KOLLAPS: decentralized and efficient network emulation for large-scale systems

Sebastião Amaro, U. Lisboa & INESC-ID, Miguel Matos, U. Lisboa & INESC-ID and
Valerio Schiavoni, University of Neuchâtel

***Abstract*—The performance and behavior of distributed systems is highly influenced by network properties, *i.e.*, latency, bandwidth, packet loss, and jitter. When developing a distributed system, questions like, "how sensitive is the application's performance to network latency and bandwidth?" commonly arise. Answering these questions systematically and in a reproducible manner is very hard due to the variability and lack of control over the network. Moreover, state-of-the-art approaches are focused exclusively on the control plane, lack support for network dynamics or do not scale beyond a single machine or small cluster, which further aggravates this problem. KOLLAPS is a distributed, scalable, and efficient network emulator addressing these limitations by hinging on two observations. First, from an application's perspective, what matters are the emergent end-to-end properties (*e.g.*, latency, bandwidth, jitter) rather than the internal state of the routers and switches leading to those properties. Second, this model is amenable to decentralized management, allowing the emulation to scale with the number of machines required by the application. This premise allows for building a simpler, dynamic emulation model that does not require maintaining the full network state. KOLLAPS is agnostic of the application language and transport protocol, scales to thousands of application nodes, and is accurate when compared against a bare-metal deployment or state-of-the-art approaches that emulate the full network state. We use KOLLAPS to accurately reproduce results from the literature and predict the behavior of complex unmodified distributed systems under different network dynamics.**

***Index Terms*—Network Emulation, Orchestration, Distributed Systems, Large Scale**

I. INTRODUCTION

Evaluating large-scale distributed systems is hard, slow, and expensive. This difficulty stems from the large number of moving parts to control at once: system dependencies and libraries, heterogeneity of the target environment, network variability and dynamics, among others.

Such an uncontrollable and poorly specified environment leads to slow experimental cycles, hardly reproducible results, and potential downstream costs worth millions of dollars [1]. It is therefore of the utmost importance to have tools that allow one to precisely describe the environment and control key parts of the system such as the network.

On the one hand, the advent of container technology (*e.g.*, Docker [2], Linux LXC [3]) and container orchestration (*e.g.*, Docker Swarm [4], Kubernetes [5]) greatly simplifies the description and deployment of complex systems and partially addresses the problem. On the other hand, there is an acute need for tools that allow to precisely control the network in complex, large-scale experiments. As a matter of fact, the inherent variability of WAN conditions (*i.e.*, failures, contention and reconfigurations), makes it hard to assess the impact of changes in the application logic. Consider Figure 1: it shows the average latency between six AWS [6] regions over a large time window measured by Cloudping[1]. Even within the infrastructure of a major cloud provider, there are significant and unpredictable variations in latency (*e.g.*, high jitter). Therefore, when developing and evaluating an application, how can a developer be sure whether the observed performance are due to improvements in the application, or rather to a *lucky* run when the network was in favourable conditions? How is performance affected by common network dynamics, such as background traffic, or link flapping? The very same questions and issues also arise in the reproducibility crisis currently plaguing the system's community [7], [8]. Different results for the same system emerge not only because systems are evaluated in different uncontrollable conditions, but also because research testbeds such as Emulab [9], CloudLab [10], PlanetLab [11], or EdgeNet [12] used to conduct experiments tend to get overloaded right before system conference deadlines [13].

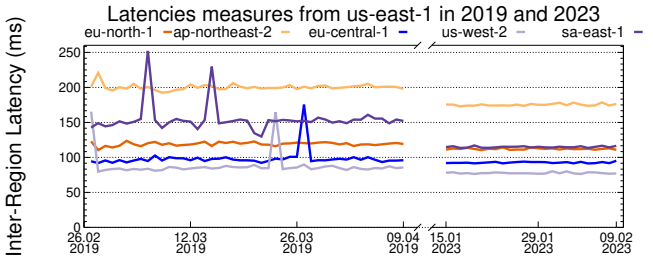

Fig. 1: Latency variability between five different AWS regions across the world. We consider 45 days in 2019, and a more recent time window of 24 days in 2023. Latencies vary on average between 90ms and 250ms, while spikes occur across all regions.

We therefore need tools to systematically assess and reproduce the evaluation of large-scale applications. One approach to systematically evaluate a large-scale distributed system is to resort to simulation, which relies on models that capture the key properties of the target system and environment [14]. Simulation provides full control of the system and environment — achieving full reproducibility — and allows to study the model of the system in a variety of scenarios. However, simulations suffer from several well-known problems, as there is a large gap between the simulated model and the real-world deployment, usually leading to several unforeseen be-

[1] https://www.cloudping.info

haviors not captured by the model [15]–[18]. Moreover, even if the simulated model is accurate, the real implementation is not guaranteed to faithfully follow it. Moreover, despite some efforts to model complex systems either through formal method analysis [19] or simulation, this is, to the best of our knowledge, seldom the case for large-scale systems.

The alternative is to resort to network emulation. As such, the real system is run against a model of the network that replicates real-world behavior by modeling a network topology together with its network elements, including switches, routers, and their internal behavior. Emulation has been used for decades [20] to help reason about the behavior of the real system in a concrete scenario rather than its model. Unfortunately, state-of-the art network emulators suffer from several limitations. MiniNet [21] is limited to a single physical machine and therefore cannot be used to emulate a large-scale resource-intensive system. MiniNet-Hifi [22] addressed the high resource usage of MiniNet, however, it is still limited to one machine. MaxiNet [23], the multi-host version of MiniNet supports distributed clusters but scales poorly. ModelNet [24] and alike rely on a dedicated cluster of nodes to maintain the emulation model to which the application nodes must connect. However, accuracy is *highly* dependent on application traffic patterns and can quickly degrade with a modest increase in the number of application nodes [25]. CrystalNet [26] accurately emulates the *control-plane* of large-scale networks (*e.g.*, routing tables, software switch versions, or device firmwares) but cannot be used to emulate the data-plane (*e.g.*, latency, bandwidth), and hence evaluate the behavior of large-scale distributed applications. SEED [27] is an Internet emulator aiming to emulate the fundamental infrastructure components of the Internet such as autonomous systems, BGP routing, and others. However, and similarly to CrystalNet, SEED lacks support to emulate network properties such as bandwidth or latency and does not provide dynamic behaviors. While emulation testbeds (*e.g.*, Emulab [9]) provide a semi-controlled environment and network, they cannot model network dynamics and thus one cannot assess their impact on application behavior. In summary, existing approaches are focused exclusively on the control plane, lack support for network dynamics or do not scale beyond a small cluster.

In this paper, we detail the design, implementation and evaluation of KOLLAPS, a decentralized, scalable and efficient network emulator. KOLLAPS overcomes the limitations of state-of-the-art solutions through two key insights. First, from the perspective of a distributed application, the observable end-to-end properties, *e.g.*, latency, jitter, bandwidth and packet loss, are more relevant to its behavior than the underlying state of each networking element leading to these properties. This enables us to build a simplified model that does not require emulating the full-state of the internal network elements (*e.g.*, routers, switches), while providing equivalent behavior. Second, it is possible to accurately maintain this emulation model in a fully-distributed fashion thus allowing the emulation to scale with the application nodes without sacrificing accuracy. Moreover, the simplified model lends itself to quick changes, enabling the emulation of dynamic events such as link removals and additions or background traffic in a timely manner (*i.e.*, milliseconds).

Finally, KOLLAPS can run in two modes: *managed* and *unmanaged*. In the former, KOLLAPS controls the whole lifecycle of the application under test, similarly to the other state-of-the-art tools (*i.e.*, KOLLAPS deploys and starts the application, runs the experiment, and so on). In the latter, KOLLAPS can be attached to an already running system and bring its emulation capabilities to it. This allows, for instance, to subject an already running system to controlled network faults (*e.g.*, packet loss) or bandwidth constraints, and opens the door to a wider range of experiments. Overall, KOLLAPS is agnostic of the implementation language, transport protocol and supports multiple (heterogeneous) deployment units (*i.e.*, container, virtual machine or physical machine) within the same experiment. For clarity, and when it is not clear from the context, we use the term application node, or *node* for short, to refer to an application running in a single deployment unit in the emulated network, and we use the term *machine* to refer to the physical hardware host, possible part of a cluster, which supports the experiment.

Previous Work. This paper extends an earlier version of Kollaps [54] with a more efficient and scalable architecture and implementation, and adds support for the unmanaged mode. We discuss the improvements in more detail in §VII.

Contributions. Our main contributions are:

1) KOLLAPS, a network topology emulator that enables the evaluation of large-scale applications in dynamic networks
2) We integrate KOLLAPS with Docker Swarm [4] and Kubernetes [5], to deploy and evaluate unmodified containerized (distributed) systems, and showcase both managed and unmanaged deployment modes;
3) A comparison of KOLLAPS's emulation accuracy versus bare-metal deployments and state-of-the-art approaches. Our evaluation scenarios include static and dynamic topologies, various workload patterns (*i.e.*, short and long-lived data flows), and different TCP congestion models (*i.e.*, TCP Reno and Cubic);
4) A showcase of the new types of experiments that KOLLAPS enables. Namely, we reproduce results from published papers [55], [56], and assess how Apache Cassandra [57], [58] is affected by different network characteristics *as-if* it were deployed on AWS;

Roadmap. §II provides background information on network emulation and also on the key technologies used in KOLLAPS's design. Related work is surveyed in §III. The design and system architecture of KOLLAPS are described in §IV, with implementation details presented in §V. §VI present our in-depth evaluation, §VII discusses some lessons learned, §VIII the limitations, and finally §IX concludes the paper.

II. BACKGROUND

We describe here the key enabling technologies used by KOLLAPS to run and maintain an accurate emulation. KOLLAPS can be deployed in different deployment units, *i.e.*, physical machines, virtual machines and containers, and with different orchestrators, *i.e.*, Docker Swarm [4] and Kubernetes [5]. To abstract the disparities between these deployment units, our approach relies exclusively on their common

TABLE I: Classification of network emulation tools. NetEm [28] uses a different queueing discipline to implement bandwidth shaping. Dockemu and DockSDN use ns-3 [29] for link-level features. For tools with multiple papers, we put the year of the first and the features of the most recent one. ↯: ability to dynamically change this property. P=physical machine, V=virtual machine, C=container.

