TÉCNICO LISBOA

Scalable Network Emulation

Sebastião Santos Boavida Amaro

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. Miguel Ângelo Marques de Matos
Dr. Shady Issa

Examination Committee

Chairperson: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. Fernando Manuel Valente Ramos

October 2021

Acknowledgments

On a first note, I would like to thank Professors Miguel Matos and Shady Issa for the chance to work on this project, and for all the time invested in developing this work.

Secondly, I would like to thank Professor Valerio Schiavoni from the University of Neuchâtel, for his time and insights, and for making available the cluster where this work was developed and evaluated.

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

Um dos principais problemas ao avaliar o desempenho de sistemas distribuídos de larga escala é a constante variabilidade das condições de rede. Esta constante variação de um elemento não possível de ser controlado, causa uma frequente diferença entre o resultado da avaliação destes sistemas. Apresenta-se então a necessidade de conseguir emular correctamente o estado da rede para conseguir avaliações reproduzíveis. Existem actualmente várias limitações com o estado da arte em termos da emulação de rede, grande parte delas devido ao facto de os emuladores tentarem emular totalmente o estado da rede. O Kollaps é um emulador de rede descentralizado que não envolve gerir as camadas de transporte da rede, e portanto é capaz de emular sistemas que envolvam várias máquinas. Contudo o Kollaps, sofre de várias limitações, uso elevado do CPU, uma dificuldade a escalar para múltiplas máquinas e não pode ser introduzido em sistemas que já estão em funcionamento. Esta tese propõe o Kollaps 2.0, uma nova versão do Kollaps que procura resolver as limitações da primeira versão. Apresentamos então os novos mecanismos que o Kollaps 2.0 implementa. E uma avaliação com micro e macro benchmarks que demonstram como o Kollaps 2.0 é uma melhoria em relação à sua primeira versão. Nesta avaliação fomos capazes de demonstrar que o uso do CPU é reduzido substancialmente em topologias de largas e pequenas escala mantendo ao mesmo tempo uma emulação precisa. Mostramos também que o Kollaps 2.0 é agora capaz de ser introduzido em sistemas em funcionamento, mantendo a sua capacidade de realizar uma emulação precisa.

Palavras-chave: Sistemas Distribuídos, Reproducibilidade, Contentores, Emulação de rede

Abstract

One of the main problems of measuring the performance of large scale distributed systems is the ever-changing state of the network. This ever-changing uncontrollable component of the environment causes results from experimentation to differ from the expectation many times. There is a need then for a way to precisely emulate the network state. With a stable network state, we can now obtain reproducible results. There are many problems with the current state of the art approaches when it comes to network emulation. Most of them come from trying to completely emulate the state of the network. Kollaps is a decentralized emulator that does not involve managing the application and transport protocol layers. And therefore can emulate systems spread across several distinct physical machines. However, Kollaps still suffers from several limitations such as high resource usage, difficulty to scale to a large number of physical nodes, and lack of support for bare metal deployments. This thesis proposes Kollaps 2.0, a new iteration of Kollaps, which improves upon the limitations of its first version. We present an overview of the new mechanisms. And finally, an evaluation with micro and macro benchmarks showing how Kollaps 2.0 improved upon Kollaps. In this evaluation, we were able to show that the CPU usage is substantially reduced in both large and small scale topologies while maintaining an accurate emulation. We also demonstrate that bare metal deployments can now use Kollaps 2.0 to emulate network states while maintaining an accurate emulation.

Keywords: Distributed Systems, Reproducibility, Network Emulation, Containers

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today's large-scale distributed systems have lots of different components, libraries frameworks, system dependencies, etc. When combining this complexity and heterogeneity with uncontrollable environment aspects, such as hardware resources and network variability, developing, debugging, and evaluating distributed systems becomes an increasingly difficult task. Let us take as an example a system developer wanting to change a specific component of the system. He cannot simply deploy the change and hope for the best. First, an experiment has to be built and tested, and then maybe it will be deployed. However, testing with uncontrollable variables such as network latency or packet loss causes assessing the impact of such changes to be extremely difficult. There is an obvious need for tools that provide reproducible experimentation of these large-scale distributed systems.

With the introduction of container technology such as Docker [1] and its orchestration system Docker Swarm [2], there is a way to manage the heterogeneity of the controllable components. But the problem related to the uncontrollable ones such as the network remains. Imagine that a system administrator wants to move his deployment from one geographical place to another, basically introducing the system to a new network without experimentation, he cannot know how the system will react to the new network conditions. Container technology mitigates the problem by providing a simple way to deploy these complex heterogeneous systems. Even so, how can an experimental result be attributed to a system change and not a specific network state? Was a good result just a lucky run because the network is less congested? Or was it a bad result due to packet loss caused by a high load of the network? This constant variability removes the reproducibility that system developers want when experimenting. Therefore to accurately interpret experimental results, we must control the network properties.

A way to try to achieve this reproducibility is the use of research testbeds. Testbeds are an environment made up of dedicated network hardware such as PlanetLab [3] or EmuLab [4],

and give researchers a way to test their systems. But these systems are expensive and time-consuming to set up and maintain. Another problem is that they can become overloaded during system conference deadlines [5] due to their resource constraints, which can cause experiments to give different results, not achieving the intended reproducibility. Therefore there is a need for tools that provide a consistent way to recreate these conditions.

Another approach to evaluate large-scale distributed systems is the use of simulation. Simulation relies on correctly modeling the system and environment by capturing its main properties. Simulation in comparison to research testbeds has the advantage of providing full control of the system and environment therefore it achieves full reproducibility. Even though it can achieve full reproducibility, simulation suffers from a well-known set of problems. The first problem comes in the form of the large gap that exists in simulated models and real-world deployments, which can lead to unforeseen behaviors [6]. The other one is that even if the results from the simulation are correct, the real world implementation may not fully follow the simulated model.

The other possibility to try to evaluate large-scale distributed systems is to use network emulation. In network emulation, the emulated system runs in a model of the network. This model tries to accurately replicate the real-world behavior by modeling the state of the network topology together with its network elements, including switches, routers, and their internal behavior.

The current state-of-the-art suffers from different limitations. Mininet [7] is limited to the use of a single host, and therefore cannot accurately emulate large-scale geo-replicated distributed systems. Maxinet [8] which expands upon Mininet, uses multiple Mininet instances to provide emulations with several hosts. However, it scales poorly due to its approach to workload management. Modelnet [9] introduces a separation between application nodes and nodes responsible for maintaining the emulation accuracy. But it still suffers from the same scalability limitations due to the centralization of the nodes responsible for managing the emulation. SplayNet [10] differs from these previous systems, introducing a decentralized network emulator. However, it is limited to the Splay [11] framework and the use of the Lua programming language. Another angle for networking evaluation is the control plane, which is responsible for routing packets. CrystalNet [12] accurately emulates it but cannot emulate the data-plane, and therefore it cannot emulate network properties such as bandwidth, latency. NEeaS [13] is an innovative cloud-based network emulation platform that aims at providing users with Network Emulation as a Service (NEaaS). However, it does not present results that confirm their platform works at large scale scenarios, and as a cloud-based platform, there are costs involved. Given all these limitations, we can conclude to the best of our knowledge that they are not suitable to systematically reproduce

the evaluation of large scale distributed systems.

Kollaps [14] is a fully decentralized emulator. It emulates a network topology on containers, is agnostic to application language and transport protocol, and can scale to thousands of application nodes while maintaining an accuracy similar to the previously mentioned centralized solutions. And to our knowledge is the most suitable solution for network emulation. However, it suffers from multiple limitations, for instance. (i) high CPU usage due to the communication mechanism between components. (ii) high CPU usage in large scale deployments due to the retrieval of information from the kernel. (iii) the model of sharing metadata causes an excessive amount of metadata to be shared between Kollaps components (iv) bare metal deployments are not supported, which could allow for use cases of applications that do not run in containers.

In this thesis, we propose Kollaps 2.0 as an improvement to Kollaps by addressing its main limitations. The ideas behind our solutions to solve the main ones were as follows. (i) to lower CPU usage, we will replace the existing mechanism of communication (Aeron [15]) with a new one. (ii) use eBPF [16] to provide a new way to retrieve information from the kernel. (iii) solve the problem of excessive amounts of metadata circulating in long-lived flows by moving to a new data dissemination model. (iv) introduce bare metal deployments by allowing the introduction of Kollaps to an already running system, and to integrate it with the already running network to do accurate emulation.

1.1 Contributions

Our main contributions are:

- Substantially lowered the CPU usage of Kollaps in small scale topologies by removing the existing mechanism of communication (Aeron [15]).
- In large scale topologies we substantially lowered the CPU usage of Kollaps by replacing the way Kollaps retrieved information from the kernel.
- The amount of metadata shared in small scale topologies is significantly less by introducing a new model of metadata dissemination.
- Introduced the support for bare metal deployments while keeping the ability to provide an accurate emulation.

1.2 Thesis Outline

The remainder of the document is structured as follows.

Chapter 2, presents background on the main technologies that Kollaps 2.0 uses. Chapter 3, describes other state-of-the-art network emulation systems. Chapter 4, explains our improvements to Kollaps 2.0 and how they addressed the limitations. Chapter 5, presents an evaluation of Kollaps 2.0 shows results from experiments and discusses the results and how they tackled the limitations. Chapter 6, concludes the document and discusses future work.

Chapter 2

Background

2.1 Summary

In this Section, we cover the key technologies used by Kollaps 2.0 to run and maintain an accurate emulation. First, we cover the possible ways to emulate applications. Second, we present how Docker provides networking between containers. Third, we give an introduction to Linux's traffic control [17] and how it provides a way to manipulate network properties. Finally, we will go over eBPF [16] and how it provides a way to monitor the network.

2.2 Emulating Applications

Network emulators must find a way to correctly and accurately emulate the network and at the same time provide a way to run application code. Several mechanisms for running the application code exist we can run on bare metal, use virtual machines or containers. Running on bare metal consists of injecting the emulator code in the machines running the application's code and having the emulation change the network properties of the machines. Virtual Machines (VMs) are software applications that provide the illusion that the application code runs on an isolated physical machine. Typically, they have a guest Operating System (OS) that applications run on top of [18]. Finally, Containers are software packages that contain everything necessary to run user software.

Docker [1] is a widely used container platform that leverages lightweight virtualization techniques provided by the Linux kernel, such as kernel namespaces used to provide processes the ability to run in an isolated environment. Another advantage of Docker is its orchestration system Docker Swarm [2], or alternatives such as Kubernetes [19]. That allows users to deploy a network of containers on multiple physical machines. Kollaps supports both of the previously mentioned orchestration systems.

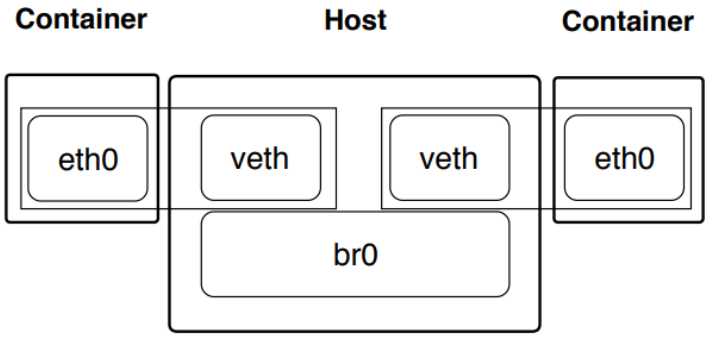

Figure 2.1: Illustration of the bridge driver

When comparing VMs and containers, containers have the significant advantage of providing a way to run applications on their network stack and storage without the need to build and run the entire OS. However, VMs have an advantage regarding isolation since they are fully isolated from the host since they have their kernel. Containers run on top of the host OS and therefore do not have the same isolation.

2.3 Docker Network Isolation

To provide the networking between containers Docker uses what it defines as the Container Network Model (CNM) drivers. The CNM manages the connectivity between containers and hides the diversity and complexity behind these complex network mechanisms [20].

However, there are many drivers to choose from: bridge, overlay, host, macvlan, and ipvlan. We are going to focus on overlay and bridge. First, overlay because it is swarm-scoped, meaning that it operates across an entire Swarm rather than individual hosts. With the overlay driver, there is no need for extra provisioning or components since IPAM(IP Address Management System), service discovery, multi-host connectivity, encryption, and load-balancing are all built-in. Second, the bridge driver since overlay is an extension of the bridge driver [20].

The bridger driver creates a bridge interface [21] on the host and creates a virtual ethernet pair [22] for each container that is then attached to the host's bridge (Figure 2.1). Since the bridge exists only inside a single host, this approach does not work for multi hosts deployments [23].

The overlay driver provides a way to deploy multi-host deployments. It extends the bridge driver and achieves connectivity between nodes through the use of VXLAN tunnels [24].

