or poll experiments.

The Symphony client communicates with the Symphony server, but as the response from the server might occur multiple times, to not stop its execution ahead of time, the server will send a flag in its response dictating if it's the last one.

This component also executes a custom validator that checks for errors such as files not existing, docker images not being found locally, and throws an exception to notify the user of their incorrect input if an error is found, stopping the execution.

3.4 Service Architecture

A Service architecture can also be modelled, as seen in figure 3.8.

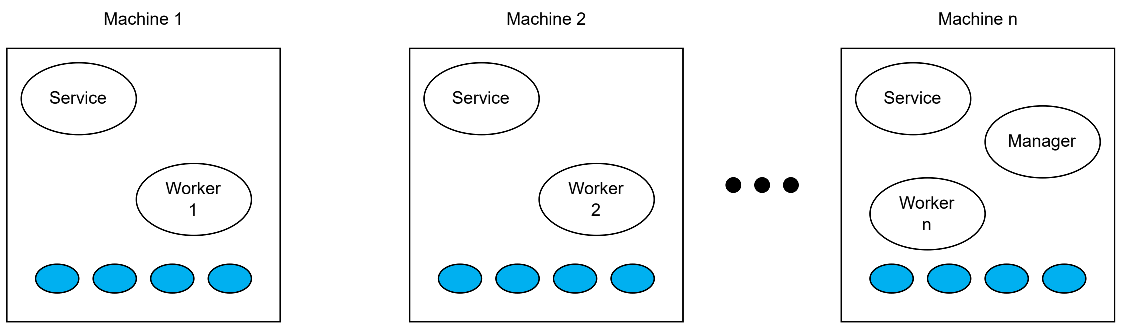


Figure 3.8: Symphony system architecture implemented as a service.

This architecture is a representation on how to model the system as a service on a cluster of multiple physical machines. To achieve this architecture each machine is running a server component, and each server has the full membership of the physical cluster. This architecture allows for experiments to be submitted to any server on the cluster, while workers can be specified to run on specific physical machines belonging to the cluster, allowing for flexibility on running experiments.

3.5 Summary

In this chapter we presented Symphony architecture, starting with its requirements. Then followed by presenting how the architecture is devised in multiple components and how each component interacts with each other.

We presented the implementation details of all the symphony components, their execution life cycle, and the experiment synchronization barriers. We provided an insight on each component contributed to the system, and how they interact with each other.

We latter presented an application of this system architecture to a real world service in a cluster of physical machines.

In the following chapter, we present a guideline on how to configure and use the system from an user perspective.

4 Configuring Symphony

In this chapter we present how to use and configure Symphony. More detailed, we start by looking at an example experiment while explaining each parameter and its possible values. We will dive into the reasoning behind each parameter and how it contributes to fulfilling an experiment.

Furthermore, we detail how to interact with the running service, how to execute an experiment and listing some best practices to follow.

Lastly, we provide a few example experiment configuration files and discuss them.

4.1 Experiment file

In Symphony experiments are described in a YAML configuration file. Since this files can have some similarities between experiments (such as the same physical environment), we implemented the possibility of adding a default and override YAML file. An experiment composed of both a default and a override file will first read all attributes from the default file and then the override file, replacing all attributes of the override file in the default file.

To better understand how to create these experiment files we first need to understand each of the configuration files parameters.For that, we will analyze an example experiment consisting of nine iperf3 clients and three iperf3 servers, this example configuration file can be seen in figure 4.1.

4.1.1 Parameters

As we can see from figure 4.1, an experiment configuration file has a total of fourteen main parameters: *dockerImgs*, *seed*, *machines*, *groups*, *runTime*, *network*, *latencyFile*, *bandwidthFile*, *rules*, *outputVolume*, *startCommand*, *dynamicEvents*, *onFinish*, *startCommand*. We will now further analyze each paramater and its possible configuration.

```
dockerImgs:
  -
    name: ipltc
    tag: latest
seed: 4542355562136458828
machines:
  - address: 172.16.66.2
    number: 0
groups:
  -
    name: client
    numberDocker: 9
    imgName: ipltc
    machines: [0]
    startAddress: 172.30.10.0
    totalBandwidth: 1000
  -
    name: server
    numberDocker: 3
    imgName: ipltc
    machines: [ 0 ]
    startAddress: 172.30.20.0
    totalBandwidth: 1000
runTime: 300
network:
  name: testnet
  subnet: 172.30.0.0/16
latencyFile: /home/aatalaia/expTese/expLinkSharing/latency.txt
bandwidthFile: /home/aatalaia/expTese/expLinkSharing/bandwidth.txt
rules:
  path: /home/aatalaia/expTese/expLinkSharing/rules/
  generated: false
  serverClient: true
  jitter: 0.1;distribution normal
  serverToServerBandwidth:
/home/aatalaia/expTese/expLinkSharing/serverToServerBandwidth.txt
  serverToServerLatency:
/home/aatalaia/expTese/expLinkSharing/serverToServerLatency.txt
outputVolume: home/aatalaia/expTese/expLinkSharing/logs
startCommand: startServer.sh
onStart:
  actions:
    - 1000;client; collectData.sh 172.30.20.0
    - 0;server;startServer.sh
dynamicEvents:
  actions:
    - 15000;client but {node-client-0,node-client-1};changeServer.sh
onFinish:
  actions:
    - 15000;client;writeLogs.sh
```

Figure 4.1: Experiment of nine iperf clients and nine iperf servers.

4.1.1.1 Docker Images

The parameter *dockerImgs* is a list of all docker images used in the experiment. As seen in figure 4.2, we can describe a docker image by providing its name and tag. All docker images added to the list must be present in the server machine where the experiment is submitted.

```
dockerImgs:
  -
    name: ipltc
    tag: latest
```

Figure 4.2: Docker images snippet of the example experiment.

4.1.1.2 Rules

```
rules:
  path: /home/aatalaia/expTese/expLinkSharing/rules/
  generated: false
  serverClient: true
  jitter: 0.1;distribution normal
  serverToServerBandwidth:
/home/aatalaia/expTese/expLinkSharing/serverToServerBandwidth.txt
  serverToServerLatency:
/home/aatalaia/expTese/expLinkSharing/serverToServerLatency.txt
```

Figure 4.3: Rules snippet of the example experiment.

As we can see from figure 4.3, the rules parameter is composed of five sub parameters: *path*, *generated*, *jitter*, and *serverClient*, *serverToServerBandwidth*, *serverToServerLatency*.

The *path* parameter is path string where the emulation rules are read and generated to. This path needs to be a shared directory among all workers and manager of the experiment, as they all need to access the emulation rules in order to enforce the correct network emulation.

The *generated* parameter is a boolean that indicates if the rules are already generated, so the generation step can be skipped.

The *jitter* parameter is a composed of two semicolons values, the percentage of jitter to be applied and the corresponding distribution. The percentage value is written in a range from 0 to 1, and works in the following manner: in a delay of 150ms with a jitter of 0.1, the actual delay is 135-165 ms, or 150ms +/- 15ms. The jitter distribution relates to what the probability of the picked value within its possible range. Possible jitter distribution are the same as netem offers in their documentation [25].

The *serverClient* is flag parameter that when set to true Symphony will search for two other parameters: the *serverToServerBandwidth* and *serverToServerLatency*. This parameters are path strings to the corresponding matrix configuration files, as we can see in the example 4.3. This will result in instead of using a peer-to-peer approach when generating the rules, it will use the server client simplification, meaning that it will emulate the all network links except the client to client links as they do not communicate with each other.

4.1.1.3 Seed

The *seed* parameter its to enable repetition of experiences, as all random values will be calculated using the seed value. If there is no seed value defined, symphony will generate one for the experiment.

4.1.1.4 Run Time

The *runTime* parameter determines for how long an experiment will execute after all applications finished their starting phase. This parameter is mandatory for the experiment to start.

4.1.1.5 Latency and Bandwidth Files

```
latencyFile: /home/aatalaia/expTese/expLinkSharing/latency.txt
bandwidthFile: /home/aatalaia/expTese/expLinkSharing/bandwidth.txt
```

Figure 4.4: Latency and Bandwidth files snippet of the example experiment.

As we can see from the figure above 4.4, the parameters for latency and bandwidth files are strings of the absolute path leading to those files. These parameters can be omitted if the rules are already generated (i.e. the parameter generator in rules has value true).

4.1.1.6 Network

```
network:
  name: testnet
  subnet: 172.30.0.0/16
```

Figure 4.5: Network snippet of the example experiment.

The parameter network has two sub parameters: *name* and *subnet*, as we can see from the figure above 4.5. This experiment parameter will use the docker interface to create an overlay network with the given name and its corresponding CIDR subnet value. This CIDR value needs to provide an amount of possible IP addresses for the containers within the experiment, and should not be greater than /16 network, meaning that Symphony is limited to sixty five thousand unique IP addresses (the maximum unique IP addresses of a /16 network).

4.1.1.7 Machines

```
machines:
  - address: 172.16.66.2
    number: 0
```

Figure 4.6: Machines snippet of the example experiment.

As we can see from the figure above 4.6, the parameter machines is a list of machines where each machine contains an IP address and a number to which they are referred inside the experiment. Each machine needs to have a service of symphony running and connected to where the experiment is submitted.

4.1.1.8 Groups

```
groups:
  -
    name: client
    numberDocker: 9
    imgName: ipltc
    machines: [0]
    startAddress: 172.30.10.0
    totalBandwidth: 1000
  -
    name: server
    numberDocker: 3
    imgName: ipltc
    machines: [ 0 ]
    startAddress: 172.30.20.0
    totalBandwidth: 1000
```

Figure 4.7: Groups snippet of the example experiment.

The parameters *groups* is a list of a container groups, as we can observe from the figure 4.7. A container group is defined with a *name*, *numberDocker*, *imgName*, *machines*, *startAddress*, *totalBandwidth*, and *customConf*.

Name: The group name identifies the group and must be unique among the list of groups in a experiment.

NumberDocker: The *numberDocker* defines how many application containers the group will execute, and the imgName tells which docker image to use.

ImgName: The docker image name that the group will use.

Machines: The *machines* parameter is a list of machine numbers from the machines parameter, that tells in which physical machines the group will be evenly distributed.