Name	Year	Mode	HW ind.	Orch	Concurrent deployments	Path congestion	Link-Level emulation capabilities				App. Agnostic	Topology dynamics	Unit	Type
							BW	Delay	Packet loss	Jitter				
DelayLine [30]	1994	User	✓	Central	✗	✗	✗	✓	✓	✗	✓	✗	P	OTHER
Dummynet [31], [32]	1997	Kernel	✓	Central	✗	✓	✓↯	✓↯	✓↯	✗	✓	✗	P	GRAPH
Emulab [9], [33]	2002	Kernel	✗	Central	✗	✓	✓↯	✓↯	✓↯	✗	✓	✓	V	SW+RT
ModelNet [25]	2002	Kernel	✓	Central	✗	✓	✓↯	✓↯	✓↯	✗	✓	✓	P	GRAPH
Nist NET [34]	2003	Kernel	✓	Central	✗	✗	✓↯	✓↯	✓↯	✓↯	✓	✗	P	GRAPH
NetEm [28]	2005	Kernel	✓	Central	✗	✗	✗	✓	✓	✓	✓	✗	P	SINGLE
Trickle [35]	2005	User	✓	Decentral	✗	✗	✓	✓	✓	✗	✓	✗	P	SINGLE
EmuSocket [36]	2006	User	✓	Central	✗	✗	✓	✓↯	✓↯	✗	✗	✓	P	GRAPH
ACIM/FlexLab [37]	2007	Kernel	✓	Central	✗	✓	✓↯	✓↯	✓↯	✓↯	✓	✓	V	GRAPH
NCTUns [38]	2007	Kernel	✓	Central	✗	✓	✓	✓	✓	✓	✓	✗	P	OTHER
IMUNES [39]	2008	Kernel	✗	Decentral	✗	✗	✓	✓	✓	✗	✓	✗	P	OTHER
MyP2P-World [40]	2008	User	✓	Central	✗	✗	✓	✓	✗	✗	✗	✗	P	OTHER
P2PLab [41]	2008	Kernel	✓	Decentral	✗	✗	✓	✓	✓	✗	✗	✗	P	SINGLE
CORE [42]	2008	Kernel	✓	Decentral	✗	✗	✓↯	✓↯	✓↯	✗	✓	✓	P	GRAPH
DFS [43]	2009	User	✓	Central	✓	✗	✓↯	✓↯	✓	✗	✗	✓	P	OTHER
Mininet [21], [22]	2010	Kernel	✓	Central	✗	✓	✓↯	✓↯	✓↯	✓↯	✓	✓	P	SW+RT
SliceTime [44]	2011	Kernel	✗	Central	✗	✓	✓	✓	✗	✗	✓	✓	V	GRAPH
SPLAYNET [45]	2013	User	✓	Decentral	✓	✓	✓	✓	✓	✗	✗	✓	P	GRAPH
Distem [46]	2013	Kernel	✓	Central	✓	✗	✓	✓	✗	✗	✓	✗	C	GRAPH
MaxiNet [23]	2014	Kernel	✓	Central	✗	✓	✓↯	✓↯	✓↯	✓↯	✓	✓	P	SW+RT
EvalBox [47]	2015	User	✓	Central	✗	✗	✓	✓	✗	✗	✓	✓	P	OTHER
Dockemu [48], [49]	2015	User	✓	Central	✗	✗	✓	✓	✓	✓	✓	✗	C	OTHER
ContainerNet [50]	2016	Kernel	✓	Central	✗	✓	✓↯	✓↯	✓↯	✓↯	✓	✓	V,C	SW+RT
NEeaaS [51]	2020	Kernel	✓	Decentral	✗	✗	✓	✓	✗	✓	✓	✗	V,C	SW+RT
DockSDN [52]	2021	Kernel	✓	Decentral	✗	✗	✓	✓	✓	✓	✗	✗	V,C	SW+RT
Testground [53]	2022	Kernel	✓	Central	✓	✗	✓↯	✓↯	✓↯	✓↯	✗	✓↯	P,C	GRAPH
KOLLAPS	**2024**	**Kernel**	✓	**Decentral**	✓	✓	✓↯	✓↯	✓↯	✓↯	✓	✓	**P,V,C**	GRAPH

denominator *i.e.*, the Linux kernel. Next, we briefly introduce the two technologies that make this possible.

Linux TC provides users with a set of traffic control (TC) functions to manage the transmission of packets. Before sending a packet to the network interface, Linux places the packet into one of multiple possible queues, called queueing disciplines or `qdisc` for short.

The kernel offers several `qdisc` by default - we briefly discuss the ones relevant for our work. The *Hierarchy Token Bucket* (`HTB`) [59] is responsible for controlling outbound bandwidth on a given link. The *Priority* `qdisc` (`prio`) [60] dequeues packets in order of their priority, packets with higher priority are dequeued before packets with lower priority. Finally, the *Network Emulator* `qdisc` (`netem`) [28] allows the introduction of delay, loss, duplication, and other characteristics in 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` associated to it. The most relevant filter in the context of network emulation is the `u32` filter [61] that allows matching traffic against any field in a network packet. The `qdiscs` and filters can be managed through the `tc` command [62].

Extended Berkeley Packet Filter (`eBPF`) [63] is a highly flexible and efficient construct in the Linux kernel that allows the execution of bytecode at various hook points, such as system calls, in a safe manner. Systems use `eBPF` in many kernel subsystems, most commonly in networking, tracing, and security.

Even though `eBPF` has its origins in packet filtering, the current instruction set is generic and flexible enough to support many use cases. `eBPF` also provides an infrastructure and utilities around its instruction set such as maps with efficient put/get operations, general-purpose helper functions (such as getting the current time), tail calls to call another `eBPF` program, a pseudo-file system to pin objects such as maps and programs, and an infrastructure that allows offloading an `eBPF` programs to a network device.

In the context of KOLLAPS, we leverage the `SO_ATTACH_BPF` function [64] to monitor the application traffic. Every time the kernel receives a packet, it populates a structure (`sk_buff`) with metadata about the packet, such as origin and destination, or packet size. As we detail later in §V this data is used to populate our model. The key takeaway is that because eBPF executes in kernel mode it enables lightweight fine-grained monitoring.

III. RELATED WORK

We categorize network emulators along several dimensions (Table I): where the link shaping is executed (user/kernel mode), independence from the underlying hardware, type of orchestration across the cluster (centralized or decentralized), support for concurrent experiments and users, support for path congestion (*i.e.*, multiple independent flows sharing the same emulated link), link-level emulation features (bandwidth, delay, packet loss, jitter), implementation-language restrictions for the programs under emulation, ability to dynamically adjust network properties on the fly as well as to change the topology itself (*i.e.*, add/remove links, switches and nodes), and the supported orchestration unit (physical machines, virtual machines, or containers). We also categorize these tools into four different types: single-link emulators (`SINGLE`), Graph of Queues (`GRAPH`), tools emulating full network stack including routers and switches (`SW+RT`), and `other` approaches not fitting into any of these criteria. Due to space limitations, we only detail how KOLLAPS

compares with some representative systems, and we consider simulation-based tools (*e.g.*, ns-3 [29], PeerSim [65]) out of the scope. A few recent systems cover orthogonal aspects of network emulation and illustrate the relevance of controlled experiments. CrystalNet [26] focuses on large-scale emulation of the control-plane, enabling network engineers to evaluate changes to the control-plane before production deployment. Other control-plane approaches [66] are orthogonal to this work, focusing only on network conditions. Kathara [67] focuses on data-plane as well but not on emulating network conditions. KOLLAPS only deals with data-plane, and it is hence complementary to CrystalNet. To enhance the efficiency of large-scale control-plane analysis, Bonsai [68] leveraged the idea of network compression while persevering the network properties. KOLLAPS' network collapsing achieves a similar goal, however, it targets a different set of network properties, again in the data-plane. Pantheon [69] is used to evaluate Internet congestion-protocols. It gathers ground-truth data and compares it with results obtained from several emulators for a variety of congestion control algorithms. Pantheon demonstrates that it is possible to approximate the behavior of a wide range of congestion algorithms by relying only on a small number of end-to-end properties. We rely on the same insight to provide a network emulator able to emulate large-scale topologies with accuracy.

There are other emulation systems dedicated to specific scenarios. Celestial [70] is a virtual testbed for the LEO (Low Earth orbit) satellites. It efficiently emulates individual satellites and their movement as well as ground station servers with realistic network conditions and in an application-agnostic manner. Instead, we focus on general-purpose network emulators that can be used in any scenario.

User-space approaches. Trickle [35] uses dynamic linking and preloading functionality of Unix-based systems to insert its code between unmodified binaries and the system calls to the sockets API. It performs bandwidth shaping and delay before delegating to the actual underlying socket calls, based on a simple configuration process. Although multiple instances of Trickle can cooperate, setting up a multi-physical machine system to emulate large networks involves tedious and error-prone manual configuration, given the lack of central deployment control system. Further, Trickle does not support statically linked binaries. In contrast, KOLLAPS is independent of the application as it works with unmodified binaries, either dynamically or statically linked. EmuSocket [36] and DelayLine [30] are user-space tools, similar in design and features to Trickle. DelayLine supports the deployment of complex topologies, but it lacks several important network emulation features (such as bandwidth or jitter).

MyP2P-World [40] is a Java-based application-level emulator aimed at peer-to-peer protocols. Applications must be implemented in Java and rely on Apache Mina [71] to intercept and emulate large-scale network conditions. While KOLLAPS can be used for Java applications, it can be used for systems implemented in any other language.

SPLAYNET [45] extends SPLAY [72] to allow emulation of arbitrary network topologies, deployed across several physical machines in a fully decentralized manner. SPLAYNET, is fully distributed as it does not rely on dedicated processes for network emulation. To emulate the network topology, SPLAYNET relies on graph analysis and distributed emulation algorithms, effectively collapsing the inner topology and delivering packets directly from one emulated process to the destination process. However, it requires developers to implement their programs in a Domain Specific Language using the Splay framework and the Lua programming language, precluding its usage to evaluate real-world and legacy systems. Moreover, it does not support dynamics nor does it emulate packet loss upon congestion. KOLLAPS adopts a similar fully decentralized approach while completely overcoming its limitations. In fact, KOLLAPS can be used with unmodified, off-the-shelf applications and assess their performances under different network conditions.

Kernel-space approaches. Next, we survey network emulators that require explicit or specialized support from the underlying OS and kernel. Dummynet [73] operates directly on a specific network interface. It was used as a low-level tool to build full-fledged emulators, such as Modelnet [25]. Modelnet allows the deployment of unmodified applications. Applications are deployed on *edge* nodes and all network traffic is routed through a set of *core routers* - dedicated machines that collectively emulate the properties of the desired target network before relaying the packets back to the destination's edge nodes. KOLLAPS relies on Linux's Traffic Control (`tc`) [62] to offer similar low-level traffic shaping features, but *(i)* without requiring dedicated physical machines and *(ii)* at the same time providing a complete testbed integrating with large-scale container orchestration tools.

The Emulab [33] testbed supports the deployment of user-provided operating systems. As ModelNet and KOLLAPS, it leverages Linux's `tc` to shape the traffic directly at the edge nodes. Emulab supports large topologies over shared clusters while maintaining the user requested resource allocation, and the ability to perform this scheduling optimally. Its graph coarsening technique is similar in principle to KOLLAPS approach for collapsing the topology.