VXLAN tunnels encapsulate data between the endpoints of each host. The tunnels encapsulate data from layers two and above, generated by the containers into a layer three packet on

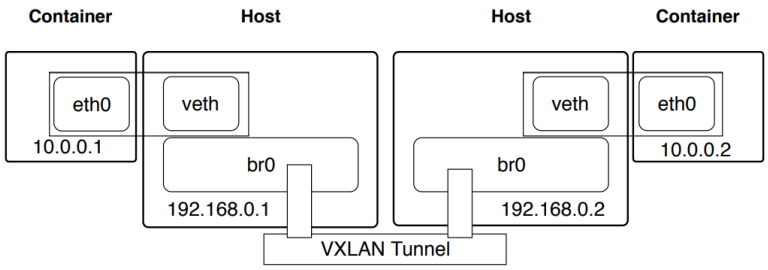

Figure 2.2: Illustration of the overlay driver

the underlying infrastructure.

VXLAN tunnels make routing through the underlying physical network infrastructure to be transparent and only requiring traffic to be encapsulated/de-encapsulated at the tunnel end-points (Figure 2.2).

2.4 Linux TC

Linux provides users with a set of traffic control (TC) functions to manage the transmission of packets. The `tc` command [17] allows users to modify the behavior of queues, referred to as queueing disciplines (`qdisc`). Before being sent to the respective network interface, Linux places packets into one of the multiple possible queues The Linux kernel offers several `qdisc` by default. Namely, Hierarchy Token Bucket (`HTB`) [25], Priority `qdisc` (`prio`) [26], and Network Emulator (`netem`) [27]. `HTB` is the one responsible for controlling outbound bandwidth on a given link. `Prio` dequeues packets in order of their priority, packets with higher priority are pulled from the queue before packets with lower priority. `Netem` allows the introduction of delay, loss, duplication, and other characteristics to outgoing packets.

TC also provides filters to match specific traffic to a `qdisc` this is necessary because a single network interface may have multiple `qdiscs` in it. The tc command is the one that provides these filters. The most important one for the context of network emulation is the use of the u32 filter [28] that allows matching traffic against any field in a network packet.

2.5 Berkeley Packet Filter

Berkeley Packet Filter(`BPF`) [29] is a highly flexible and efficient construct in the Linux kernel, that allows the execution of bytecode at various hook points in a safe manner. Systems use eBPF in many kernel subsystems, most commonly in networking, tracing, and security.

In 1992 BPF was introduced, but it was with the 3.18 kernel version that a newer version

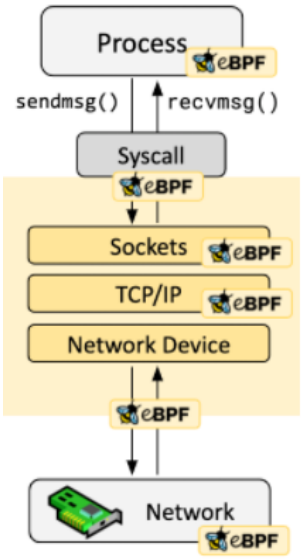

Figure 2.3: Illustration of the eBPF environment, originally from [16]s

appeared. Extended Berkeley Packet Filter(`eBPF`) [16] which will be the one we focus on this rendered the original version, now mentioned as classic BPF(cBPF) obsolete.

Even though Berkeley Packet Filter hints at a packet filtering purpose from a networking perspective, the instruction set is generic and flexible enough to provide many use cases for `eBPF`. `eBPF` does not only provide an instruction set but also offers infrastructure around it. Such as maps with efficient key/value stores, helper functions to interact with (such as getting current time), tail calls to call another `eBPF` program, a pseudo-file system to pin objects such as maps and programs, and infrastructure that allows the offloading of `eBPF` programs to a network device for example. A depiction of the eBPF environment can be seen in figure 2.3.

To load these programs into the kernel, `llvm` [30] provides an `eBPF` backend, so that tools like `clang` [31] can be used to compile C into an `eBPF` object file, which then can be loaded into the kernel.

For our work, we are going to focus on the networking and tracing kernel subsystems. On the networking subsystem, we focus on eXpress Data Path(`XDP`) [32] programs, Linux `tc`, and `socket` [33]. And on the tracing subsystems, we focus specifically on kprobes(kernel-probes) [34].

An example of an eBPF program can be seen below.

```
1 #include <uapi/linux/ptrace.h>
2 #include <net/sock.h>
3 #include <bcc/proto.h>
4 struct ipv4_key_t {
5     u32 saddr;
```

```
6       u32 daddr;
7   };
8   BPF_HASH(ipv4_send_bytes, struct ipv4_key_t);
9
10  int kprobe__tcp_sendmsg(struct pt_regs *ctx, struct sock *sk,
11      struct msghdr *msg, size_t size)
12  {
13
14      if (family == AF_INET) {
15          struct ipv4_key_t ipv4_key = {.pid = pid};
16          ipv4_key.saddr = sk->__sk_common.skc_rcv_saddr;
17          ipv4_key.daddr = sk->__sk_common.skc_daddr;
18          ipv4_send_bytes.increment(ipv4_key, size);
19      }
20      return 0;
21  }
```

This program tracks the number of bytes sent by TCP using the tracing subsystem, specifically kprobes. In line 85, we declare a map that will hold the information, in line 87 we specify the function to probe, in lines 92-95, we retrieve the information from the kernel structure and update the map.

2.5.1 Network Subsystems

eXpress Data Path

eXpress Data Path(`XDP`) provides a framework for `eBPF` that enables high-performance programmable packet processing in the Linux kernel. It runs the `eBPF` program at the earliest possible point in software, namely, at the moment the network driver receives the packet.

`XDP` combined with `eBPF` allows access to the packet data through 'direct packet access' meaning, the program holds data pointers directly in registers and loads content into registers. This access gives us a way to retrieve information from the packet or alter it if we wish to and therefore also provides a way to monitor traffic.

Linux TC

The kernel `tc` layer in the networking data path can also use `eBPF`, but this approach has many differences when compared to `XDP`. Mainly, the `eBPF` input context is a sk_buff [35] and not an

xdp_buff [36]. When the kernel receives a packet, it allocates a buffer and parses the packet to store metadata about the packet. This representation is known as the sk_buff. This structure is then exposed in the `eBPF` input context so that programs from the `tc` layer can use the metadata that is residing in the packet. The other important difference is that while in `XDP` only ingress(from the outside to the host) could trigger the execution of `eBPF`. In tc, the egress(from the host to the outside) point in the networking data patch can trigger the execution of the `eBPF` code.

`eBPF` programs that run in the `tc` layer are run from the `cls_bpf` classifier [37]. While `tc` terminology describes the `eBPF` attachment point as a classifier. The description is not completely accurate since it is not just a simple classifier, it is a fully programmable packet processor being able to read and change the packet data. It can also terminate the `tc` processing with an action, such as dropping the packet.

`cls_bpf` can be attached to different points in the traffic control subsystem. Three types of hook points exist, the ingress hook, the egress hook, and the classification hook. Our focus resides on the egress hook, which runs centrally from __dev_queue_xmit, which is the data link layer function called when any packet is to be delivered to an external destination [38].

A workflow of how to use `cls_bpf` can be seen below:

1.`clang` -O2 -target bpf -o foo.o -c foo.c

2.`tc qdisc` add dev eth0 `clsact`

3.`tc` filter add dev eth0 egress bpf da obj foo.o

In line 1, we compile foo.c with clang, which outputs foo.o our `eBPF` object file. Then we add a `cls_act qdisc` [39] to a network device, then we add the object file as a classifier to the egress side.

Sockets

Linux 2.2 for cBPF and 3.19 for `eBPF`, added the SO_ATTACH_FILTER and SO_ATTACH_BPF functions [33]. They provide a way to attach BPF or `eBPF` programs respectively to filter data on a chosen socket. As in `tc`, we handle a `sk_buff` structure and can access its content to make traffic decisions and monitor traffic.

2.5.2 Tracing Subsystems

Kernel Probes

Kprobes enables the user to dynamically break into any kernel routine and collect debugging and performance information in a non-disruptively way, users can trap almost any kernel code

address. To insert a kprobe that will run when a breakpoint hits, the user needs only to specify the address.

There are two types of probes: kprobes and kretprobes (also called return probes) [34]. We can insert a kprobe virtually anywhere in the kernel, while a return probe fires when a specified function returns.

Users can use kernel probes to attach `eBPF` programs almost anywhere in the kernel or user applications [16], allowing users to gather information regarding many kernel subsystems, networking, tracing, security, etc.

Perf

Perf [40] also called perf_events, is a tool that allows to instrument CPU performance counters, tracepoints, kprobes, and uprobes (userspace probes) it is capable of lightweight profiling. Performance counters are CPU hardware registers that count hardware events such as instructions executed, cache-misses suffered, or branches mispredicted. Tracepoints are instrumentation points placed at logical locations in code. Such as for system calls, TCP/IP events, file system operations, etc. Perf can also dynamically create tracepoints using the kprobes and uprobes frameworks for kernel and userspace dynamic tracing. eBPF can use Perf to be able to pass information from kernelspace to userspace via events as shown by many tools in bcc [41].

Chapter 3

Related Work

In Section 3.1 we describe some of the state-of-the-art network emulators and Kollaps. Next, in Section 3.2 we summarize the state-of-the-art and explain Kollaps main limitations.

3.1 State of the Art

In this Section, we will describe other network emulations tools, as well as Kollaps, and cover their main advantages and disadvantages. We also contrast these systems against the current Kollaps version.

3.1.1 MiniNet

Mininet [7] is a prototyping tool for large networks, that run on a single machine with limited resources. Mininet accomplishes this by using lightweight OS-level virtualization techniques. These techniques consist of leveraging processes and virtual Ethernet pairs in separated network namespaces, allowing the user to launch networks with gigabit bandwidth and hundreds of nodes.

Mininet has four major components: hosts, links, switches, and controllers. Hosts are simple shell processes moved into their network namespace, each containing a virtual ethernet interface. Links are a virtual ethernet pair and behave as a connection through two virtual interfaces. Switches are OpenFlow switches that provide behavior similar to a hardware switch. Controllers manage flow control for improved network management and application performance.

One of the advantages of Mininet is that it allows for interactivity which means that changes to the network, such as packet loss, changing the throughput of a connection, can be applied during the execution of the emulation.

The most significant limitation of Mininet is the lack of performance fidelity, as mentioned in Section 5 of [7]. This limitation occurs because the Linux scheduler manages the CPU resources.

Therefore we cannot assure that the scheduler will schedule a packet promptly. Scalability is another limitation in Mininet due to the O(n) linear lookup on the routing tables, causing the packet forwarding rate to drop significantly with an increase in size.

These limitations are addressed in Mininet-Hifi [42], allowing a single entity to treat a group of processes. For each group to have a fair share of CPU usage Mininet-Hifi provides CPU bandwidth limits. As proposed by the Mininet authors, Mininet-Hifi started to use `tc` (see Section 2.1.2) to configure the network properties.

Mininet's capability of running thousand of hosts in a single host comes at the expense of the limited scalability when the hosts run intensive CPU applications. Therefore Mininet is not the most suitable for large scale distributed systems.

3.1.2 MaxiNet

Maxinet [8] is a distributed emulator of software defined networks, based on Mininet, that makes emulation over several physical machines possible. Using this method, Maxinet can emulate networks with thousands of nodes on just a few physical machines.

Maxinet works as an abstraction layer connecting the deployed Mininet instances running on different workers, also providing an API to access the cluster of Mininet instances. RPC Calls provide the communication between the Maxinet and the instances. Generic Routing Encapsulation (GRE) tunnels assure the communication between hosts running on different workers. GRE tunnels are an IP encapsulation protocol, and Maxinet takes advantage of these to hide from the Mininet instances that they are running on different physical machines.

One advantage of using Maxinet is their usage of time dilation [43], making it possible to emulate bandwidth values higher than the actual physical links.

When it comes to limitations, GRE tunnels introduce a problem because they only work between switches, so all emulated nodes that connect to the same switch must be deployed on the same worker and switch. Another limitation is that the workers are assumed to have an evenly distributed workload. The system has no notion of either the performance of the workers or of the physical network. In environments where the workload is not even, overloaded machines will become the bottleneck of the environment, therefore hampering its scalability.

3.1.3 ModelNet

ModelNet [9] is an Internet emulation environment that provides users a way to deploy unmodified software prototypes in configurable Internet-like environments, and to control the network conditions and subject them to faults.

ModelNet has two main components: `edge nodes` that contain many `virtual nodes` and where any user-defined software runs, and the `core routers` which are servers equipped with large memories and modified FreeBSD kernels. They cooperate to emulate the target network and enforce the desired properties.

To correctly emulate the network, ModelNet runs in five different phases. The first one is the Create phase, where ModelNet creates a graph based on the network topology. The second is the Distillation phase that transforms the graph into a pipe topology that models the target network. Next comes the Assignment phase, mapping pieces of the previously distilled topology to ModelNet core nodes, partitioning the pipe graph to distribute the emulation load across the core nodes. The Binding phase comes after and has the job of assigning the `virtual nodes` to `edge nodes`, configuring them to run the user's software and providing the core routers with the shortest path between all the `virtual nodes`. In the end, the Run phase happens and ModelNet executes the user's software, and the emulation begins.