StartAddress: The *startAddress* is to tell the service where to start attributing the IP addresses of the container applications belonging to that group. This attribution occurs in a linear way, meaning that a group of 3 applications starting from the IP address of 172.30.10.254 will end at the ip address of 172.30.11.01 . If no start address is defined, symphony will start attributing addresses from the beginning of the subnet value. The user must have special attention when giving each group a starting address, as to not having group addresses overlap, leading to the experiment giving an error in the setup phase.

TotalBandwidth: The *totalBandwidth* is to set a total bandwidth that each container within that group can allocate as a maximum. This value should be an integer, and it is in the scale of Mbps.

CustomConf: the parameter *customConf* is a string of parameters that will be passed to the container when start executing. One example of the usage of this parameter is to pass environment variables to the entire group of containers or limit their cpu or memory usage through their corresponding docker parameters.

4.1.1.9 Output Volume

Experiment logs in Symphony are collected through the output volume parameter. This parameter is a string of the absolute path of for the output volume directory. When setting up an experiment, Symphony will mount an output volume in each application container and bind it with the given parameter directory. After an experiment is executed and cleanup the output volume will not be deleted so that the user can retrieve the experiments logs when he desires.

4.1.1.10 Start and Finish Commands

The parameters: *startCommand* and *finishCommand*, are mostly used to avoid repetition of start and finish actions, if they all execute the same command at the same time. These parameters are optional, meaning that if left blank the experiment will ignore them.

```
onStart:
  actions:
    - 1000;client; collectData.sh 172.30.20.0
    - 0;server;startServer.sh
dynamicEvents:
  actions:
    - 15000;client but {node-client-0,node-client-1};changeServer.sh
onFinish:
  actions:
    - 250000;client;writeLogs.sh
```

Figure 4.8: Events snippet of the example experiment.

4.1.1.11 Events

Symphony offers events as to empower the user to create actions that allow for more detailed and realistic experiments. As we can see from the figure above 4.8, events are devised into three categories: *onStart* events, *Dynamic* events, and *onFinish* events. We will further detail each category bellow, so lets focus on how to create an event action. An action is defined by three semi colon separated values:

At: This field determines where in time the action should start executing. This value is a positive integer of milliseconds that should be within 0 and the experiment run time.

Target: This field specifies each application where the action will execute. To facilitate and allow the user for more complex actions without having to name each container, a set of keywords was devised. This set of keywords are: *all*, *but*, N, $\{N\}$,and %.

The keyword *all* refers to all applications within the experiment, this keyword can be using with a combination of *but* to exclude some applications or with % to only use a percentage of all applications.

The keyword *but* is to except applications from being selected into the action target, it can be utilized in a combination of keyword N and $\{N\}$.An example taken from the dynamic events of figure above 4.8, is saying to execute the script *changeServer.sh* on all clients except client-0 and client-1.

The keyword N refers to an application or to an application group. Taken the above example, this keyword is referring to the application group, but it could also be a single application like client-node-01.

The keyword $\{N\}$ is to denominate a colon separated group of keywords N. This group of keywords N can contain single applications mixed with application groups.

The keyword % is to select a percentage of applications from all applications or a group of applications.

Command: This field defines which command to execute. This command can either be a custom command defined by the user to execute on the application or a default command. The given default commands are the following: *duplicate*, *corruption*, *isolation*, *linkflapping*, *netdown*, *reorder*, and *crash*. All commands except crash have as input parameter their time duration, in ms. These commands were implemented in the following manner:

The command *isolation* is implemented as a loss of 100% of all network packets to other applications. The command *linkflapping* is to apply spikes of isolation within its duration.

The command *netdown* blocks all incoming and receiving traffic from the experiment network, except for the docker swarm traffic on port 2344.

The command *crash* will send a *SIGKILL* signal for the application to stop executing.

The commands *duplicate*, *reorder*, *corruption* are mirrors from netem qdisc offers and they have the same input with the additional input of duration.

On Start

The *onStart* parameter dictates when the applications should start executing, if left blank all applications will start with the start command at the beginning of the experiment (i.e. at experiment time zero). This parameter should be used to start applications in different times, as seen from the example figure 4.8, it first starts the server group at time 0, then at time 1000 symphony starts the client group.

Dynamic

The *Dynamic* parameter is used to defined actions that occur during the experiment. This parameter should be used to contain all actions that are neither start or finish actions.

On Finish

The *onFinish* parameter dictates when the applications should start executing their finish action, if left blank and a finish command is defined, all applications will execute the finish command at the end of the experiment.

4.2 How to configure Symphony as a service

In this section we will learn each step to configure Symphony, and add it as a service on *systemctl* on your cluster of physical machines. First, we will look on each configuration parameter, there are seven parameters that can be configured.

interfaceName: This field is a name of a network interface to which the service will bind and use to send messages. There is no default value, so this field is required for the service to be correct.

port: The port number of the IP address given to the interface to which the servers will listen and send messages to. Default value is port 10000.

localPort: The port number of the localhost to which the servers will listen to client requests. Default value is port 10100.

localManagerPort: The port number of localhost to which the servers will listen to manager messages. Default values is port 10200.

masterPort: The port number of the IP address given to the interface to which the manager will listen and send messages to workers. Default values is port 10003.

servers: This field corresponds to the servers membership of the physical clusters, it should include the IP address of all the physical machines. It will try to connect to a symphony server on that IP and on port mentioned above. If given its own IP address the server will ignore it.

```
[Unit]
Description=Symphony Service
After=syslog.target network.target

[Service]
SuccessExitStatus=143

User= current_user


Type=fork

ExecStart=/PATH_TO/execStart.sh
ExecStop=/bin/kill -15 $MAINPID

[Install]
WantedBy=multi-user.target
```

Figure 4.9: A template service configuration file

membershipTimeout: This field is a timeout to which retry opening the connections on the server membership. Default value is 5000 ms.

After configuring the values mentioned above, the service can be started on a cluster of physical machines. For that we must create a service file to describe our service, an example can be seen on figure 4.9. In the figure there is just a need to replace the path to where symphony is located and the user, or group in which the service will operate.

Now we must add the service file to each machine and reload the *systemctl* daemon. The steps required for that are:

1. Add the symphony.service file to etc/systemd/system and reload, with the command *systemctl reload*.

2. Use the *systemctl* commands: *start*, *stop*, and *reload* to start or stop the Symphony service.

After the steps mentioned above, you can check if the service is running by running the command *systemctl status*. The results should match figure 4.10, if the service is all steps steps were done successfully. Now we are ready to start writing our experiment files.

4.3 User Interaction

The user interaction with the symphony service is through the experiment file and the symphony client, either the java class or through the client command. Both commands

```
● symphony.service - Symphony Server Service
     Loaded: loaded (/etc/systemd/system/symphony.service; disabled; vendor preset:
enabled)
     Active: active (running) since Wed 2022-11-23 20:04:50 CET; 30s ago
   Main PID: 20544 (execStart.sh)
      Tasks: 77 (limit: 113989)
     Memory: 595.1M
     CGroup: /system.slice/symphony.service
             ├─20544 /bin/bash /home/aatalaia/symphony/execStart.sh
             └─20548 /usr/lib/jvm/java-1.11.0-openjdk-amd64//bin/java -DlogFilename=
-cp Symphony-server/package/symphony-server.jar Main
```

Figure 4.10: Output of a running Symphony service

```
java -cp Symphony-client/package/symphony-client.jar Main submit
-file Symphony-server/package/ExpCassandra.yaml
```

Figure 4.11: An example of the java command to start an Symphony client

receive two parameters, the $-file$ for the configuration file and - $-default$ for a default configuration file. An example of the java command can be seen in figure 4.11.

The client command is an extension of the client java class, both of them provide the same functionality. This functionality is to query the symphony server if a experiment is running and to start or stop a running experiment. The submit command requires you to provide a path to the experiment file. More information on the symphony client can be found in 3.3.4.

The symphony service, when already configured and running, is used in the following steps:

1. Create the YAML experiment file, defining each parameter in order for the experiment to have your expected behaviour.

2. Launch a symphony client, either with the $launchClient$ command or be executing the client java, and passing it the submit parameter with the path of the YAML experiment file.

3. That is it! Now the experiment will start executing after completing its setup phase.

4.3.1 Best Practices

In this section we will advise on some of the best practices when using the symphony service, them being:

- Name each application group uniquely, as to not refer to both groups at once when defining the experiment actions.

- Check each group starting address, as to not have containers overlap addresses and stopping the setup phase.

- When defining a path of your local directory always use the full path as to avoid aborting the setup phase because of a missing parameter.
- After executing an experiment, if you want to execute it again you can check the generated boolean to true, as a way to reduce the setup time.

4.4 Examples

In this section we will analyze examples of configuration files that can be found in annex I.3 and I.3. This file starts by describing the docker images that will take part in the experiment, in this case is the *ipltc* with tag latest. Followed by enumerating all the physical machines where the experiment will be executed and attributing them a number. This number will serve as an identifier to refer to this machine later in the experiment configuration file. Then we specify our groups, filling each parameter with the desired value. Then we specify the experiment $runTime$ parameter to be thirty five seconds, the network to be named testnet and have a CIDR of 172.30.0.0/16, and the corresponding paths to the latency and bandwidth file. We the, have to give a path for our rules files to be generated to, and since we have set to true the flag of server client, we will need to specify where are the server to server latency and bandwidth files. Followed by specifying where the output volume should bind to in the local file system, and say a default start command to be used in case a start action is not specified. Lastly, we specify the start actions for both the experiment groups, where the server group will start an iperf3 server with the corresponding start command starting at time zero, and the client group will execute a script to extract data for thirty seconds at one thousand milliseconds into the experiment or 1 second into the experiment.

4.5 Summary

In this chapter we started by showing an example configuration file, explaining how each parameter contributes to the experiment and its possible values. Then we provided instructions on how to configure the symphony service, followed by how to use the service, and some best practices that should be followed. After reading this chapter an user should be able to configure and utilize the service to run experiments, and to have knowledge to build the experiment configuration file, knowing how each experiment parameter is defined and their possible values. In the next chapter we will look into how Symphony compares against a state of the art emulator and if it can replicate real world experiments.

5 Evaluation

In this chapter we present the experimental evaluation of our work, as to validate that our work is able to operate experiments of large distributed systems. To further test our work, we will compare it against a current state of the art network emulator. This chapter is organized as follows:

In section 5.1, we will present the experiment goals to which we evaluated the experimental results.