Container-based approaches. Finally, we look at emulation tools used with containers. Mininet [22] emulates network topologies on a single physical machine. It relies on Linux's lightweight virtualisation mechanisms (*i.e.*, `cgroups`) to emulate separated nodes. Similarly to Docker, it creates virtual Ethernet pairs running in separated namespaces and it assigns processes to those. Mininet can emulate hundreds of nodes (instances) on a single physical machine, with dedicated instances for switches and routers running on their own processes. Conversely, KOLLAPS does not require these additional network instances, relying instead on maintaining and updating the state of the emulation at each container. Mininet is limited to a single physical machine deployment hence preventing its use for large-scale resource-intensive applications that cannot fit in a single machine. Besides, even with a simple topology, Mininet's accuracy quickly degrades under certain workloads such as short-flows. Maxinet [23] extends Mininet to allow for cluster deployments of worker physical machines with native support for Docker containers. It does so by tunneling links that cross different workers. However, it requires all emulated

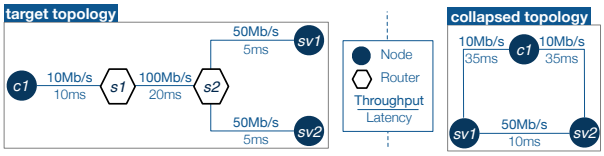

Fig. 2: Left: deployment with one client (*c1*), two servers (*sv1*, *sv2*) and three network elements (*s1*, *s2*). Right: collapsed topology with links describing maximum bandwidth and minimum latency between each two nodes.

nodes that connect to the same switch to be deployed on the same worker as the switch. In contrast, KOLLAPS does not impose co-located deployments of workers and switches. ContainerNet [50], [74] extends Mininet to add native support for Docker containers and dynamic topologies. Still, it is limited to single machine deployments. A similar limitation is present in Dockemu [75], a network emulation tool based on Docker containers. Dockemu 2.0 [49] improved upon its predecessor, however, this limitation remains.

DockSDN [52] is a network emulator with the goal of creating a software-defined network testbed. It relies on Docker containers to emulate virtual nodes, controllers, and switches which enables lightweight and multi-platform deployments. According to DockSDN's evaluation, it has better resource utilization than Mininet without hindering the performance of the emulated network. However, it lacks support for jitter, packet loss and path congestion emulation and does not support network dynamics.

Testground [53] is a platform for testing, benchmarking, and simulating distributed systems. Like SPLAYNET, Testground provides developers with a specific API that must be used for experiment description and orchestration. The experiments are described in an imperative way, using the Go programming language, rather than in a declarative way as in KOLLAPS and other systems such as ModelNet. Moreover, Testground lacks support for path congestion and link dynamics.

NEaaS [51] is a cloud-based network emulation platform aiming at providing users with a Network Emulation as a Service (NEaaS). NEaaS can be used on both public and private clouds and has support for Docker containers. However, NEaaS lacks support for topology dynamics and its accuracy with large-scale scenarios is unclear.

To the best of our knowledge, KOLLAPS is the only system that can be used to evaluate unmodified large-scale applications over arbitrary topologies, supporting a richer set of emulation features, and providing good accuracy when compared to bare metal and state-of-the-art systems. While some existing approaches (in particular ModelNet [25] and ContainerNet [50]) show characteristics similar to KOLLAPS, they do either fail to meet the criteria of scalability, supporting multiple physical machines or supporting path congestion, two fundamental features that prevents the adoption of such tools for modern large-scale distribute systems. Finally, it is worth noting that the decentralized design of KOLLAPS' metadata exchange is similar in spirit to the hose model of ElasticSwitch [76], which was used to provide bandwidth guarantees for virtual machines in cloud environments.

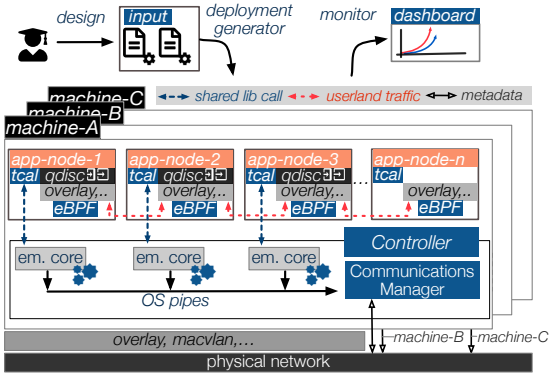

Fig. 3: Kollaps architecture

IV. THE KOLLAPS SYSTEM

We now describe the design of KOLLAPS which, as we discussed earlier, rests on two key insights. The first insight is that from an application's standpoint, what matters are the end-to-end network properties such as bandwidth or latency, rather than the underlying state of routers and switches leading to these properties. This enables us to forego the need to emulate the full state of the internal network elements (*e.g.*, routers, switches) and hence build a leaner and simplified emulation model. Second, this leaner emulation model is amenable to distribution which allows the emulation to scale with the application nodes without sacrificing accuracy. To illustrate this, consider the topology in Figure 2 (left), with two network elements, switch *s1* and *s2*, and three nodes, client *c1* and servers *sv1*, and *sv2*. Rather than emulating the full network and the state of the switches (*s1* and *s2*), we rely on a *network collapsing* technique to collapse the topology to virtual end-to-end links that retain the properties of the original topology, as depicted in Figure 2 (right). Note that the nominal bandwidth and latency in the collapsed network correspond to the maximum available bandwidth and the minimum achievable latency between each two nodes, as if it was the only active flow. The actual link properties are then maintained through a *distributed network emulation* algorithm that models latency, bandwidth, jitter and packet loss. The distributed nature of KOLLAPS and its simplified emulation model allows it to scale to thousands of nodes with accuracy on par with centralized state-of-the-art emulators. Next, in §IV-A we present an overview of KOLLAPS's architecture and then in §IV-B we discuss how the distributed network emulation works.

A. Overview

Figure 3 depicts KOLLAPS's architecture in a deployment over several machines. KOLLAPS consists of several components. The Emulation Core has a single instance per application node and is responsible for maintaining and enforcing the emulation model. The Communications Manager has a single instance per machine and is responsible for sharing metadata among all the Emulation Cores across the system. The TC Abstraction Layer (TCAL) is deployed once per application node, and it retrieves and sets the link properties.

The Dashboard exposes a web-based interface to monitor and control the experiments. Finally, the Deployment Generator is a command line tool that transforms an experiment description (*e.g.*, Listing 1 and Listing 2) into a deployment plan. Next, we detail these components.

The **Deployment Generator** translates a topology description into an actual deployment plan. KOLLAPS supports an XML Modelnet-like syntax [25] to facilitate porting of existing topology descriptions, as well as a YAML-based syntax that we show here. Listing 1 describes the network topology from Figure 2 (left). The topology description language supports `services`, `bridges`, `links`, and `dynamic` elements. For managed deployments, the `services` correspond to sets of containers sharing the same image. The image names must be valid and available from private or public Docker registries (Listing 1, lines 4 and 6). Each service supports several parameters (*e.g.*, total replicas or additional parameters to pass to the running containers once deployed). For unmanaged deployments, the service specifies the machine details such as IP, the paths to the KOLLAPS installation and the topology file.

The `bridges` map to networking devices, *e.g.*, routers and switches that have unique names (lines 9–10) and are arbitrarily connected, to realize complex topologies, via `links`. Links can be uni– or bi—directional, with mandatory attributes to specify the source, destination, and the network properties (*i.e.*, latency, bandwidth, packet loss, jitter, lines 12–17). In case of jitter, the link latency follows by default a normal distribution but others are supported, with mean and standard deviation to match the specified latency and jitter attributes. Internally, all links are unidirectional: declaring a bi-directional link results in the creation of two identical links in opposite directions, with same attributes except for the bandwidth capacity where upload and download attributes might differ. The `dynamic` part (Listing 2) injects changes into the topology dynamically while the experiment progresses. KOLLAPS supports several dynamic events, namely modification of any of the properties of the links, addition and removal of links, bridges and services. This captures a wide range of dynamics, not only in the application itself, whose nodes may come and go during the experiment, but also in the network topology. For example, the rapid removal and insertion back into the topology of a link emulates a flapping link [77]. Each event maps to an `action` element, either for changes to link properties (lines 21–23) or for addition and removal of services, links and bridges (lines 24–36). A dedicated DSL to simplify the programming of more complex dynamics is detailed in [78].

The **Emulation Core** (EC) is the main component of KOLLAPS. Each instance is responsible for emulating end-to-end network properties. Since KOLLAPS does not directly emulate network elements nor their internal state, we must accurately describe the topology at the application nodes. This is achieved as follows. We start by parsing the topology description (*e.g.*, Listing 1) into a graph structure, maintained throughout the emulation. Next, we compute the shortest paths between every pair of reachable nodes. Each shortest path is composed of several links, whose properties are used to determine the end-to-end network properties. We further detail how these properties are computed in §IV-B.

Listing 1: Static topology.

```
1  experiment:
2   services:
3    name: c1
4     image: "iperf"
5    name: sv
6     image: "nginx"
7      replicas: 2
8   bridges:
9    name: s1
10   name: s2
11  links:
12   orig: c1
13    dest: s1
14    latency: 10
15    up: 10Mbps
16    down: 10Mbps
17    jitter: 0.25
18 #others not shown
```

Listing 2: Dynamic events.

```
19 dynamic:
20  orig: c1
21   dest: s1
22   jitter: 0.5
23   time: 120
24  action: leave
25   name: s1
26   time: 200
27  action: join
28   orig: c1
29   dest: s2
30   up: 100Mbps
31   down: 100Mbps
32   latency: 10
33   time: 210
34  action: leave
35   name: sv
36   time: 240
```

Dynamic Topologies. The Emulation Core also enforces the dynamic features of the topology. The `dynamic` topology elements are reflected by modifications to the graph structure discussed above. Rather than computing modifications to the graph on the fly while the experiment executes, we precompute offline all the modifications before the experiment starts, as an ordered sequence of graphs. We resort to this approach because while computing all the required metadata (*e.g.*, all-pairs shortest paths, end-to-end properties, *etc.*) is fast for small graphs (*e.g.*, few milliseconds), for large graphs with thousands of nodes it requires several seconds, precluding accurate emulation of sub-second dynamics.

The **Communications Manager** (CM) is responsible for sharing metadata across all the ECs in the deployment. In KOLLAPS, ECs can either be in the same machine or in different machines therefore the CM has two responsibilities. The first, is intra-machine communication, ensuring the sharing of metadata among ECs in the same machine. To achieve it, the CM will start two pipes for each EC running in the same machine, one pipe to read metadata related to that EC and another pipe to write data related to all the other ECs. Secondly, the CM is also responsible for inter-machine communication guaranteeing the sharing of metadata among ECs in different machines. This is achieved by starting a socket for each other CM in the deployment. This way, the CM guarantees that all the ECs observe the same metadata regarding the flows in the system, used to update the emulation model.

The **TC Abstraction Layer** (TCAL) interfaces with Linux's Traffic Control (§II) and is used to enforce the emulation constraints. For each destination, KOLLAPS creates a `htb qdisc` that enforces the bandwidth allocated to flows towards that destination. To enforce latency, jitter, and packet loss KOLLAPS relies on a `netem qdisc`.

The **Dashboard** allows users to control and monitor the progress of their experiments via a graphical web-based interface (not shown). This dashboard shows a graph-based representation of the emulated topology, the status of the

services, ongoing traffic, and dynamic events. In unmanaged deployments, the Dashboard also provides a command to start and stop the emulation, allowing the developers to control their experiments.

Finally, the **Controller** component has a dual role depending on the deployment mode. In unmanaged deployments, the Controller is responsible for receiving commands from the Dashboard via `ssh`, and relaying these commands, through TCP sockets to every EC in the system. In managed deployments, the Controller is responsible for interfacing with the containers and starting the Emulation Core and the Dashboard in the respective containers.

B. Emulation Model

We now discuss how KOLLAPS's emulation model maintains the end-to-end network properties. Formally, given a path $\mathcal{P}$ composed of links $\mathcal{P} = \{l_1, l_2, \ldots, l_n\}$, its end-to-end properties can be computed as follows:

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

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

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

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

For latency, packet loss and jitter (assuming a uniform distribution), it is enough to sum or multiply the properties of the links (variance for the jitter case).

