

**KOLLAPS:
Decentralized and Dynamic Topology Emulation**

Paulo Jorge Louseiro Gouveia

Thesis to obtain the Master of Science Degree in
Information Systems and Software Engineering

Supervisor: Prof. Miguel Ângelo Marques de Matos

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva

Supervisor: Prof. Miguel Ângelo Marques de Matos

Member of the Committee: Prof. Fernando Ramos

November 2019

Acknowledgments

First, I would like to thank Professor Miguel Matos for the opportunity to work on this project and for all the time and effort invested in this work.

Second, I would like to thank Professor Valerio Schiavoni, from the University of Neuchâtel, for all the helpful insights and for granting access to the cluster on which this work was evaluated.

Finally, I would like to thank João Neves, the original author of NEED, for all the help throughout the development and evaluation of KOLLAPS.

This work was partially funded by Fundo Europeu de Desenvolvimento Regional (FEDER) through Programa Operacional Regional de Lisboa, and Fundos Nacionais through FCT - Fundação para a Ciência e Tecnologia, for the project Lisboa-01-0145-FEDER-031456 (Angainor) and UID/CEC/50021/2019.

Resumo

O comportamento de aplicações distribuídas de grande escala é fortemente influenciado pelas propriedades da rede como latência, largura de banda e perda de pacotes. É, portanto, fundamental averiguar o desempenho destas aplicações de uma forma sistemática e reproduzível em cenários controlados.

Uma abordagem possível é o uso de emuladores ou testbeds. Infelizmente, o atual estado-da-arte em emulação de redes e plataformas de teste apresentam várias lacunas tais como escalabilidade limitada (ex.: MiniNet), falta de suporte a dinamismo na rede (ex.: EmuLab), ou pelo foco exclusivo no plano de controlo (ex.: CrystalNet). Nós argumentamos que estas limitações vêm da emulação do estado total da rede, nomeadamente o estado de *routers* e *switches*.

Esta tese propõe o KOLLAPS, um emulador descentralizado, agnóstico da aplicação e protocolo de transporte, e capaz de escalar a milhares de processos mantendo uma precisão equivalente às abordagens do estado-da-arte centralizadas.

Apresentamos uma avaliação com micro- e macro-benchmarks que demonstram a escalabilidade e precisão do KOLLAPS.

Palavras-chave: Emulação de Redes, Contentores, Descentralizado, Sistemas Distribuídos, Reproducibilidade

Abstract

The performance and behavior of large-scale distributed applications is highly influenced by network properties such as latency, bandwidth, packet loss, and jitter. It is, therefore, fundamental to assess the performance of these applications in a systematic and reproducible manner.

One possible approach is to resort to network emulation or testbed environments. Unfortunately, the current state-of-the-art approaches do not scale beyond a single machine or small cluster (*e.g.* MiniNet), are focused exclusively on the control-plane (*e.g.* CrystalNet) or do not support network dynamics (*e.g.* EmuLab). We argue that these limitations come from the emulation of the complete state of the network, namely router and switches.

We propose KOLLAPS, a decentralized emulator, agnostic to application and transport protocol, capable of scaling to thousands of processes and achieves an emulation accuracy on-par with centralized approaches.

We present an evaluation with micro- and macro-benchmarks that show the scalability and accuracy of KOLLAPS.

Keywords: Network Emulation, Containers, Distributed Systems, Decentralized, Reproducibility

Contents

| | |
|---|-----------|
| Acknowledgments | iii |
| Resumo | v |
| Abstract | vii |
| List of Tables | xi |
| List of Figures | xiii |
| 1 Introduction | 1 |
| 1.1 Contributions | 3 |
| 1.2 Thesis Outline | 3 |
| 2 Background | 5 |
| 2.1 Experimentation Platforms | 5 |
| 2.2 Linux Traffic Control | 6 |
| 2.3 RTT-aware Min-Max | 7 |
| 2.4 Docker | 8 |
| 3 Related Work | 11 |
| 3.1 Dummynet | 11 |
| 3.2 Trickle | 12 |
| 3.3 IMUNES | 12 |
| 3.4 Mininet | 13 |
| 3.5 Maxinet | 14 |
| 3.6 ModelNet | 14 |
| 3.7 SplayNet | 16 |
| 3.8 CrystalNet | 16 |
| 3.9 NEED | 17 |
| 3.10 Summary | 18 |

| | | |
|----------|---|-----------|
| 4 | Architecture | 21 |
| 4.1 | Overall architecture | 21 |
| 4.2 | Deployment Generator | 22 |
| 4.3 | Bootstrapper and Master Container | 22 |
| 4.4 | Emulation Manager | 23 |
| 4.5 | tc abstraction layer (TCAL) | 25 |
| 4.6 | Dashboard | 25 |
| 4.7 | Summary | 25 |
| 5 | Implementation | 27 |
| 5.1 | Deployment Generator | 27 |
| 5.2 | Master Container | 28 |
| 5.3 | Emulation Manager | 28 |
| 5.4 | Aeron | 30 |
| 5.5 | tc abstraction layer | 31 |
| 5.6 | Summary | 32 |
| 6 | Evaluation | 33 |
| 6.1 | Link-level emulation accuracy | 33 |
| 6.2 | Emulation reaction time | 34 |
| 6.3 | Metadata bandwidth usage | 36 |
| 6.4 | Scalability of bandwidth emulation accuracy | 37 |
| 6.5 | Scalability of latency emulation | 38 |
| 7 | Conclusion | 41 |
| 7.1 | Achievements | 41 |
| 7.2 | Future Work | 42 |
| | Bibliography | 43 |
| A | Kollaps topology example | 49 |

List of Tables

| | | |
|-----|---|----|
| 6.1 | Study of the bandwidth shaping accuracy for different emulated link capacities and tools on a (client – server) topology. | 34 |
| 6.2 | Mean squared error exhibited on latency tests with large scale-free topologies in KOLLAPS, NEED, Mininet and Maxinet. | 39 |

List of Figures

| | | |
|-----|--|----|
| 3.1 | NEED architecture and workflow. | 18 |
| 4.1 | KOLLAPS architecture. | 22 |
| 4.2 | Example of a topology and an equivalent collapsed topology. | 23 |
| 4.3 | Example of the information reported by the Dashboard. | 26 |
| 6.1 | Dumbbell topology with 3 clients, 3 servers, where the links connecting the clients to switch s1 all have different properties. | 35 |
| 6.2 | Reaction time to throttle flows that vary in time. | 35 |
| 6.3 | KOLLAPS metadata bandwidth with an increasing number of flows and containers, 1 source and 1 destination per flow, F : concurrent data flows, C total deployed containers. | 36 |
| 6.4 | Comparison of generated metadata between KOLLAPS and NEED, 1 source and 1 destination per flow. F : concurrent data flows, C total deployed containers. . . . | 37 |
| 6.5 | Emulation accuracy for different update periods and total number containers. The topology allows at most 20 Mbit/s per flow. | 38 |

Chapter 1

Introduction

As applications leveraging large-scale distributed systems become more complex and resource intensive, so does their evaluation become harder, slower and expensive. This difficulty stems from the large number of moving parts one has to be concerned about: system dependencies and libraries, heterogeneity of the target environment, network variability and dynamics, among others.

The advent of container technology (*e.g.*, Docker [Mer14], Linux LXC [lin19]) and container orchestration (*e.g.*, Docker Swarm [doc19b], Kubernetes [k8s19]) greatly simplifies the deployment and partially addresses environment heterogeneity. The main challenge is thus how to systematically evaluate the system in a WAN environment. As a matter of fact, the inherent variability of WAN conditions (*i.e.*, failures, contention and reconfigurations), makes it hard to assess the impact of changes in the application logic. Can we actually be sure the observed performance improvement was really due to the changes made to the application, or was it just due to a *lucky* run when the network was lightly loaded? How is performance affected by common network dynamics, such as background traffic, or link flapping? The very same questions and issues also arise in the reproducibility crisis currently plaguing down the distributed systems' community [Boi16, Pen11]. Different results for the same system emerge not only because systems are evaluated in different uncontrollable conditions, but also because research testbeds such as Emulab [HRS⁺08], CloudLab [RET14], or PlanetLab [CCR⁺03] where such experiments are conducted tend to get overloaded right before system conference deadlines [KRLP11]. We therefore need tools to systematically assess and reproduce the evaluation of large-scale applications.

One approach to systematically evaluate a large-scale distributed system is to resort to simulation, which relies on models that capture key properties of the target system and environment [BCN10]. Simulation provides full control of the system and environment — and

hence it is fully reproducible — and allows to study the model of the system in a variety of scenarios. However, simulations suffer from several well-known problems. On one hand, there is a large gap between the simulated model and the real-world deployment, usually leading to several unforeseen behaviors not captured by the model [CGR07, FK03, FP01, PF97]. On the other hand, even if the simulated model yields correct results, the real implementation is not guaranteed to faithfully follow the simulated model. Moreover, despite some efforts to model complex systems either for formal method analysis [NRZ⁺15] or simulation, this is, to the best of our knowledge, seldom the case for large-scale systems.

The alternative is to resort to network emulation. In network emulation, the real system is run against a model of the network that replicates real-world behavior by modeling a network topology together with its network elements, including switches, routers and their internal behavior. Emulation thus allows to reach conclusions about the behavior of the real system in a concrete scenario rather than its model. Unfortunately, state-of-the art network emulators suffer from several limitations. MiniNet [HHJ⁺12a] is limited to a single physical machine and therefore cannot be used to emulate a large-scale system. MaxiNet [WDS14a] and the multi-host version of MiniNet support distributed clusters but scale poorly [LZP⁺17]. Approaches such as ModelNet [VGVEY09a] rely on a dedicated cluster of nodes to maintain the emulation model to which the application nodes must connect. However, accuracy is highly dependent on application traffic patterns and quickly degrades with a modest increase in the number of application nodes. CrystalNet [LZP⁺17] accurately emulates the *control-plane* of large-scale networks (*e.g.* routing tables, software switch versions or device firmwares) but cannot be used to emulate the data-plane (*e.g.* latency, bandwidth), and hence evaluate the behavior of large-scale distributed systems.

One could also resort to emulation testbeds (*e.g.*, Emulab [HRS⁺08]). While this provides a semi-controlled environment and network, it cannot model network dynamics and thus one cannot assess their impact on application behavior.

To summarize, to the best of our knowledge, existing tools do not allow to systematically assess and reproduce the evaluation of large-scale applications subject to network dynamics. This is where KOLLAPS comes in. KOLLAPS is a decentralized network topology emulator. It emulates a network topology beneath unmodified containerized applications, is agnostic to the application language and transport protocol, and can scale to thousands of containers, maintaining an emulation accuracy that is on par with other state-of-the-art systems that emulate the full state of the network.

KOLLAPS builds on previous work (NEED [Nev18a]) and addresses the limitations of it,

namely scalability and efficiency, which are the goal of this thesis to solve. First, the initial iteration of KOLLAPS requires a broadcast of information regarding the flows and bandwidths used by each container to every other container. This hurts the system’s scalability because with a significant amount of application containers the impact of these packets on the network is no longer negligible. Second, it is possible for the computations to find the maximum bandwidths on paths to be repeated on different containers. The idea behind KOLLAPS is to share the flow information through shared memory for containers on the same physical host, reducing the network overhead, and to merge emulation processes into one, being the computations done only once for every physical host.