ModelNet has several limitations. The first one is scalability since all the packets must go through the core router, therefore, the hardware resources of the core limit the scalability of the system. Secondly, the lack of resource isolation among `virtual nodes` in the same `edge nodes` enables the possibility of interference.

Take as an example a `virtual node` sending UDP packets at the maximum rate the hardware can handle. Since UDP flows are not affected by congestion signals, the `virtual node` will never drop the packets, while another `virtual node` possibly sending TCP packets would suffer from drastic packet loss. This interference causes the `virtual node` to get less than his proper share of bandwidth.

Lastly, ModelNet assumes the existence of a perfect routing protocol that calculates the shortest path between all pairs of hosts. If a link failure happens, ModelNet assumes that the routing protocol can instantly recalculate the shortest paths.

ModelNet 2009 [44] introduced time dilation [43] to ModelNet, providing a way to emulate bandwidth larger than the physical links in the cluster allow.

3.1.4 SplayNet

Splaynet [10] is a user-space network emulation system, built on Splay [11]. It allows users to deploy several topologies on shared physical nodes, with minimal setup complexity in a fully decentralized way.

The authors choose to build the system on Splay because it allows for quick prototyping, deployment, and management, of distributed experiments. The emulation process begins with

the user defining a network topology, specifying the network properties, then submitting it to the Splay Controller along with the application code to run and any additional files required to drive the experiment. Then an all-pairs-shortest-path algorithm is run and for every two reachable nodes. It calculates the maximum bandwidth, the overall delay, and the packet loss probability. In the end, SplayNet deploys the application code on the Splay Daemons (`splayds`), responsible for creating a sandboxed environment for the applications, along with the topology information necessary to start the network emulation layer. SplayNet ensures that this allocation does not affect the accuracy of other deployed topologies by finding the minimal set of `splayds` necessary.

When compared to the previously described emulators, SplayNet differentiates itself by its decentralized architecture. It accomplishes this by having each node maintain an up to date view of ongoing data flows. This view is efficiently modeled as an n-ary tree, rooted at the local node, with inner nodes being the routers, leafs the virtual end-nodes, and edges labeled with maximum bandwidth and latency. Other advantages come in the form of how to handle intermediate network devices. SplayNet does not create processes to emulate routers and switches. Instead, it collapses the given topology and delivers the packets from one emulated host to another. These advantages make SplayNet have equal emulation accuracy in terms of bandwidth and latency accuracy to state-of-the-art while allowing the deployment of multiple experiments, each under a different network condition. The main issue with SplayNet is that it implies the user is familiar with the Lua programming language and the Splay framework.

3.1.5 NEeaS

NEeaS is a cloud-based network emulation platform aiming at providing users with Network Emulation as a Service (NEaaS). NEeaS can deploy experiments on both public and private clouds. To emulate networks of larger scale and to reduce the hardware cost of the proposed platform, NEaaS uses light-weighted virtualization technology. Namely, it uses Docker containers to supplement virtual machines (VM) to emulate networking nodes in a hybrid manner.

The architecture of NEeaS can be divided into 4 main layers as can be seen in Figure 3.1.

The Resource Virtualization Layer, the role of this layer is to abstract, virtualize, and pool all sorts of underlying hardware resources provided by the emulation platform. The three main modules include:(i) Compute Virtualization, which creates VMs that will act like a real computer. (ii) Network Virtualization, which combines hardware network resources and network functionality onto a single software-based administrative entity. (iii) Storage Virtualization presents a logical view of the physical storage resources to a host computer. These three virtualization modules construct the resource basis for the upper layer emulation functionalities.

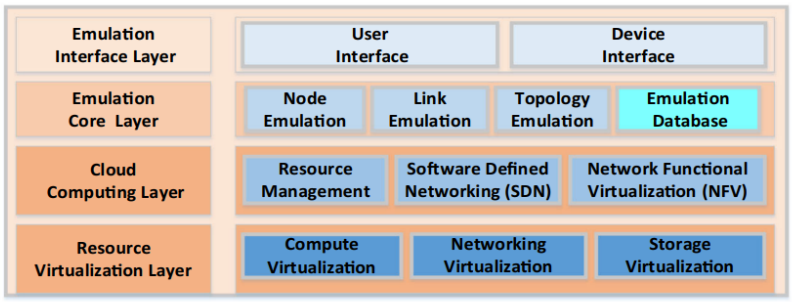

Figure 3.1: Architecture of NEeaS

The Cloud Computing Layer is a cloud operating system responsible for providing resources in different forms, according to the Emulation Core Layer requirements in real-time. This layer consists of three modules: (i) Resource Management, which allocates and frees compute, network, and storage resources to satisfy emulation needs. (ii) SDN which contains two types of entities. The SDN controller and virtual switch these entities in unison with the network virtualization module of the lowest layer, providing traffic control and flow steering functionalities. (iii) NFV which provides the node emulation, each node in the target network can be emulated by a generic VM, with NFV enhancement, which includes dedicated functionality implementations of the target node.

The Emulation Core Layer contains three modules covering node, link, and topology emulations, respectively, plus one database storing the emulation status and parameter values of all the emulated instances during the emulation process.

The Emulation Interface Layer offers two types of interfaces. (i) User Interface which is the interface presented to the emulation users. It provides a graphic interface for users to accomplish a series of typical emulation operations, such as creating nodes, links, topologies, configuration, and management of emulation scenarios, etc...(ii) Device Interface, the emulation platform supports the connections to real network nodes.

NEeaS implements the resource virtualization layer using Linux based Kernel Virtual Machine. NEeaS installs and configures the KVM in each emulation node, to virtualize and pool the compute, storage, and network resources of all the emulation nodes. NEeaS uses OpenStack [45], OpenDaylight[46] , and Open vSwitch[47] to implement the three modules of the cloud computing layer. It implements the Emulation Core Layer as a browser/server model. Users can utilize any browser to access the server to conduct their emulation tasks. It uses Linux TC to set network properties. Finally, NEeaS implements the emulation interface layer, this layer contains a user interface module and a device interface module. The user interface is the webserver front-end. Technologies, such as HTML5, jQuery, Ajax, jTopo, etc... are employed

to construct the web based user interface.

NEeaS experiments show they can accurately emulate network states in certain scenarios. But, they do not provide any results that show they can scale to thousands of nodes while maintaining an accurate emulation. Moreover, with a cloud based platform there are obvious monetary costs involved.

3.1.6 Kollaps

Kollaps[14] is a fully distributed decentralized network emulator, capable of scaling to thousands of processes, while staying accurate when compared to the state-of-the-art centralized solutions, and bare-metal deployments.

Kollaps builds on two major premises. The first one is that from an application viewpoint, the only thing that matters is the end-to-end network properties, (e.g., latency, bandwidth, packet loss, and jitter), instead of the internal state of routers and switches. Secondly, this simplified approach allows Kollaps to be fully decentralized, allowing the emulations to scale with the needs of the application. Kollaps leverages Docker containers for the lightweight deployment of applications, and the usage of Linux traffic control (`tc`) to perform the point to point emulation of the network properties.

Kollaps has six main components as seen in Figure 3.2:

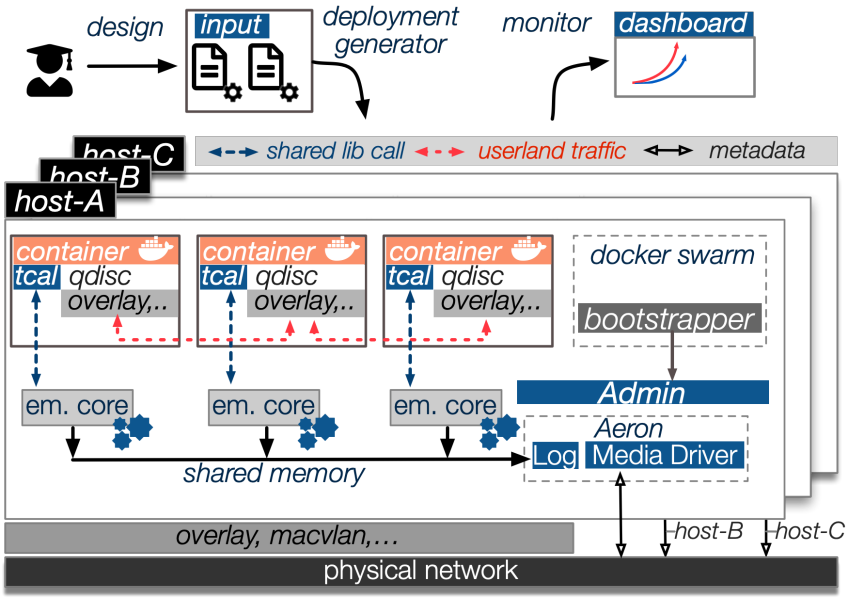

Figure 3.2: Kollaps Architecture

The Deployment Generator consists of a Python program that receives as input an XML file containing the system topology. It then builds a graph of the given topology and outputs either a Docker Swarm file or a Kubernetes [19] manifest file, depending on the user's needs. This XML

structure is specific to Kollaps and based on the one used by ModelNet [9]. Kollaps supports static topologies specifications and provides a rich set of dynamic events, such as changing the properties of links, removing and adding links, bridges, and services.

To accurately emulate network properties, Kollaps uses the `tc` command (see 2.4). But for an application to use it inside a docker container, the Docker container has to execute with the `CAPNETADMIN` [48] capability which is not available for Docker Swarm.

The task of the Bootstrapper is to deploy on that physical machine a unique container, the Administrator (Admin), outside of Docker. This container shares the pid namespace with the host and has the elevated privileges needed for the usage of tc. The Admin has access to the Docker daemon and is responsible for the local creation of new containers. When a new Kollaps container appears, the Admin will request Docker to inject the appropriate Kollaps process (emulation core, dashboard) within the same pid namespace of the initial container. The Admin although it may have added complexity to the architecture, brings two advantages. Firstly, the application container images do not need to be changed to accommodate Kollaps. Secondly, it allows the use of shared memory between Kollaps processes, due to them sharing the same file system. Therefore we do not need to use the network to share metadata which would add overhead to the emulation.

The Emulation Core (EC) is the main component of Kollaps. It runs on the Admin pid namespace and in the network namespace of the container. Kollaps does not directly emulate internal network devices or their state, instead, it collapses the topology as can be seen in Figure 3.3, on the left we can see the initial topology, and on the right, the collapsed one without internal network devices and with the links adapted to the network values. The process necessary to achieve this consists of first parsing the topology into a graph structure, then the EC calculates all the shortest paths between every two reachable containers. Each path contains multiple links, whose properties the EC uses to determine the end-to-end network properties. In the case of dynamic events, the EC precomputes the graphs offline. During the emulation, the EC changes the graph each time an event occurs. We show the formulas for each property below:

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

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

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

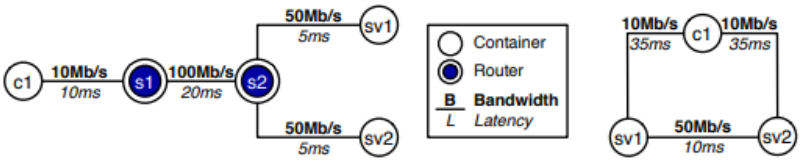

Figure 3.3: Example of a topology collapsed into an equivalent one

$$maxBandwidth(P) = \min_{\forall \in l_i} Bandwidth(l_i)$$

Latency, packet loss, and jitter (assuming a uniform distribution) are straightforward to calculate. The latency of a path is the sum of all the latencies in a link. Jitter is the variance of those latencies. Packet loss on a given link is a probability, therefore to calculate the packet loss rate of the path, we must multiply all the packet loss rates of the links in the path. The bandwidth, however, can not be calculated offline as it depends on the state of the flows that are sharing each link. In scenarios where the bandwidth required surpasses the bandwidth available on a given link, congestion will happen.

TCP manages the competition for bandwidth in a link with its congestion control mechanism. There are many TCP congestion control mechanisms such as TCP Reno [49], TCP Vegas, [50] etc. These mechanisms are responsible for adjusting the throughput. To allow all of the competing flows to get a fair share of the bandwidth.

The RTT-Aware Min-Max model [51] receives a flow and returns the share of the link, which is inversely proportional to its round-trip time. The formula for calculating the fair share of a flow f is:

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

where

$$f \in \{f1, f2, ..., f\}$$

are the active flows on a link.

The ECs are responsible for calculating the network properties of their paths. They do this by maintaining a data structure with the bandwidth of each flow in the topology. The ECs leverage the RTT model to calculate the share for each flow. The EC obtains the bandwidth of the container and sends this metadata to the other ECs via the Aeron Media Driver. The Aeron Media Driver [15] is an open-source UDP and IPC message transport protocol. There is a single instance of Aeron in every physical machine. Shared memory assures communication

between ECs on the same machine and UDP messages communication between ECs on different physical machines.

To maintain an accurate emulation, the ECs run an emulation loop. This procedure is to be periodically run and has five steps: (i) Clear the state of all local active flows. (ii) Obtain the bandwidth usage of each flow by querying the tc abstraction layer (described below). (iii) The Aeron Media Driver disseminates this information. (iv) Compute the bandwidth usage on each path and the links that reside on that path. (v) Enforce bandwidth restrictions on each path.