Bandwidth requires more considerations though, because it is limited not only by the physical capacity of the path, but also by all active flows on each link. The maximum bandwidth in the path is determined by the link with the least bandwidth. However, the bandwidth allocated to each active flow depends on all active flows in the same path and thus it must be dynamically recomputed at runtime. Moreover, when the bandwidth required by each flow surpasses the maximum available bandwidth, the links become congested and therefore we need a mechanism to ensure a fair allocation of bandwidth among the competing flows. In a real deployment, when competing flows require more bandwidth than the available capacity, network elements such as routers and switches buffer packets to accommodate the excess load up to a point where the buffers overflow and packets are dropped. Unreliable transport protocols (*i.e.*, UDP) ignore packet loss but reliable transport protocols (*i.e.*, TCP) have congestion control mechanisms to adjust the throughput with the goal of allowing all competing flows to get a fair bandwidth share. In KOLLAPS, rather than modeling the internals of network elements, which is expensive, we rely instead on a model to compute a fair share of the bandwidth available for each competing flow. In particular, we leverage the RTT-Aware Min-Max model [79], [80], which gives a share to each flow that is proportional to its round-trip time and is inspired by TCP Reno [81], a widely adopted congestion control implementation. Formally, the fair share of a long-lived flow f is given by:

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

where $f \in \{f_1, f_2, \ldots, f_n\}$ are active flows on a link.

This bandwidth sharing model gives the percentage of the maximum bandwidth any flow is allowed to use at capacity. However, it does not guarantee that the available bandwidth on a link will be fully utilized, for instance when a given flow does not consume all its available share. Hence, when the sum of shares of all active flows is less than the maximum bandwidth on the link, a maximization step occurs, to increase the share of the other flows, proportionally to their original shares. Note that KOLLAPS enforces bandwidth sharing per destination, not per connection.

V. IMPLEMENTATION

KOLLAPS components are implemented in Python (v3.19), C and Rust(1.67.0), and available at https://kollaps.dev. For managed deployments, it requires a Docker daemon (v1.12) running on each machine. The Deployment Generator currently supports Docker Swarm (v1.12) and Kubernetes (v1.14) by generating Docker Compose or Kubernetes Manifest files, respectively. Before proceeding with the experiment deployment, users can adjust these configuration files as need, which is in fact required by many real applications [82]. For unmanaged deployments, it requires key-based `ssh` access to the machines, which must have KOLLAPS pre-installed.

A. Integration with container orchestrators

In addition to producing Compose/Manifest files, the KOLLAPS deployment toolchain must configure the following orchestrator-dependent resources: *(1)* access to the orchestrator APIs, used at runtime for name resolution, *(2)* the topology descriptor file, read by each Emulation Core instance to setup the initial network state and compute the graph of the dynamic changes, and *(3)* the setup of multiple virtual networks.

Privileged bootstrapping. In order for an application running inside a Docker container to use `tc` (as the TCAL does), it must be executed with `CAP_NET_ADMIN` capability [83]. Although Docker allows executing applications in standalone containers with user-specified capabilities, this feature is currently unavailable for Docker Swarm. We circumvent this limitation as follows. We deploy a bootstrapping container on every Swarm node (not shown in Figure 3) whose job is to launch, on that machine and outside Swarm itself, the Controller, as depicted in Figure 3. The Controller shares the PID namespace with the host and has elevated privileges. It has access to the local Docker daemon and monitors the local creation of new containers. Upon the creation of a new container, the Controller launches an Emulation Core in the network namespace of the container which then becomes the responsible for that container, as discussed in §IV. We distinguish between containers whose network should be emulated by KOLLAPS and regular containers through a tag injected in the configuration by the Deployment Generator. We expect future releases of Docker Swarm to allow for a simplified mechanism. When using Kubernetes, such restrictions do not hold and the Controller is therefore deployed automatically.

B. Emulation Core and TCAL

The TCAL library provides an interface to setup the initial networking configuration and set the network restrictions dictated by the emulation model, namely bandwidth, latency, packet loss and jitter.

The Emulation Core execution is split into two stages: initialization and emulation loop. Once the initial graph representation is built, this component resolves the names of all services to obtain their IP addresses. In managed deployments it uses the internal Swarm discovering service or Kubernetes's API. In unmanaged deployments this step is skipped since we know at startup the IPs. Then, it runs an all-pairs shortest path graph traversal [84] between the local node and all the other reachable nodes. Finally, it computes the properties of the collapsed topology as described previously. The properties of the paths are then set up by the TCAL, before moving to the emulation loop stage. The emulation loop maintains a data structure with the bandwidth usage of each instance. It works by periodically executing the following steps: *(1)* obtain the bandwidth usage through `eBPF`; *(2)* disseminate the local bandwidth usage to the other nodes, *(3)* compute bandwidth usage on each path and its constituent links; *(4)* enforce bandwidth restrictions. The period is defined by the system parameter `ec_period`. Additionally, the bandwidth usage considered in step (1) is subject to an hysteresis factor controlled by the system parameter `max_age`. We choose the default value of `max_age=2`, although using `max_age=1` is fine for a lot of scenarios, in systems with very short and intermittent flows this could result in over-sensitivity in adapting the model to changes resulting in the model flip-flopping between two states. A `max_age` greater than one indicates that the model considers bandwidth reports from the previous emulation cycles, if no more recent value is available, which prevents unwanted switching. We study the impact of these parameters in §VI-D.

Monitoring. Outbound traffic is matched to `netem qdiscs` through an `u32` universal 32 bit [61] traffic control filter. The filter is a two-level hashtable that matches against the destination IP address of packets and directs them to their corresponding `netem qdisc`. This two-level design is due to limitations in the `u32`, which does not provide a real hashing mechanism (for speed reasons) but just a simple index in a 256 position array. With a `/16` netmask this could result in several collisions, degrading performance. We map the third octet of the IP address to the first level and the fourth octet to the second level of the hashtable, achieving constant lookup times. Traffic directed to the `netem qdisc` will first be subjected to the `netem` rules to enforce latency, jitter and packet loss. When packets are dequeued from `netem`, they are immediately queued in the parent `htb qdisc`, to enforce bandwidth restrictions.

Congestion Effects. The model calculates the maximum available bandwidth for each flow at any given time. While effective when bandwidth usage is below or at capacity, it produces unrealistic results when the cumulative bandwidth required by flows surpasses the maximum bandwidth capacity. This stems from a complex interaction between Linux TC and the implementation of congestion control algorithms in the kernel. In a real deployment with TCP, the protocol throttles its throughput dynamically by observing reported packet loss or delay when the buffers at network devices overflow. In contrast, UDP is insensitive to packet loss and simply continues to send packets at the application sending rate. A first approach to model packet loss due to congestion would be to dimension the buffers (queue sizes) in the TCAL following known network buffer sizing strategies [85], [86]. However, there are some differences between `tc` queues behavior and those found in a switch or router. Routers and switches drop further incoming packets when the buffers become full[2]. In the kernel, when the buffer becomes full `htb qdisc` back-pressures the application rather than dropping packets. This is because packet loss is done in `netem` [87] rather than in `htb qdiscs`(see §II), and also due to Linux's TCP Small Queues (TSQ) [88] (since kernel v3.6). TSQ reduces the number of packets in `qdiscs` and device queues with the goal of reducing the RTT, hence mitigating buffer-bloat. It tracks the amount of data waiting to be transmitted, and when this surpasses a given limit, the socket is throttled down. The impact of this is application-dependent, with blocking-IO applications blocks when writing to the socket, while with non-blocking I/O would applications observes zero bytes written.

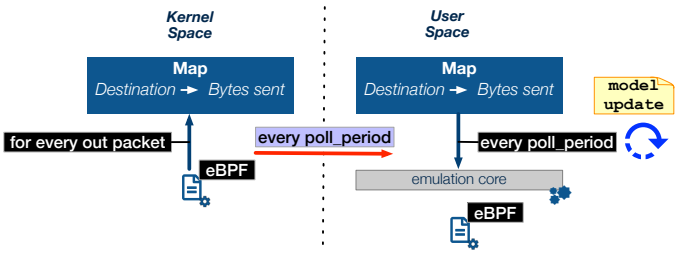

Fig. 4: Bandwidth retrieval using `eBPF` and `perf`.

C. Kernel Information Retrieval

For the emulation to be accurate, and adapt quickly to changes, it is crucial to have timely information about the ongoing network traffic. Our approach, depicted in Figure 4, leverages `eBPF` and `perf` and works as follows.

The information related to the flows of a node is a pair of IPs (source and destination addresses), and a number representing how many bytes were sent from the source to the destination. This allows us to represent the information related to flows as a map, where the key is the destination address, and the value is the number of bytes sent.

We use `eBPF`'s socket filter, and more precisely RedBPF [89] which allows attaching to a `raw_socket` [90] and filter all outgoing traffic of the node. Each time a node sends a packet, the `eBPF`'s socket filter is triggered and the map is updated. Since all of this is done in kernel space using `eBPF` it is extremely lightweight. To expose this information to userspace, so that it can be accessed by the Emulation Core, we use `perf_events`, which allows us to notify

[2]This is a simplification: in practice packets already in the queue can be dropped to allow for incoming traffic with higher priority.

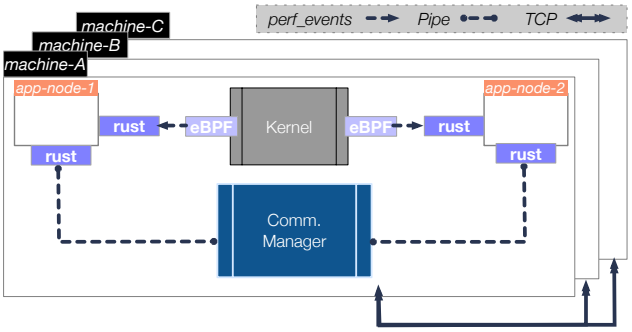

Fig. 5: Metadata dissemination flow.

userspace of events in kernel space. Specifically, we use a RedBPF PerfMap [91] and produce an event (*i.e.* update the map in userspace with the information from kernel space) every `ec_period`. Upon starting the Emulation Core spawns a dedicated thread to process the perf events and keep the map in userspace updated. The main advantage of our approach is that it does not require to constantly query the Kernel for information related to traffic. Instead, eBPF's socket filter only updates the userspace map every `ec_period` with new metadata, resulting in accurate monitoring with minimal overhead.

D. Metadata dissemination

The metadata about bandwidth usage mentioned in the previous section is disseminated across containers by the Communications Manager. The Communications Manager is implemented in Rust and consists of 817 SLOC. It uses sockets [92] and pipes created using `mkfifo` [93]. Upon startup, the Communications Managers waits for all of Emulation Cores to start. After this, it opens a socket and starts one thread for each other CM in the system. The metadata is received from the other CMs though the sockets and sent to the local ECs through pipes. This design reduces contention and synchronization which is key for efficiency. The flow of metadata in KOLLAPS is displayed in Figure 5.