In section 5.2, we will present the different experiment use cases, providing a brief explanation of the chosen application and how we will used to extract key metrics that provide insight on the performance of the proposed work.

In section 5.3, we will state the environment where the experiment were executed.

Lastly, in section 5.4, we will present the experimental results of the use cases previously defined. We will use this experiment results to compare Symphony against Kollaps, a state of the art network emulator.

5.1 Evaluation Goals

In this section we will revise the goals proposed for this work and how we will evaluate each network emulator with those requirement in mind. These goals are:

Emulation Scalability: as to measure how each network emulator scales with different sizes of distributed systems.

Ease of use: to measure the complexity of each experiment configuration file, and how much boilerplate lines are needed to represent different experiments.

Produce reliable results: to assure that all network emulators produce reliable results, and that users can trust such results.

Generalist: to ensure that the network emulator is agnostic of application. This means that the network emulator treats the application as a black box.

Realism: to measure the grade of realism of the applied network emulation by the network emulators.

5.2 Experiment Use Cases

In order to evaluate all of the criteria mentioned above 5.1, we propose the following experiment use cases:

5.2.1 Iperf/Ping

In this use case we use the tools iperf3 and ping, in order to make a decision on which emulation architecture would be implemented and to evaluate the criteria of which the proposed work produces reliable results.

The first evaluation will be to provide an insight on the performance of each emulation architecture. This experiment was done with multiple pings while retrieving key performance metrics, such as CPU usage, RAM allocation, and total CPU context switches. After collecting those key performance metrics, a decision about which emulation architecture to choose could be made.

Lastly, the other evaluation will be to measure the average error of bandwidth and latency for each sent packet and compare them with the state of the art network emulator: Kollaps 2.5.1. The average error of bandwidth is measured by the iperf3 tool, more precisely we will connect an iperf3 client to an iperf3 server through an emulated network link with the specified maximum bandwidth. The average error of latency is measured by the ping command, more precisely we will have two applications connected through an emulated network link with the specified latency, send a thousand ping requests and calculate the average error of those ping requests.

5.2.2 Cassandra

In this use case we will run an experiment with the Cassandra database. The Cassandra database is a distributed database that is used by thousands of companies. Therefore Cassandra is a suitable database to test the realism and repeatable criteria of the work done in this dissertation. To evaluate those criteria we propose to replicate an experiment done in "Practical and fast causal consistent partial geo-replication"[15], and see if we can achieve similar results when using the framework developed in this thesis. If we achieve similar conclusions when looking at the emulated data charts, we can successfully say that the work done in this dissertation is able to replicate experiments which were done in a real world scenario.

5.2.3 IPFS

In this use case we run multiple experiments consisted of IPFS applications. Since IPFS is a decentralized system that operates in a peer-to-peer manner, it is capable of storing and retrieving data without relying on a central server. Therefore, IPFS is well-suited for testing distributed systems that require the deployment and operation of a large number of applications.

These IPFS experiments are to assess key properties of the tools presented. More precisely, we compare both emulators scalability, how they behave under different application workloads, and how they perform over long experiment times.

We used the distributed property of IPFS to test the emulation scalability of both emulators. More precisely, we run IPFS applications and execute a constant payload, while increasing the number of IPFS applications, to a thousand IPFS applications. To evaluate the scalability, we measure each physical machine resources and how they increase with the number of application containers.

For the different application workloads, we ran experiment with a constant number of machines and IPFS applications, only changing each IPFS application workload between experiment. We measured the machine resources, and the average latency of each get command.

To evaluate the performance of each emulator in handling long-running experiments, we conducted a two-hour experiment with a fixed number of applications and physical machines, each with a consistent workload. During the experiment, we monitored the resource usage of each machine every minute.

5.3 Environment

The environment where the most of the experiments were executed was on **GRID Cluster** testbed [5]. They were executed on the *gros* cluster, where each physical machine has the following hardware specifications: Intel Xeon Gold 5220 (Cascade Lake-SP, 2.20GHz, 1 CPU/node, 18 cores/CPU) cpu, 96 GiB of memory, 25 Gbps network and a 480 GB SSD SATA Micron MTFDDAK480TDN.

The other environment where experiments were executed was on DI Cluster. The experiment was executed on two physical machines each with 128Gib of RAM, AMD EPYC 7281 CPU, and a 2x10 Gbps network.

5.4 Experimental Results

In this section we will present the experimental results of each use case and discuss the results obtained. More precisely, we will analyse each specific use case and discuss each configuration file.

5.4.1 Emulation Architecture

This subsection refers to the first experiment in the use case of ping/iperf 5.2.1, and its purpose is to make a choice between different emulation architectures. This experiment was executed on DI Cluster with the specifications being defined in 5.3.

To achieve a conclusion, we made an experiment with 20,100,200 container applications varying the emulation methods: no emulation, emulation using an outer container, and emulation through a master process. We measured key performance metrics in each experiment using the snmp protocol, those values were RAM usage, CPU values, and context switches. Each experiment was run 3 times and the results from the 200 container experiment can be found in figures 5.1, and 5.2. It is due to note that *Char*1 and *Char*2 represent the two different physical machines were the experiment was executing.

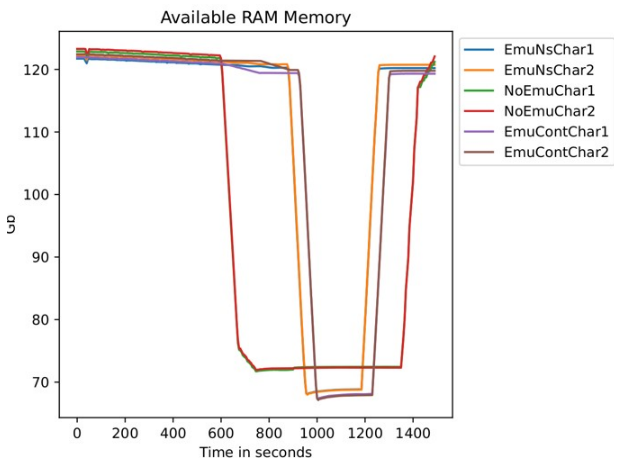


Figure 5.1: Available Ram throughout the experiments.

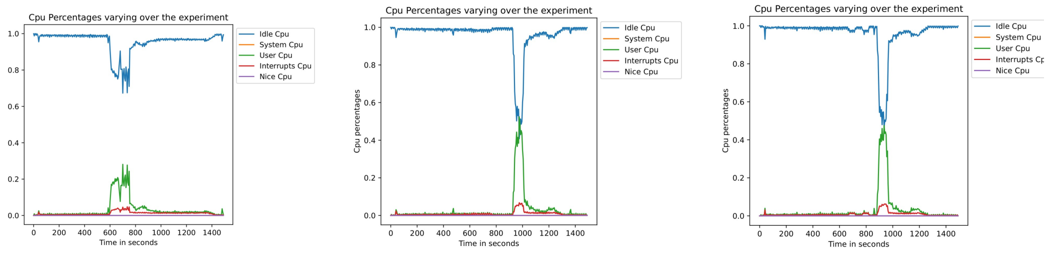


Figure 5.2: CPU values throughout the experiments.

Observing the results, we can conclude that there is no major difference between emulating using an external container or emulating through a master process. For that reason we decided to emulate using an external container attached to the each application network namespace, as later these containers can be used to collect metadata and because of its ease of implementation.

5.4.2 Link Results

This subsection refers to the last experiment in the use case of ping/iperf 5.2.1, and its purpose is to test the reliability of the results provided by this work. As mentioned in the use case explanation, we setup two application containers through an emulated link and measured either latency or bandwidth with each respective measuring tool.

Configuration

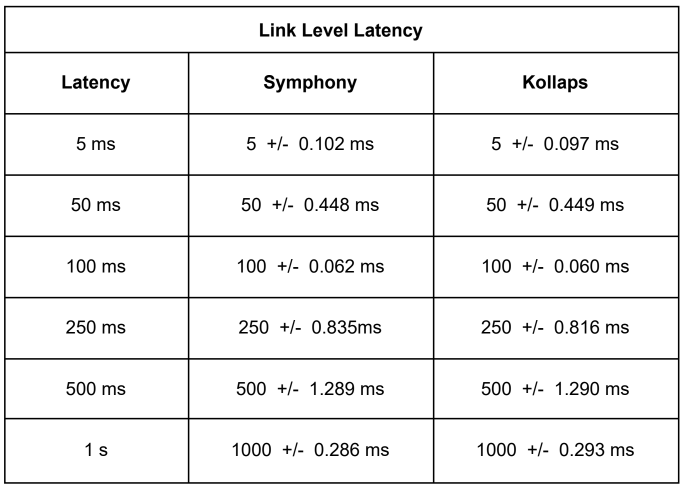
Now we will take a look at both emulators configuration, more precisely what was needed to run the experiments and their respective configuration files.

The Kollaps configuration for running both emulation experiments were simple and their configurations files were small and easy editable between experiments, being respectively: the latency configuration file I.1 and the bandwidth configuration file I.2. One minor setback was that we could not find a way to automatically bind an output volume with the container running the data gathering command. This had to be circumvented by entering inside the container and fetching the output file.

The Symphony configuration for both experiments used the same configuration file with changes on the bandwidth and latency files. The configuration file can be found in annex I.3, we can see that this file contains more configuration parameters than the previous ones, resulting in a bigger file.

Latency

The results for the average link level latency were obtained by running the Linux *ping* command, with a count of 500 round trips, then we present the rounded value with its respective deviation interval. This results can be seen in table 5.1



Link Level Latency		
Latency	**Symphony**	**Kollaps**
5 ms	5 +/- 0.102 ms	5 +/- 0.097 ms
50 ms	50 +/- 0.448 ms	50 +/- 0.449 ms
100 ms	100 +/- 0.062 ms	100 +/- 0.060 ms
250 ms	250 +/- 0.835ms	250 +/- 0.816 ms
500 ms	500 +/- 1.289 ms	500 +/- 1.290 ms
1 s	1000 +/- 0.286 ms	1000 +/- 0.293 ms

Table 5.1: Link Level Latency emulation comparison.

Analysing the values from table 5.1 we can see that they do not have a significant difference between emulators. This can be explained as both emulators use the netem qdisc to enforce their respective latency.