The TC abstraction layer (TCAL) is a library written in C and serves as a high-level API. It provides a way for the ECs to set up the initial network conditions, retrieve the bandwidth usage necessary to serve as emulation data, and modify the maximum available bandwidth on paths. The API consists of the following functions:

- `init` function initializes the tc infrastructure, receiving as an argument the size of the packet queue to be created
- `initDestination` function for a given IP sets up the infrastructure necessary to enforce the delay, packet loss, and bandwidth throttling on traffic directed to the specified IP.
- `changeBandwidth` function modifies the maximum allowed bandwidth to the given IP address.
- `updateUsage` retrieves the metadata related to how much information was sent by the container to others.
- `queryUsage` function does the same as the previously described function but for a specified IP.
- `tearDown` function destroys the previously created tc infrastructure.

The Dashboard is a web application available to users through HTTP, which provides a GUI to start and finish the experiment. While also providing a way to monitor it through its execution. It shows a graphical representation of the topology while also providing the status of each service and showing the real-time active flows in the experiment.

Kollaps simplified topology model provides emulations with accuracy comparable with other systems such as MiniNet, and due to being fully distributed, and decentralized it provides linear scalability with the number of flows and physical hosts in the cluster.

Table 3.1: Comparison of Network Emulators. P= Process, V = Virtual Machine, C = container, B = bare metal.

Name	Orchestration	Application Agnostic	Application Deployment	Efficiency
Mininet	Centralized	Yes	P	Low
Mininet-Hifi	Centralized	Yes	C	Low
Maxinet	Centralized	Yes	P	Low
ModelNet	Centralized	Yes	P	Low
SplayNet	Decentralized	No	P	N/A
NEeaS	Decentralized	Yes	P,C,V	Medium
Kollaps(2020)	Decentralized	Yes	C	Low
Kollaps 2.0	**Decentralized**	**Yes**	**C,V,B**	**High**

3.2 Discussion

Table 3.1 summarizes the comparison of the state of the art systems with Kollaps 2.0 and shows how it enhances it. Orchestration alludes to how the emulators structure themselves. Centralized emulators are more likely to have problems in terms of scalability. Application Agnostic refers to the type of application running in the system. An emulator is agnostic when any application can run in the emulator. An agnostic emulator provides significantly more use cases. Application Deployment mentions what type of use cases the emulators serve. The more different types of deployments, the more use cases they can cover. Efficiency is a qualitative measure that represents the resource usage of the network emulators. The centralized emulators, due to their nature, have low efficiency on the central components. SplayNet does not mention resource usage. NEeaS mentions that it still faces challenges against limited hardware resources. While Kollaps uses all of the available CPU.

The network emulators described in the previous Section, although having varying advantages, suffer from different problems. Mininet has scalability problems when it comes to running resource-intensive applications. MaxiNet, which extends upon Mininet, although providing Mininet with a way to run on multiple machines, it has problems when it comes to managing heterogeneous load, causing the overloaded machines to become a bottleneck. ModelNet also suffers from scalability issues due to the problem of having all packets go through the core nodes, which limits scalability to the hardware resources of these machines. SplayNet, due to its decentralized architecture, does not suffer from the same scalability issues as the previous systems but implies users are familiar with the Lua programming language and the Splay framework. NEeaS has the cost of being a cloud based network emulator, and provides no results that show it can scale to thousands of nodes while maintaining an accurate emulation.

Kollaps, to the best of our knowledge, is the only system with a decentralized architecture while using container technology, allowing Kollaps to be application-agnostic while maintaining

emulation accuracy and scalability when compared to the state of the art. However, Kollaps has several limitations, such as high CPU usage, due to the constant polling of information done by Aeron, causing the CPU usage to always be close to 100%.

Large scale deployments have a high CPU usage due to the TCAL updateUsage function, which runs every emulation loop in each EC. Update Usage makes a request to the kernel for every other container in the experiment, therefore causing quadratic scaling with the number of containers.

Another limitation is the periodic dissemination of information during the emulation, which causes the ECs to disseminate metadata, even if nothing changes. Therefore, causing a lot of metadata traffic. Periodic dissemination also causes flows, shorter than a single iteration of the emulation loop, to go undetected. Other limitations exist, such as marshaling the metadata from C to Python to C and vice-versa in every emulation loop, Kollaps does not support bare-metal deployments leaving a lot of use cases unexplored.

Chapter 4

Approach

In this Chapter, we will explain in detail the main limitations of Kollaps [14] and discuss how we solved them in Kollaps 2.0. Kollaps provides a way of doing emulation in a fully decentralized way, it uses Docker containers to provide lightweight deployment of applications and `tc` to emulate network properties. Figure 4.1 depicts the Kollaps 2.0 architecture.

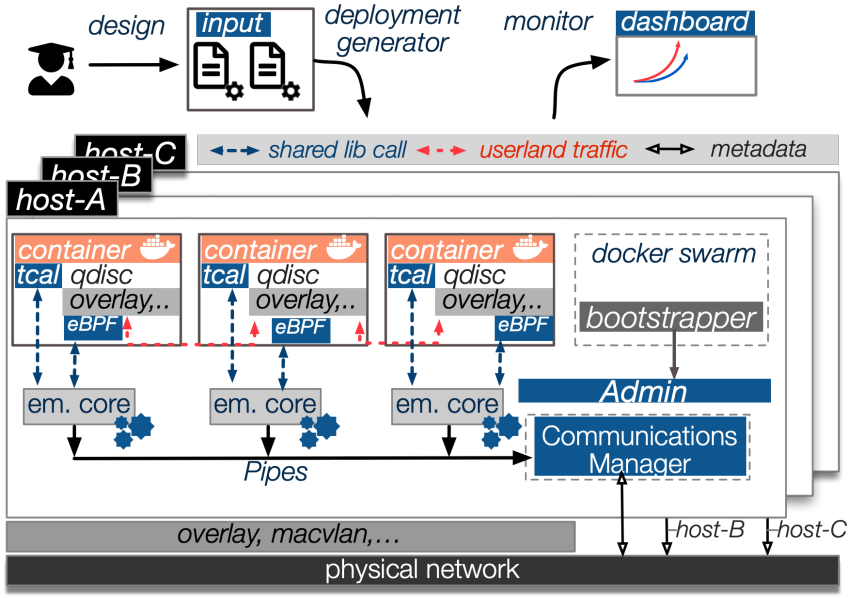

Figure 4.1: Kollaps 2.0 architecture

4.1 Metadata Dissemination Mechanisms

In Kollaps, CPU usage is always close to 100%, because of the polling of metadata from the Aeron Media Driver [15]. Kollaps polls with a thread in an infinite loop in busy waiting until new messages appear. If no other processes are running, the scheduler will always pick up the Aeron thread causing high CPU usage. Therefore Aeron is unfit with Kollaps and will be

removed. With the removal of Aeron, there was a need for new communication mechanisms, these mechanisms need to provide communication between Emulation Cores (ECs) in the same machine(inter node communication), and in different physical machines (intra node communication). To implement these mechanisms, we must understand the flow of metadata in Kollaps.

The current flow of metadata in Kollaps is complex. The flow starts in the TCAL reading from the Linux `tc` (see 3.1.6), to the EC then, to Aeron, then to all the other ECs.

Figure 4.2 shows a depiction of the flow of metadata in Kollaps.

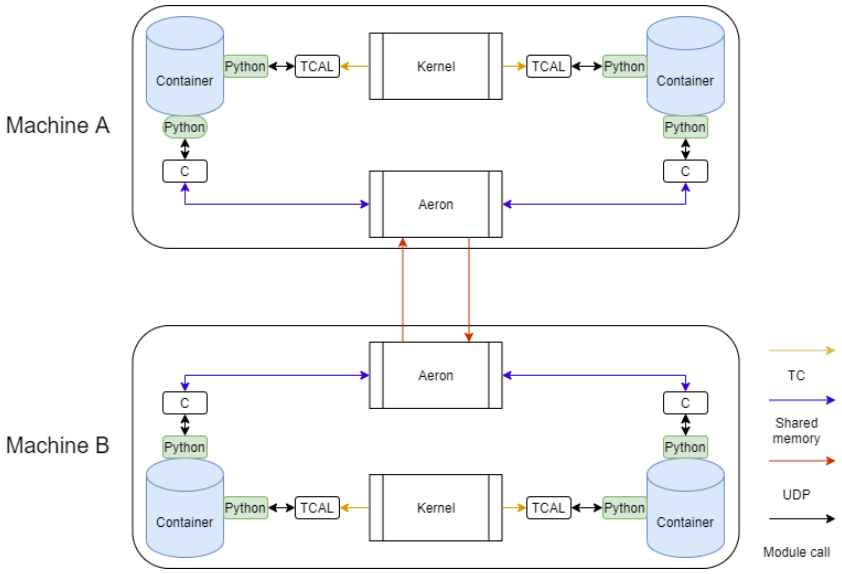

Figure 4.2: The flow of metadata in Kollaps.

In the first step, conversion from C to Python is done, then to C again, specifically through shared memory for ECs on the same machine, and UDP for ECs in other machines. When the EC receives the metadata from Aeron, it goes from C to Python.

4.1.1 Communication Manager

Aeron is an independent C++ component, therefore we could either use Python to implement the component that would replace Aeron, since most of Kollaps is written in Python, however, this is not optimal, since Python is considered a language not fitted for systems where performance is a major metric [52], or we could use another language more fit to Kollaps needs. Rust is a multi-paradigm programming language designed for performance and reliability [53]. Aeron worked as a component that receives metadata and shares this metadata with the system, so, we decided to implement the Communication Manager that would work similarly to Aeron but implemented in Rust.

The Admin will start the CM, and a single CM exists per physical machine. The CM will have two tasks. The first is for inter node communication, which means that it must share

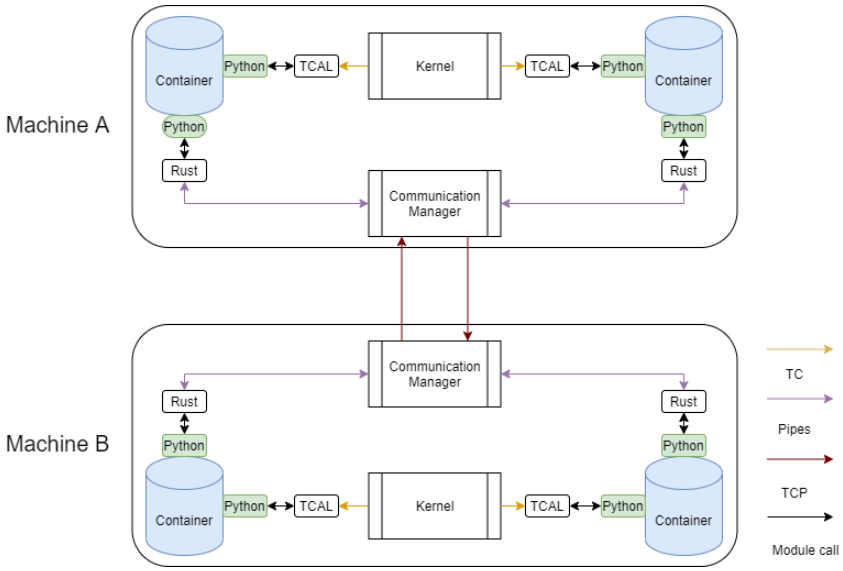

Figure 4.3: The flow of metadata in Kollaps 2.0 .

the metadata among ECs in its machine. The second one is for intra node communication, to achieve this, it must share the information of its ECs with other CMs, as well as the information it receives from other CMs to its ECs.

Communication Mechanisms

For intra node communication, each EC creates two pipes, one in read mode to receive metadata from the CM and one in write mode to send metadata to the CM.

For inter node communication, the CMs use TCP sockets [54]. For each other CM in the deployment, there is a socket connecting them. Figure 4.3 shows a depiction of the flow of metadata in Kollaps 2.0.

Functionality

During startup, the CM waits for all the containers to start. When all the containers have started, it opens the pipes, one in write mode and one in read mode for each container, and saves them in internal data structures.

Then it creates a thread that will accept connections from other CMs. This thread will run until it has accepted connections from all the other CMs. For each connection received, the CM starts a thread, which is always reading from the socket and writing to the pipes.

Meanwhile, in the main thread, the CM starts the loop with two steps. First, the CM does select call with all of the pipes open in read mode. Then the main will block until a single pipe contains a message to be read. After unblocking, we go to the second step, where we iterate over the pipes that contain metadata. We read the metadata from the pipes and write to all of

the pipes, previously opened in write mode. The CM uses the select call instead of an iterator, due to the blocking nature of the read call of the pipes we created [55]. If we used an iterator, the system would block if a single container stopped writing metadata.

4.1.2 Emulation Core

The EC needs to receive the metadata from the other ECs, therefore it needed to change to accommodate the new metadata dissemination mechanisms introduced in the previous section. The EC will now have the task of receiving metadata from the CM, and sending his metadata to the CM. As we now use Rust, the fact that the EC runs Python code stood as a problem, therefore the first step was to create a way for Python to interact with Rust.