1.1 Contributions

Our main contributions are:

- We present KOLLAPS, a network topology emulator that allows evaluation of large-scale applications in dynamic networks;
- We integrate KOLLAPS with Docker Swarm and Kubernetes, to deploy and evaluate unmodified containerized (distributed) applications;
- We implement a system to exchange information between containers on the same physical host using shared memory;
- We evaluate KOLLAPS with a series of micro- and macro-benchmark that show the precision of our emulation model;
- We show that the emulation accuracy of KOLLAPS is on par with other centralized state-of-the-art approaches.

1.2 Thesis Outline

The remainder of the document is structured as follows. Chapter 2 provides background knowledge regarding the technologies KOLLAPS is currently employing. Chapter 3 presents a review of existing network experimentation tools described in the literature. In Chapter 4 we explain the design and architecture of KOLLAPS followed by a detailed explanation of its implementation in Chapter 5. In Chapter 6 we present an evaluation of KOLLAPS, showing experiment results and discussing them. Finally, in Chapter 7 we conclude the document and discuss future work.

Chapter 2

Background

In this chapter, we cover the technologies used by KOLLAPS in order to run and control experiments. First, we distinguish testbeds, simulators and emulators, the tools at disposal of developers to test their network applications. Second, we will go over Linux’s traffic control [Hub01] and the RTT-aware Min-Max, which KOLLAPS uses to set the emulated link properties and to define per flow bandwidths on shared links, respectively. Third, we cover Docker [doca] and Docker Swarm [docd], which KOLLAPS uses to deploy and coordinate containers over a cluster. Finally we cover Aeron, the system KOLLAPS uses for the metadata dissemination architecture.

2.1 Experimentation Platforms

When trying to replicate network conditions in a controlled environment there are usually three main types of systems: network simulators, testbed networks, and network emulators.

Network simulators such as ns-3 [RH10] or SENSE [BHK07], imitate the behavior of real systems using models that simplify the study of interactions between system components. These tools are generally highly configurable, extensible and provide controlled and repeatable evaluation conditions. However, they usually make simplifying assumptions resulting in them working over an abstract model of the platform, which can lead to an inaccurate representation of traffic dynamics [BRNR15]. Therefore, due to their limited level of realism, they often offer little help with real-world experiments but rather focused experiments that would be too costly on more realistic platforms.

Testbed networks are environments made up of dedicated network hardware (routers and hosts), interconnected and configured to form a network with the desired topology and properties of the target production network [ZM04]. Since everything from the devices to the protocols

and applications is real, testbeds allow the study of a system in realistic conditions, exposing complex interactions between network components that could pass unnoticed on simulators. However, these systems are expensive and time-consuming to set up and maintain [Que15]. Physical resources are scarce and expensive, which leads to a limited technology variety when compared with software alternatives. PlanetLab [CCR⁺03] is an example of a geographically distributed testbed, which presents yet another restriction. Similarly to deploying experiments on virtual machines rented from services like Amazon EC [ec2], Microsoft’s Azure [azu19], etc., these environments do not behave in a fully controllable or predictable manner. Therefore, achieving experiment repeatability and reproducibility becomes difficult, if possible at all.

Network emulators stand in the middle of the approaches described above: they subject real network traffic and systems to a synthetic network environment [ZM04]. Emulation provides a controlled and reproducible experiment environment but most importantly, applications and protocols implemented are tested without any modification, which means they can be deployed as they were tested [BRNR15]. The benefits and drawbacks of emulation when compared with simulators and dedicated testbeds vary according to the complexity and scope of the emulator. Dummynet [Riz97] is able to emulate packet losses and delays, bandwidth constraints and different queuing policies. However, it lacks the ability to emulate network topologies. Mininet [DOSSP14] is another example of an emulator, which can emulate such topologies in a single physical host. In general, the major adversary of emulation environments is scalability. Past a certain number of emulated devices, data traffic starts competing with control traffic for the available resources resulting in loss of accuracy and consequently affecting the validity of the experiment results.

2.2 Linux Traffic Control

Linux offers a rich set of traffic control functions for managing and manipulating the transmission of network packets. Before being dispatched to the respective network interfaces, packets are placed into queues. Through the `tc` command [Hub01], the Linux kernel allows users to modify the behaviour of packets within these queues, referred to as **qdiscs** (queueing disciplines). **Qdiscs** are categorised as **classless** and **classful** [deb]. **Classless qdiscs** are queues with a straightforward behavior. They do not possess any mechanisms for finer control of packet behavior, and an example of these would be a simple FIFO. **classful qdiscs** can contain classes, providing handles to which to attach filters, which in turn have classifiers. All these components make it possible to create a hierarchy of qdiscs that a packet must traverse, providing a select degree of flexibility and fine control.

The Linux kernel offers several qdiscs by default. Within the scope of this work, three are of notable importance as they are the ones specifically used by KOLLAPS [Nev18a]. These are: **HTB** (Hierarchy Token Bucket) [MD02], **prio** (Priority qdisc) [ANK01], and **netem** (Network Emulator) [FL11]. **HTB** allows the user to control the outbound bandwidth on a given link. It shapes traffic based on the Token Bucket algorithm which does not depend on interface characteristics and therefore does not need to know the underlying bandwidth of the outgoing interface. **Prio** is a classful queueing discipline made up of multiple classes of differing priority. It dequeues packets in the order of their priority: packets assigned to a higher priority class are always pulled from the queue before packets from lower priority classes. **Netem** is a queueing discipline that allows the user to add delay, loss, duplication and more characteristics to outgoing packets.

Regarding filters, KOLLAPS takes advantage of the **u32** filter [u3201] to place the packets into the correct qdisc. The **u32** filter allows matching against any arbitrary bitfields in a network packet, namely the destination IP address. Another important feature of **u32** is that filter delegation is done through the use of hash tables, which allows traffic to be matched against several different rules in constant time, proportional to the number of rules.

2.3 RTT-aware Min-Max

In real networks, there are different variants of TCP that handle link congestion between devices in different ways. However, KOLLAPS cannot rely on such algorithms because it does not directly emulate internal network devices but rather their effects directly on the source nodes. In order to correctly emulate the constraints of the different links that comprise a path (the set of links packets must traverse between the source and destination), KOLLAPS must know a priori the effects flows (sequences of packets) will have on the topology such as how much bandwidth a flow is allowed to take on some link. To achieve this KOLLAPS resorts to the Round-Trip Time (RTT) Aware Min-Max model, described in [Kel97] and [MR99], and also employed by Splaynet [SRF13], another network emulator that does not directly emulate internal network devices.

This model consists of computing a fair share of the link’s available bandwidth for each flow. In KOLLAPS, every node maintains a graph extracted from the original topology, comprised of the shortest paths between every pair of nodes. When a flow takes a path where links do not need to be shared with other flows, the flow will take a bandwidth dictated by the link in the path with the lowest bandwidth. When links do need to be shared, however, each flow f gets a share of the bandwidth inversely proportional to its RTT according to the following formula, which gives us a percentage of the maximum bandwidth that any given flow is allowed to use.

$$Share(f) = \frac{RTT(f)^{-1}}{\sum_{i=1}^n RTT(f_i)^{-1}} \quad (2.1)$$

However, a flow might be using less bandwidth on a specific link than the share it is given because its actual limitation is another link in the path. Therefore this formula does not guarantee that all the available bandwidth on a link will be utilized. Consequently, a maximization step is required where all other unrestricted flows have their maximum allowed bandwidth increased, proportionally to their original shares.

2.4 Docker

Containers, much like virtual machines, are a solution to the problem of how to get software to run reliably when moved from one computing environment to another, for example moving from the developers' workstations to a production network. With virtual machines, the entire image must be transferred. This includes the operating system and all required applications, which can easily take up gigabytes of space as well as have a severe impact on the performance of virtualization when multiple VMs are running on the same host. By contrast, a container consists only of application code and all dependencies required for it to run reliably on any environment. A container image is a standalone executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings. That means containers are much more lightweight than virtual machines. Multiple options exist to deploy container-based applications, of which the most popular and widely used is Docker [doca]. KOLLAPS takes advantage of Docker containers and Swarm to deploy a network of containers over multiple physical hosts, so this section will focus on Docker.

Docker is a widely used container platform that leverages lightweight virtualization techniques provided by the Linux kernel, such as kernel namespaces [man18], to run processes in an isolated environment. On Docker, regulation of the container's connectivity is done through Container Network Model (CNM) plugins, referred to as network drivers. These drivers are the ones responsible for creating the necessary Linux bridges, internal interfaces, iptables rules, and host routes to make connectivity possible [Chu].

There are currently five drivers provided by default: **bridge**, **host**, **overlay**, **macvlan** and **none** [docc]. They work by creating a virtual Ethernet pair, which consists of two virtual interfaces connected together. Docker then moves one end of the pair to a unique network namespace and assigns the processes running inside the container to that network namespace [Hau16]. All plugins perform this setup step with the exception of **host**, which gives the container direct

access to the network interfaces of the host, and **none**, which disables all container networking. This operation gives the container a unique network interface together with an independent IP stack, routing tables, firewall rules, and other kernel networking data structures, and isolates the container from other existing interfaces (both on the host and on other running containers) [Nev18a].

Within the scope of this work, the **overlay** driver is of particular interest as it is the default driver for multi-host deployments and the one used by KOLLAPS to connect together containers that are running across a cluster of physical machines. **Overlay** extends the normal **bridge** with a peer-to-peer communication system using VXLAN tunnels [MDD⁺14] and uses a key-value store, reachable by all hosts in the cluster, to distribute its state [Mer]. When using this driver, Docker has a built-in IPAM system that can automatically manage IP pools, assigning containers a unique address upon start. VXLAN tunnels encapsulate data between endpoints that are created at each node in the cluster. The tunnels encapsulate data from the data layer and above, generated by the containers connected to the network, into a network layer packet on the underlying infrastructure. This allows for routing through the underlying physical network infrastructure to be done transparently and requires traffic to be encapsulated/de-encapsulated only at the tunnel endpoints [Nev18a].

When deploying containers on clusters, container orchestration tools are essential to automate configurations and manage the resources in a fair and efficient manner. To achieve this, Docker provides a tool called **swarm mode** [docd], which works with **services** [docb]. **Services** can be a single container or a collection of replicated containers based on the same image. The user defines the desired state of the service using a declarative model, specifying things such as how many containers participate in the service and whether any ports are exposed to clients outside the swarm, and relies on Docker to manage the service’s state.

Chapter 3

Related Work

As covered in Chapter 2, network experimentation systems from literature can be categorized as dedicated testbeds, simulators or emulators. KOLLAPS and its predecessor NEED are network emulation systems. In this section, we will cover some other network emulation tools and outline their strengths and weaknesses. These systems differ in terms of the scope in which they operate (kernel or user space) and the network properties they are able to emulate (bandwidth limiting, packet loss, etc.).

3.1 Dummynet

Dummynet [Riz97] is a network emulator that can be used to enforce desired link properties on network interfaces.

Dummynet is based on kernel modifications and it works with a structure called a **pipe**, provided by its emulation engine. **Pipes** combine a queue with finite size and a communication link with fixed bandwidth and programmable propagation delay. Packets are intercepted and delivered to the packet classifier, **ipfw**, which uses a list of rules to match them and decide what to do with them, such as delay the packet or drop it. When packets are passed to a pipe, if there is capacity, the packet is queued, otherwise, it is dropped. The queue is drained at a rate corresponding to the link’s allowed bandwidth and once outside the queue, a packet is staged in a delay line for a time corresponding to the propagation delay of the link.

Dummynet’s low-level operations of packet classification and control allow it to achieve better accuracy than similar userspace implementations. The number of allowed pipes is only limited by the systems’ memory but Dummynet lacks the mechanisms to emulate entire network topologies. It is, however, widely used by other systems like Modelnet [VGVEY09b] to achieve that goal.