VI. EVALUATION

We evaluated KOLLAPS through a series of micro- and macro-benchmark experiments in our cluster. Furthermore, to validate the soundness of our approach against realistic scenarios, we compare the behavior of applications running on Amazon EC2 and under KOLLAPS. Overall, our results show that: *(1)* KOLLAPS emulation accuracy is comparable with, and in some scenarios better than, tools that emulate the full network state such as Mininet; *(2)* KOLLAPS has low CPU and Memory usage in large-scale scenarios; *(3)* running an application with KOLLAPS in a cluster or in Amazon EC2 yields similar results; *(4)* KOLLAPS unmanaged deployments have comparable results to managed deployments.

The companion supplemental material provides additional experiments evaluating the emulation accuracy.

Evaluation settings. Our cluster is composed of 6 Dell PowerEdge R630 server machines, with 64-cores Intel Xeon E5-2683v4 clocked at 2.10 GHz CPU, 128 GB of RAM and connected by a Dell S6010-ON 40 GbE switch. The machines run Ubuntu Linux 22.04 LTS, kernel v5.15.0-60-generic. The `ec_period` and `max_age` are set to 50ms and 2, respectively, unless otherwise stated. We assess the impact of adjusting these parameters in §VI-D This configuration is used in all experiments, unless stated otherwise. The tests conducted on Amazon EC2 use `r4.16xlarge` instances, the closest type in terms of hardware-specs to the machines in our cluster. We used Docker Engine version 20.10.23, and the Docker overlay network driver. For comparison with other state-of-the-art tools we used Mininet (v2.22) and Maxinet (v1.2). We also conducted some experiments in IMEC Virtual Wall 1 [94] testbed, which we further describe in §VI-F3.

TABLE II: Bandwidth shaping accuracy for several emulated link capacities on a point-to-point client-server topology.

Link BW	KOLLAPS	Mininet	trickle (def.)	trickle (tuned)
Low (Kb/s)				
128 Kb/s	123 (-4%)	123 (-4%)	262 (+104%)	131 (+2%)
256 Kb/s	245 (-4%)	286 (+11%)	472 (+184%)	262 (+2%)
512 Kb/s	489 (-4%)	490 (-5%)	717 (+40%)	525 (+2%)
Mid (Mb/s)				
128 Mb/s	122 (-5%)	122 (-5%)	250 (+95%)	131 (+2%)
256 Mb/s	244 (-5%)	245 (-5%)	493 (-4%)	261 (+1%)
512 Mb/s	487 (-5%)	486 (-5%)	952 (+85%)	518 (+1%)
High (Gb/s)				
1 Gb/s	0.954 (-5%)	933 (-7%)	1.67 (+67%)	1.00 Gb/s
2 Gb/s	1.91 (-5%)	N/A	1.93 (-3%)	1.97 (-1.5%)
4 Gb/s	3.79 (-5%)	N/A	4.12 (+3%)	3.61 (-10%)

A. Bandwidth Emulation Accuracy

First we evaluate the accuracy of our bandwidth shaping mechanism under a topology that consists of two services running iPerf3 [95], connected by a single link. iPerf3 is a tool that measures the maximum bandwidth between its client and server instances. We use iPerf3 to assess the accuracy of KOLLAPS in emulating different target bandwidths, and compare the results with Mininet [22] and Trickle [35], a userspace bandwidth shaper. During the experiment, we run iPerf3 for 60 seconds, and report the average bandwidth in Table II. The values obtained with KOLLAPS and Mininet are similar since both systems rely on the `htb qdisc` to perform the bandwidth shaping. Mininet however does not allow imposing bandwidth limits greater than 1Gb/s. KOLLAPS does not impose that restriction and ensures the same level of accuracy of both systems at lower bandwidth rates ($\approx 95\%$). Results using the default Trickle settings deviate significantly from the specified bandwidth rates. After a more detailed investigation, we were forced to tune iPerf3 to use smaller TCP sending buffers to achieve accuracy comparable with the other systems.

B. Jitter Emulation Accuracy

Next, we evaluate the accuracy of jitter emulation. For this, we set up a sequence of experiments using the same topology of two nodes connected through a single link, with one sending 10,000 `ping` requests to the other. We assign the link different latency values according to the measured latencies between services deployed on `us-east-1` and other Amazon AWS

TABLE III: Jitter shaping accuracy for several emulated links with source at `us-east-1`.

→ Destination	Latency (ms)	EC2 Jitter (ms)	KOLLAPS Jitter (ms)
us-east-1	6	0.5607	0.6367
us-east-2	17	1.2411	1.4018
ca-central-1	24	1.2451	1.3872
us-west-1	70	1.3627	1.5438
eu-west-1	78	1.2000	1.3684
eu-west-2	85	1.6609	1.8592
eu-north-1	119	1.2850	1.4479
ap-northeast-1	170	1.4217	1.6031
ap-south-1	194	2.0233	2.2758
ap-northeast-2	200	1.8364	2.0888
ap-southeast-2	208	1.4277	1.6290
ap-southeast-1	249	1.2111	1.3728

regions Table III shows for each destination AWS region (2nd column), the measured latency and jitter values in the 3rd and 4th columns, respectively. On the right-most column, we present the jitter value emulated by KOLLAPS using the same latency. The overall mean squared error between the observed and emulated jitter is 0.2029 which is negligible for most practical purposes. While smaller errors could be achieved by directly controlling the network infrastructure on which KOLLAPS is deployed, this is beyond the scope of this work.

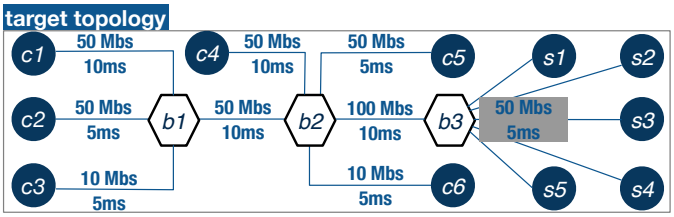

Fig. 6: Topology used to validate the decentralized bandwidth throttling in Fig. 7).

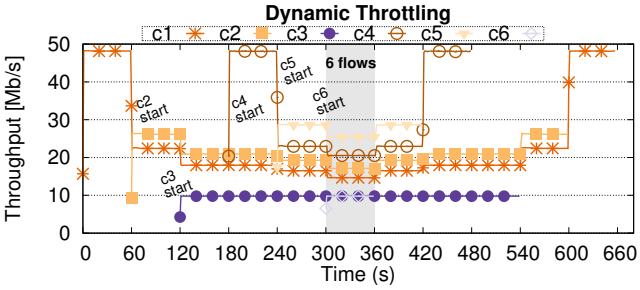

Fig. 7: Decentralized bandwidth throttling: several clients compete on a shared link. Each gets a different share of bandwidth, adjusted at runtime.

C. Decentralized Bandwidth Throttling

Next, we investigate the effectiveness of our bandwidth sharing model when the bandwidth requested by the application exceeds the available capacity. To assess this, we set the topology depicted in Figure 6. It consists of six clients (`C1 - C6`), three bridges (`B1 - B3`) and 6 servers (`S1 - S6`). The first three clients are connected to `B1` through links with bandwidths of 50, 50 and $10Mb/s$ and latencies of 10, 5 and $5ms$, respectively. The other three clients are connected to `B2` with the same links properties. All servers are linked to `B3` through equal links with $50Mb/s$ bandwidth and $5ms$ latency. Finally, `B1` is connected to `B2` by a $50Mb/s$ link with $10ms$ latency, and `B2` is connected to `B3` by a $100Mb/s$ link with $10ms$ latency.

Figure 7 shows the bandwidth of each of the established flows over time. We use iPerf3 to establish continuous TCP flows between clients and servers, while the experiment proceeds as follows. In the first half of the experiment we start each client sequentially in 60 second intervals. Initially, only `C1` has an active flow, and hence it uses all the available bandwidth. Upon starting `C2`, both clients will compete for the bandwidth over the shared links. At this point, since `C2` has a smaller RTT than `C1`, it gets a proportionally higher share of the bandwidth. Following the model in §IV, these shares are 23.08 Mb/s and 26.92 Mb/s, respectively. When `C3` starts, it will be allowed an equal share of the bandwidth to `C2`. However, `C3` is limited by a $10Mb/s$ link prior to the contended one. Consequently, the bandwidth share of the other two clients is increased proportionally to their original shares, resulting in 18.45, 21.55, and 10 Mb/s, respectively.

At 180 seconds, `C4` starts. It can reach $50Mb/s$ because the throughput of all other three clients is limited by the 50 Mb/s link connecting the bridges `B1` and `B2`. Hence, the link between `B2` and `B3` can accommodate all four clients. When `C5` starts, this is no longer the case. Now, all five clients are competing for the 100 Mb/s link. `C3` remains limited to 10 Mb/s, below its allowed share. The shares for all other clients is increased accordingly resulting in 16.89, 19.75, 10, 23.74, and 29.62 Mb/s, respectively. At 300 seconds, `C6` starts and, like `C3`, the maximum bandwidth it can use is lower than its given share. The expected bandwidths therefore become 15.04, 17.55, 10, 21.06, 26.33, and 10 Mb/s for clients `C1-C6`, respectively.

On the second half of the experiment (from $360s$ until the end) we sequentially shutdown the clients every $60s$ in the reverse order of arrival. Despite the decentralized emulation model, KOLLAPS is able to quickly adjust the bandwidth shares to the dynamic behavior of clients.

D. Impact of System Parameters

We now study the impact of the system parameters `ec_period` and `max_age`. The `ec_period` affects three aspects of the emulation. The first is CPU usage as smaller periods result in the execution of more emulation loops and hence more CPU usage. The second is accuracy - in scenarios with short or intermittent flows a smaller `ec_period` allows KOLLAPS to react faster to changes. The third one is related to metadata freshness. A small `ec_period` might result in not receiving new metadata from some of the Communication Managers since the last emulation loop. If we discard this data right away, this could result in unnecessary adaptations as explained in §V-B. The `max_age` counteracts this by considering older metadata if newer one is not available. Next we study the impact of these parameters on CPU usage and reaction time.

1) CPU Usage: To assess CPU usage we use a star topology (40Mbps upload and 40Mbps download bandwidth) with an increasing number of nodes where each node sends

TABLE IV: Impact of different `ec_periods` on CPU load.

#Active Flows	#Nodes	50 ms		25 ms		10 ms	
		usr	sys	usr	sys	usr	sys
20	5	7%	4%	8%	4%	9%	5%
42	7	11%	5%	12%	6%	14%	6%
90	10	20%	8%	24%	9%	30%	10%

traffic to all the others. This is the worst case for the Emulation Core from a resource usage perspective since there are a quadratic number of active flows. The experiment is deployed in a single machine and we measure CPU usage with `dstat`. We selected iPerf as the target application because it has a minimal CPU usage and therefore the observed usage is mostly due to KOLLAPS.

The results are reported in Table IV, and since the monitoring is done in kernel space through eBPF we show both `%usr` and `%sys` CPU usage.

As expected, smaller periods result in more CPU KOLLAPS uses, however, the increase is sublinear - for instance, changing the period from 50ms to 10ms for 10 nodes resulted in the total CPU usage only increasing from 28% to 40%. The default of 50ms has a low CPU usage, specially with lower number of application nodes per machine, and still has a reasonable reaction time for most scenarios we considered.