4.1.3 Communication Core

To do this, we used PyO3 [56] which enables the user to generate a native Python module from a Rust library. So we created the Communication Core(CC) module. The EC will use this module as a Linux shared library.

The CC module provides a way for the EC to write metadata to the CM and to read from it. To achieve this, when the CC starts it creates pipes and opens one in write mode and one in read mode. Then it starts a thread with the task of reading from the pipe the CM writes to. When the thread reads metadata from the pipe, it hands it over to the EC. The module also provides ways for the EC to write its metadata to the pipe, allowing it to share metadata with the CM. Figure 4.4 shows a depiction of this part of the flow of metadata in Kollaps 2.0.

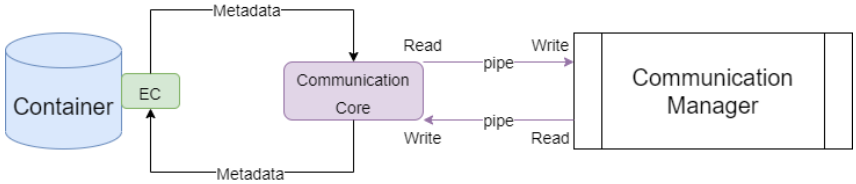

Figure 4.4: The flow of metadata in a container.

4.2 Kernel Information Retrieval

With the removal of Aeron, we significantly reduced the CPU usage, as we later show in Section 5.1. However, when it came to large-scale topologies, CPU usage was still at 100%. Using pyspy, [57] we were able to see that the TCAL update_usage function was causing the high CPU usage.

From the perspective of an EC, the function does the following for every other EC in the deployment. First, it does rtnl_dump_request [58] to know how many bytes it sent to the other

EC. Second, it uses rtnl_dump_filter [58] passing a callback as an argument. The kernel will then call the callback with the number of bytes associated. After receiving the information, the TCAL calls the EC with the number of bytes sent.

Therefore the system scales quadratically since each EC does two requests to the kernel for every other EC in the topology, which causes the high CPU usage.

There was a need to change update_usage, instead of performing multiple requests, to perform one single request to get all of the information necessary.

4.2.1 eBPF

eBPF as described in Section 2.5 allows the execution of bytecode at various hook points in a safe manner. eBPF provides a generic and flexible instruction set combined with infrastructure around it.

eBPF appears as a possible solution to the problem of having too many requests to the kernel. To that effort, we came up with the idea of maintaining a hashmap. The hashmap will have as its key the addresses of containers, and as its value, the number of bytes sent. With this map, we now would have the EC perform only one request, where we retrieve the entire map instead of one request per other EC in the system.

We will start by explaining the solutions that did not entirely fit Kollaps, with the goal of helping future work by explaining why some solutions that at first glimpse may seem ideal, however when implemented may not work very well. The final solution is explained in Section 4.2.2.

Kernel Probes

As previously explained in Section 2.5.2, kernel probes give us a way to break into any kernel routine and specify a handler routine to be called, so as a first approach to the problem, we decided to try and probe the functions of the kernel responsible for sending messages. With a probe inserted in a kernel function, we would have a handler routine that retrieves the source, destination address, and size from the messages, and inserts this information in the hashmap.

We decided to leverage bcc [41], which provides an easy way to write eBPF programs and a simple Python front-end. With this front-end, we can retrieve the hashmap to userspace.

With bcc, we decided to probe the tcp_send_msg [38] kernel function and create the mechanism previously described. However, Kollaps is supposed to be protocol agnostic, so we could not pursue this solution since it only detects TCP messages.

We moved into trying to probe the function that runs before tcp_send_msg which is secu-

rity_socket_sendmsg [59]. Again we were successful at implementing the mechanism. However, we were counting the size of the message and not of the packets. A TCP message does not exactly exist since TCP is stream oriented. TCP can bundle up data from many send() requests [60], and this is what we were handling in the `eBPF` program, multiple packets in a single stream instead of a packet at a time.

Because we are counting messages instead of packets, the updates to the hashmap would be slow compared to the frequency of the emulation loop, to understand why this is a problem we must first understand how the EC calculates throughputs. The EC does this by first calculating the delta between the number of bytes sent, and the previous number of bytes sent, and second dividing this delta by pool_period time. When we have slow updates, we will see the same number of bytes sent during many loops causing the delta value to be 0, if a delta value is 0 so is the throughput, therefore the EC would think a connection dropped when in reality it did not.

Therefore we need a function responsible for sending packets and not messages. This was __dev_queue_xmit . This function, as described in Section 2.5.1 is the last one called before the kernel sends a packet into the network.

When we are probing functions, we can access the contents of its context. In both tcp_sendmsg, and security_socket_sendmsg we had a sock.h structure [61]. Although not trivial as some investigation in the kernel code was necessary to know where the information is, it was a simple task to retrieve both the destination and source addresses. Below we can see an example of these accesses, where sk is a sock.h structure.

```
source=sk->__sk_common.skc_rcv_saddr;
destination=sk->__sk_common.skc_daddr;
```

With __dev_queue_xmit(), the context is a sk_buff although having documentation in the kernel. It is not a trivial task in any manner to retrieve the source and destination addresses. Because we do not have a similar structure to sock where we can directly get the addresses. We have to use bpf_probe_read [62] which allows us to read values from the kernel. To do so, it receives as arguments the size that will read and the kernelspace address. Although the size is straightforward because both the destination and source addresses are four bytes, calculating the kernelspace addresses where they reside using offset [63] is extremely hard.

When we run an `eBPF` program, a verification step is always run. It is responsible for making sure the `eBPF` program is safe to run [64]. The verification step made it extremely hard to design a program capable of retrieving the addresses from the skb_buff. Because it would be frequent that the verifier would reject the program, since we were accessing addresses we were

not allowed to. Therefore, normal debugging was not an option.

We eventually managed to retrieve the source and destination addresses as well as the packet size through the skb.len parameter [35]. The next step was testing this new mechanism. However, during testing, the number of bytes reported by it was lower than supposed we believe this was because we were accessing the packets from the host's namespace. And at that point in the packet's lifetime, it was already stripped of its VXLAN header, as explained in Section 2.3, causing the lower values therefore we had to move away from these types of solutions.

eXpress Data Path

`XDP`, as explained in Section 2.5.1, is designed for high performance. This fact, combined with the possibility of using `eBPF` to access the packet data, seemed to be a solution to our problem. BCC provides a way to offload an `eBPF` program to a network device, so we would use a program similar to the one we did with the __dev_queue_xmit() kprobe and offload it to a network device.

However, we need to specify which network device we offload the program to. In Section 2.3 we explain how the bridge and overlay driver worked, specifically that for each container, a veth pair [22] is created and attached to the bridge, leaving one end on the container network namespace and one on the host network namespace.

Since we have two different network devices, we have two locations to offload the program. We could either offload to the corresponding veth on the host network namespace, or the `eth0` on the container's network namespace. When we offloaded to the veth in the host network namespace, we did not capture any packets. This behavior happens because containers create the packets in their network namespaces. So the correct solution is to offload them to the `eth0` in the container's network namespace.

Since we were running from the network namespace of each container, our key will now be the destination address since the source is always the same. After implementing this mechanism, and testing, we noticed that what would be reported instead of packets sent was packets received. This behavior is because `XDP` only works on the ingress side of the network device, therefore it does not fit Kollaps since Kollaps does its calculations by packets sent.

Linux Traffic Control

Linux TC provides a way to run `eBPF` programs via the cls_bpf classifier as explained in Section 2.5.1. BCC, in combination with iproute2, simplifies the workflow shown in Section 2.5.1, making it possible to hook our `eBPF` program to the egress side of the network device. After implementation, it provided a mechanism that increments a hashmap after the container sends

a packet.

However, when testing, the mechanism did not display a stable behavior. During the lifetime of connections, the hashmap starts to update slower and slower in userspace, and therefore caused a similar problem to the one we had, when probing the security_socket_sendmsg. We suspected this might have been a behavior caused by Python because as mentioned before, Python has problems when used in these types of systems.

So we decided to use the same type of solution but this time in Rust, using RedBPF [65]. RedBPF works similarly to bcc, it provides users with a way to create `eBPF` programs via the redbpf-probes and redbpf-macros modules, and to compile them into .elf files [66] to be used by the kernel via the cargo-bpf module. Finally, to load and access bpf programs from userspace, it provides the redbpf module.

Still, after implementing this mechanism in Rust, the problem remained. We concluded that the map was not updated in userspace at the same rate as it was in kernelspace, which deemed the idea of accessing the map from userspace at every emulation loop not suitable for Kollaps.

Socket Filters

The `socket` interface as described in Section 2.5.1, provides a way to attach `eBPF` programs to them. Following the same ideas behind Linux TC, we will attach a filter to sockets. However, we do not want to attach a filter to every socket the container creates, because this would add an extremely high complexity to the mechanism. RedBPF allows attaching to a raw_socket [67] allowing us to filter all the traffic of the container.

4.2.2 Final Solution

Socket filters looked like another possible solution. However, we had to make changes to the `eBPF` program due to the problem presented in the previous Section. We need a different way to obtain the changes to the hashmap in userspace, therefore, we decided to use perf events, as explained in Section 2.5.2, which allows us to notify userspace of events in kernel space. Specifically, we use a RedBPF PerfMap [68], which provides us a way to get the hashmap values from kernelspace, in userspace.

The solution, Figure 4.5 shows a depiction of the solution we designed, and works as follows, the `eBPF` program still maintains the same map, however, it sends its updates via the PerfMap to userspace. We do not want to generate an event each time the container sends a packet since this causes CPU usage to be high, and the EC only needs new values every pool_period (see Section 3.1.6. So we define that we only send events every perf_event period. Perf_event has a

default value of 0.025 seconds, half of the pool_period value.

In userspace, we have to read these events, therefore at startup, the Communication Core (see Section 4.1.3) will start a new thread. This thread is responsible for reading the events the `eBPF` program will generate. To save the values in userspace, we create a map similar to the one in the kernel. The map updates each time the `eBPF` program sends an event. Therefore, we now have a way to force a userspace map to update, with this mechanism, we do not rely on kernelspace changes to be available in userspace.

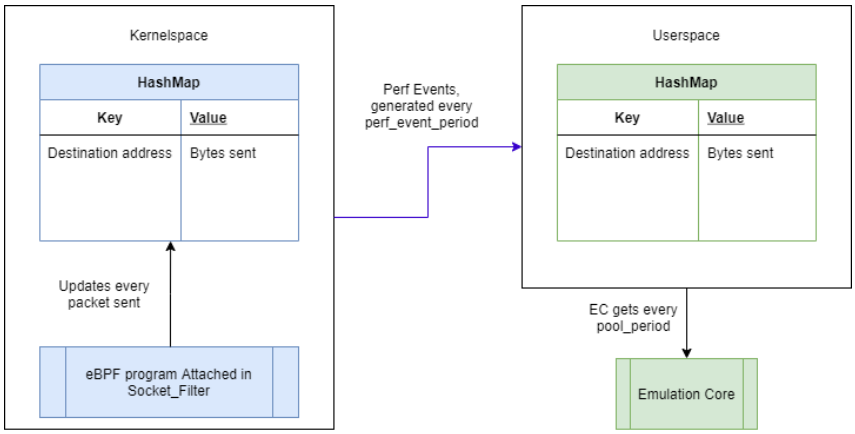

Figure 4.5: Mechanism to retrieve information from the Kernel .

4.3 Metadata Dissemination Model

The model of metadata dissemination in Kollaps is a periodic one. The model is always exchanging metadata between cores, even if nothing changes, which brings two problems.

The first problem is the excessive amounts of metadata circulating in long-lived flows. The second is that each time we receive metadata, we need to react to the possible changes that might have happened. This design decision, causes the ECs to calculate new values each time they receive new metadata. Therefore, causing a higher CPU usage than necessary.

To solve this problem we looked into changing this model into a reactive one. This change involves deciding if the information we got from the kernel is relevant to share or not. Now, the EC will only share the metadata, if the variation from the previously seen throughput for a path, is superior to a certain percentage, 5% will be the default value, 5% lowers the metadata shared and maintains an accurate emulation as well. However, due to the way Kollaps is designed, we had to make several changes to accommodate this.

To explain these problems, first, we must revisit the emulation loop explained in Section 3.1.6. Firstly in step (iv), after computation, the EC would delete the metadata related to flows of other ECs. Since we are reactive, the ECs must maintain the metadata, and assume that if

the stored flows did not change, then nothing changed.

Secondly, in step (iv), the ECs only compute bandwidth usage on active paths. However, being active in Kollaps means that the EC will share the metadata related to this path with other ECs. In the new model, it is not certain the EC will share the metadata with other ECs. However, we still want to do calculations because even though one EC does not have relevant changes to report, other ECs might have sent relevant metadata. In the new model, the ECs do calculations if either the throughput of the path changed by 5% or if other ECs sent new metadata, therefore solving this problem.