3.2 Trickle

Trickle [Eri05] is a portable, cross-platform approach to download and upload rate limiting that runs entirely in userspace. Trickle instances can collaborate with each other to enforce network-wide policies for delays and bandwidth limits, or they may run independently. Using dynamic linking and loading, Trickle stands as a proxy between the process whose traffic is to be shaped and the socket implementation provided by the system.

In `trickled`, a daemon that coordinates among multiple instances of Trickle, every collaborating process is represented by an entity and every entity is assigned a priority according to a user-specified policy. When an entity is ready to perform an I/O operation it consults the scheduler. The scheduler may then advise the entity to delay its request, to partially complete the request (truncate the I/O operation), or a combination of the two. After an entity has performed the I/O operation, it notifies the scheduler with the length and type of the operation.

One big difference between Trickle and in-kernel rate limiters is that Trickle only has access to much higher levels of the OSI layers. This is an advantage because it means it is easy to configure and apply specific rules for individual processes. However, it is also a disadvantage because while in-kernel rate limiters typically schedule discrete packets and reside in or close to the network layer, Trickle only has access to sockets. This means it must rely on the underlying layer semantics in order to be effective. Furthermore, it is affected by local buffering in every OSI layer, and this effectively reduces the reaction time it can provide. Together, these conditions severely reduce the granularity at which Trickle can operate and hence, its accuracy.

Trickle's approach to delaying I/O operations tends to result in very bursty network behavior that interferes with the computation of the rate limits. For this reason, burst smoothing techniques are employed, which results in yet another tradeoff: it becomes likely to lose some accuracy because of timing. To avoid sparse large flows, I/O operations are often divided into smaller ones. Timers in userland are not very accurate, and thus when sleeping on a timer for several I/Os the inaccuracies may become more impactful than when waiting only once for one single operation.

3.3 IMUNES

IMUNES [ZM04] is a network emulator that consists of independent virtual nodes and links, which can be individually configured, interconnected, accessed and observed just like nodes in a real network. It operates in real-time and does the network emulation at the packet level using a real TCP/IP stack.

A virtual node is composed of a network stack instance and zero or more associated user space processes. Each network stack is functionally independent of all others, so each instance maintains its own private set of state variables such as the list of interfaces, routing cache and tables, communication sockets, etc.

During the OS boot process, real network interfaces are automatically assigned to the default network stack instance and after additional virtual nodes have been instantiated, IMUNES relies on the **netgraph** framework [net] from the FreeBSD kernel to reassign network interfaces from one virtual node to another. All virtual links pass the packets through an instance of the **ng_pipe** traffic shaper, which is also implemented as a **netgraph** node, so that bandwidth limits, delays, and bit errors can all be emulated.

IMUNES enables each virtual node to run a private copy of unmodified user-level applications, including routing protocol daemons, traffic generators, analyzers, or application servers [imu]. IMUNES leverages lightweight virtualization techniques provided by the FreeBSD kernel, resulting in more efficient use of resources and less redundancy, since only the network related functions are node specific. Fidelity is also high since all emulated nodes use real system calls for network traffic processing. However, IMUNES is confined to only one machine, which limits its scalability due to both CPU and memory bottlenecks.

3.4 Mininet

Mininet [LHM10] started out as a system for prototyping large networks on the constrained resources of a single laptop. Its lightweight approach of using OS-level virtualization features, such as processes and network namespaces, allows it to scale to hundreds of nodes on a single machine. This contrasts with the use of virtual machines, which have a significant memory overhead per VM.

Mininet is composed of **links**, **hosts**, **switches** and **controllers**. **Hosts** are simple shell processes running in their own network namespaces. Each of these network namespaces provides processes with exclusive ownership of interfaces, ports and routing tables. **Links** are virtual Ethernet pairs that act as a connection between two virtual interfaces. **Switches** are OpenFlow [MAB⁺08] software switches equivalent to regular hardware switches. Lastly, **controllers** control the experiment environment.

The most significant limitation of the first implementation of Mininet is a lack of performance fidelity, especially at high loads. CPU resources for processes are dictated by the default Linux scheduler, which provides no guarantee that a host that is ready to send a packet will be scheduled promptly. Not only other applications running on the host impact the accuracy of

Mininet, packet forwarding rate drops significantly with the size of routing tables because of the $O(n)$ lookup that is required. Mininet-HiFi [HHJ⁺12b] however, has improved upon the original implementation of Mininet with resource isolation, provisioning, and monitoring to tackle these issues. It groups processes hierarchically and enforces CPU bandwidth limits for those groups so that CPU time is fairly shared among them. Mininet-HiFi also uses the `tc` command to configure link properties such as bandwidth, delay, and packet loss.

Although Mininet can scale to hundreds of nodes in a single physical host, the fact that it is limited to one single host means its scalability is reduced when experiments require resource-intensive applications. Therefore, it is not a favored choice for testing large scale distributed systems.

3.5 Maxinet

Maxinet [WDS⁺14b] is a distributed network emulation tool that, to tackle Mininet’s scalability limitations, was created as an abstraction layer connecting multiple, unmodified Mininet [LHM10] instances running on different worker machines.

Communication between Maxinet and Mininet instances is done through RPC calls to the Mininet API and communication between workers is achieved by Generic Routing Encapsulation tunnels (GRE tunnels) [FLH⁺00]. These GRE tunnels are an IP packet encapsulation protocol and are used to hide from Mininet switches the fact that they are running on different hosts. Maxinet also employs a mechanism to deal with the CPU constraints of Mininet called time dilation.

Maxinet successfully deals with Mininet’s restrictions but it does so with some notable trade-offs. First, GRE tunnels only work between switches. Therefore, all emulated hosts that connect to the same switch must be deployed on the same worker as the switch. Furthermore, Maxinet has no notion of either the performance of workers or the physical network. This restricts its usage in heterogeneous environments. The load is aimed to be evenly distributed over all workers, so weak machines will become the bottleneck of the emulation.

3.6 ModelNet

ModelNet [VYW⁺02] is an emulation environment that enables researchers to deploy software prototypes in a configurable Internet-like environment and subject them to faults and varying network conditions.

The environment is comprised of **core** and **edge nodes**. Edge nodes contain a number of

Virtual Edge Nodes (VNs), each running arbitrary user-specified OS and application software. All their traffic is routed through a group of core nodes, also referred to as the **ModelNet core**, that must cooperate between themselves to emulate the bandwidth, congestion, latency, and packet loss properties of the desired target network before relaying the packets back to the destination edge nodes.

ModelNet is based on Dummynet [Riz97] and runs in five phases: *Create*, *Distillation*, *Assignment*, *Binding* and *Run*. The first phase, *Create*, generates a network topology consisting of a graph whose edges represent network links and whose nodes represent network devices. The *Distillation phase* takes the graph and creates a network of **pipes** within the ModelNet core according to the target topology. The *Assignment phase* maps pieces of the distilled topology to ModelNet core nodes, partitioning the pipe graph to distribute emulation load across the core nodes. The *Binding phase* pre-computes the shortest-path routes among all pairs of VNs in the distilled topology and installs them in a routing matrix in each core node. In this process, it also assigns VNs to edge nodes and configures them to execute applications. And finally, the *Run phase* executes the target application code on the edge nodes.

One of the key features of ModelNet is the ability to get increased emulation scalability for reduced accuracy. This is done at the *Distillation phase* where, as an option, the network may be simplified, trading accuracy for reduced emulation cost. Reduced emulation cost can be achieved by reducing the complexity of the emulated topology and/or multiplexing multiple VNs onto a single physical machine, and emulation accuracy can be increased by introducing synthetic background traffic to dynamically change network characteristics and to inject faults.

ModelNet allows the use of unmodified end-user applications and more recent efforts have also implemented time dilation in order to manipulate the time it takes to run experiments [VG09]. However, there are two notable issues with ModelNet. The first is that all packets must go through the ModelNet core, meaning the scalability will be limited by the core’s hardware resources. The second is that since there is no resource isolation among competing VNs running on the edges, it is possible for a single VN to transmit UDP packets as fast as allowed by the hardware configuration, preventing TCP flows originating from the same physical host to obtain their proper share of emulated bandwidth. Finally, ModelNet assumes the existence of a perfect routing protocol that calculates the shortest paths between all pairs of nodes. This choice means that upon link failure it is also assumed that the routing protocol is able to instantly discover resulting shortest paths.

3.7 SplayNet

SplayNet [SRF13] is an integrated user-space network emulation system based on the SPLAY framework [LRF09].

The first step in a SplayNet experiment is to define a network topology to emulate. Users write an abstract description that maps vertices and edges of an undirected cyclic graph to the physical connections of a network, specifying the physical properties of the links (delays, bandwidth, and packet loss rate). Next, the user submits to the Splay controller the topology description, the code to execute, and any additional files required for the experiments. An all-pairs-shortest-path algorithm based on link delays is then used to determine every shortest path and compute the maximum available bandwidth, the overall delay, and the packet loss probability. Finally, SplayNet is responsible for allocating resources for executing the applications on the emulated topology, which corresponds to selecting a minimal set of **splayds** for executing the job whilst avoiding saturating the bandwidths of the physical links and ensuring that the deployment of a topology does not impair the accuracy of other deployed topologies.

SplayNet runs entirely in userspace and allows for emulation of arbitrary network topologies deployed across several physical hosts. It also supports the deployment of several experiments simultaneously, each under different network emulation conditions, as long as the underlying physical infrastructure can provide enough resources. The main standout difference, when compared to other emulators, is that with SplayNet inner nodes from network topologies (*i.e.* routers and switches) are never instantiated on real machines. The topologies are collapsed into just end nodes and the emulation of real network conditions is achieved in a decentralized manner using distributed congestion emulation algorithms. SplayNet offers equivalent performance to state-of-the-art systems, both in terms of latency emulation and bandwidth shaping accuracy, and was shown to scale well for concurrent deployments of real-world distributed protocols and large topologies. However, experiments must be set up as scripts using the SPLAY framework and written in the Lua programming language, requiring the user to be familiar with both.

3.8 CrystalNet

CrystalNet [LZP⁺17] is a high-fidelity, cloud-scale network emulator, developed by Microsoft in order to test changes in their networks.

To accurately mimic the control plane, CrystalNet runs real network device firmware in virtualized sandboxes (containers and virtual machines) and interconnects them with virtual links to mimic real topologies. It loads real configurations into the emulated devices and injects

real routing states into the emulated network. This enables a faithful replication of the network control plane and a reasonable approximation of the data plane.

The **orchestrator** is the main component of CrystalNet. It is responsible for reading the description of production networks and provisioning VMs (virtual machines) accordingly, starting device virtualization sandboxes in the VMs with the appropriate virtual interfaces, building an overlay network among the sandboxes, and introducing some external device sandboxes to emulate external networks. In addition, CrystalNet creates a management overlay which connects all devices to a Linux Jumpbox VM that can access the devices in the same way as in production. To tackle the restriction that its API must work for all emulated devices, CrystalNet runs the tested devices inside **PhyNet** containers. These containers run their binaries independently of the devices being tested and are connected between themselves as in the target topology. These containers are also used to hold all virtual interfaces and tools for emulation control, such as packet injection and pulling.

CrystalNet offers uniform management over heterogeneous device software, can simultaneously integrate with multiple public clouds, private clusters, and physical devices, and runs real device firmware on emulated devices. Unfortunately, CrystalNet is exclusively concerned with the control plane and does not guarantee a faithful emulation of the network data plane (latency, link bandwidth, traffic volume etc.). Although Microsoft has announced their plans to rename CrystalNet to ONE (Open Network Emulator) and release it as open source software [ms218], at this time CrystalNet is still a proprietary technology.