2) Reaction Time: We now assess how long KOLLAPS takes to react to changes in the bandwidth, and how this is affected by the `ec_period` and `max_age`. To to this we repeat the experiments of §VI-C and measure how long it takes for client `C1` to react to the appearance of client `C2` in the network. Note that both clients share a path to the servers and hence will contend for the available bandwidth. We ensure that the clients are running on different machines (and hence take into account the communication cost between the Communication Managers) and synchronized their clocks using the NTP protocol [96]. Then we measure the time since client `C2` sends its first metadata message (*i.e.*, after the Emulation Core becomes aware of the change), until client `C1` instructs TCAL to change its bandwidth.

For the default value of `ec_period=50ms` we observe an average reaction time of 54.5±2 ms and for an `ec_period=10ms` we observe an average reaction time of 7±3 ms. The results are explained by several factors. First, the periods of the Emulation Cores in both machines are not synchronized and hence we expect some variations in the results, up to an `ec_period`. Second, we have to account for KOLLAPS's overhead, namely the processing, sending and receiving of the metadata messages. Finally, we have the experimental errors due to NTP inaccuracies and concurrent traffic going on the cluster, which is shared. Nonetheless, the take away is that the reaction time in both scenarios is still below the 60ms round-trip time between the clients and servers.

E. Geo-replicated Systems

We turn our attention to macro-benchmarks to assess and motivate the behavior of KOLLAPS in real-world scenarios.

In this experiment, we reproduce results obtained for two Byzantine fault tolerant state machine replication libraries: BFT-SMaRt [56], and its optimized version Wheat [55]. The authors of these systems evaluate and compare them through a geo-distributed deployment on Amazon EC2 instances spanning 5 regions [55]. The experiment consists of placing one server and one client at each region, with servers running a simple replicated counter. Aside from the experimental results, the authors also provide the measured average latency and jitter between regions ([55], Table II), which we use to model a topology in KOLLAPS that mimics the one observed in their experiments.

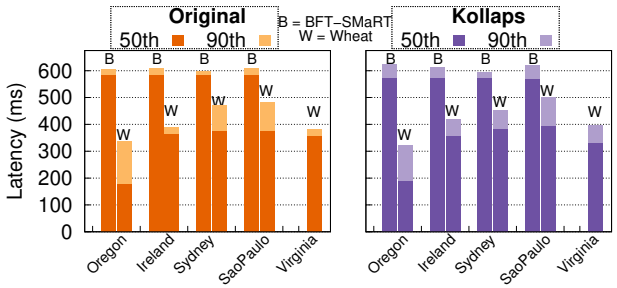

Fig. 8: Reproduction of an experiment with a geo-replicated deployment of BFT-SMaRt and Wheat. The experiment measures the latencies of clients located in different Amazon EC2 regions (left: results from [55], right: same experiments with KOLLAPS).

Figure 8 shows the results of the original experiment on EC2 (left), and using KOLLAPS (right). As we can observe, the results of executing the experiment in KOLLAPS are close to the results achieved by the authors on EC2, with a maximum difference of 7.3% observable between the 90th percentiles of the Wheat client in Ireland. BFT-SMaRt results were even closer, with a maximum difference of 2.7%.

We attribute the difference to the following. Since the authors only provide the average and standard deviation latency measurements, we assumed for the KOLLAPS experiments a normal jitter distribution. However, the Amazon EC2 `t1.micro` instances used by the authors in their experiments are prone to jittery behavior [97], potentially not following a normal distribution. This is relevant in particular to Wheat as its more latency sensitive than BFT-SMaRt and thus more affected by the jitter distribution.

F. Scalability

We now study KOLLAPS's scalability by analyzing the resource usage and emulation accuracy with increasing system sizes. We generate large-scale topologies using the preferential attachment algorithm [98]. This algorithm constructs scale-free networks, which are representative of the characteristics of Internet topologies. We consider topologies of the following sizes: 1000 (666 nodes, 334 bridges), 2000 (1344 nodes, 656 bridges), 4000 (2668 nodes, 1332 bridges) and 8000 (5336 nodes, 2664 bridges). The experiment consists of nodes sending ICMP echo requests (`ping`) to other random nodes for 10 minutes.

1) Resource Usage: The CPU usage was retrieved using dstat [99]. Since part of KOLLAPS's logic is on kernel space due to `eBPF`, we report both user (`usr`) and system (`sys`) CPU usage, *i.e.* the time spent in user and kernel mode

TABLE V: CPU and Memory usage per physical machine in large-scale topologies.

Topology size	#Nodes	#Switches	CPU usr	CPU sys	Memory Usage
1000	666	334	0%	0%	454MB
2000	1344	656	1%	0%	1072MB
4000	2668	1332	1%	1%	3095MB
8000	5336	2664	1%	1%	9768MB

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

Topology size	#Nodes	#Switches	KOLLAPS	Mininet	Maxinet
1000	666	334	0.0683	0.0079	28.0779
2000	1344	656	0.0770	NA	347.5303
4000	2668	1332	0.0813	NA	NA
8000	5366	2664	0.1110	NA	NA

respectively. The results are depicted in Table V and, as it is possible to observe, KOLLAPS's CPU usage in this scenario is negligible even in large-scale topologies.

The results for memory usage are also reported in Table V. Overall, the memory usage per node is very small, allowing KOLLAPS to scale to large topologies.

2) Emulation Accuracy: We now compare KOLLAPS with Mininet [22] and Maxinet [23] in large-scale topologies. To assess the accuracy of the emulation with increasing system sizes we compare the reported round-trip times (`RTT`) obtained with ping, with the theoretical ones statically computed from the topology. The results are presented in Table VI as a mean squared error between these two quantities. KOLLAPS and Maxinet are deployed on four machines while Mininet is deployed in a single machine as a multiple machine deployment is not supported. We observe that Mininet produces smaller errors than KOLLAPS for the 1000 topology. The reasons are twofold. First, the container networking in Docker introduces small yet measurable delays. Second, because KOLLAPS is running on different physical machines, there is also a small but measurable delay when packets need to traverse the physical links. Despite these two factors, the largest deviations from the theoretical `RTTs` observed with KOLLAPS were $0.34ms$, $0,37ms$, $0.37ms$, $0.45ms$ for the 1000, 2000, 4000, and 8000 elements topologies, respectively. For reference, the minimum theoretical `RTTs` in the four topologies are $20ms$, $16ms$, $14ms$ and $24ms$, respectively. Accordingly, the deviation values correspond to a MSE of 0.0683, 0.0770, 0.0813, and 0.110 respectively.

Due to the current limitations with Mininet, it was not possible to gather results for the larger topologies. Maxinet requires an external controller to manage the emulated switches. We experimented with several configurations with POX [100] modules, Floodlight [101], and Opendaylight [102] to find out which one yielded the best results. The controller configuration used for these experiments rely on 4 distinct POX controllers executing the `forwarding.l2_nx module`, the best performing one for this scenario. The error obtained for Maxinet is significantly higher than both KOLLAPS and Mininet, with the largest deviation reaching 11ms and 40ms on the 1000 and 2000 topologies respectively, larger than the minimum theoretical delays in each topology. We attribute this to the overhead of having an external controller, as well as to the type of controller (with other configurations producing even worse results). For these reasons, we did not run further experiments for Maxinet in the 4000 topology. To a lesser extent, Maxinet also suffers from the small yet measurable delay when packets need to traverse the physical links.

TABLE VII: Average BW per container and % Error of BFT Smart with varying machines and servers.

#Machines	#Nodes	Average BW (Kbps)	Error (%)
15	45	4925	1,5%
25	75	4851	2,98%
50	150	4613	7,74%

3) Scalability of Path Congestion: The experiments up to this point are either with a small amount of active end nodes or with many nodes producing a low amount of traffic. To assess KOLLAPS scalability in more intensive workloads and with more machines we use BFT-Smart [56] as our application. We choose BFT-Smart because it is a CPU-intensive application with all-to-all communication which therefore stresses both the Emulation Core (as the model needs to keep track of N^2 flows where N is the number of nodes) and the Communication Manager (which needs to disseminate metadata to the M machines). To be able to use more machines than what is available in our testbed, we used IMEC Virtual Wall 1 testbed [94]. The machines have the following characteristics: 2x Dual core AMD opteron 2212 (2GHz) CPU and 8GB RAM. Given that the machines only have 4 cores relatively slow cores, we were only able to deploy 3 BFT-Smart nodes per machine before reaching saturation. The nodes are connected in a star topology so that they all contend with each other, with 5Mbps of bandwidth and 1ms of latency. We chose 5Mbps because it is easily reached by BFT-Smart and hence it puts a substantial strain on the emulation and allows us to better assess emulation accuracy in difficult conditions. We use a single client, which is sufficient to saturate the servers. The client is connected to the servers through an additional switch with a bandwidth of 100Mbps and latency of 0.01ms. Due to the high-contention among servers, we used a `ec_period` of 5ms/15ms/50ms and a `max_age` of 2 for the experiments with 15, 25, and 50 machines, respectively. The reason for these parameters, as further discussed in §VI-D, stems from the high contention of the workload since all servers are competing for bandwidth. Since the hardware is being use at close to capacity, we also increased the `ec_period` to allows each emulation core to have more time to receive and process the metadata and hence maintain the emulation accuracy.

For the experiment, we progressively increase the number of nodes (BFT-Smart servers) and machines and measure the average bandwidth used by each server with iftop [103]. The observed bandwidth should not surpass 5Mbps since this is the limit established in the topology. The results are reported in Table VII. The main observation is that the error rate is below 10% in all experiments. We note that this scenario represents a worst case scenario for KOLLAPS path congestion mechanism since all of the active end nodes generate traffic for all of the other end nodes. Thus there is a permanent high-contention for a share of bandwidth in the same links. For instance, for the experiments with 75 nodes each connection amounts to only 66,6b Kbps of the total 5Mbps. Overall, this

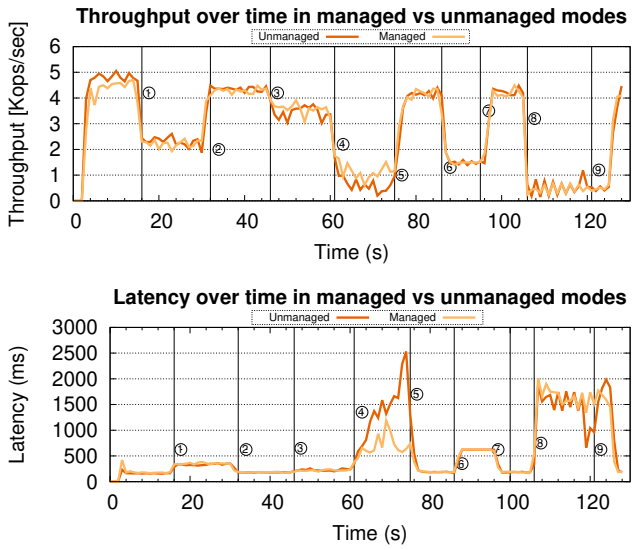

Fig. 9: YCSB throughput and read latency in a dynamic environment in managed and unmanaged deployment modes. The circled numbers correspond to `dynamic` events from §VI-G.

experiment illustrates KOLLAPS ability to provide accurate emulation under high-contention and as the system size (both in terms of physical machines and application nodes) scales.