Thirdly, as explained before, we maintain metadata sent by other ECs. The metadata in Kollaps specifies a bandwidth value and the links in the graph it transversed. Integer values represent links. The integer values of the links are specific to a specific state of the graph. However, as explained in Section 3.1.6 the graph changes every time a dynamic event happens. Therefore, when a graph change happens, its configuration changes and an integer value that the EC assigned to a specific link between two nodes might not represent the same link anymore. Therefore, every time a dynamic event happens, the metadata stored becomes obsolete. Which means we must delete it.

Fourthly, containers can crash at any time in the experiment. In the previous model, as we mentioned in step (iv), we delete metadata related to flows of other ECs after computation. This means that should a container leave, no problem occurs since it will discard the metadata eventually. But in our new model, we must force the EC to discard the metadata. To achieve this, we have the EC that is going to leave, send a special message which informs the others to discard its metadata.

4.4 Baremetal Deployments

Kollaps did not support bare metal deployments. Hence a user that would want to emulate a network state in an already running system would not be able to. To enable this use case in Kollaps 2.0, we introduced support for bare metal deployments.

One debate that appeared when addressing bare metal deployments was if more than one EC per network namespace should be possible. If we have more than one EC per network namespace, we can allow for different applications in the same network namespace. Therefore, allowing them to have different links and therefore different network values. However, this is not realistic because, in a real deployment, which is the purpose of bare metal, the network that is in between two network namespaces is the same. Therefore we abandoned this idea, and we initialize one EC per physical machine.

We do not need to make changes to emulation mechanisms and metadata dissemination. However, all of the others need changes since they are for containerized deployments. We changed the Deployment Generator to accept bare metal topologies. Baremetal topologies are different because the user must specify the IP of the service, the hostname that is the name for ssh access to the machine, and if the user wants, a path to a logfile that Kollaps 2.0 can retrieve later. To the others, some design details had to be changed.

The dashboard (Figure 4.6) now has more responsibilities since it is the way we have to interact with the machines.

Figure 4.6: The new dashboard for baremetal deployments.

To run bare metal deployments we assume that each machine has the Kollaps code in a specific directory. To interact with the machines, we use ssh, specifically a Python ssh library [69]. We also introduced two new commands and changed the start and stop commands. The install command that has the task of installing the Kollaps components on the machine, the initialize command initializes the emulation and data dissemination mechanisms. The start command will now start a script given by the user. The stop command stops the emulation and data dissemination mechanisms, after, it runs a script to stop the experiment if the user provides one.

4.5 Discussion

Kollaps is a fully decentralized network emulator with a well-established architecture, but it has multiple limitations. In Kollaps 2.0, we looked to address these limitations. To do so, we introduced: (i) New data dissemination mechanisms replacing Aeron with a new Rust component that uses pipes, and sockets to share metadata, lowering the CPU usage of Kollaps. (ii) A new way to retrieve data from the kernel combining `eBPF` and the `socket` subsystem, lowering even more the CPU usage specifically, in large scale deployments. (iii) A new data dissemination model, replacing the periodic one with a reactive one lowering the amounts of metadata circulating. (iv) Finally, we introduced Kollaps 2.0 to bare metal deployments, allowing a lot of new use cases.

Chapter 5

Evaluation

In this chapter, we present the evaluation of Kollaps 2.0 through a series of experiments. The results show that:

- Kollaps 2.0 lowered the CPU usage of both small scale, and large scale topologies, while at the same time maintaining accuracy.
- Kollaps 2.0 lowered the amounts of metadata shared in long-lived flows while maintaining the fast reaction time of the Kollaps.
- Kollaps 2.0 introduced bare metal deployments while managing to maintain the emulation accuracy of Kollaps, hence enabling novel use cases.

The first benchmark presented is the comparisons in CPU usage in differents scales of deployments, presented in Section 5.1. Then we present a comparison between metadata shared in Kollaps and Kollaps 2.0 in Section 5.2. In the next three sections, we compare Kollaps and Kollaps 2.0 in metrics related to the accuracy of the emulation. In Section 5.3 we present bandwidth emulation accuracy, in Section 5.4 emulation reaction time, and finally in Section 5.5 scalability of latency emulation accuracy.

To test for these different benchmarks, we used 4 Dell PowerEdge R630 server machines, with 64-cores Intel Xeon E5-2683v4 clocked at 2.10 GHz CPU, 128 GB of RAM. These machines are connected by a Dell S6010-ON 40 GbE switch, unless specified otherwise these were the used machines. The Docker Engine version used was 19.03.4, and the docker network driver used was overlay.

Table 5.1: Comparison of CPU usage per physical machine in smallscale deployments running iperf3.

Nodes	**Kollaps 2.0**		**Kollaps w/o Aeron**		**Kollaps**	
	usr	sys	usr	sys	usr	sys
10	1%	1%	1%	1%	7%	1%
50	2%	1%	2%	1%	23%	1%
100	2%	1%	2%	1%	40%	1%
200	4%	2%	4%	2%	82%	2%

5.1 CPU Usage

One of the primary limitations of Kollaps was CPU usage, to tackle this, we removed Aeron because as explained in Section 4.1 it was the primary source of high CPU usage.

To test for the improvements, we set up various deployments with n-clients/n-servers dumbbell topology. The clients execute an iperf3 client, and the servers execute an iperf3 server. The CPU usage was retrieved using dstat [70], we separate results into user (`usr`)the percentage of time spent running user processes, and system (`sys`) percentage of time spent running system processes. Table 5.1 presents the results.

The values from Kollaps at first glimpse seem acceptable. However, we must not forget we have four machines with 64 cores each. We compared the original Kollaps with a version of Kollaps 2.0 that only has Aeron substituted with the new mechanisms and fully implemented Kollaps 2.0. As we can see by the results, we were able to reduce CPU usage significantly with the removal of Aeron, and the introduction of both the Communication Manager and the Communication Cores. The results for both Kollaps 2.0 and Kollaps without Aeron are the same since the rest of the new mechanisms do not affect this scale of topologies.

However, as explained in Section 4.2 the mechanism to retrieve the information from the kernel, did not fit Kollaps since it scaled quadratically. To measure the CPU usage of large scale topologies, we did different experiments using scale-free network topologies generated with [71]. This method constructs scale-free networks, which are representative of the characteristics of Internet topologies. The experiment consists of end-nodes running ping [72] to another end-node for 10 minutes. We did this tests with 500 nodes(332 services, 168 bridges),1000 nodes(666 services, 334 bridges), 2000 nodes (1344 services,656 bridges) and 4000 nodes(2668 services, 1332 bridges). During this test, we again run dstat.

Table 5.2 presents the results for Kollaps 2.0, Kollaps without Aeron, and Kollaps. Kollaps originally had the Aeron problem which, caused a lot of `usr` CPU usage. However, after being removed, revealed the problem with the mechanism to retrieve information from the kernel, which caused the high `sys` CPU usage. After being substituted by the mechanism combining

Table 5.2: Comparison of CPU usage per physical machine in largescale deployments running ping.

Nodes	**Kollaps 2.0**		**Kollaps w/o Aeron**		**Kollaps**	
	usr	sys	usr	sys	usr	sys
500	1%	0%	1%	95%	97%	3%
1000	1%	0%	1%	95%	97%	3%
2000	3%	0%	1%	95%	97%	3%
4000	5%	1%	1%	95%	97%	3%

both `eBPF` and the `socket` subsystem. We can see that the usage is negligible. Although we can expect significant CPU usage from these experiments, we must not forget pings do not cause significant bandwidth values. Therefore, Python calculations are not done, and this is the reason why we don't have significant values in Kollaps 2.0.

5.2 Metadata Generation

Kollaps RTT bandwidth model, as explained in Section 3.1.6 relies on having all the bandwidth values of competing flows. To meet this requirement, Kollaps relies on metadata sharing. However, the model used to share the metadata did not fit Kollaps, since metadata was always shared across every other EC in the system, even if nothing changed.

To test improvements to the reduction of metadata shared, we created simple n-clients/n-servers dumbbell topologies to measure the amounts of metadata shared. We use iperf3 to generate steady TCP traffic for 60 seconds.

As explained in Section 4.3 in Kollaps, we have a periodic metadata sharing model. Thus every time the EC has new information, it will share it with others. With the new model, the EC only shares metadata if it changes by at least 5%. Figure 5.1 presents the results of these changes. To calculate the amount of metadata sent, we multiply the number of messages each EC sent by the size of the default Kollaps message which, is 256 bytes.

Observing the results, we can see that we managed to drop metadata shared at deployments of a lower scale. However, we can also see that when the number of containers increases, the difference between the models shrinks. Because, as explained in Section 4.3", every time an EC reports a change, the others must do new calculations and share their new values. Obviously, with an increase in containers, we will also have an increase in changes, and therefore a larger amount of metadata is shared, eventually reaching the point where we are again sharing at every emulation loop.

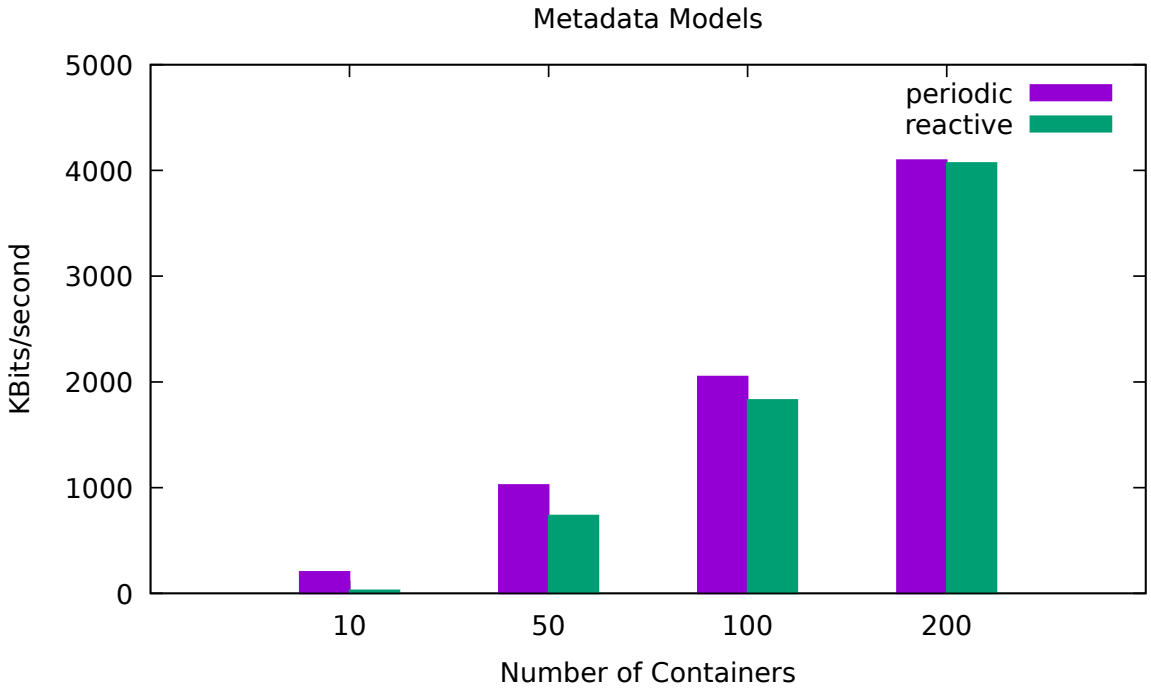

Figure 5.1: Comparison between metadata generated in a periodic and reactive model.

5.3 Bandwidth Emulation Accuracy

Due to changes in the mechanism used to retrieve information from the kernel (see Section 4.2), we must revisit this test to assure that Kollaps 2.0 can still accurately emulate different values of bandwidths.

To benchmark this accuracy, we run a test composed of two services connected by a single link. One service will execute an iperf3 [73] server in one machine, and another service, an iperf3 client in a different physical machine. We access bandwidth at different values, from low to high. The test consists of having the client execute iperf3 over a period of 60 seconds.

Observing the results presented in Table 5.3, we can see the bandwidth values of Kollaps 2.0, in both containerized and bare metal deployments, are always in a 5% error across all ranges. Showing that Kollaps 2.0 maintains the precise accuracy reported in Kollaps, we expect this because the information we retrieve using `eBPF` and with the `rtnetlink` library is the same.

5.4 Emulation Reaction Time

5.4.1 Docker Deployment

Given the changes previously mentioned in Section 4, mainly, the change in the Communication Mechanisms and the change in the information retrieval from the kernel, we must test the

Table 5.3: Study of the accuracy bandwidth shapping accuracy for different link values on a client-server topology.