3.9 NEED

NEED [Nev18a] is a network emulator that leverages Docker containers to emulate the macro behavior of networks rather than its internal details.

Given an existing container image, the user extends it using a provided template to produce a new image. Then, the user defines a topology description, which is given as input to the **Deployment Generator**, that produces a Docker Compose File to be used by Docker Swarm [docd]. Once the containers are deployed over the cluster, the Emulation Core is responsible for enforcing the characteristics of the topology and executes alongside the application code inside each container. On Figure 3.1 we can see an overview of NEED’s architecture.

Similar to SplayNet [SRF13], NEED does not directly emulate links and needs a way to accurately describe the topology at the end hosts. It achieves this by picking the shortest paths between every pair of hosts, which are composed of several links, and computing the resulting characteristics of the entire path. To keep the entire system updated, every host exchanges

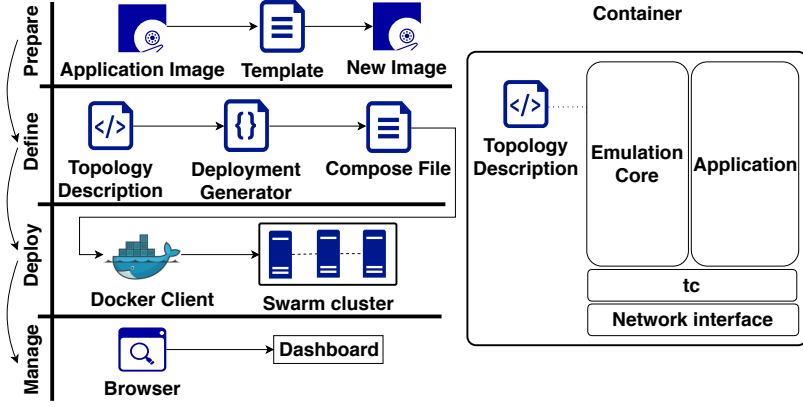


Figure 3.1: NEED architecture and workflow.

metadata regarding the outgoing bandwidth usage and then locally computes the bandwidth restrictions on flows.

NEED is able to deploy unmodified container-based systems over emulated topologies without any centralized nodes while providing emulation accuracy on par with existing state-of-the-art systems. However, the scalability of NEED is limited because every **Emulation Core** must send to and receive from every other **Emulation Core** in the system, information to maintain its distributed model. The biggest obstacle to NEED’s scalability is, therefore, the amount of information that must constantly be shared between every container.

3.10 Summary

The topology emulators presented above vary in terms of advantages and disadvantages. Systems like Mininet and IMUNES use lightweight virtualization techniques but are limited to running on a single machine. They are therefore unable to scale to large topologies with resource-intensive applications. Other systems have high entry barriers, like ModelNet, that requires dedicated clusters, or SplayNet, that requires the user to be familiar with a specific language or framework.

CrystalNet is likely the most comprehensive network emulator available to date. However, one could argue that application developers should not be concerned with network specificities such as the firmware running on which routers. Similarly, regarding the emulation of internal network devices, although network topologies and link characteristics have directly observable and measurable effects on distributed applications, internal behavior of the underlying network elements is difficult to capture from an application perspective and therefore less relevant to application developers.

To the best of our knowledge, NEED is the only system that can be used to deploy container based unmodified applications over emulated topologies without the need of any centralized node

or the need to directly emulate routers and switches, supporting a rich set of emulation features and still providing emulation accuracy on par with existing state-of-the-art systems. However, this scalability is limited to a couple thousand containers at most because after a certain point the overhead of control messages exchanged between every container starts affecting the accuracy of the emulation. The goal of this work is to address these limitations and build a large-scale and accurate network emulator.

Chapter 4

Architecture

KOLLAPS is a network emulation system that takes advantage of Docker to emulate arbitrary network topologies on a cluster of physical machines. The main standout feature of KOLLAPS is the combination of containers, for lightweight deployment of applications on a cluster, with techniques for performing point-to-point emulation in a decentralized way. This work was based on an already existing project, NEED [Nev18a]. The design employed by NEED suffers from two problems that limit its scalability. The first is the loss of the metadata UDP packets containing information with the flows. The second is the delay of the metadata packets that can be caused by heavy networking load or by the fact that windows of time between cycles of the emulation loops are not guaranteed to be constant across all instances. In this chapter we describe the architecture of KOLLAPS and explain how the new design addresses NEED’s limitations.

4.1 Overall architecture

The major challenge to NEED’s scalability is the huge amounts of exchanged metadata between every container. Further developments by the original author over the initial prototype of NEED, have made all **Emulation Managers** accessible from a separate container, as opposed to strictly running alongside applications in their respective container. A special container, named **Master Container**, now has access to all emulation cores running on a given physical host. This makes it possible to replace the exchange of UDP messages between instances running on the same physical host with shared process memory.

A general overview of the KOLLAPS architecture can be seen in Figure 4.1. KOLLAPS is made up of six main components: the **Deployment Generator**, the **Bootstrapper**, the **Master Container**, the **Emulation Manager**, the **tc Abstraction Layer** and the **Dashboard**.

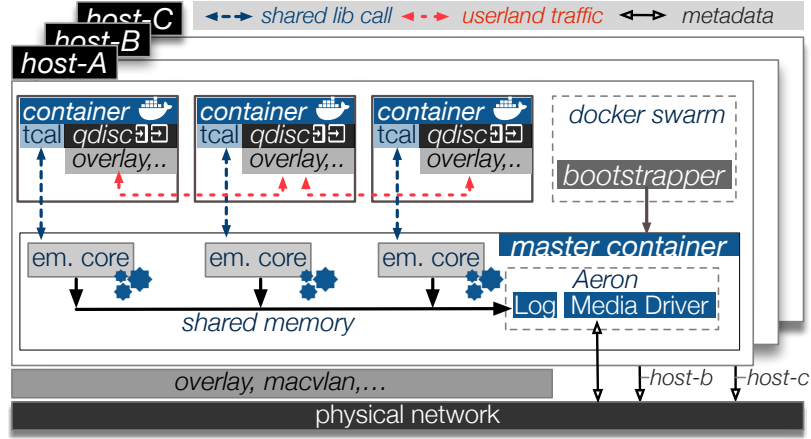


Figure 4.1: KOLLAPS architecture.

4.2 Deployment Generator

The **Deployment Generator** is a Python script that takes as input an XML file containing a topology description, builds a graph structure and writes to the standard output either a Docker Swarm Compose file or a Kubernetes Manifest file, depending on the desired orchestrator. The specification for the XML file is specific to KOLLAPS and is inspired by the one used in ModelNet. It was developed in order to include the parameters to emulate in the description of the network graph, as well as taking into consideration the particularities of deploying applications as Docker containers. This method was chosen over using the Docker API to immediately deploy the experiment on a cluster because many real applications require further customization of the Docker Compose or Kubernetes Manifest files.

4.3 Bootstrapper and Master Container

In order for an application running inside a Docker container to use `tc` (as we do via the TCAL), it must be executed with the `CAP_NET_ADMIN` capability [doc19a]. Although Docker allows executing applications in standalone containers with user-specified capabilities, this feature is currently unavailable in Docker Swarm.

We circumvent this limitation as follows. We deploy a bootstrapping container (the **Bootstrapper**) on every Swarm node. The job of the **Bootstrapper** is to launch, on that machine, a special container (the **Master Container**), itself outside of the Swarm. It shares the Pid namespace with the host and has elevated privileges. This new container has access to the local Docker daemon and monitors the local creation of new containers. Upon the creation of a new KOLLAPS container, the **Master Container** requests the Docker daemon to launch the appropriate KOLLAPS process (*e.g.*, the **Emulation Manager**, **Dashboard**) within the same network namespace

of the starting container. We expect future releases of Docker Swarm to allow for a simplified mechanism. In the case of Kubernetes, such restrictions do not hold and the **Master Container** can be started immediately and operate by itself, without the need for the **Bootstrapper**.

Having this separate container that launches all emulation related processes has two main advantages. First, usability wise, application container images do not need to be changed to accommodate KOLLAPS; Second, KOLLAPS processes are able to share the same file system, which we leverage to exchange information between processes through shared memory, as opposed to using strictly the network;

4.4 Emulation Manager

The **Emulation Manager** executes in parallel with the application being tested and is the main component of KOLLAPS. It takes as input the network topology description and is responsible for parsing the topology description, creating all the local **tc** infrastructure (required to enforce the network limitations) and for cooperating with all the other **Emulation Managers** in order to maintain an accurate distributed emulation.

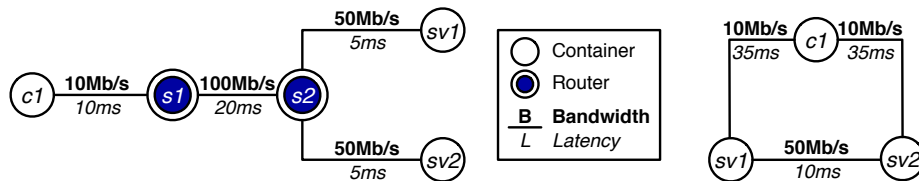


Figure 4.2: Example of a topology and an equivalent collapsed topology.

Bridges (*i.e.* routers or switches) do not really exist in the emulation model so the **Emulation Manager** executes only on **Docker services** [docb] corresponding to application containers. The specified topology is collapsed, into a different topology where the **services** are directly connected to all the other reachable **services** from the original topology. An example of this topology collapsing can be seen in Figure 4.2. However, the properties of the original topology still need to be preserved.

The **Emulation Manager** first builds the network graph structure from the XML topology description file. Second, using the Docker Swarm name resolution system, it resolves the names of all **services** to obtain their IP addresses and finds the service instance for which it is responsible. Third, it executes the Dijkstra algorithm to find all shortest paths between the instance it is responsible for and all the other reachable **services**. Fourth, following these paths, the **Emulation Manager** calculates the properties of the collapsed topology.

$$Latency(\mathcal{P}) = \sum_{i=1}^n Latency(l_i) \quad (4.1)$$

$$Jitter(\mathcal{P}) = \sqrt{\sum_{i=1}^n Jitter(l_i)^2} \quad (4.2)$$

$$Loss(\mathcal{P}) = 1.0 - \prod_{i=1}^n (1.0 - Loss(l_i)) \quad (4.3)$$

$$\max Bandwidth(\mathcal{P}) = \min_{\forall l_i \in \mathcal{P}} Bandwidth(l_i) \quad (4.4)$$

Fifth, the properties of the paths are then applied over the end-to-end links using the `tc abstraction layer`. Finally, after all the steps above, emulation is ready to begin and the `Emulation Manager` moves over to the emulation loop.

The loop starts by clearing all flows older than `max_flow_age`. This age represents the number of loops that this information was used in. Then we query the `tc abstraction layer` for the amount of data sent to other service instances. That information is used to calculate how much bandwidth is currently being used on each path, and therefore also on each link of the original topology. This information is referred to as the emulation metadata. The metadata is saved locally and then offloaded to a side process so that it can be shared with all other `Emulation Managers`. We then sleep for a period of time referred to as `pool_period` to allow for the metadata from other instances to arrive and be processed.