Bandwidth

The results for the average link bandwidth error uses *iperf*3 tool to measure bandwidth. It is worth noting that for bandwidth values of 1 Mbps and below, we had to adjust the TCP window size to ensure accurate readings. Specifically, we used window sizes of 4 Kbps, 8 Kbps, 12 Kbps, and 24 Kbps for bandwidth readings of 50 Kbps, 128 Kbps, 256 Kbps, and 1 Mbps, respectively.

Link Level Bandwidth			
labels	**Bandwidth**	**Symphony**	**Kollaps**
Low	50 Kbps	47 Kbps	48 Kbps
	128 Kbps	123 Kbps	124 Kbps
	256 Kbps	246 Kbps	248 Kbps
Medium	1 Mbps	963 Kbps	962 Kbps
	256 Mbps	244 Mbps	243 Mbps
	512 Mbps	488 Mbps	489 Mbps
High	1 Gbps	963 Mbps	962 Mbps
	5 Gbps	4,76 Gbps	4,75 Gbps
	10 Gbps	9,54 Gbps	9,54 Gbps

Table 5.2: Link Level Bandwidth emulation comparison.

Taking a further look into the table above 5.2, we can see that all results have an error bellow or equal to 5 % and have negligent changes when changing emulator. This can be explained, as all the network emulators used in this experiment have the same underlying layer, the Linux htb qdisc.

Summary

In this subsection we took a look into the results of the first use case presented 5.2.1, the link level emulation. This use case was analyzed to see if the work proposed in this thesis can successfully emulate the desired network properties and does not increase the error of a single network link emulation properties, more precisely the properties of bandwidth and latency.

After collecting and analyzing the results for each network emulator, we can clearly see that the work proposed in this thesis does not introduce any errors when emulating a single network link. This outcome is to be expected as both emulators use the same underlying layer, the netem qdisc and htb qdisc.

Looking back at the evaluation goals we can say that both emulators provide realism and produce reliable results as they both have a small error margins when emulating key network properties. Although Kollaps has the advantage on its ease of use through keeping their configuration files smaller and with less boilerplate configurations than Symphony on small experiments.

5.4.3 Cassandra

Configuration

The configuration for this experiment was extracted from the paper mentioned on the use case explanation section 5.2. This experiment configuration was translated to the Symphony configuration file, which can be found in annex I.6.

The configuration uses 12 physical machines from the environment **GRID Cluster** where 9 machines will be executing four Cassandra server instances and 3 machines will execute 4 client instances. The Cassandra server instances were split into 9 groups, in order to provide each group a custom configuration related to their respective data center. The clients remained as a single group as they all have the same custom configurations. The remaining part of the configuration file is composed of the configuration of emulation files and rules, which is followed by the start actions of the experiment.

We can observe there are a lot of similar actions but with different times and recipients. This was done to avoid collisions between Cassandra servers joining the clusters while another server was still joining, which can only happen if the server in question is a seed Cassandra server. The solution resulted in launching each Cassandra server on a single action, generating 36 lines of actions just to start each server instance.

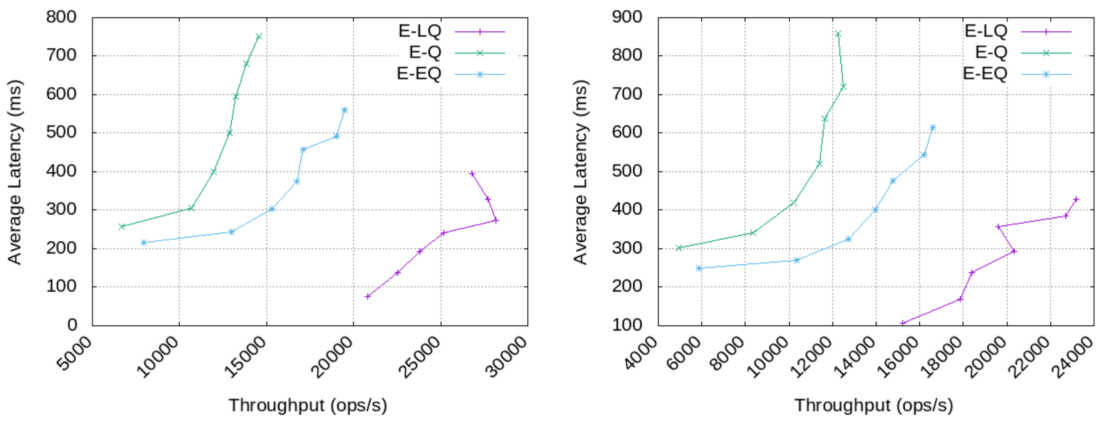
This configuration file has a total of 192 lines, which 121 are application related lines, 20 are emulation related lines, 40 boilerplate lines and the others are blank lines for better readability. We notice that the size of this experiment configuration file mostly comes from the groups parameter and the start actions parameter. The group parameter could not be reduced as each group requires different configuration, but the start actions parameters could be substantially reduced if do not require to start every server instance individually. This matter of delaying the time where each application within a group starts is a feature that will be implemented in future work 6.

Experimental Results

In this subsection we present the experimental results of the Cassandra experiment use case. As mentioned in the Cassandra use case, we will replicate the results of the work

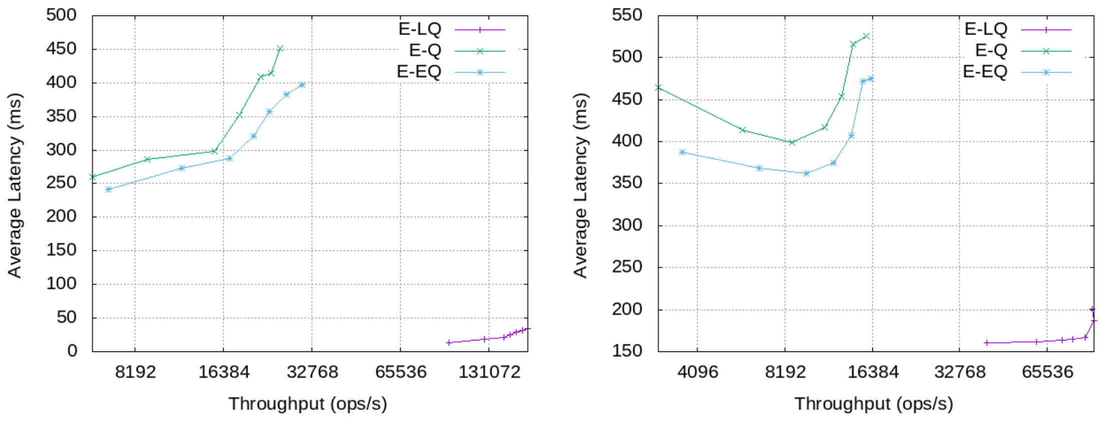
"Practical and fast causal consistent partial geo-replication"[15] while using the proposed work in this dissertation. More precisely, we will compare the charts generated from our replicated version against the ones with the published data and see if the same conclusions can be reached.

We will start by comparing the throughput charts bellow, being the original data figures 5.3 and the emulated data figure 5.4.



(a) Only local operations. (b) Local and remote operations.

Figure 5.3: Throughput and latency figures from the Cassandra experiment paper [15].

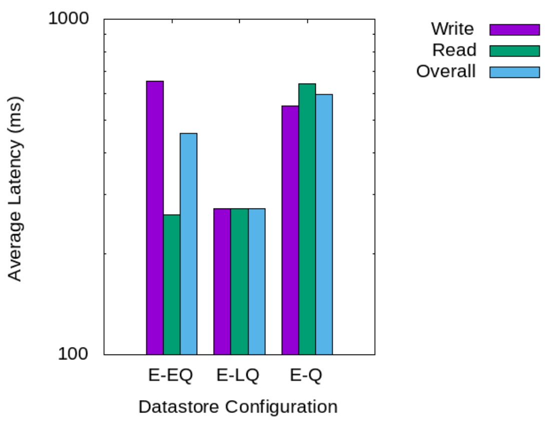


(a) Only local operations. (b) Local and remote operations.

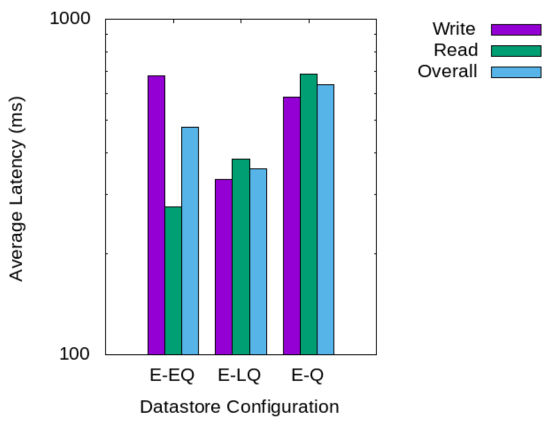
Figure 5.4: Throughput and latency figures from the emulated Cassandra experiment.

Looking at both figures we can clearly see that they are not identical. This was expected because the experiments were executed with different physical machines with different speeds. This impacts mostly the local operations, as the devices speed have the most impact in average latency when comparing with the network latency. Although they are not equal, the same conclusions can be extracted from both figures. These conclusions being:

- the lowest latency is for local quorum configuration and the highest latency is for quorum configuration
- With the increasing number of client threads, the system will eventually saturate and start to loose throughput and increase latency. This can be observed as number of client threads increase, the average latency starts increasing when the systems appears to be saturated.

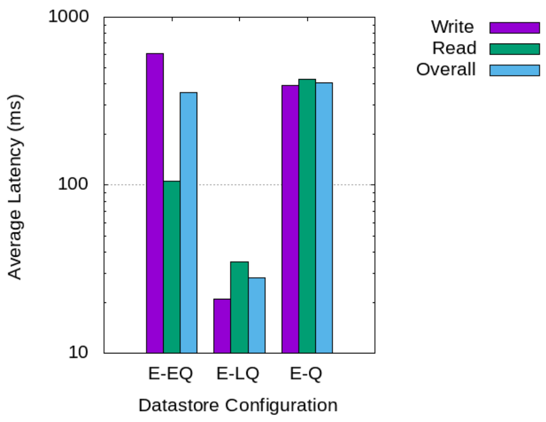


(a) Only local operations.