Link BW	Kollaps 2.0 Docker	Kollaps 2.0 Baremetal	Kollaps 1.0
Low:			
128 Kb/s	123 Kb/s	123 Kb/s	122 Kb/s
256 Kb/s	245 Kb/s	245 Kb/s	245 Kb/s
512 Kb/s	489 Kb/s	490 Kb/s	489 Kb/s
Mid:			
128 Mb/s	122 Mb/s	122 Mb/s	122 Mb/s
256 Mb/s	244 Mb/s	245 Mb/s	244 Mb/s
512 Mb/s	489 Mb/s	489 Mb/s	489 Mb/s
High:			
1 Gb/s	954 Mb/s	955 Mb/s	954 Mb/s
2 Gb/s	1.9 Gb/s	1.9 Gb/s	1.9 Gb/s
4 Gb/s	3.79 Gb/s	3.81 Gb/s	3.78 Gb/s

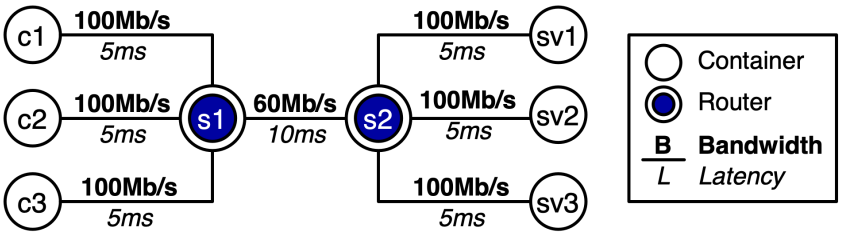

Figure 5.2: Dumbbell topology with 3 clients, 3 servers.

reaction time of Kollaps 2.0. To assure that the new mechanisms are as fast at reporting correct values as the ones in the original Kollaps. To this goal, we set up a simple 3-clients/3-servers dumbbell topology, depicted in Figure 5.2. The link connecting both switches has a maximum bandwidth of 60Mbps, all the clients have the same bandwidth, and latency so they should get a similar share of the link.

Kollaps provides a rich set of dynamic events (Section 3.1.6). To implement these events, it uses Timers [74] in each container. Each event in an experiment is a timer object, with an action and a specific time to run the action. When the user starts the experiment, the Timer objects start counting. When the timer object responsible for starting the experiment hits his time, it will start the experiment. Specifically, calling a script in this example an iperf3 client, however, this causes one problem. Because the experiment defines the timer at second X, the EC will assume it started the iperf3 client at X second. While in reality, the iperf3 client might take some time to start, on average, it takes one second. Therefore, the iperf3 client starts at X+1, causing a desync between scripts and the events, making Kollaps look like it is predicting the future. Let us take as an example an event happening 10 seconds after the iperf3 client starts, since the client started at X+1, in the iperf3 client logs the event will happen at second 9.

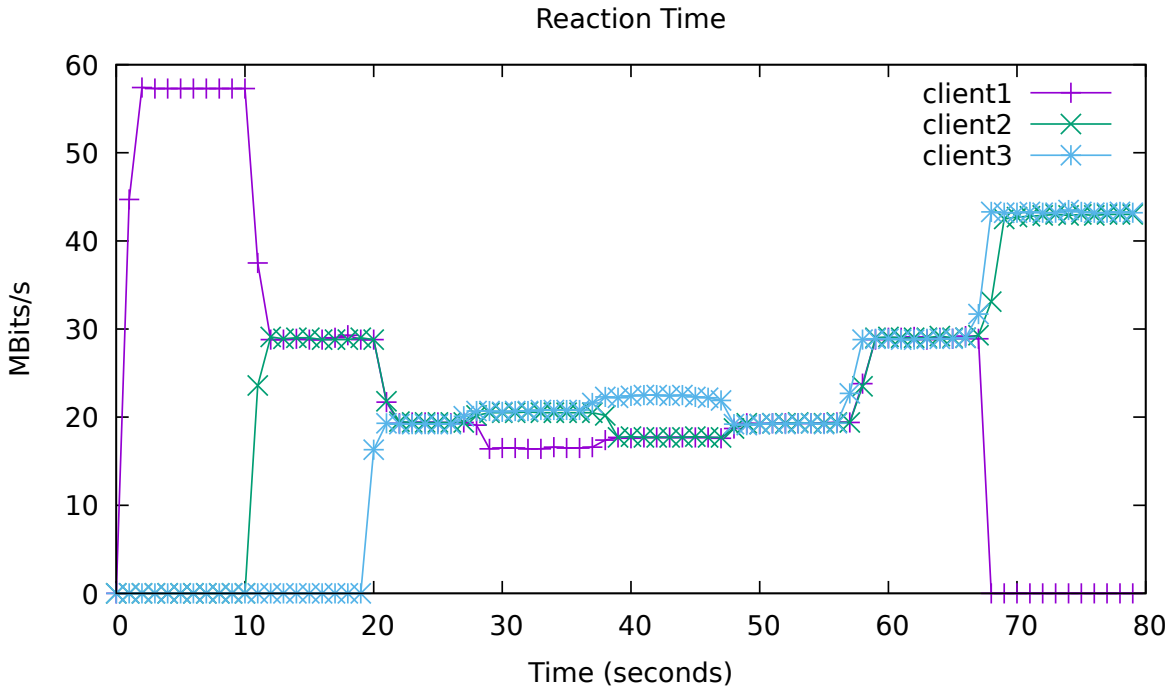

Figure 5.3: Reaction time in a container deployment.

The experiment, as seen in Figure 5.3 proceeds as follows. At the start, client1 be the only one with an active flow. So it will use all of the available bandwidth. After 10 seconds, client2 starts, and both the nodes will compete for a share of the link. Since they have the same latency and therefore RTT, the model as explained in Section 3.1.6, will award them with the same bandwidth. At second 20, client3 enters the experiment, and following the same logic, the three clients will have similar bandwidths. In both cases, we can see that Kollaps 2.0 took around a second to adjust the bandwidth of the clients. At second 30, the link between client1 and the s1 switch will change in latency from 5 to 10 since its RTT went up. It will have a smaller share of bandwidth. Causing his bandwidth to drop, since there is now available bandwidth to be used, both client2 and client3 will start using that bandwidth. At 40 seconds, the link between client2 and the s1 switch will have the same increase in latency. Therefore the bandwidth for client2 will be lower, causing the clients to pick up the available bandwidth. Finally, at 50 seconds, the same happens to client3, and the three clients will now have the same bandwidth since the links are identical. At second 60, the link connecting the switches s1 and s2 increases in bandwidth from 60 to 90 Mbs causing the available bandwidth to increase. And therefore, the bandwidth of the three clients will increase similarly. In second 70, client1 crashes leaving his share of the bandwidth available which, will be picked up by both client1 and client2 in a similar manner.

Given these results, we can conclude that we retained the ability of Kollaps to react under a second to changes in the topology in Kollaps 2.0. While also validating that the kernel

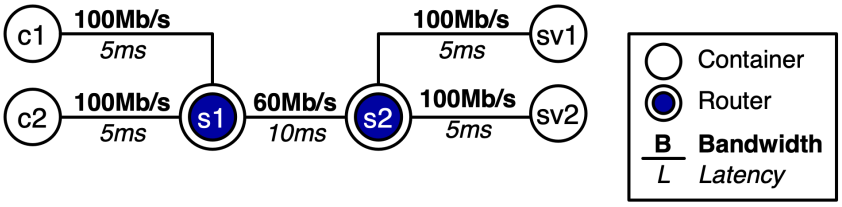

Figure 5.4: Dumbbell topology with 2 clients, 2 servers.

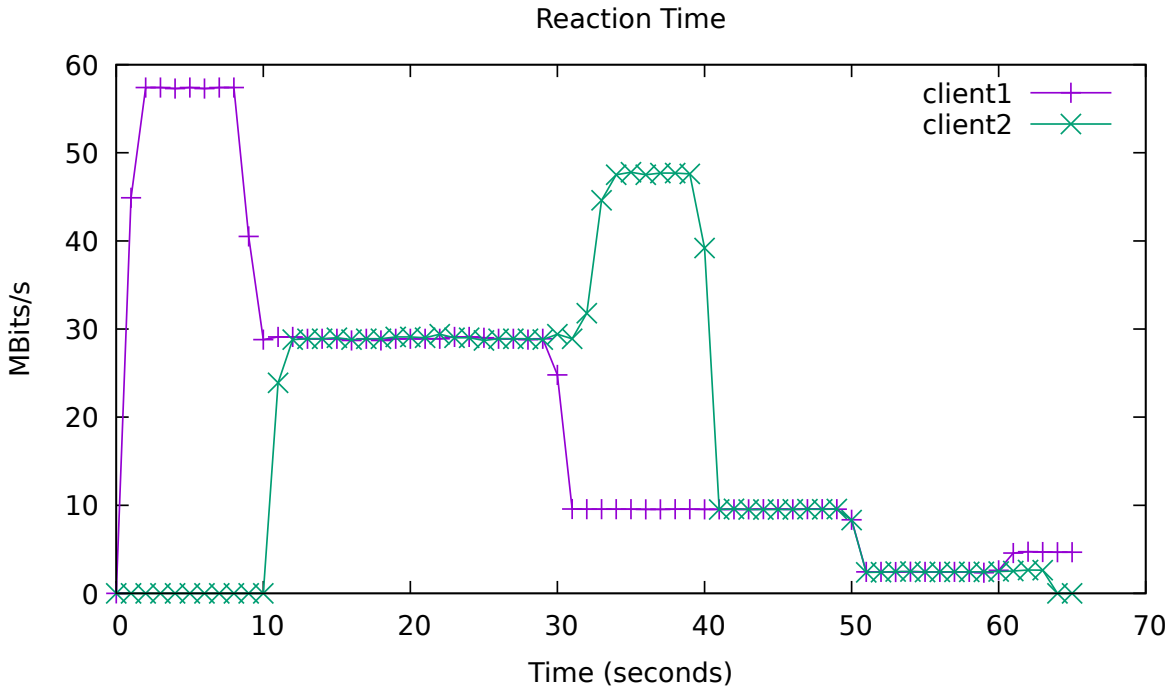

Figure 5.5: Reaction time in a baremetal deployment.

information we are retrieving is correct and updated at a fast pace.

5.4.2 Baremetal Deployment

In Kollaps 2.0, we introduced bare metal deployments. In the previous section, we showed how we retain the link-level emulation capabilities of Kollaps. Now we must take a look at the validation of the RTT model, but also check if we maintained the reaction time accuracy. For this, we created a simple 2-clients/2-servers dumbbell topology, depicted in Figure 5.4. The clients have the same bandwidth values, and the link connecting both switches has a 60 Mbps bandwidth.

To support bare metal deployments, we choose ssh to communicate with the machines. However, this adds extra overhead to the starting of experiments. This overhead causes the EC of each machine to not start at the exact time. For this reason, we do not have similar syncing behavior to the one we had in Docker deployments.

Table 5.4: Mean squared error exhibited on latency tests with large scale-free topologies in Kollaps 2.0 and Kollaps.

Nodes	Kollaps 2.0	Kollaps
500	0.0300	0.0257
1000	0.0369	0.0361
2000	0.0462	0.0400
4000	0.0647	0.045

This experiment, as depicted in Figure 5.5 and starts with client1, using all of the available bandwidth. At second 10, client 2 joins, and since they have similar links, they will obtain similar shares of bandwidth. At second 30, the link between client1 and s1 changes from 100Mbps to 10Mbps. This constraint in the bandwidth causes client1 bandwidth usage to drop to 10Mbps. A second 40 a similar thing happens to client2. At second 50, the link between the switches changes from 60 Mbps to 5 Mbps causing the available bandwidth of the clients to only be 2.5Mbps for each. Finally, at second 60, the link between client2 and s1 disappears, causing the 5Mbps of available bandwidth to be allocated only to client1.

Given these results, we can confirm that we retained the reaction time of Kollaps 2.0 in bare metal deployments.

5.5 Scalability of Latency Emulation Accuracy

With the changes to the mechanism used to retrieve information from the kernel, there was a massive drop in CPU usage in large scale deployments as mentioned in Section 5.1. We now assess whether if Kollaps 2.0 can still scale while maintaining accurate latency values. To this avail, we did different experiments using scale-free network topologies generated using [71]. The experiment consists of end-nodes running ping [72] to another end-node for 10 minutes. We did these tests with the same large scale topologies as mentioned in Section 5.1.

In Table 5.4 we observe the results, and can see that Kollaps 2.0 retains the same latency scalability as Kollaps. Because the mechanism by which Kollaps performed the emulation of latency did not change. However, as we can see by the results, the MSE still increases with the size of the topology, even though we stay at low CPU usage as reported in Section 5.1. Because with the increase in the number of services, and bridges the probability of two services that ping each other being on the same machine diminishes. Therefore, the effect of the actual network comes into play, adding a small error increase with the increase of nodes.

5.6 Discussion

After this evaluation, we can confirm that Kollaps 2.0 is far more efficient in resource usage, specifically CPU usage, than the original one. It also started moving away from a periodic to a new reactive model of metadata dissemination with noticeable improvements already in small scale scenarios. Finally, Bare metal deployments can now use Kollaps to emulate networks.

Chapter 6

Conclusions

To develop large scale distributed systems, one of the most important tools that developers have is testing. However, if testing does not provide reproducible results it is not useful, because developers can not assess the impact of the changes made. One reason for unreproducible results is the network. The network stands as an uncontrollable variable that developers have to take into account every day, therefore, a reliable way to control the network is necessary to achieve reproducibility. This method is called network emulation.