In `NEED`, `Emulation Managers` would use Python UDP sockets to send and receive metadata to and from every other `Emulation Manager` in the system. `KOLLAPS` replaces this Python communication processes with calls to a library that interacts with Aeron [aer19]. Aeron is an OSI layer 4 Transport for message-oriented streams that implements reliable channels over UDP and IPC. Later on, in Section 5.4, we describe how Aeron is set up in `KOLLAPS` and how the `Emulation Manager` interacts with it.

After the steps above, the `Emulation Managers` uses all the collected metadata to decide on new bandwidth limits for each path. These calculations are also offloaded to another process, which computes the new bandwidth limits according to the RTT Aware Min-Max model explained in Section 2.

4.5 tc abstraction layer (TCAL)

The `tc abstraction layer` is implemented as a library, written in C, and provides a concise high-level API to setup initial networking conditions, retrieve bandwidth usage, and modify the maximum available bandwidth on paths. The API for this library can be seen in Listing 4.1.

The `init` function initializes the `tc` infrastructure. It receives as an argument the size of the packet queue to be created. The `initDestination` function sets up the `tc` infrastructure for enforcing delay, packet loss and bandwidth throttling on all traffic directed to the specified IP address. The `changeBandwidth` function modifies the `tc` infrastructure changing the maximum allowed bandwidth to the specified IP address. The `updateUsage` function takes a snapshot of the number of bytes sent to each previously setup destination. The `queryUsage` function retrieves this information for a specific destination. Finally the `tearDown` function destroys all the created `tc` infrastructure.

Listing 4.1: API for the `tc abstraction layer`.

```
void init(int txqueuelen);
void initDestination(unsigned int ip, int bandwidth, int latency,
                    float jitter, float packetLoss);
void changeBandwidth(unsigned int ip, int bandwidth);
void updateUsage();
unsigned long queryUsage(unsigned int ip);
void tearDown();
```


4.6 Dashboard

The `Dashboard` is a web application, made available through HTTP, that provides a GUI for starting and shutting down experiments, as well as monitoring information about the deployed experiment. It provides a graphical representation of the topology, displays the current status of all service instances in the experiment, and shows in real time the currently active flows in the topology.

4.7 Summary

KOLLAPS is composed of several interconnected systems. The topology specification and the deployment generators are designed to integrate seamlessly with Docker and Kubernetes workflows, and to be easily adapted to work with other container orchestration systems. The `tc abstraction layer` is designed to be a simple interface for enforcing point-to-point network

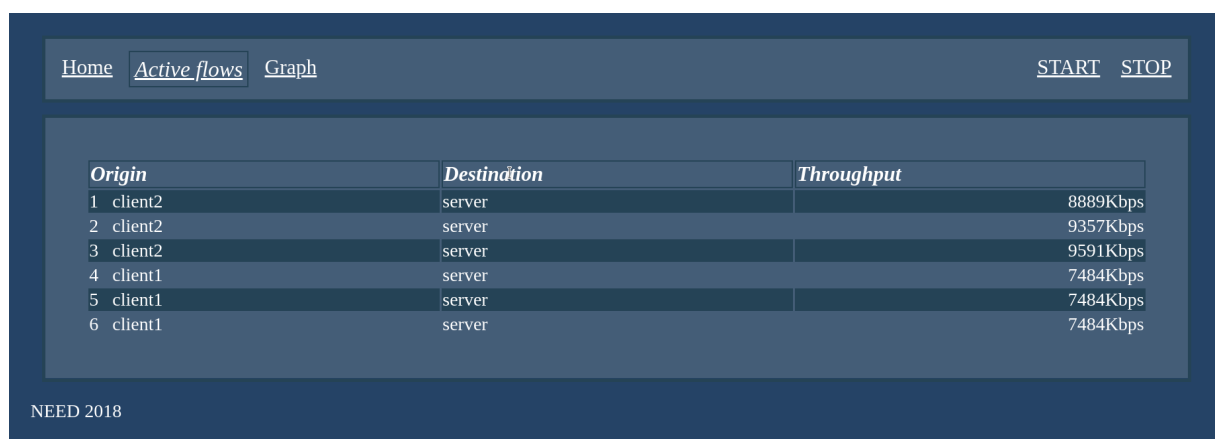
characteristics. The **Emulation Manager** was designed to operate in a fully decentralized fashion, and implement the logic necessary to perform the emulation with a collapsed topology, that is equivalent to the original one. Finally, a library was developed to allow communication between **Emulation Managers** and **Aeron Media Drivers** allowing the significant reduction of metadata usage.



The screenshot shows the 'Home' view of a dashboard. At the top, there are navigation links: 'Home' (active), 'Active flows', and 'Graph'. On the right, there are 'START' and 'STOP' buttons. Below the navigation bar is a table with three columns: 'Hostname', 'Address', and 'Status'. The table contains six rows of data. At the bottom left, the text 'NEED 2018' is visible.

| | <i>Hostname</i> | <i>Address</i> | <i>Status</i> |
|---|-----------------|----------------|---------------|
| 1 | client1.0 | 10.1.0.7 | Ready |
| 2 | client2.0 | 10.1.0.8 | Ready |
| 3 | client3.0 | 10.1.0.9 | Ready |
| 4 | server.0 | 10.1.0.10 | Ready |
| 5 | server.1 | 10.1.0.11 | Ready |
| 6 | server.2 | 10.1.0.12 | Ready |

NEED 2018



The screenshot shows the 'Active flows' view of the dashboard. At the top, there are navigation links: 'Home', 'Active flows' (active), and 'Graph'. On the right, there are 'START' and 'STOP' buttons. Below the navigation bar is a table with three columns: 'Origin', 'Destination', and 'Throughput'. The table contains six rows of data. At the bottom left, the text 'NEED 2018' is visible.

| | <i>Origin</i> | <i>Destination</i> | <i>Throughput</i> |
|---|---------------|--------------------|-------------------|
| 1 | client2 | server | 8889Kbps |
| 2 | client2 | server | 9357Kbps |
| 3 | client2 | server | 9591Kbps |
| 4 | client1 | server | 7484Kbps |
| 5 | client1 | server | 7484Kbps |
| 6 | client1 | server | 7484Kbps |

NEED 2018

Figure 4.3: Example of the information reported by the Dashboard.

Chapter 5

Implementation

In this chapter, we discuss the implementation of the various components of KOLLAPS. We go into detail on how the various components were implemented and discuss the reasons behind the decisions that were taken.

5.1 Deployment Generator

The **Deployment Generator** takes a topology description and builds a compose file for a given orchestrator (currently Docker Swarm and Kubernetes are supported). This method was preferred over immediately deploying experiments on a cluster with the Docker API because many real applications require further customization of the compose files. The **Deployment Generator** parses the XML file with the description of the topology and builds a graph structure. There is one Python class for every supported orchestrator. After the graph structure is built, one of these classes is called to generate the appropriate YAML compose file. An alternative would be to translate the XML directly to YAML. However the current approach has two advantages. First, it easily supports other Docker orchestration systems by writing code to generate new deployment descriptions. Second, it allows for customization of the emulation environment through arguments that add environment variables into specific containers.

One example of this customization are the environment variables passed to the **Master Container**. The `KOLLAPS_ORCHESTRATOR` lets the **Master Container** know what Python client to use in order to interact with the deployed containers on the specified orchestrator. Another good example is the fact that every **Master Container** needs to know how many nodes are in the cluster. With Kubernetes, any node can be given permissions to access information about the cluster. However, with Docker Swarm, a node must be a Manager to do so. To avoid all nodes having to be Managers, we run the **Deployment Generator** on a node

with permissions to read cluster state and write onto the `NUMBER_OF_GODS` variable the number of machines in the cluster. `POOL_PERIOD` and `MAX_FLOW_AGE` can be used to configure parameters of the emulation loop. Finally, `AERON_THREADING_MODE` can be used to specify how many threads are used by Aeron.

5.2 Master Container

The **Master Container** is a privileged container that has access to the host namespace and network interfaces. It consists of a Python script that on boot reads all environment variables starting by the `NUMBER_OF_GODS`. Then every **Master Container** broadcasts a message with its IP and collects the messages received from the others. After finding the IPs for every **Master Container**, it starts broadcasting a message saying it is ready to proceed and when everyone is ready the broadcasts are stopped and the Aeron Media Driver is launched. The Aeron Media Driver is a process used for the dissemination of emulation metadata and is further explained in Section 5.4. Then, it reads the `KOLLAPS_ORCHESTRATOR` variable and creates a bootstrapper object accordingly. Each of these objects is a Python class that inherits an interface and must implement its methods with the appropriate Docker client for the orchestrator. Every service has a unique name and every experiment has a unique string identifier. The **Master Container** will look for containers that match a specific experiment identifier and on those it will inject the corresponding KOLLAPS processes. If the identifier of the service matches a supervisor, the specific KOLLAPS process will be injected. For example, if the name of the service is a combination of "dashboard" with the identifier of the experiment, the process corresponding to the **Dashboard** will be injected into that container. Any service that does not match a known KOLLAPS component is presumed to be an application container and we inject the **Emulation Manager** into it to enforce the emulation. When no containers with the expected experiment identifier are present, the **Master Container** terminates.

5.3 Emulation Manager

The **Emulation Manager** is the component that runs the algorithms required to manage the emulation. It is implemented in Python and is split into two parts: initialization and the emulation loop. The initialization step starts when the **Emulation Manager** is launched. It parses the XML topology description file and builds an internal graph data structure. This is the same code the **Deployment Generator** uses to create its own graph structure. After the graph is built, using the Docker Swarm built-in name resolution system, the **Emulation Manager** then

resolves the names of all services to obtain their IP addresses. Once the addresses of all services are known, the **Emulation Manager** must find the service instance for which it is responsible. Then, it sets up the subscriptions and publications for Aeron that will be used to collect and send information regarding bandwidth usages by the services. With the IP addresses, the **Emulation Manager** executes an implementation of Dijkstra shortest paths algorithm to find the shortest paths between the instance it is responsible for and all the other reachable service instances. In the case of ties, one of the alternative paths is picked deterministically based on the order of links on the topology description. With the shortest paths found, the **Emulation Manager** proceeds to calculate the properties of the collapsed topology using the methods described in Section 4.4. The initial properties of the paths are then applied using the `tc abstraction layer`.

The information about the links of the original topology that compose each path, as well as the original graph is kept in memory, as this information will be necessary for calculating the bandwidth congestion restrictions during emulation. As soon as the path properties are applied, emulation is ready to begin, and the **Emulation Manager** moves to the emulation loop.

The loop starts by clearing all flows older than `max_flow_age`. Then the `tc abstraction layer` is queried for the amount of data sent to other service instances. That information allows us to calculate how much bandwidth is currently being used on each path, and therefore also on each link of the original topology. We refer to this information as the emulation metadata. There is a separate thread running that is responsible for collecting metadata from other emulation core instances. The metadata is saved locally and shared with all the other **Emulation Manager** instances through this thread responsible for interacting with the Aeron library. The program then sleeps for `pool_period` milliseconds. This period, referred to as pool period, is configurable and should allow enough time to pass for metadata from other instances to arrive and be processed. Then, the **Emulation Manager** uses all the collected metadata to decide on new bandwidth limits for each path. The calculations involved in deciding new bandwidth limits are also offloaded to another parallel process. This is done to minimize the impact of lengthy calculations on large topologies on the intervals between sending metadata. To decide on new bandwidth limits for the paths, the RTT Aware Min-Max model is employed, which has been described in Section 2.3.