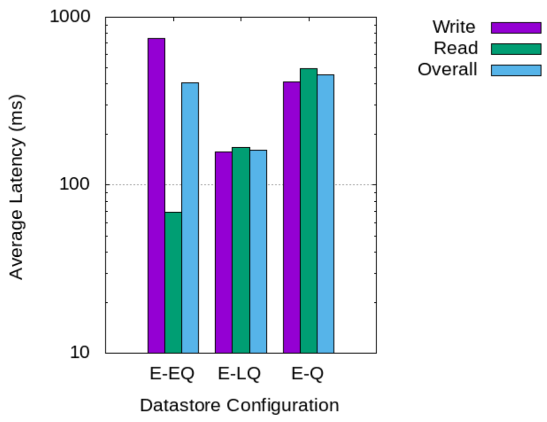


(b) Local and remote operations.

Figure 5.5: Average latency figures from the Cassandra experiment paper [15].



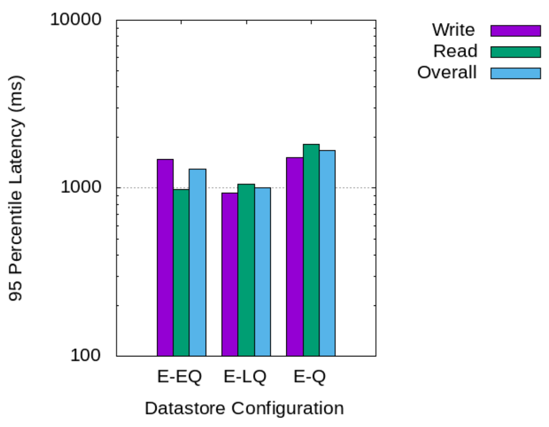
(a) Only local operations.



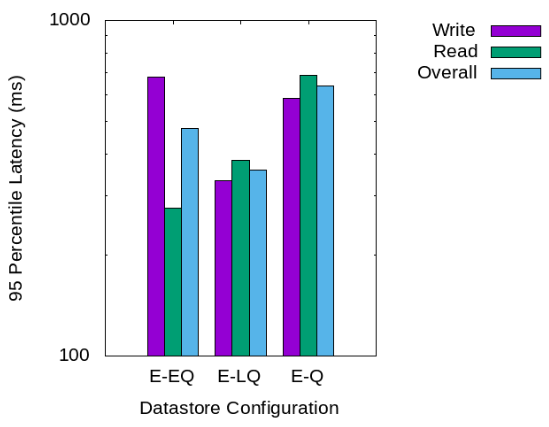
(b) Local and remote operations.

Figure 5.6: Average latency figures from the emulated Cassandra experiment.

When taking a look into the latency charts, figure 5.5 and figure 5.6, we can see again that we have a lower latency on local operations. As we mentioned above, this is due to faster hardware, as the disk speed is several times faster in the emulated experiment than the original experiment. The 95 percentile latency also follows the same pattern as the average latency chart and the figures 5.7 and 5.8 can seen bellow.

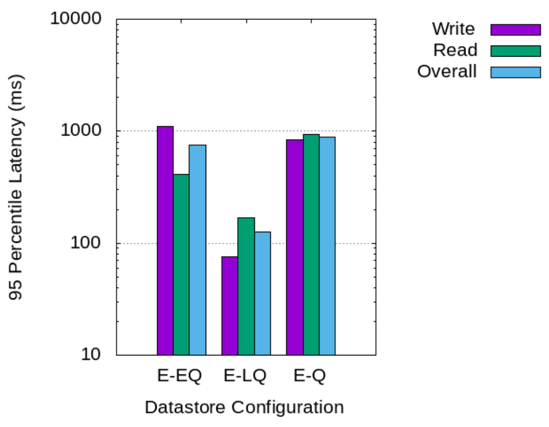


(a) Only local operations.

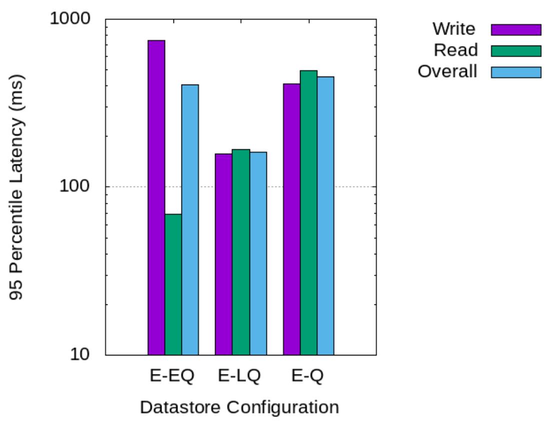


(b) Local and remote operations.

Figure 5.7: 95 percentile latency figures from the Cassandra experiment paper [15].



(a) Only local operations.



(b) Local and remote operations.

Figure 5.8: 95 percentile latency figures from the emulated Cassandra experiment.

Summary

In these experiment use case, we replicated the Cassandra experiment [15]. We first mapped the configuration mentioned into a Symphony configuration file, which although it did not present much of a challenge, it revealed an issue that the configuration does not allow for applications with a group or groups to be sequentially started with a delay between each other resulting in needing to specify each application individually.

When looking at the experiment results between the emulated experiment and the original experiment, we can see although they are not the same results, the same conclusions can be drawn from both experiments. Although the results were not equal, some amount of difference was to be expected as the underlying hardware was not the same in the emulated experiment and the original experiment. Since the same conclusions can be achieved in both experiments, we can say that Symphony is able to provide realism while producing reliable results.

When creating the configuration file required to execute this experiment, we noticed a few improvements on how we describe a group start actions separated by a constant time. Taking into account the experiment results and the difficulties of creating the experiment configuration file, we can still successfully say that Symphony is able to replicate real world scenario experiments using its own emulated environment.

5.4.4 IPFS

Configuration

This use case experiments required configuring both emulators to launch multiple IPFS peers and emulate the desired network properties between them.

Symphony configuration consisted of writing its configuration file I.4. This configuration file is composed of the physical machines taking part in the experiment, a group of docker containers running the IPFS image and paths to the latency and bandwidth files. Then adding an action for all peers to start executing at the given time. This configuration can be easily modified to accommodate different number of peers by changing the number of dockers in the desired group and provide a different latency and bandwidth file. As for the configuration metrics, this configuration file has a total of 40 lines where 23 are application focused lines, 7 are emulation focused lines and 10 are boilerplate lines.

The configuration process for Kollaps has the following steps: preparing the Docker swarm and overlay network, and writing the configuration file. The configuration file, which can be seen in I.5, had a few problems that needed to be addressed, being that a link's origin and dest attributes can only be given valid names of declared services or bridges. This lead to having each peer have their own service, link and schedule, meaning that for higher number of peers this file became to big to be manually written. To solve this problem, we created a script to generate a topology file that would receive the number of peers, their latency and bandwidth file and output the corresponding topology file. As for the configuration metrics, the configuration file has a total of 1385 lines where 150 lines are application focused lines, 1224 lines are emulation focused lines and 11 are boilerplate lines.

We analyzed both configuration files for the 50 peers experiment, coming to the conclusion that Symphony offers a more compact configuration file, being 34 times smaller than Kollaps configuration file. Another factor that we can take into account is how much bigger the configuration file is when compared to a system with a higher number of peers. Symphony requires the user to just add the extra physical machines while respectively add them in their desired groups and increase the number of applications, while Kollaps configuration file grows exponentially with the increase of the number of peers. Resulting in Symphony configuration file more robust to the increase of the number of peers than Kollaps.

Experimental Results

In this subsection we present the experimental results of the IPFS use case. We will start with the emulation scalability experiment. This experiments focus on measuring each physical machine CPU and ram, while also measuring the amount of setup time required for each number of applications. Before we advance into analysing the experimental results, we should detail that the lack of value in the bar chart represents that the system was unable to complete an experiment with those parameters, either by nonsupporting such a high number of applications or being unable to advance from the experiment setup.

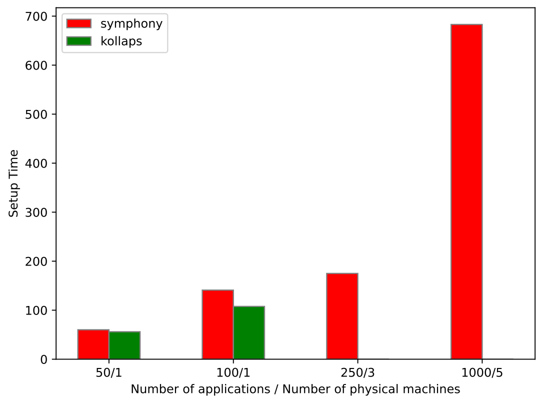


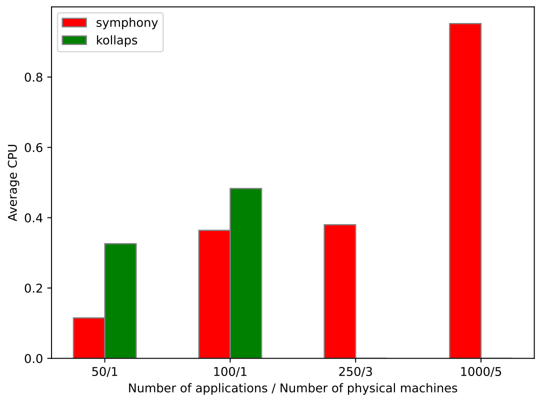
Figure 5.9: Setup Time throughout the experiments.

We can see the setup time in the following figure 5.9. As we can see, symphony presents an overall higher setup time than Kollaps, where Kollaps successfully completes the emulation of an experience.

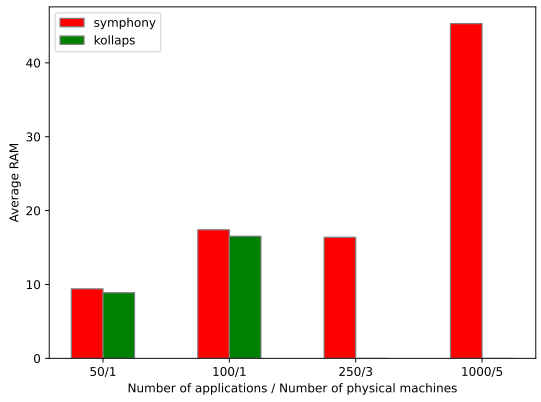
This can be explained as Kollaps transforms its experiment setup file into a docker compose deployment file, while requiring an existent docker swarm setup and overlay network. This allows for faster setup times because the emulator does not have to execute each docker run command, while also not having to execute the synchronization barriers of creating a docker swarm and its respective overlay network.