G. Unmanaged Mode

So far all the experiments were conducted in managed mode, where KOLLAPS controls the full lifecycle of the applications. We now compare the emulation accuracy between managed and unmanaged modes. To this end, we manually deployed Apache Cassandra (v4.0.5) on five physical machines for the unmanaged mode, and on Docker Swarm on one physical machine for the managed mode. The experiment models a deployment with: *(i)* two server machines located in one data center in North Virginia, USA, *(ii)* two server machines located in one data center in Frankfurt, Germany, and *(iii)* a client which is located in Stockholm, Sweden. The latencies between data centers were obtained from Cloudping and the client runs the YCSB benchmark [104]. This produces a topology with five `services` (four servers and one client) and two `bridges` connecting the data centers to the client. The experiment injects several `dynamic` events to model multiple *what-if* scenarios, described next. Throughput and read latency[3] results are depicted in Figure 9. We start with the scenario previously described and then apply the following events: ① at 15 seconds we model what would happen if we moved one data center from Frankfurt to Mumbai, India which has higher latency and hence negatively affects the application performance, ② at second 30 we reduce the latency from the Mumbai datacenter to the US datacenter in half, which results in an increase in performance, ③ at second 45, we introduce a 1% drop rate of all packets in the same link, causing the system to be disrupted as we observe a substantial decline in throughput and a rise in latency, ④ at second 60 we increase the drop rate in the link to 10% increasing the level of disruption in the network. As expected we observe an even bigger drop in throughput and a substantial rise in latency, ⑤ at second 75 the drop rate in the link is set to zero, ⑥ at second 85 we increase by 4× the latency in the links from the client to both data centers causing a significant drop in performance, ⑦ at second 95 we normalize the conditions again, ⑧ at second 105 we introduce a 100% drop rate loss in the link from the client to one of the servers, disrupting the system and causing a significant drop in throughput, ⑨ at second 120 we reset the network to the original state before concluding the experiment.

We draw two main conclusions from these experiments. First, the managed and unmanaged modes achieves very similar results, demonstrating how KOLLAPS provides accurate emulation in unmanaged mode. Second, the ability to easily deploy KOLLAPS in an already running system opens the door to a novel set of experiments, *e.g.*, injecting network faults in a production system in line with Chaos Engineering methodologies.

[3]The benchmark reports read and update latency separately. The obtained results for both are similar and hence, for clarity, we report only read latency

VII. LESSONS LEARNED

Efficiency. While we had efficiency concerns in the early designs, we noticed some inefficiencies over the years as KOLLAPS [54] was used in several scenarios, from teaching distributed system classes up to deployment in production-grade systems. Originally, metadata dissemination relied on Aeron [105], an efficient and reliable UDP unicast framework. While Aeron was designed to be efficient, its usage in KOLLAPS proved to be very demanding on the CPU. This is because each emulation core polls with a thread in busy-waiting for new metadata messages. This resulted in the scheduler always selecting the Aeron thread causing high CPU usage which could affect the application.

With this problem solved, we noticed that `sys` CPU usage could significantly increase in large-scale deployments. Upon a thorough investigation, we pinpointed this to the mechanism used to retrieve the flow metadata. In earlier versions, we used `tc` to retrieve this data. Specifically, each emulation core would query about the metadata from itself to every other emulation core. This quadratic cost resulted in high `sys` usage, particularly when emulating large-scale topologies. We redesigning and reimplementing this mechanism by leveraging `eBPF` (§V-C).

Finally, we also reimplemented the KOLLAPS's critical components in Rust which resulted in an overall more efficient (CPU- and memory-wise) and safer system when compared with the previous Python implementation.

Unmanaged Scenarios. Since the early designs, we aimed for KOLLAPS to be accessible to use and it supported Docker Swarm [4] and Kubernetes [5] from the outset. Interestingly, potential users of the tool felt this was limiting since it did not allow them to use KOLLAPS in their existing (legacy) deployments or within their continuous integration pipelines. To address this, we extended KOLLAPS to support unmanaged deployments which allows to deploy KOLLAPS alongside

existing systems and control their network. This enables new use cases such as using KOLLAPS to control the network as part of a Chaos Engineering [106] approach.

VIII. LIMITATIONS

Interactivity. For dynamic topologies, we compute off-line, and locally at each node, the sequence of all graph states over time. While this approach allows to achieve sub-second emulation precision, it also prevents the support to establish an interactive testing session for which a precise crash plan is not defined statically by configuration but rather decided by the user on the fly. In principle, it is possible to support interactive experiments by computing and applying the graph changes online at the expense of some accuracy.

Multipath routing and multicast. While the emulated topology itself can include multiple paths between each two pairs of nodes, KOLLAPS uses a shortest path algorithm to compute the collapsed links between every pair of application nodes, effectively discarding any multipath routing [107] considerations. We plan to support this in the future by: *(i)* extending the language to allow the specification of multiple paths, *(ii)* use a k-shortest paths algorithm for link collapsing, and *(iii)* extend the emulation model to embed this. Note also that KOLLAPS does not currently support multicast because the multicast tree is maintained at the network elements such as switches and bridges, which we do not model.

Beyond the physical links. KOLLAPS only emulates network topologies whose aggregate capacity fits into the limits of the underlying physical cluster (*i.e.*, it is impossible to emulate a link of 10Gb/s if KOLLAPS is running on a cluster with 1Gb/s connections). Moreover, since the bandwidth sharing is updated upon each iteration of the Emulation Core, there is a lower bound on the minimum latency that KOLLAPS can emulate. Concretely, KOLLAPS will either fail to capture and update the bandwidth sharing for short flows that span a time interval shorter than a single iteration, or would react after the flow has ended. In this sense, our design and implementation are better suited for emulating WAN deployments rather than emulating data-center environments. Recent work explores such support time-dilution techniques optimized for containers in the context of SDN [108], which we plan to leverage.

IX. CONCLUSION

KOLLAPS is motivated by the need to simplify the evaluation of large-scale geo-distributed applications. Rather than emulating the full network state, we argue that application-level metrics are mostly affected by the macro network properties, such as end-to-end latency, bandwidth, packet loss and jitter. Our experiments, on small and large-scale Internet-like topologies, in both static and dynamic settings, show that KOLLAPS is able to accurately reproduce real-world deployments of off-the-shelf popular systems, such as Cassandra. To our community, reproducibility of results is increasingly important and we believe KOLLAPS can be a useful tool to achieve this goal. We showed this by reproducing results from a geographically distributed state machine replication system presented in the literature [55]. Finally, KOLLAPS can also be used by engineers to predict application performance and correctness under hypothetical, but fully controlled, network conditions.

ACKNOWLEDGMENTS

This work was supported by Fundação para a Ciência e Tecnologia (FCT) under grant PTDC/CCI-COM/4485/2021 (Ainur).

REFERENCES

[1] P. Bailis and K. Kingsbury, "The Network is Reliable," *Queue*, vol. 12, no. 7, p. 20, 2014.
[2] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.
[3] "Linux LXC," https://linuxcontainers.org/, 2024, accessed: 2024-06-04.
[4] "Docker Swarm," https://docs.docker.com/engine/swarm/, 2024, accessed: 2024-06-04.
[5] "Kubernetes," https://kubernetes.io/, 2024, accessed: 2024-06-04.
[6] "Amazon Elastic Compute Cloud," https://aws.amazon.com/ec2/, 2024, accessed: 2024-06-04.
[7] R. F. Boisvert, "Incentivizing Reproducibility," *Communications of the ACM*, vol. 59, no. 10, pp. 5–5, Sep. 2016. [Online]. Available: http://doi.acm.org/10.1145/2994031
[8] R. D. Peng, "Reproducible research in computational science," *Science*, vol. 334, no. 6060, pp. 1226–1227, 2011.
[9] 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.
[10] R. Ricci, E. Eide, and C. Team, "Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications," *; login:: the magazine of USENIX & SAGE*, vol. 39, no. 6, pp. 36–38, 2014.
[11] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.
[12] B. C. Senel, M. Mouchet, J. Cappos, T. Friedman, O. Fourmaux, and R. McGeer, "Demo: Edgenet, a production internet-scale container-based distributed system testbed," in *ICDCS*. IEEE, 2022, pp. 1298–1301.
[13] W. Kim, A. Roopakalu, K. Y. Li, and V. S. Pai, "Understanding and characterizing planetlab resource usage for federated network testbeds," in *Internet Measurement Conference*. ACM, 2011, pp. 515–532.
[14] J. Banks, J. S. C. II, B. L. Nelson, and D. M. Nicol, *Discrete-Event System Simulation, 5th New International Edition*. Pearson Education, 2010.
[15] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *PODC*. ACM, 2007, pp. 398–407.
[16] S. Floyd and E. Kohler, "Internet research needs better models," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 29–34, 2003.
[17] S. Floyd and V. Paxson, "Difficulties in simulating the internet," *IEEE/ACM Transactions on Networking*, vol. 9, no. 4, pp. 392–403, Aug 2001.
[18] V. Paxson and S. Floyd, "Why we don't know how to simulate the internet," in *In Proceedings of the 1997 Winter Simulation Conference*, 1997, pp. 1037–1044.
[19] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How amazon web services uses formal methods," *Commun. ACM*, vol. 58, no. 4, pp. 66–73, 2015.
[20] B. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz, "Trace-based mobile network emulation," in *SIGCOMM*. ACM, 1997, pp. 51–61.
[21] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *CoNEXT*. ACM, 2012, pp. 253–264.
[22] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *HotNets*. ACM, 2010, p. 19.

[23] 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*, June 2014, pp. 1–9.

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

[25] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, "Scalability and accuracy in a large-scale network emulator," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 271–284, 2002.

[26] 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 *SOSP*. ACM, 2017, pp. 599–613.

[27] W. Du, H. Zeng, and K. Won, "SEED Emulator: An Internet Emulator for Research and Education," in *Proceedings of the Twenty-First ACM Workshop on Hot Topics in Networks (HotNets)*, Austin, Texas, USA, November 14-15 2022.

[28] S. Hemminger, "Network emulation with NetEm," in *Proceedings of the Linux Conference*, 2005.

[29] G. F. Riley and T. R. Henderson, "The *ns-3* network simulator," in *Modeling and Tools for Network Simulation*. Springer, 2010, pp. 15–34.

[30] D. B. Ingham and G. D. Parrington, "Delayline: a wide-area network emulation tool," *Computing Systems*, vol. 7, no. 3, pp. 313–332, 1994.

[31] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, 1997.

[32] M. Carbone and L. Rizzo, "Dummynet revisited," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 2, p. 12, 2009.

[33] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 255–270, 2002.

[34] M. Carson and D. Santay, "NIST net: a linux-based network emulation tool," *Comput. Commun. Rev.*, vol. 33, no. 3, pp. 111–126, 2003.

[35] M. A. Eriksen, "Trickle: A userland bandwidth shaper for unix-like systems," in *USENIX Annual Technical Conference, FREENIX Track*. USENIX, 2005, pp. 61–70.

[36] M. Avvenuti and A. Vecchio, "Application-level network emulation: the emusocket toolkit," *Journal of network and computer applications*, vol. 29, no. 4, pp. 343–360, 2006.

[37] R. Ricci, J. Duerig, P. Sanaga, D. Gebhardt, M. Hibler, K. Atkinson, J. Zhang, S. K. Kasera, and J. Lepreau, "The flexlab approach to realistic evaluation of networked systems," in *NSDI*. USENIX, 2007.