Although the use of Aeron prevents metadata packets from being lost, they can still suffer delays. This might cause them to arrive too late at a given instance to be considered for the RTT Aware Min-Max model. This can happen since the windows of time between cycles of the emulation loop are not guaranteed to be constant across all instances. Also, under heavy networking load it is common to observe such packet delays. We attempt to mitigate the impact

of such delays by not deleting all known flows at every loop iteration. If we receive a flow with the same origin and destination as one that is already in the known flows from the previous iteration, we update the bandwidth usage of that flow and reset its `flow_age`. During the computation of the bandwidth restrictions we increment the flow's age and at the start of every loop we delete flows that are deemed too old by the `max_flow_age`. The knowledge of flows is kept in a python dictionary with strings as keys and updating existing entries, oposed to deleting and creating new ones, has proven to lead to faster loop iterations.

5.4 Aeron

All **Emulation Managers** are now launched from within the **Master Container** and share the same file system. Therefore, a solution to reduce the amount exchanged metadata over the network comes from using shared memory. However, **Emulation Managers** must remain completely independent processes so that they can be freely moved into different namespaces, otherwise we would not be able to injected these processes into the Docker containers. For Python 3.7 and lower, shared memory is only possible between processes generated from a common parent process, which means we cannot use this functionality.

Aeron [aer19] is the system that was chosen to share metadata between **Emulation Managers**, which replaced the all-to-all broadcasts from **NEED**. It relies on a central process that coordinates the exchange of messages between every other process interacting with it. This process is the **Aeron Media Driver**, which has two different releases, one in **JAVA** and one in **C**. We chose to use the one written in **C** for performance reasons. The configuration of this process is done through environment variables that can be specified on the compose file. The `AERON_THREADING_MODE` tells the **Aeron Media Driver** how many dedicated threads it should use. With the `DEDICATED` value it will use three threads: one for receiving messages, one for sending messages and another for coordinating which messages need to be delivered to whom and which can be overwritten. The `SHARED_NETWORK` uses only two threads, where the sending and receiving of messages will be done by the same thread. Finally, with the `SHARED` value the **Aeron Media Driver** operates with one single thread. The other Aeron related environment variables are used to specify the use of the maximum memory supported by Aeron, although these can also be customized from the compose file if the machines happen to be memory constrained.

Emulation Managers interact with the **Aeron Media Driver** through a shared library implemented in **C++**. The initialization consists of setting up a subscription and publication to the local **Aeron Media Driver** via the shared memory channel, and publications to every

other **Aeron Media Driver** on other machines via UDP, using the IPs collected by the **Master Container**.

The other two available functions exposed by the library are: one for the publication of messages containing the information of the flows being used and another to request the messages not yet read.

The result is a system where there is one **Aeron Media Driver** per machine in the cluster and a process per application container (*i.e.* **Emulation Manager**) that communicates with every **Aeron Media Driver**. This way, the bandwidth used for the sharing of emulation metadata scales with the number of machines and flows, and not the number of application containers being tested, given that each instance send its information only once for each machine in the cluster, regardless of the number of instances.

5.5 tc abstraction layer

The **tc abstraction layer** is implemented as a library, written in C. Since we are now relying on Aeron to share metadata, we no longer require the qdiscs associated with the metadata packets. Therefore we remove the **prio qdisc** and the rule from the **u32** filter for placing packets in it. The **tc abstraction layer** contains a hierarchy of queuing disciplines (**qdiscs**) for enforcing the topology restrictions. For each destination, we create a **netem qdisc** [FL11] and a **htb qdisc** (hierarchical token bucket) [MD02]. The former is used to apply latency, jitter, and packet loss rate, while the latter enforces bandwidth constraints.

Outbound traffic is matched to **netem qdiscs** through **u32** universal 32bit [u3218] traffic control filters. The filter is a two-level hashtable that matches against the destination IP address of packets and directs them to their corresponding **netem qdisc**. This two-level design is due to limitations in the **u32**, which does not provide a real hashing mechanism (for speed reasons) but just a simple index in a 256 position array. With a /16 netmask this could result in several collisions, degrading performance. We map the third octet of the IP address to the first level and the fourth octet to the second level of the hashtable, achieving constant lookup times. Traffic directed to the **netem qdisc** will first be subjected to the **netem** rules to enforce latency, jitter and packet loss. When packets are dequeued from **netem**, they are immediately queued in the parent **htb qdisc** and the network restrictions are enforced.

The **tc abstraction layer** is queried and updated very frequently during each experiment, namely to retrieve bandwidth usage and enforce the dynamic properties during runtime. To minimize the overhead of these calls, we rely on **netlink** sockets [KKKS03] that communicate directly with the kernel, circumventing the need to periodically spawn a new **tc** process.

5.6 Summary

The **Deployment Generator** transforms topology descriptions written in XML into Docker Compose or Kubernetes Manifest files written in YAML, that are ready to deploy on either Docker Swarm or Kubernetes. The **Bootstrapper** and **Master Container** are responsible for overcoming limitations of the orchestrators and container models such as access to networks and file systems. We leverage this to share a file system between all containers in the same physical machine and to manipulate the network stack of the containers. The **Emulation Manager** enforces the characteristics of the emulated topology. To do so, it makes use of the **tc abstraction layer**, which is a library that interacts with the kernel to allow for modification of point-to-point network properties. **Emulation Managers** need to share bandwidth usage between them in order to compute network limitations during emulation. For this, we implemented a new library that uses Aeron, a reliable UDP and IPC messaging protocol. This new method to share metadata drastically reduces the network overhead that **NEED** suffers from. Beyond these components, other small tweaks were made to the supervisor processes of **NEED** to accommodate the new process injection by the **Master Container**.

Chapter 6

Evaluation

In this chapter we evaluated KOLLAPS through a series of micro- and macro-benchmark experiments in a cluster. Overall, our results show that:

- KOLLAPS scales linearly with the number of flows and physical hosts in the cluster;
- for a number of physical hosts it has constant cost regarding bandwidth usage and number of emulated application containers;
- emulation accuracy is comparable with other common network emulation systems such as Mininet.

We start by comparing bandwidth emulation accuracy between KOLLAPS and other systems (Section 6.1). Next, we verify that KOLLAPS emulates latency and bandwidth with precision and accuracy in simple topologies (Section 6.2). Next, we compare the amount of bandwidth that emulation metadata used by KOLLAPS and NEED for a given experiment (Section 6.3). Then we assess the scalability of KOLLAPS. First, we assess the scalability regarding bandwidth congestion scenarios (Section 6.4). And finally, we assess the scalability regarding only latency emulation by showing results from executing large scale-free topologies (Section 6.5).

Evaluation settings. Unless otherwise specified, tests were executed on a cluster composed of 4 Dell PowerEdge R630 server machines, with 64-cores Intel Xeon E5-2683v4 clocked at 2.10 GHz CPU, 128 GB of RAM and connected by a Dell S6010-ON 40 GbE switch. The Docker Engine version used was 19.03.4 and the network driver was overlay.

6.1 Link-level emulation accuracy

We begin by evaluating the accuracy of our bandwidth shaping mechanism under a simple scenario that consists of only two services connected by one link. One of the services executes

| Link BW | Kollaps | NEED | Mininet | trickle (def.) | trickle (tuned) |
|----------|-----------|-----------|----------|----------------|-----------------|
| Low: | | | | | |
| 128 Kb/s | 122 Kb/s | 122 Kb/s | 123 Kb/s | 262 Kb/s | 131 Kb/s |
| 256 Kb/s | 244 Kb/s | 245 Kb/s | 286 Kb/s | 472 Kb/s | 262 Kb/s |
| 512 Kb/s | 489 Kb/s | 490 Kb/s | 490 Kb/s | 717 Kb/s | 525 Kb/s |
| Mid: | | | | | |
| 128 Mb/s | 122 Mb/s | 122 Mb/s | 122 Mb/s | 250 Mb/s | 131 Mb/s |
| 256 Mb/s | 245 Mb/s | 245 Mb/s | 245 Mb/s | 493 Mb/s | 261 Mb/s |
| 512 Mb/s | 488 Mb/s | 487 Mb/s | 486 Mb/s | 952 Mb/s | 518 Mb/s |
| High: | | | | | |
| 1 Gb/s | 954 Mb/s | 954 Mb/s | 933 Mb/s | 1.67 Gb/s | 1.00 Gb/s |
| 2 Gb/s | 1.89 Gb/s | 1.91 Gb/s | N/A | 1.93 Gb/s | 1.97 Gb/s |
| 4 Gb/s | 3.79 Gb/s | 3.79 Gb/s | N/A | 4.12 Gb/s | 3.61 Gb/s |

Table 6.1: Study of the bandwidth shaping accuracy for different emulated link capacities and tools on a (client – server) topology.

an iPerf [ipe19] server and the other executes an iPerf client. We access accuracy across a range of different target bandwidths, and compare the results with the same experiment executed with NEED, Mininet and Trickle. On all cases the iPerf client is configured to execute for 60 seconds before terminating, the average throughput reported at the server can be seen in Table 6.1.

As we can see from these results the changes made to the `tc abstraction layer` did not affect the accuracy or precision of the bandwidth emulation. The values obtained with KOLLAPS, NEED and Mininet are very similar. This is because these systems rely on the `htb qdisc` to perform the bandwidth shaping. Mininet does not allow imposing bandwidth limits greater than 1Gb/s so we were unable to collect results for that experiment. Neither KOLLAPS nor NEED impose this restriction, and in fact accuracy is for both always maintained within 5% of error across all ranges of target bandwidths.

We also compare results against executing the same experiment with Trickle, a user space bandwidth shaper. Results using the default settings deviate significantly from the specified throughput rates. However, tuning iPerf to use smaller TCP sending buffers led to accuracy comparable with the other systems. This shows how dependent the accuracy of user space tools is on application behavior, something that is not observed with kernel space tools as they operate directly at the network packet level.

6.2 Emulation reaction time

The `pool_period` impacts the reaction time for traffic throttling, in particular when the number of competing flows varies over time. To measure this, we set up a simple 3-clients/3-servers dumbbell topology, depicted in Figure 6.1. The link connecting both switches has a maxi-

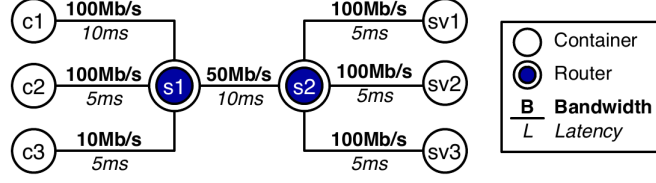


Figure 6.1: Dumbbell topology with 3 clients, 3 servers, where the links connecting the clients to switch s1 all have different properties.

mum bandwidth of 50Mbit/s . This results in connections between the clients and servers with bandwidths of 50Mbit/s , 50Mbit/s and 10Mbit/s , and RTTs of 50ms , 40ms , and 40ms , respectively. Note that clients C1 and C2 have the same bandwidth but different RTTs. Given our bandwidth sharing model, under network contention client C2 will get a higher share than client C1, inversely proportional to their respective RTTs.

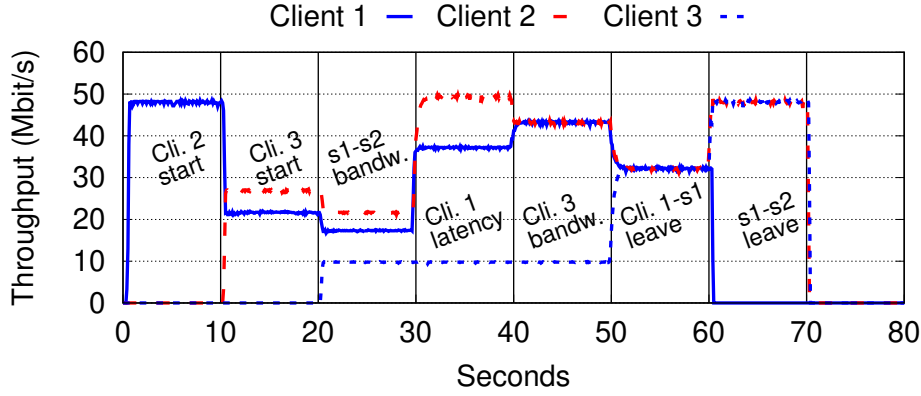


Figure 6.2: Reaction time to throttle flows that vary in time.