Next we take a look into the average CPU usage of the scalability experiments 5.10. We can observe that Kollaps presents a increase of the CPU usage average. We believe that this extra consumption of CPU resources is due to Kollaps approach to compute a fair share of bandwidth per link. This is due to Kollaps observing each link bandwidth allocation, meaning that all emulation cores it will have to constantly monitor every link in the experiment.

Now we look at the average RAM usage of the scalability experiments 5.10. As expected, both emulators require an identical amount of RAM per experiment, although Symphony presents a slight increase in RAM usage. This slight increase of RAM usage when comparing to Kollaps can be explained as Symphony is running as a service, meaning that it has a server process running.



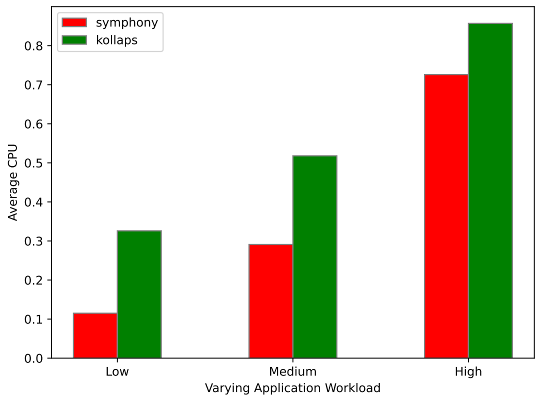
(a) CPU usage.



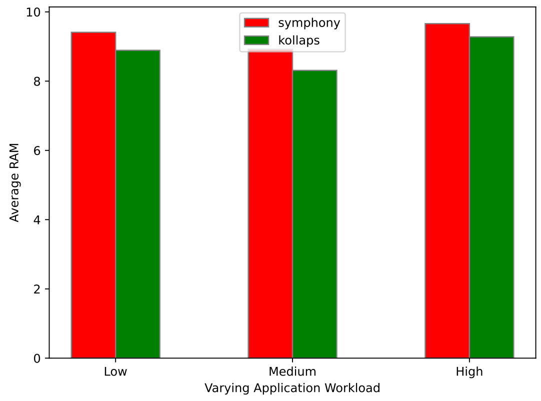
(b) RAM usage.

Figure 5.10: Resource usage throughout the scaling experiments.

On the workload experiments we analyze how the systems behaves under the conditions described in 5.2.3, meaning that we will keep the same number of physical machines and applications while changing the workload of each application. We will being by looking at the CPU usage figure 5.11.



(a) CPU usage.



(b) RAM usage.

Figure 5.11: Resource usage throughout the workload experiments.

As we can see from figure 5.11, the CPU average gap between Symphony and Kollaps appears to become shorter as the workload increases. Although this gap becomes shorter, Kollaps will be first reach a threshold where CPU averages become almost 100% and starts back pressuring the application. From the used RAM point of view, it seems that increasing the amount of workload has minimal impact on the RAM usage, while also having no impact on the RAM usage gap between both emulators.

We also measured the average amount of latency for each *get* command, as can be seen in figure 5.12. Both emulators present a similar amount of perceived latency, and as expected, this average latency increases as the workload of the system increases.

On the long running experiments we wanted to observe if the time of the experience

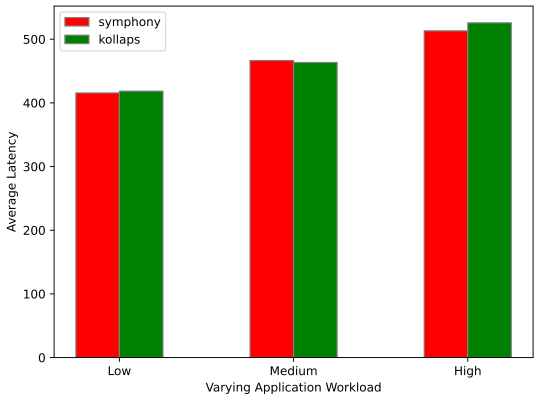
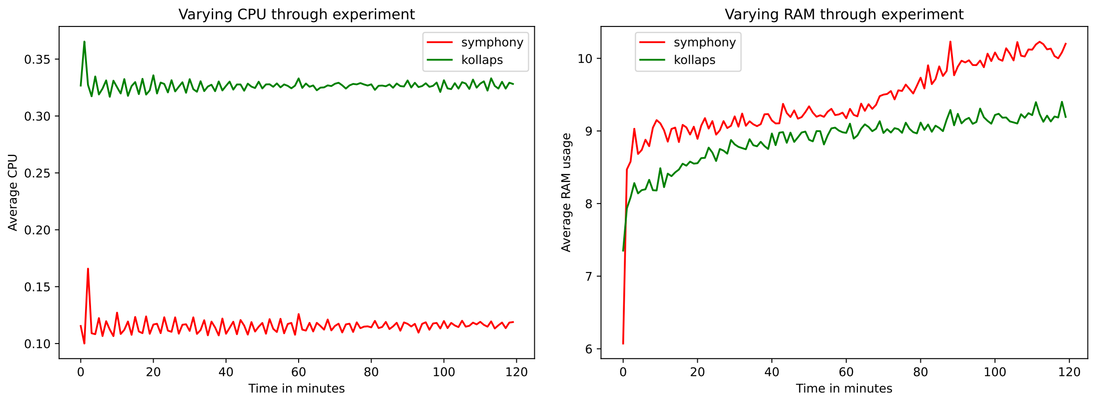


Figure 5.12: Average *get* latency throughout the experiments.

had an impact on the physical machine resources. So we run experiments for two hours and collected the CPU and RAM usage every one minute, as we can see in figure 5.13.



(a) CPU usage.

(b) RAM usage.

Figure 5.13: Resource usage throughout the long workload experiments.

As we can conclude from the figure 5.13 a), there is no increase in the average CPU usage over time. This means that it stays constant within the experiment, without taking into account the initial spike when all the applications are starting and trying to connect to the bootstrap node. Although the RAM usage presents a slight increase throughout the experiment, since this increase can be seen in both emulators it is due to the application RAM

The same cannot be said for the RAM usage, as we can see in figure 5.13 b), that both emulators present a slightly increase of RAM usage throughout the experiment.

Summary

In this experiment use case we did three types of experiments in order to measure and compare different comparisons of the emulators. First, we discussed both emulators configuration file and concluded that Symphony holds an edge against Kollaps because its configuration file is more compact and adapts better to the changes in the number of peers.

Then, we started by running a scalability experiment where we increased the number of peers to see which emulator provides a better scalability, one of the stated evaluation goals. In this experiment, Symphony becomes a better choice of emulator because of the inability of Kollaps to emulate systems with an higher number of links then 65535.

We followed by executing the experiment of changing the workload of each application and fixing the system size. On this experiment we cannot conclude that one emulator has an edge on the other, as both emulators provided similar results on the average latency of *get* commands and with opposing results in the CPU average and RAM usage average metrics.

Lastly, we executed the experiment of long experiment run time. In this experiment both emulators provided similar results, so we concluded that there is no difference in the experiment run time when choosing an emulator.

After analyzing these experiment results and combining them, we can come to the conclusion that Symphony has a clear advantage when emulating large distributed systems. This advantage is due to Kollaps eventually becoming incapable of executing with such a large system size, and for its exponentially increase of configuration file size with the increase of system size.

5.5 Summary

In this chapter we presented the criteria behind our evaluation and its results. We first detailed all the evaluation goals and how we plan to evaluate them. We then detailed our use case experiments, and how each configuration file complies with the evaluation criteria. Followed by the experiment methodology, where we detail the environment where we ran the experiments, discussed the evaluation results and how they relate to the evaluation goals.

In the next chapter we conclude this thesis, and present some improvements and possible features that can be implemented as future work.

6 Conclusion

Conclusion

As current distributed systems become larger and more complex, pressure increases for conducting systematic and detailed performance assessments. Conducting experiments in distributed systems is a challenging task because you need to retain control over multiple processes while providing an emulated environment.

With those challenges in mind, in this thesis we studied how to evaluate distributed systems, how each assumptions are made and to measure key system performances there is a need to control the underlying network layer. We dived into how we could control this underlying network layer using Linux traffic control to filter network packets to their correct qdiscs, applying the correct network emulation. We studied the current state of the art network emulators on how they deliver network emulation and control experiments. As a result of this analysis, we presented Symphony a tool capable of executing experiments and offering an emulated network environment while keeping the experiment configuration to a minimum.

The work done in this dissertation leverages Linux traffic control and docker containers to allow users to create experiments in realistic emulated environment. This realistic emulated environment is solidified with the implementation of actions, such that it allows for users to add a degree of realism to their experiments. We implemented the configuration file to be as natural as possible, containing few lines of configuration while empowering the users to further customize their experiments.

The experimental evaluation conducted in this thesis allowed us to conclude that Symphony is capable of replicating existing experiments that were executed in a real world scenario using an emulated environment. We can also say that Symphony has an edge when emulating large distributed system over Kollaps, a state of the art emulator. This advantage is due to Symphony providing a smaller and more concise experiment configuration file, and the ability to emulated experiments until 65353 applications.

Future Work

In this final section we detail possible future work directions that improve our current solution. Those future work directions are:

Shared volume directory: Adding a a shared volume directory, so experiments do not rely on having a previously shared directory for the emulation and logs files. An example of a tool capable of implementing a shared volume directory would be GlusterFS [17], as we require a scalable network file system capable of handling the emulation rules and logs files.

Configuration helper: For now the server configuration has to be done manually. To better improve the configuration process, a configuration script needs to be implemented that allows the user to input the desired configuration and the configuration helper would install that configuration on a cluster of physical machines.

Create an alternate ending to experiments: Implementing an alternate ending to experiments, meaning that after an experiment is complete the application would stop and await for starting executing again. This would allow for executing the same consecutive experiment while skipping the setup phase.

Unit management: Implement a unit management feature so parameters like actions could receive their respective unit instead of being forced to all follow the same unit.

Delay between containers launching: Implement a new action group feature that would allow users to specify a constant time that would be required to wait between the launch of applications in the same group.