In this document, we analyze the current state-of-the-art and leverage it against our network emulator Kollaps [14]. Kollaps is a fully decentralized distributed emulator agnostic to application and transport protocols, that can scale to thousands of nodes while maintaining an accuracy similar to the current centralized solutions and bare-metal deployments. Nevertheless, like all systems, Kollaps has its limitations. Due to its decentralized nature, Kollaps needs to share information, the current communication framework and communication model have many problems. On the engineering level, Aeron [15] causes the CPU usage to be at 100%. On an algorithmic level, the current model causes Kollaps to have a high amount of metadata in the system in long-lived flows. Finally, Kollaps does not support bare metal deployments, therefore, leaving a lot of use cases unexplored.

In this dissertation, we improved Kollaps and introduced Kollaps 2.0. Kollaps 2.0 brings new communication mechanisms and a new communication model, lowering CPU usage and the amount of metadata shared. It introduces a new mechanism to retrieve data from the Kernel built on eBPF and the socket subsystem, lowering CPU usage on large scale deployments. Kollaps 2.0 also brings bare metal support and allows for a vast new number of use cases.

6.1 Future Work

Kollaps 2.0 is one of the main tools for network emulation for large scale distributed systems. Still, it has several limitations. We now present some limitations, and discuss possible approaches to solve them.

The first one is CPU usage, in Kollaps 2.0, we were able to significantly lower CPU usage. However, since Kollaps still uses Python to do the calculations for the RTT model, and the use of Python still causes a high CPU usage. As mentioned, Rust is a much better-suited language for this type of system. We began the transition by introducing the Communications Manager and Communication Core. But, to further bring Kollaps 2.0 resource usage down, a full transition is necessary.

Metadata shared, we introduced a reactive model in Kollaps 2.0. However, it proved not to be enough to reduce the high amounts of metadata shared in large scale topologies. To further bring down metadata shared, one possible solution is to move the RTT model to the Communications Manager, and have the Communications Manager do the calculations with all the values retrieved from the ECs. This way, the metadata shared to containers would not be metadata to run the model, but instead their new bandwidth values, which would also help with the previously mentioned limitation. Another possible improvement for metadata shared is to move away from fixed-size messages. Kollaps 2.0 always creates a buffer with 256 bytes for metadata, although, most of the times entries are empty, a possible solution is to either send more small messages, or calculate a maximum message size given the topology.

Interactivity, Kollaps 2.0 does not support the ability to change deployments during run-time at the user's will. With interactivity, we would be able to remove containers or machines (in bare metal) from the experiment, add new ones, change the properties of links etc.

Bibliography

[1] "Docker." accessed: 2021-10-13. [Online]. Available: https://www.docker.com/products/container-runtime

[2] "Docker swarm." accessed: 2021-10-13. [Online]. Available: https://docs.docker.com/engine/swarm/

[3] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: An overlay testbed for broad-coverage services," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 3, p. 3–12, Jul. 2003. [Online]. Available: https://doi.org/10.1145/956993.956995

[4] 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 2008 Annual Technical Conference*, ser. ATC'08. USA: USENIX Association, 2008, p. 113–128.

[5] W. Kim, A. Roopakalu, K. Y. Li, and V. S. Pai, "Understanding and characterizing planetlab resource usage for federated network testbeds," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 515–532. [Online]. Available: https://doi.org/10.1145/2068816.2068864

[6] S. Floyd and V. Paxson, "Difficulties in simulating the internet," vol. 9, no. 4, p. 392–403, Aug. 2001. [Online]. Available: https://doi.org/10.1109/90.944338

[7] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," ser. Hotnets-IX. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1868447.1868466

[8] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, and H. Karl, "Maxinet: Distributed emulation of software-defined networks," in *2014 IFIP Networking Conference*, 2014, pp. 1–9.

[9] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, "Scalability and accuracy in a large-scale network emulator," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, p. 271–284, Dec. 2003. [Online]. Available: https://doi.org/10.1145/844128.844154

[10] V. Schiavoni, E. Rivière, and P. Felber, "Splaynet: Distributed user-space topology emulation," in *Middleware 2013*, D. Eyers and K. Schwan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 62–81.

[11] "Splay." accessed: 2021-10-13. [Online]. Available: http://wwwa.unine.ch/iiun/cs/splay/documentation.html

[12] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "Crystalnet: Faithfully emulating large production networks," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 599–613. [Online]. Available: https://doi.org/10.1145/3132747.3132759

[13] J. Lai, J. Tian, K. Zhang, Z. Yang, and D. Jiang, "Network emulation as a service (neaas): Towards a cloud-based network emulation platform," *Mobile Networks and Applications*, vol. 26, no. 2, pp. 766–780, Apr 2021. [Online]. Available: https://doi.org/10.1007/s11036-019-01426-0

[14] P. Gouveia, J. a. Neves, C. Segarra, L. Liechti, S. Issa, V. Schiavoni, and M. Matos, "Kollaps: Decentralized and dynamic topology emulation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387540

[15] "Aeron media driver." accessed: 2021-10-13. [Online]. Available: https://github.com/real-logic/aeron

[16] "extended berkeley packet filter." accessed: 2021-10-8. [Online]. Available: https://ebpf.io/what-is-ebpf

[17] "Linux traffic control." accessed: 2021-10-13. [Online]. Available: https://linux.die.net/man/8/tc

[18] "What is a virtual machine (vm)?" accessed: 2021-10-22. [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine

[19] "Kubernetes." accessed: 2021-10-13. [Online]. Available: https://kubernetes.io/

[20] "Understanding docker networking drivers and their use cases," accessed: 2021-10-8. [Online]. Available: https://www.docker.com/blog/understanding-docker-networking-drivers-use-cases/

[21] "bridge(8) — linux manual page," accessed: 2021-10-8. [Online]. Available: https://man7.org/linux/man-pages/man8/bridge.8.html

[22] "veth - virtual ethernet device." accessed: 2021-10-10. [Online]. Available: https://man7.org/linux/man-pages/man4/veth.4.html

[23] M. Hausenblas, *Docker-Networking-and-Service-Delivery*, 1^{st} ed. O'Reilly Media, 2016.

[24] L. Kreeger, T. Sridhar, M. Bursel, and C. Wright, *RFC 7348*, 2014.

[25] "Hierarchy token bucket." accessed: 2021-10-13. [Online]. Available: https://linux.die.net/man/8/tc-htb

[26] "Priority qdisc." accessed: 2021-10-13. [Online]. Available: https://man7.org/linux/man-pages/man8/tc-prio.8.html

[27] "Netem - network emulator." accessed: 2021-10-13. [Online]. Available: https://www.linux.org/docs/man8/tc-netem.html

[28] "universal 32bit traffic control filter," accessed: 2021-10-13. [Online]. Available: https://tldp.org/HOWTO/Adv-Routing-HOWTO/lartc.adv-filter.u32.html

[29] "Berkeley packet filter." accessed: 2021-10-8. [Online]. Available: https://www.kernel.org/doc/html/latest/networking/filter.html#networking-filter

[30] "Llvm project." accessed: 2021-10-8. [Online]. Available: https://llvm.org/

[31] "Clang: a c language family frontend for llvm," accessed: 2021-10-8. [Online]. Available: https://clang.llvm.org/

[32] "express data path," accessed: 2021-10-8. [Online]. Available: https://docs.cilium.io/en/stable/bpf/

[33] "Linux socket interface," accessed: 2021-10-8. [Online]. Available: https://linux.die.net/man/7/socket

[34] "Kernel probes (kprobes)," accessed: 2021-10-8. [Online]. Available: https://www.kernel.org/doc/html/latest/trace/kprobes.html

[35] "sk_buff structure." accessed: 2021-10-8. [Online]. Available: https://paginas.fe.up.pt/~mrs01003/skb.html

[36] "xdp_buff structure." accessed: 2021-10-8. [Online]. Available: https://elixir.bootlin.com/linux/latest/source/include/net/xdp.h#L69

[37] D. Borkmann, "On getting tc classifier fully programmable with cls bpf," 2016.

[38] A. K. Chimata, "Path of a packet in the linux kernel stack," University of Kansas, Tech. Rep., 2015.

[39] "net, sched: add clsact qdisc," accessed: 2021-10-8. [Online]. Available: https://lwn.net/Articles/671458/

[40] "Perf wiki." accessed: 2021-10-10. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[41] "Bpf compiler collection (bcc)." accessed: 2021-10-10. [Online]. Available: https://github.com/iovisor/bcc

[42] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," ser. CoNEXT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 253–264. [Online]. Available: https://doi.org/10.1145/2413176.2413206

[43] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker, "To infinity and beyond: Time warped network emulation," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 1–2. [Online]. Available: https://doi.org/10.1145/1095810.1118605

[44] K. V. Vishwanath, D. Gupta, A. Vahdat, and K. Yocum, "Modelnet: Towards a datacenter emulation environment," in *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*, 2009, pp. 81–82.

[45] "Openstack." accessed: 2021-10-14. [Online]. Available: https://www.openstack.org/

[46] "Opendaylight," accessed: 2021-10-14. [Online]. Available: https://www.opendaylight.org/

[47] "Open vswitch," accessed: 2021-10-14. [Online]. Available: https://www.openvswitch.org/

[48] "Docker security capabilities." accessed: 2021-10-13. [Online]. Available: https://docs.docker.com/engine/security/

[49] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose, "Modeling tcp reno performance: a simple model and its empirical validation," *IEEE/ACM Transactions on Networking*, vol. 8, no. 2, pp. 133–145, 2000.

[50] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "Tcp vegas: New techniques for congestion detection and avoidance," in *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, ser. SIGCOMM '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 24–35. [Online]. Available: https://doi.org/10.1145/190314.190317

[51] F. Kelly, "Charging and rate control for elastic traffic," *European Transactions on Telecommunications*, vol. 8, no. 1, pp. 33–37, 1997.

[52] "Python perfomance." accessed: 2021-10-13. [Online]. Available: https://wiki.python.org/moin/PythonSpeed

[53] "The rust programming language introduction." accessed: 2021-10-13. [Online]. Available: https://doc.rust-lang.org/book/ch00-00-introduction.html

[54] "Async standard library." accessed: 2021-10-13. [Online]. Available: https://docs.rs/async-std/1.8.0/async_std

[55] "Linux pipes." accessed: 2021-10-10. [Online]. Available: https://linux.die.net/man/3/mkfifo

[56] "The pyo3 user guide," accessed: 2021-10-8. [Online]. Available: https://pyo3.rs/v0.14.5/

[57] "py-spy: Sampling profiler for python programs." accessed: 2021-10-10. [Online]. Available: https://github.com/benfred/py-spy

[58] "rtnetlink(7) — linux manual pag." accessed: 2021-10-12. [Online]. Available: https://man7.org/linux/man-pages/man7/rtnetlink.7.html

[59] "security_socket_sendmsg in the kernel code." accessed: 2021-10-10. [Online]. Available: https://elixir.bootlin.com/linux/v5.15-rc4/C/ident/security_socket_sendmsg

[60] "tcp(7) — linux manual page," accessed: 2021-10-14. [Online]. Available: https://man7.org/linux/man-pages/man7/tcp.7.html

[61] "sock.h," accessed: 2021-10-14. [Online]. Available: https://elixir.bootlin.com/linux/latest/source/include/net/sock.h#L349

[62] "bpf-helpers(7) — linux manual page," accessed: 2021-10-14. [Online]. Available: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html

[63] "offsetof(3) — linux manual page." accessed: 2021-10-8. [Online]. Available: https://man7.org/linux/man-pages/man3/offsetof.3.html

[64] "Verification step in ebpf." accessed: 2021-10-10. [Online]. Available: https://ebpf.io/what-is-ebpf#verification

[65] "A rust ebpf toolchain." accessed: 2021-10-10. [Online]. Available: https://github.com/foniod/redbpf

[66] "elf - format of executable and linking format (elf) files." accessed: 2021-10-10. [Online]. Available: https://man7.org/linux/man-pages/man5/elf.5.html

[67] "raw(7) - linux man page." accessed: 2021-10-10. [Online]. Available: https://linux.die.net/man/7/raw

[68] "Perfmap implementation in github." accessed: 2021-10-10. [Online]. Available: https://github.com/foniod/redbpf/blob/main/redbpf/src/perf.rs

[69] "openssh-wrapper 0.4." accessed: 2021-10-10. [Online]. Available: https://pypi.org/project/openssh-wrapper/

[70] "dstat." accessed: 2021-10-13. [Online]. Available: https://linux.die.net/man/1/dstat

[71] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999. [Online]. Available: https://science.sciencemag.org/content/286/5439/509

[72] "Ping command." accessed: 2021-10-12. [Online]. Available: https://linux.die.net/man/8/ping

[73] "iperf." accessed: 2021-10-10. [Online]. Available: https://iperf.fr/

[74] "Timer objects." accessed: 2021-10-14. [Online]. Available: https://docs.python.org/3/library/threading.html