The experiment seen in Figure 6.2 proceeds as follows. Initially, only C1 has an active flow, and hence it uses all the available bandwidth. After 10 seconds, C2 starts and thus it will compete for bandwidth over the shared link. At this point, since C2 has a smaller RTT than C1, it gets a proportionally higher share of bandwidth, following the model we describe in Section 4.4. The reaction time, *i.e.*, how long it takes for KOLLAPS to throttle down the bandwidth available to C1 due to the competing flow from C2, is around one second. At second 20, C3 starts and quickly reaches its bandwidth limit of 10Mbit/s . The bandwidth of the other two clients gets proportionally adjusted to cope with this new competing flow also, also in around one second.

Reaction time is similarly around one second when dynamically changing properties of links or the topology itself. At second 30, we double the bandwidth of the link connecting the switches. Since C3 is already sending at full capacity, the extra bandwidth is shared between C1 and C2, again inversely proportional to their respective RTTs. At second 40, the link between C1 and

s1 is set to have latency of 5ms instead of 10ms. Since on the server side of the topology, every link is identical, that makes the properties of paths from **C1** and **C2** to any server identical, which is reflected in their bandwidth becoming equal. At second 50, the link between **C3** and **s1** has its bandwidth set to 100Mbps as well. All links from clients to **s1** are now identical, resulting in three identical flows. At second 60, we remove the link between **C1** and **s1** from the topology. In other words, the graph is not connected anymore and **C1** is no longer able to put flows on the network anymore. The freed up bandwidth is equally shared between **C2** and **C3**. Finally, at second 70, also the link connecting the switches is removed. Naturally, we do not measure any traffic anymore.

These results allow us to conclude that ongoing flows are adjusted under one second with the presence of new flows and validate our bandwidth sharing model that assigns bandwidth proportionally to the RTT.

6.3 Metadata bandwidth usage

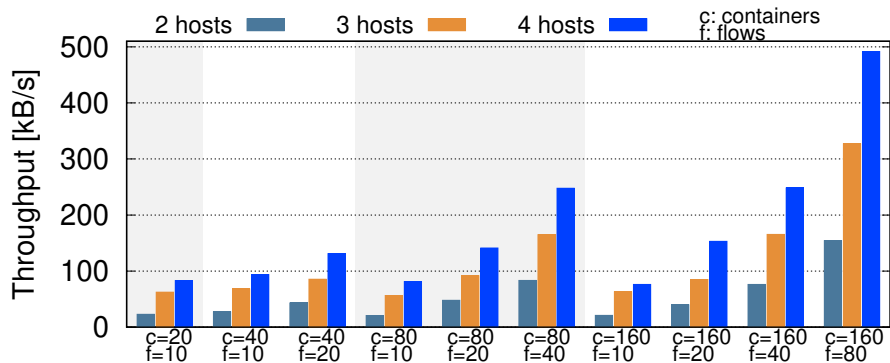


Figure 6.3: KOLLAPS metadata bandwidth with an increasing number of flows and containers, 1 source and 1 destination per flow, **F**: concurrent data flows, **C** total deployed containers.

KOLLAPS relies on metadata dissemination to accurately model and emulate bandwidth restrictions for competing flows. In this section, we study the cost of this metadata dissemination by deploying a simple dumbbell topology. We use iPerf [ipe19] to generate steady TCP traffic at 50Mbit/s , the maximum capacity of the shared link. We denote each configuration by a tuple **M/C/F** with **M** number of physical machines in the cluster, **C** total deployed containers and **F** number of concurrent data flows. Results are shown in Figure 6.3.

Every container submits the metadata to the **Aeron Media Drivers** in the system and every subscriber receives the metadata from the local **Aeron Media Driver**. For experiments running on a single machine, all messages are exchanged through shared memory and no network band-

width is used. From Figure 6.3, we can see that the bandwidth used by the metadata is not related to the amount of containers as the messages are only exchanged between **Aeron Media Drivers**. Metadata bandwidth grows with the number of flows since increasing the number of flows means that there is more information to be shared. It also grows with the number of physical machines since this means there are more **Aeron Media Drivers** to share the information with. However, metadata bandwidth does not increase with the emulated application bandwidth because the messages will still have the same size.

Note that the amount of links in the topology can impact metadata bandwidth. As we use a varying number of bytes to represent the links, metadata bandwidth is affected if the number of links is larger than 256 (2^8), practically doubling the used bandwidth required to send the UDP packets per dissemination period.

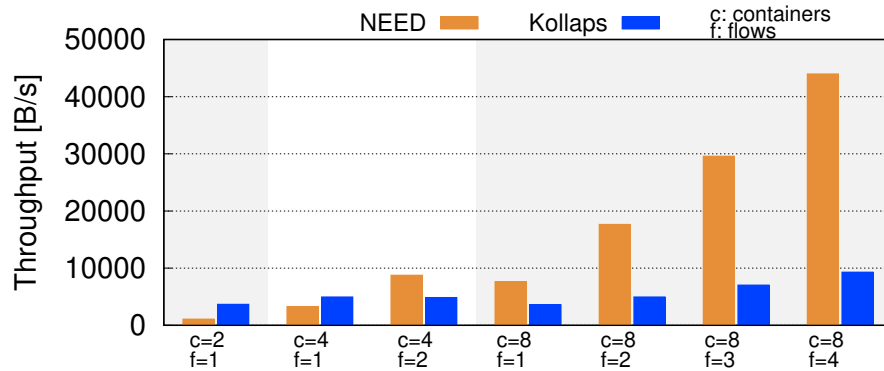


Figure 6.4: Comparison of generated metadata between KOLLAPS and NEED, 1 source and 1 destination per flow. F: concurrent data flows, C total deployed containers.

On Figure 6.4 we can see a comparison between the metadata generated by KOLLAPS and NEED for the same topologies and workload. In this experiment, we run both systems on a cluster with only two nodes. In NEED every container sends the metadata to every other container. This means the metadata circulating in the system will scale with the number of containers with $O(c^2)$, c being the number of containers. With KOLLAPS, the metadata disseminated scales with the number of nodes in the cluster (*i.e.* **Aeron Media Drivers**). In this example, with only two physical hosts, every packet of metadata travels to only one destination: the other host. All other exchanges of metadata are done through shared memory.

6.4 Scalability of bandwidth emulation accuracy

The `pool.period` affects metadata bandwidth but it can also affect, if configured incorrectly, emulation accuracy. Shorter periods increase the responsiveness of the system but also shorten

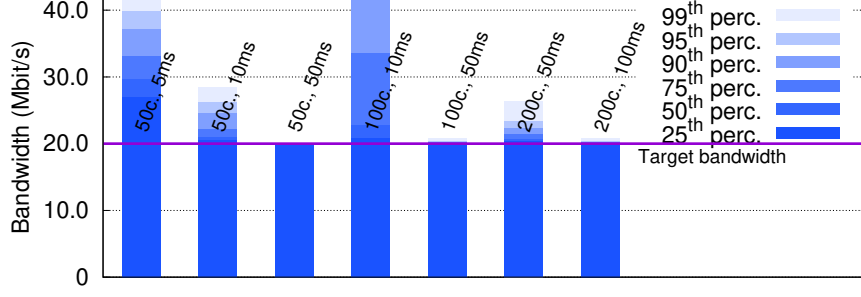


Figure 6.5: Emulation accuracy for different update periods and total number containers. The topology allows at most 20 Mbit/s per flow.

the time window available to receive and process metadata packets. We study the impact of the period in the next experiment, shown in Figure 6.5. The topology is a dumbbell with the containers evenly split across two switches (clients on one side, servers on the other). Each client runs iPerf [ipe19] without a throughput limit which will saturate the inter-switch link. This link is configured such that, for each topology, clients get a bandwidth share of 20Mbit/s .

We start by running an experiment with only 50 containers and a `pool.period` of 5ms . We can see that the containers often surpass their allowed share. As we increase the period for the next two configurations we see the bandwidths getting closer to the expected value. With a period of 50ms barely ever surpass the 20Mbit/s limit. From the subsequent configurations, we can see that increasing the number of containers decreases accuracy while increasing the `pool.period` increases accuracy. This is because increasing the `pool.period` increases the time KOLLAPS has to receive any delayed metadata and to compute the bandwidth limits, which evidently comes at the cost of reaction time. Decreasing the period is, therefore, something that should be avoided unless high-accuracy response times to the rearrangement of flows is required. A good choice for the dissemination period will depend on the number of containers and also on the resources of the underlying cluster. This parameter can be passed to the system by setting an environment variable for the `Bootstrapper`. The default 50ms dissemination period was chosen for its good balance between accuracy and reaction times. However, the `Dashboard` reports a percentage of delayed metadata upon stopping an experiment. This gives the user some notion of how accurate the experiment was so that the user can then decide whether or not to change the parameters.

6.5 Scalability of latency emulation

So far our experiments have focused on simple dumbbell topologies. In this section, experiments use scale-free topologies generated through the preferential attachment method described

in [BA99]. This method yields scale-free networks, which are representative of the characteristics of Internet topologies. The experiment consists of end-nodes sending ICMP echo requests (**ping**) to other random end-nodes for 10 minutes. We compare the obtained round-trip times (RTT) with the theoretical ones statically computed from the topology. The results are presented in Table 6.2 as a mean squared error between these two quantities. We consider three topologies, respectively with 1000 elements (666 services, 334 bridges), 2000 elements (1344 services, 656 bridges) and 4000 elements (2668 services, 1332 bridges).

| Topology size | Kollaps | NEED | Mininet | Maxinet |
|---------------|---------|--------|---------|----------|
| 1000 | 0.0235 | 0.0595 | 0.0079 | 28.0779 |
| 2000 | 0.0388 | 0.0799 | NA | 347.5303 |
| 4000 | 0.0721 | NA | NA | NA |

Table 6.2: Mean squared error exhibited on latency tests with large scale-free topologies in KOLLAPS, NEED, Mininet and Maxinet.

Results for NEED, Mininet and Maxinet were taken from NEED’s evaluation [Nev18b]. KOLLAPS, NEED and Maxinet are deployed on four machines while Mininet is deployed in a single machine as a multiple machine deployment is not supported. We observe that Mininet produces smaller errors than both KOLLAPS and NEED. We attribute this to two factors. First, the container networking in Docker introduces small yet measurable delays. Second, because KOLLAPS and NEED are running on different physical machines, there is also a small but measurable delay when packets need to traverse the physical links. Despite these two factors, the largest deviation from the theoretical RTT observed with KOLLAPS was $0.55ms$, on the third topology. Due to the limitations with Mininet, it was not possible to gather results for the larger topologies.

With these results we can observe an increase of the error with the size of the topology. Even though the largest deviation from the expected values is just $0.55ms$, increasing the topology size generates more frequent errors due to higher resource utilization. This is where we see the difference between NEED and KOLLAPS. NEED generates significantly more metadata packets that are all sent through the network, flooding the hosts’ interfaces. Reducing the metadata inundating the network allows KOLLAPS to have less frequent errors.