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 (2011-08), pp. 242–253. ISSN: 0146-4833. DOI: 10.1145/2043164.2018465. URL: https://doi.org/10.1145/2043164.2018465 (cit. on p. 8).

[5] D. Balouek et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Ed. by I. I. Ivanov et al. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20. ISBN: 978-3-319-04518-4. DOI: 10.1007/978-3-319-04519-1_1 (cit. on p. 56).

[6] 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).

[7] 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. 7).

[8] 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).

[9] 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).

[10] 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).

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

[12] G. DeCandia et al. "Dynamo: Amazon's Highly Available Key-Value Store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (2007-10), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: https://doi.org/10.1145/1323293.1294281 (cit. on p. 2).

[13] *Docker documentation: Docker Overview*. https://docs.docker.com/get-started/overview/, note=Accessed: June. 2021 (cit. on pp. 19, 21).

[14] *docker-java, a wrapper api for docker written in java*. https://github.com/docker-java/docker-java, note=Accessed: March. 2022 (cit. on p. 32).

[15] P. Fouto, J. Leitao, and N. Preguiça. "Practical and fast causal consistent partial geo-replication". In: *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE. 2018, pp. 1–10 (cit. on pp. 55, 61–63).

[16] P. Fouto et al. "Babel: A Framework for Developing Performant and Dependable Distributed Protocols". In: *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*. 2022, pp. 146–155. DOI: 10.1109/SRDS55811.2022.00022 (cit. on p. 33).

[17] *GlusterFS, scalable network filesystem*. https://www.gluster.org/, note=Accessed: March. 2022 (cit. on p. 70).

[18] 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. 22).

[19] 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 p. 15).

[20] 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. 26).

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

[22] 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).

[23] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[24] *Man HFSC website*. https://man7.org/linux/man-pages/man7/tc-hfsc.7.html. Accessed: July. 2022 (cit. on p. 11).

[25] *Man netem page website*. https://man7.org/linux/man-pages/man8/tc-netem.8.html. Accessed: July. 2022 (cit. on p. 45).

[26] 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).

[27] 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).

[28] 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. 14).

[29] 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).

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

[31] 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).

[32] 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).

[33] I. Stoica, H. Zhang, and T. E. Ng. “A hierarchical fair service curve algorithm for link-sharing, real-time and priority services”. In: *ACM SIGCOMM Computer Communication Review* 27.4 (1997), pp. 249–262 (cit. on p. 11).

[34] 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).

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

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

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

[38] J. Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014 (cit. on p. 18).

[39] *Universal 32bit Filter.* https://man7.org/linux/man-pages/man8/tc-u32.8.html, note=Accessed: May. 2021 (cit. on pp. 11, 40).

[40] A. Vahdat et al. “Scalability and Accuracy in a Large-Scale Network Emulator”. In: *SIGOPS Oper. Syst. Rev.* 36.SI (2003-12), pp. 271–284. ISSN: 0163-5980. DOI: 10.1145/844128.844154. URL: https://doi.org/10.1145/844128.844154 (cit. on p. 24).

[41] 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).

I

Experiment examples

```
<?xml version="1.0" encoding="UTF-8" ?>
<experiment boot="kollaps:1.0">
    <services>
     <service name="dashboard" image="kollaps/dashboard:1.0" supervisor="true" port="8088"/>
     <service name="n1" image="kollaps/ping-client:1.0" command="['n2', '500', '500']" />
     <service name="n2" image="kollaps/ping-client:1.0" command="['n2', '1', '1']" />
    </services>
    <links>
        <link origin="n1" dest="n2" latency="5"
            drop="0" upload="100Mbps" network="kollaps_network" />
        <link origin="n2" dest="n1" latency="5"
            drop="0" upload="100Mbps" network="kollaps_network" />
    </links>

    <dynamic>
        <schedule name="n1" time="0.0" action="join"/>
        <schedule name="n2" time="0.0" action="join"/>

        <schedule name="n1" time="6666.0" action="leave"/>
        <schedule name="n2" time="6666.0" action="leave"/>
    </dynamic>
</experiment>
```

Figure I.1: Experiment file for Kollaps single link latency emulation

```
<?xml version="1.0" encoding="UTF-8" ?>
<experiment boot="kollaps:1.0">
    <services>
     <service name="dashboard" image="kollaps/dashboard:1.0" supervisor="true" port="8088"/>
     <service name="client1" image="kollaps/iperf3-client:1.0" command="['server','1']"/>
      <service name="server" image="kollaps/iperf3-server:1.0" share="false"/>
    </services>
    <links>
        <link origin="client1" dest="server" latency="1" upload="50Kbps" download="50Kbps" netwo
    </links>
    <dynamic>
        <schedule name="client1" time="0.0" action="join"/>
        <schedule name="server"  time="0.0" action="join"/>

        <schedule name="client1" time="6666.0" action="leave"/>
        <schedule name="server"  time="6666.0" action="leave"/>
    </dynamic>
</experiment>
```

Figure I.2: Experiment file for Kollaps single link bandwidth emulation

```
dockerImgs:
  -
    name: ipltc
    tag: latest


machines:
  - address: 172.16.66.2
    number: 0

groups:
  -
    name: client
    numberDocker: 1
    imgName: ipltc
    machines: [0]
    startAddress: 172.30.10.254
    totalBandwidth: 1000
  -
    name: server
    numberDocker: 1
    imgName: ipltc
    machines: [ 0 ]
    startAddress: 172.30.20.0
    totalBandwidth: 1000

runTime: 35

network:
  name: testnet
  subnet: 172.30.0.0/16

latencyFile: /home/aatalaia/expTese/expLinkSharing/latency.txt

bandwidthFile: /home/aatalaia/expTese/expLinkSharing/bandwidth.txt

rules:
  path: /home/aatalaia/expTese/expLinkSharing/rules/
  generated: false
  serverClient: true
 serverToServerBandwidth: /home/aatalaia/expTese/expLinkSharing/serverToServerBandwidth.txt
 serverToServerLatency: /home/aatalaia/expTese/expLinkSharing/serverToServerLatency.txt

outputVolume: home/aatalaia/expTese/expLinkSharing/logs

startCommand: bash

onStart:
  actions:
    - 1000;client; collectData.sh 172.30.20.0
    - 0;server;iperf3 -s
```

Figure I.3: Experiment file for Symphony for single link emulation

```
dockerImgs:
  -
    name: myipfsd
    tag: latest


seed: abababababab

machines:
  - address: 172.16.66.64
    number: 0
  - address: 172.16.66.65
    number: 1
  - address: 172.16.66.76
    number: 2

groups:
  -
    name: peers
    numberDocker: 250
    imgName: myipfsd
    machines: [0,1,2]
    startAddress: 172.30.19.5
    totalBandwidth: 1000

network:
  name: scalingnet
  subnet: 172.30.0.0/16

runTime: 320 #seconds

latencyFile: /home/aatalaia/expTese/ExpScaling/nodes_250/latency.txt

bandwidthFile: /home/aatalaia/expTese/ExpScaling/nodes_250/bandwidth.txt

rules:
  path: /home/aatalaia/expTese/ExpScaling/nodes_250/rules/
  generated: false
  jitter: 0.1;distribution normal
  serverClient: false


outputVolume: home/aatalaia/expTese/ExpScaling/nodes_250/logs

startCommand: /ipfs/start-daemon.sh myexp

onStart:
  actions:
    - 15000;peers but {node-peers-0};/ipfs/start-daemon.sh myexp
    - 0;node-peers-0;/ipfs/start-bootstraper.sh
```

Figure I.4: Symphony experiment file for the Scaling experiment.

```
<?xml version="1.0" encoding="UTF-8" ?>
 <experiment boot="kollaps:2.0">
  <services>
   <service name="dashboard" image="kollaps/dashboard:1.0"
   supervisor="true" port="8088"/>
   <service name="peers0" image="myipfsb:latest"
   command="['/ipfs/start-bootstraper.sh']"/>
   <service name="peers1" image="myipfsd:latest"
   command="['/ipfs/start-daemon.sh', 'myexp']"/>
   .
   .
   .
   <service name="peers49" image="myipfsd:latest"
   command="['/ipfs/start-daemon.sh', 'myexp']"/>
  </services>
  <links>
   <link origin="peers0" dest="peers1" latency="67" upload="1000Mbps"
   download="1000Mbps" network="kollaps_network"/>
   <link origin="peers0" dest="peers2" latency="60" upload="1000Mbps"
   download="1000Mbps"  network="kollaps_network"/>
   .
   .
   .
   <link origin="peers47" dest="peers49" latency="76" upload="1000Mbps"
   download="1000Mbps" network="kollaps_network"/>
   <link origin="peers48" dest="peers49" latency="147" upload="1000Mbps"
   download="1000Mbps" network="kollaps_network"/>
  </links>
  <dynamic>
   <schedule name="peers0" time="0.0" action="join"/>
   <schedule name="peers0"  time="8000.0" action="leave"/>
   <schedule name="peers1" time="30.0" action="join"/>
   <schedule name="peers1"  time="8000.0" action="leave"/>
   <schedule name="peers49" time="30.0" action="join"/>
   <schedule name="peers49"  time="8000.0" action="leave"/>
  </dynamic>
</experiment>
```

Figure I.5: Kollaps experiment file for the Scaling experiment.

```
dockerImgs:
  -
    name: cassandra
    tag: 3.11.2
  -
    name: ycsb
    tag: latest


seed: abababababab

machines:
  - address: 172.16.66.91
    number: 0
  - address: 172.16.66.93
    number: 1
  - address: 172.16.66.9
    number: 2
  - address: 172.16.66.88
    number: 3
  - address: 172.16.66.96
    number: 4
  - address: 172.16.66.89
    number: 5
  - address: 172.16.66.90
    number: 6
  - address: 172.16.66.99
    number: 7
  - address: 172.16.66.94
    number: 8
  - address: 172.16.66.95
    number: 9
  - address: 172.16.66.92
    number: 10
  - address: 172.16.66.98
    number: 11
  - address: 172.16.66.97
    number: 12


groups:
  -
    name: serverseastus
    numberDocker: 4
    imgName: cassandra
    machines: [0]
    startAddress: 172.30.10.5
    customConf: -e JAVA_OPTS=-Xmx8g -e
CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch -e CASSANDRA_DC=eus -e
CASSANDRA_SEEDS=172.30.10.5,172.30.11.5,172.30.12.5,172.30.13.5,172.30.14.5,172.30.1
```

Figure I.6: Symphony Cassandra experiment part 1/5.

```
5.5,172.30.16.5,172.30.17.5,172.30.18.5,172.30.10.6,172.30.11.6,172.30.12.6,172.30.1
3.6,172.30.14.6,172.30.15.6,172.30.16.6,172.30.17.6,172.30.18.6
    totalBandwidth: 10000

  - name: serversjap
    numberDocker: 4
    imgName: cassandra
    machines: [1]
    startAddress: 172.30.11.5
    customConf: -e JAVA_OPTS=-Xmx8g -e
CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch -e CASSANDRA_DC=jap -e
CASSANDRA_SEEDS=172.30.10.5,172.30.11.5,172.30.12.5,172.30.13.5,172.30.14.5,172.30.1
5.5,172.30.16.5,172.30.17.5,172.30.18.5,172.30.10.6,172.30.11.6,172.30.12.6,172.30.1
3.6,172.30.14.6,172.30.15.6,172.30.16.6,172.30.17.6,172.30.18.6
    totalBandwidth: 10000

  - name: serversas
    numberDocker: 4
    imgName: cassandra
    machines: [2]
    startAddress: 172.30.12.5
    customConf: -e JAVA_OPTS=-Xmx8g -e
CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch -e CASSANDRA_DC=as -e
CASSANDRA_SEEDS=172.30.10.5,172.30.11.5,172.30.12.5,172.30.13.5,172.30.14.5,172.30.1
5.5,172.30.16.5,172.30.17.5,172.30.18.5,172.30.10.6,172.30.11.6,172.30.12.6,172.30.1
3.6,172.30.14.6,172.30.15.6,172.30.16.6,172.30.17.6,172.30.18.6
    totalBandwidth: 10000

  - name: serversaus
    numberDocker: 4
    imgName: cassandra
    machines: [3]
    startAddress: 172.30.13.5
    customConf:  -e JAVA_OPTS=-Xmx8g -e
CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch -e CASSANDRA_DC=aus -e
CASSANDRA_SEEDS=172.30.10.5,172.30.11.5,172.30.12.5,172.30.13.5,172.30.14.5,172.30.1
5.5,172.30.16.5,172.30.17.5,172.30.18.5,172.30.10.6,172.30.11.6,172.30.12.6,172.30.1
3.6,172.30.14.6,172.30.15.6,172.30.16.6,172.30.17.6,172.30.18.6
    totalBandwidth: 10000

  - name: serversind
    numberDocker: 4
    imgName: cassandra
    machines: [4]
    startAddress: 172.30.14.5
    customConf:  -e JAVA_OPTS=-Xmx8g -e
CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch -e CASSANDRA_DC=ind -e
CASSANDRA_SEEDS=172.30.10.5,172.30.11.5,172.30.12.5,172.30.13.5,172.30.14.5,172.30.1
5.5,172.30.16.5,172.30.17.5,172.30.18.5,172.30.10.6,172.30.11.6,172.30.12.6,172.30.1
3.6,172.30.14.6,172.30.15.6,172.30.16.6,172.30.17.6,172.30.18.6
```

Figure I.7: Symphony Cassandra experiment part 2/5.

```
    totalBandwidth: 10000

  - name: serversca
    numberDocker: 4
    imgName: cassandra
    machines: [5]
    startAddress: 172.30.15.5
    customConf:  -e JAVA_OPTS=-Xmx8g -e
CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch -e CASSANDRA_DC=ca -e
CASSANDRA_SEEDS=172.30.10.5,172.30.11.5,172.30.12.5,172.30.13.5,172.30.14.5,172.30.1
5.5,172.30.16.5,172.30.17.5,172.30.18.5,172.30.10.6,172.30.11.6,172.30.12.6,172.30.1
3.6,172.30.14.6,172.30.15.6,172.30.16.6,172.30.17.6,172.30.18.6
    totalBandwidth: 10000

  - name: serverswus
    numberDocker: 4
    imgName: cassandra
    machines: [6]
    startAddress: 172.30.16.5
    customConf:  -e JAVA_OPTS=-Xmx8g -e
CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch -e CASSANDRA_DC=wus -e
CASSANDRA_SEEDS=172.30.10.5,172.30.11.5,172.30.12.5,172.30.13.5,172.30.14.5,172.30.1
5.5,172.30.16.5,172.30.17.5,172.30.18.5,172.30.10.6,172.30.11.6,172.30.12.6,172.30.1
3.6,172.30.14.6,172.30.15.6,172.30.16.6,172.30.17.6,172.30.18.6
    totalBandwidth: 10000

  - name: serverseu
    numberDocker: 4
    imgName: cassandra
    machines: [7]
    startAddress: 172.30.17.5
    customConf:  -e JAVA_OPTS=-Xmx8g -e
CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch -e CASSANDRA_DC=eu -e
CASSANDRA_SEEDS=172.30.10.5,172.30.11.5,172.30.12.5,172.30.13.5,172.30.14.5,172.30.1
5.5,172.30.16.5,172.30.17.5,172.30.18.5,172.30.10.6,172.30.11.6,172.30.12.6,172.30.1
3.6,172.30.14.6,172.30.15.6,172.30.16.6,172.30.17.6,172.30.18.6
    totalBandwidth: 10000

  - name: serversbr
    numberDocker: 4
    imgName: cassandra
    machines: [8]
    startAddress: 172.30.18.5
    customConf:  -e JAVA_OPTS=-Xmx8g -e
CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch -e CASSANDRA_DC=br -e
CASSANDRA_SEEDS=172.30.10.5,172.30.11.5,172.30.12.5,172.30.13.5,172.30.14.5,172.30.1
5.5,172.30.16.5,172.30.17.5,172.30.18.5,172.30.10.6,172.30.11.6,172.30.12.6,172.30.1
3.6,172.30.14.6,172.30.15.6,172.30.16.6,172.30.17.6,172.30.18.6
    totalBandwidth: 10000
```

Figure I.8: Symphony Cassandra experiment part 3/5.

```
  -
    name: clients
    numberDocker: 36
    imgName: ycsb
    machines: [9,10,11,12]
    startAddress: 172.30.19.5
    totalBandwidth: 10000

network:
  name: cassnet
  subnet: 172.30.0.0/16

runTime: 7200 #seconds

latencyFile: /home/aatalaia/expTese/expCassandra/clientsToServerLatency.txt

bandwidthFile: /home/aatalaia/expTese/expCassandra/clientsToServerBandwidth.txt

rules:
  path: /home/aatalaia/expTese/expCassandra/rules/
  generated: false
  jitter: 0.1;distribution normal
  serverClient: true
  serverToServerBandwidth:
/home/aatalaia/expTese/expCassandra/serverToServerBandwidth.txt
  serverToServerLatency:
/home/aatalaia/expTese/expCassandra/serverToServerLatency.txt

outputVolume: home/aatalaia/expTese/expCassandra/logs

startCommand: bash

onStart:
  actions:
    - 3000000;clients;/startThreads.sh
    - 2190000;node-clients-32;/createYCSB-DB.sh 172.30.18.5 createBR br
    - 2190000;node-clients-28;/createYCSB-DB.sh 172.30.17.5 createEU eu
    - 2190000;node-clients-24;/createYCSB-DB.sh 172.30.16.5 createWUS wus
    - 2190000;node-clients-20;/createYCSB-DB.sh 172.30.15.5 createCA ca
    - 2190000;node-clients-16;/createYCSB-DB.sh 172.30.14.5 createIND ind
    - 2190000;node-clients-12;/createYCSB-DB.sh 172.30.13.5 createAUS aus
    - 2190000;node-clients-8;/createYCSB-DB.sh 172.30.12.5 createAS as
    - 2190000;node-clients-4;/createYCSB-DB.sh 172.30.11.5 createJAP jap
    - 2190000;node-clients-0;/createYCSB-DB.sh 172.30.10.5 createEUS eus
    - 2160000;node-serverseastus-3;/usr/local/bin/docker-entrypoint.sh
    - 2040000;node-serversjap-3;/usr/local/bin/docker-entrypoint.sh
    - 1920000;node-serversas-3;/usr/local/bin/docker-entrypoint.sh
    - 1800000;node-serversaus-3;/usr/local/bin/docker-entrypoint.sh
    - 1680000;node-serversind-3;/usr/local/bin/docker-entrypoint.sh
    - 1560000;node-serversca-3;/usr/local/bin/docker-entrypoint.sh
```

Figure I.9: Symphony Cassandra experiment part 4/5.

```
    - 1440000;node-serverswus-3;/usr/local/bin/docker-entrypoint.sh
    - 1320000;node-serverseu-3;/usr/local/bin/docker-entrypoint.sh
    - 1200000;node-serversbr-3;/usr/local/bin/docker-entrypoint.sh
    - 1080000;node-serverseastus-2;/usr/local/bin/docker-entrypoint.sh
    - 960000;node-serversjap-2;/usr/local/bin/docker-entrypoint.sh
    - 840000;node-serversas-2;/usr/local/bin/docker-entrypoint.sh
    - 720000;node-serversaus-2;/usr/local/bin/docker-entrypoint.sh
    - 600000;node-serversind-2;/usr/local/bin/docker-entrypoint.sh
    - 480000;node-serversca-2;/usr/local/bin/docker-entrypoint.sh
    - 360000;node-serverswus-2;/usr/local/bin/docker-entrypoint.sh
    - 241000;node-serverseu-2;/usr/local/bin/docker-entrypoint.sh
    - 121000;node-serversbr-2;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serverseastus-1;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serversjap-1;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serversas-1;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serversaus-1;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serversind-1;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serversca-1;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serverswus-1;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serverseu-1;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serversbr-1;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serverseastus-0;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serversjap-0;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serversas-0;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serversaus-0;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serversind-0;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serversca-0;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serverswus-0;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serverseu-0;/usr/local/bin/docker-entrypoint.sh
    - 1000;node-serversbr-0;/usr/local/bin/docker-entrypoint.sh
```

Figure I.10: Symphony Cassandra experiment part 5/5.

2023 Symphony: A Scalable Emulator for Distributed Systems André Atalaia

NOVA SCHOOL OF SCIENCE & TECHNOLOGY