Chapter 7

Conclusion

As distributed systems become more complex, so has the effort required to validate them grown. However, this evaluation cannot be ignored as it is a crucial step to verify the correctness and performance of applications. Network emulation tools help developers reduce the costs of performing such validations but they all have limitations. NEED in particular combines the usage of containers and a decentralized design. By collapsing the topologies into end-to-end links that retain the high level properties of the original topology, users can focus on the applications and macro properties of the network that affect them. However, NEED has limited scalability due to the network overhead introduced by the sharing of data required to maintain the distributed model. KOLLAPS builds on NEED and reduces this overhead through the use of shared memory to share this data.

KOLLAPS can scale to hundreds of nodes while maintaining accuracy on all emulated properties. For specific scenarios where bandwidth contention does not occur, our system can scale as far as the underlying physical cluster allows. The most significant advantage of KOLLAPS is the ability to conduct *what-if* scenarios, allowing users to evaluate the performance and correctness of applications under hypothetical, fully controlled, network conditions.

7.1 Achievements

Two articles were written based on the present work. These were submitted and accepted on both SRDS 2019 (Symposium on Reliable Distributed Systems 2019) and INForum 2019 conferences. In the latter, our article was awarded an honorable mention for the best papers in the conference.

7.2 Future Work

KOLLAPS has shown to be a useful tool for distributed systems developers but there is still allot of room for improvement. Next we present some limitations of the system and discuss possible solutions.

CPU usage. The poling of metadata from the **Aeron Media Driver** is done with an infinite loop in busy waiting until there are new messages. For the most part, this does not affect other processes as we call on the thread to yield if there is nothing to be read. However, this causes CPU usage to always be close to 100% since this thread will keep being picked up by the scheduler if there is no other process to run. This could be mitigated by a more precise coordination between the emulation loop on the **Emulation Manager** and the message reading loop on the Aeron library.

Deployment time. Before experiments can be started, every **Emulation Manager** needs to fill the topology graph with every IP address. Although unnoticeable for modest topologies, for topologies with thousands of nodes this can take hours. We propose using the **Master Container** to find the IPs of the containers in the same host by inspection and then extending the protocol that it uses to discover every other **Master Container** in the cluster to share these IPs.

Bibliography

- [aer19] Aeron — low latency transport protocol. <https://medium.com/@pirogov.alexey/aeron-low-latency-transport-protocol-9493f8d504e8>, 2019.
- [ANK01] Bert Hubert Alexey N. Kuznetsov, J Hadi Salim. *PRIIO - Priority qdisc*, 2001.
- [azu19] Microsoft Azure. <https://azure.microsoft.com/en-us/>, 2019.
- [BA99] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
- [BCN10] J. Banks, J.S. Carson, and B.L. Nelson. *Discrete-event System Simulation*. Prentice Hall, 2010.
- [BHK07] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. Oversim: A flexible overlay network simulation framework. In *IEEE global internet symposium*, volume 2007, pages 79–84, 2007.
- [Boi16] Ronald F. Boisvert. Incentivizing Reproducibility. *Commun. ACM*, 59(10):5–5, September 2016.
- [BRNR15] Tomasz Buchert, Cristian Ruiz, Lucas Nussbaum, and Olivier Richard. A survey of general-purpose experiment management tools for distributed systems. *Future Generation Computer Systems*, 45:1–12, 2015.
- [CCR⁺03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.
- [CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

- [Chu] Mark Church. Understanding docker networking drivers and their use cases.
- [deb] Trafficcontrol.
- [doca] Docker.
- [docb] How services work.
- [docc] Networking overview.
- [docd] Run docker engine in swarm mode.
- [doc19a] Docker Security Capabilities. <https://docs.docker.com/engine/security/>, 2019.
- [doc19b] Docker Swarm. <https://docs.docker.com/engine/swarm/>, 2019.
- [DOSSP14] Rogério Leão Santos De Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. In *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6. IEEE, 2014.
- [ec2] Amazon ec2.
- [Eri05] Marius A Eriksen. Trickle: A userland bandwidth shaper for unix-like systems. In *USENIX Annual Technical Conference, FREENIX Track*, pages 61–70, 2005.
- [FK03] Sally Floyd and Eddie Kohler. Internet research needs better models. *ACM SIGCOMM Computer Communication Review*, 33(1):29–34, 2003.
- [FL11] Hagen Paul Fabio Ludovici. *NETEM(8)*, 2011.
- [FLH⁺00] Dino Farinacci, Tony Li, Stan Hanks, David Meyer, and Paul Traina. Generic routing encapsulation (gre). Technical report, 2000.
- [FP01] Sally Floyd and Vern Paxson. Difficulties in simulating the internet. *IEEE/ACM Transactions on Networking (ToN)*, 9(4):392–403, 2001.
- [Hau16] Michael Hausenblas. *Docker Networking and Service Discovery*. O’Reilly Media, Inc., 2016.
- [HHJ⁺12a] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In

- Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.
- [HHJ⁺12b] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264. ACM, 2012.
- [HRS⁺08] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX Annual Technical Conference*, 2008.
- [Hub01] Bert Hubert. *tc - show / manipulate traffic control settings*, 2001.
- [imu] Imunes project overview.
- [ipe19] iPerf. <https://github.com/esnet/iperf>, 2019.
- [k8s19] Kubernetes. <https://kubernetes.io/>, 2019.
- [Kel97] Frank Kelly. Charging and rate control for elastic traffic. *European transactions on Telecommunications*, 8(1):33–37, 1997.
- [KKKS03] Hormuzd M. Khosravi, Alexey Kuznetsov, Andi Kleen, and Jamal Hadi Salim. Linux Netlink as an IP Services Protocol. RFC 3549, July 2003.
- [KRLP11] Wonho Kim, Ajay Roopakalu, Katherine Y. Li, and Vivek S. Pai. Understanding and Characterizing PlanetLab Resource Usage for Federated Network Testbeds. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 515–532, New York, NY, USA, 2011. ACM.
- [LHM10] Bob Lantz, Brandon Heller, and Nick 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*, page 19. ACM, 2010.
- [lin19] Linux LXC. <https://linuxcontainers.org/>, 2019.
- [LRF09] Lorenzo Leonini, Étienne Rivière, and Pascal Felber. Splay: Distributed systems evaluation made simple. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 185–198, 2009.

- [LZP⁺17] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 599–613. ACM, 2017.
- [MAB⁺08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.
- [man18] *namespaces - overview of Linux namespaces*, 2018.
- [MD02] Bert Hubert Martin Devera. *HTB - Hierarchy Token Bucket*, 2002.
- [MDD⁺14] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T Sridhar, Mike Bursell, and Chris Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. Technical report, 2014.
- [Mer] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment.
- [Mer14] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment, 2014.
- [MR99] Laurent Massoulié and James Roberts. Bandwidth sharing: objectives and algorithms. In *INFOCOM’99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1395–1403. IEEE, 1999.
- [ms218] Crystalnet: Faithfully emulating large production networks. pages 599–613, Aug 16, 2018.
- [net] netgraph – graph based kernel networking subsystem.
- [Nev18a] João Neves. Container network topology modelling. Master’s thesis, Instituto Superior Técnico, UTL, Lisbon, Portugal, October 2018.
- [Nev18b] João Neves. Need : Container-based decentralized topology emulation. 2018.

- [NRZ⁺15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, March 2015.
- [Pen11] Roger D Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011.
- [PF97] Vern Paxson and Sally Floyd. Why we don’t know how to simulate the internet. In *In Proceedings of the 1997 Winter Simulation Conference*, pages 1037–1044, 1997.
- [Que15] Alina Quereilhac. *Une approche générique pour l’automatisation des expériences sur les réseaux informatiques*. PhD thesis, Nice, 2015.
- [RET14] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.
- [RH10] George F Riley and Thomas R Henderson. The ns-3 network simulator. In *Modeling and tools for network simulation*, pages 15–34. Springer, 2010.
- [Riz97] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.
- [SRF13] Valerio Schiavoni, Etienne Rivière, and Pascal Felber. Splaynet: Distributed user-space topology emulation. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 62–81. Springer, 2013.
- [u3201] *u32 - universal 32bit traffic control filter*, 2001.
- [u3218] u32 Universal Identifiers. <http://man7.org/linux/man-pages/man8/tc-u32.8.html>, 2018.
- [VGVEY09a] Kashi Venkatesh Vishwanath, Diwaker Gupta, Amin Vahdat, and Ken Yocum. ModelNet: Towards a datacenter emulation environment. In *IEEE P2P’09 - 9th International Conference on Peer-to-Peer Computing*, number May, pages 81–82, 2009.
- [VGVEY09b] Kashi Venkatesh Vishwanath, Diwaker Gupta, Amin Vahdat, and Ken Yocum. Modelnet: Towards a datacenter emulation environment. In *Peer-to-Peer Computing, 2009. P2P’09. IEEE Ninth International Conference on*, pages 81–82. IEEE, 2009.

- [VYW⁺02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 36(SI):271–284, 2002.
- [WDS14a] Philip Wette, Martin Dräxler, and Arne Schwabe. MaxiNet: Distributed emulation of software-defined networks. *2014 IFIP Networking Conference, IFIP Networking 2014*, 2014.
- [WDS⁺14b] Philip Wette, Martin Draxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl. Maxinet: Distributed emulation of software-defined networks. In *Networking Conference, 2014 IFIP*, pages 1–9. IEEE, 2014.
- [ZM04] Marko Zec and Miljenko Mikuc. Operating system support for integrated network emulation in imunes. In *Workshop on Operating System and Architectural Support for the on demand IT Infrastructure (1; 2004)*, 2004.

Appendix A

Kollaps topology example

Listing A.1: Example of KOLLAPS topology description language. This description corresponds to the topology on Figure 6.1.

```
1 <experiment boot="kollaps:1.0">
2   <services>
3     <service name="client1" image="warpenguin.no-ip.org/alpineclient:1.0" command=["'server ','0','0']"/>
4     <service name="client2" image="warpenguin.no-ip.org/alpineclient:1.0" command=["'server ','0','0']"/>
5     <service name="client3" image="warpenguin.no-ip.org/alpineclient:1.0" command=["'server ','0','0']"/>
6     <service name="server" image="warpenguin.no-ip.org/alpineserver:1.0" share="false"/>
7   </services>
8   <bridges>
9     <bridge name="s1"/>
10    <bridge name="s2"/>
11  </bridges>
12  <links>
13    <link origin="client1" dest="s1" latency="10" upload="100Mbps" download="100Mbps" network="test_overlay"/>
14    <link origin="client2" dest="s1" latency="5" upload="100Mbps" download="100Mbps" network="test_overlay"/>
15    <link origin="client3" dest="s1" latency="5" upload="10Mbps" download="10Mbps" network="test_overlay"/>
16    <link origin="s1" dest="s2" latency="10" upload="50Mbps" download="50Mbps" network="test_overlay"/>
17    <link origin="s2" dest="server" latency="5" upload="100Mbps" download="100Mbps" network="test_overlay"/>
18  </links>
19  <dynamic>
20    <schedule name="client1" time="0.0" action="join"/>
21    <schedule name="client2" time="0.0" action="join"/>
22    <schedule name="client3" time="0.0" action="join"/>
23    <schedule name="server" time="0.0" action="join" amount="3"/>
24
25    <schedule name="client1" time="120.0" action="leave"/>
26    <schedule name="client2" time="120.0" action="leave"/>
27    <schedule name="client3" time="120.0" action="leave"/>
28    <schedule name="server" time="120.0" action="leave" amount="3"/>
29  </dynamic>
30 </experiment>
```