[38] S. Wang, C. Chou, and C. Lin, "The design and implementation of the NCTUns network simulation engine," *Simulation Modelling Practice and Theory*, vol. 15, no. 1, pp. 57–81, 2007.

[39] Z. Puljiz, R. Penco, and M. Mikuc, "Performance analysis of a decentralized network simulator based on imunes," in *2008 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, June 2008, pp. 519–525.

[40] R. Roverso, M. Al-Aggan, A. Naiem, A. Dahlstrom, S. El-Ansary, M. El-Beltagy, and S. Haridi, "MyP2PWorld: Highly reproducible application-level emulation of P2P systems," in *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, ser. SASOW, 2008.

[41] L. Nussbaum and O. Richard, "Lightweight emulation to study peer-to-peer systems," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 6, pp. 735–749, 2008.

[42] J. Ahrenholz, C. Danilov, T. R. Henderson, and J. H. Kim, "Core: A real-time network emulator," in *MILCOM 2008 - 2008 IEEE Military Communications Conference*, 2008, pp. 1–7.

[43] C. Tang, "DSF: A common platform for distributed systems research and development," in *Middleware*, ser. Lecture Notes in Computer Science, vol. 5896. Springer, 2009, pp. 414–436.

[44] E. Weingärtner, F. Schmidt, H. vom Lehn, T. Heer, and K. Wehrle, "Slicetime: A platform for scalable and accurate network emulation," in *NSDI*. USENIX Association, 2011.

[45] V. Schiavoni, E. Rivière, and P. Felber, "Splaynet: Distributed user-space topology emulation," in *Middleware*, ser. Lecture Notes in Computer Science, vol. 8275. Springer, 2013, pp. 62–81.

[46] L. Sarzyniec, T. Buchert, E. Jeanvoine, and L. Nussbaum, "Design and evaluation of a virtual experimental environment for distributed systems," in *PDP*. IEEE Computer Society, 2013, pp. 172–179.

[47] V. Sinha and M. Wang, "evalbox: A cross-platform evaluation framework for network systems," in *MASCOTS*. IEEE Computer Society, 2015, pp. 15–18.

[48] M. A. To, M. Cano, and P. Biba, "Dockemu – a network emulation tool," in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*, 2015, pp. 593–598.

[49] E. Petersen, G. Cotto, and M. Antonio To, "Dockemu 2.0: Evolution of a network emulation tool," in *2019 IEEE 39th Central America and Panama Convention (CONCAPAN XXXIX)*, 2019, pp. 1–6.

[50] M. Peuster, J. Kampmeyer, and H. Karl, "Containernet 2.0: A rapid prototyping platform for hybrid service function chains," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, June 2018, pp. 335–337.

[51] J. Lai, J. Tian, D. Jiang, J. Sun, and K. Zhang, *Network Emulation as a Service (NEaaS): Towards a Cloud-Based Network Emulation Platform*, 10 2019, pp. 508–517.

[52] E. Petersen and M. A. To, "Docksdn: A hybrid container-based sdn emulation tool," in *2020 IEEE Latin-American Conference on Communications (LATINCOM)*, 2020, pp. 1–6.

[53] "Testground," https://github.com/testground/testground, 2022, accessed: 2024-06-04.

[54] P. Gouveia, J. Neves, C. Segarra, L. Liechti, S. Issa, V. Schiavoni, and M. Matos, "Kollaps: decentralized and dynamic topology emulation," in *EuroSys*. ACM, 2020, pp. 23:1–23:16.

[55] J. Sousa and A. Bessani, "Separating the WHEAT from the chaff: An empirical design for geo-replicated state machines," in *SRDS*. IEEE Computer Society, 2015, pp. 146–155.

[56] A. N. Bessani, J. Sousa, and E. A. P. Alchieri, "State machine replication for the masses with BFT-SMART," in *DSN*. IEEE Computer Society, 2014, pp. 355–362.

[57] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.

[58] "Apache Cassandra," https://cassandra.apache.org/, 2024, accessed: 2024-06-04.

[59] "Linux hierarchy token bucket," https://linux.die.net/man/8/tc-htb, 2024, accessed: 2024-06-04.

[60] "Priority qdisc." accessed: 2024-06-04. [Online]. Available: https://man7.org/linux/man-pages/man8/tc-prio.8.html

[61] "u32 Universal Identifiers," http://man7.org/linux/man-pages/man8/tc-u32.8.html, 2024, accessed: 2024-06-04.

[62] "Linux Traffic Control," https://linux.die.net/man/8/tc, 2024, accessed: 2024-06-04.

[63] "extended berkeley packet filter." accessed: 2024-06-04. [Online]. Available: https://ebpf.io/what-is-ebpf

[64] "Linux socket filter," accessed: 2024-06-04. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/filter.txt

[65] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*. IEEE, 2009, pp. 99–100.

[66] M. Pizzonia and M. Rimondini, "Netkit: easy emulation of complex networks on inexpensive hardware," in *TRIDENTCOM*. ICST, 2008, p. 7.

[67] G. Bonofiglio, V. Iovinella, G. Lospoto, and G. D. Battista, "Kathará: A container-based framework for implementing network function virtualization and software defined networks," in *NOMS*. IEEE, 2018, pp. 1–9.

[68] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Control plane compression," in *SIGCOMM*. ACM, 2018, pp. 476–489.

[69] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. A. Levis, and K. Winstein, "Pantheon: the training ground for internet congestion-control research," in *USENIX Annual Technical Conference*. USENIX Association, 2018, pp. 731–743.

[70] T. Pfandzelter and D. Bermbach, "Celestial: Virtual software system testbeds for the LEO edge," in *Middleware*. ACM, 2022, pp. 69–81.

[71] "Apache mina," https://mina.apache.org/, 2024, accessed: 2024-06-04.

[72] L. Leonini, E. Rivière, and P. Felber, "SPLAY: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze)," in *NSDI*. USENIX Association, 2009, pp. 185–198.

[73] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, 1997.

[74] M. Peuster, H. Karl, and S. Van Rossem, "Medicine: Rapid prototyping of production-ready network services in multi-pop environments," in *Network Function Virtualization and Software Defined Networks (NFV-SDN), IEEE Conference on*. IEEE, 2016, pp. 148–153.

[75] M. A. To, M. Cano, and P. Biba, "DOCKEMU - A Network Emulation Tool," *Proceedings - IEEE 29th International Conference on Advanced Information Networking and Applications Workshops, WAINA 2015*, pp. 593–598, 2015.

[76] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos, "Elasticswitch: practical work-conserving bandwidth guarantees for cloud computing," in *SIGCOMM*. ACM, 2013, pp. 351–362.

[77] R. Potharaju and N. Jain, "When the network crumbles: an empirical study of cloud network failures and their impact on services," in *SoCC*. ACM, 2013, pp. 15:1–15:17.

[78] L. Liechti, P. Gouveia, J. Neves, P. G. Kropf, M. Matos, and V. Schiavoni, "THUNDERSTORM: A tool to evaluate dynamic network topologies on distributed systems," in *SRDS*. IEEE, 2019, pp. 241–250.

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

[80] L. Massoulie and J. Roberts, "Bandwidth sharing: objectives and algorithms," *IEEE/ACM Transactions on Networking*, vol. 10, no. 3, pp. 320–328, June 2002.

[81] 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, April 2000.

[82] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Queue*, vol. 14, no. 1, p. 10, 2016.

[83] "Docker Security Capabilities," https://docs.docker.com/engine/security/, 2024, accessed: 2024-06-04.

[84] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[85] G. Vu-Brugier, R. Stanojevic, D. J. Leith, and R. Shorten, "A critique of recently proposed buffer-sizing strategies," *Comput. Commun. Rev.*, vol. 37, no. 1, pp. 43–48, 2007.

[86] A. Vishwanath, V. Sivaraman, and M. Thottan, "Perspectives on router buffer sizing: Recent results and open problems," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 2, pp. 34–39, 2009.

[87] "Linux Network Emulator," https://www.linux.org/docs/man8/tc-netem.html, 1997, accessed: 2024-06-04.

[88] "Linux TCP Small Queues," https://lwn.net/Articles/507065/, 2012, accessed: 2024-06-04.

[89] "A rust ebpf toolchain." accessed: 2024-06-04. [Online]. Available: https://github.com/foniod/redbpf

[90] "raw(7) - linux man page." accessed: 2024-06-04. [Online]. Available: https://linux.die.net/man/7/raw

[91] "Perfmap implementation in github." accessed: 2024-06-04. [Online]. Available: https://github.com/foniod/redbpf/blob/main/redbpf/src/perf.rs

[92] "Async standard library." accessed: 2024-06-04. [Online]. Available: https://doc.rust-lang.org/stable/std/net/

[93] "Linux pipes." accessed: 2024-06-04. [Online]. Available: https://linux.die.net/man/3/mkfifo

[94] "Imec virtual wall 1." accessed: 2024-5-28. [Online]. Available: https://doc.ilabt.imec.be/ilabt/virtualwall/index.html

[95] "iPerf3," https://github.com/esnet/iperf, 2024, accessed: 2024-06-04.

[96] "Network time protocol (ntp)." accessed: 2024-06-04. [Online]. Available: https://www.rfc-editor.org/rfc/rfc958.html

[97] "Amazon EC2 - T1 micro instances," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts_micro_instances.html, 2024, accessed: 2024-06-04.

[98] 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

[99] "dstat." accessed: 2024-06-04. [Online]. Available: https://linux.die.net/man/1/dstat

[100] "POX," https://github.com/noxrepo/pox, 2018, accessed: 2024-06-04.

[101] "Floodlight," https://github.com/floodlight, 2018, accessed: 2024-06-04.

[102] "Opendaylight," https://www.opendaylight.org/, 2018, accessed: 2024-06-04.

[103] "iftop - display bandwidth usage on an interface by host." accessed: 2024-06-04. [Online]. Available: https://linux.die.net/man/8/iftop

[104] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*. ACM, 2010, pp. 143–154.

[105] "Aeron," https://github.com/real-logic/aeron, 2024, accessed: 2024-06-04.

[106] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Softw.*, vol. 33, no. 3, pp. 35–41, 2016.

[107] I. Cidon, R. Rom, and Y. Shavitt, "Analysis of multi-path routing," *IEEE/ACM Transactions on Networking*, vol. 7, no. 6, pp. 885–896, Dec 1999.

[108] J. Yan and D. Jin, "A lightweight container-based virtual time system for software-defined network emulation," *Journal of Simulation*, vol. 11, no. 3, pp. 253–266, 2017.

Sebastião Amaro , is a Ph.D. student, in the topic of Fault Injection in Distributed Systems. He holds a Master in Computer Science and Engineering from IST (Instituto Superior Técnico) with a thesis in the topic of network emulation, and he worked as a researcher in 2022 on the topic of performance in persistent memory systems.

Miguel Matos (Member, IEEE) is an assistant professor at the Engineering School of the University of Lisbon (Instituto Superior Técnico) and a Senior Researcher at INESC-ID. His research interests lie in broad the area of systems. His work has been published in SOSP, Eurosys, TPDS, JPDC, ICDCS, DSN, IPDPS and Middleware among others.

Valerio Schiavoni (Member, IEEE), is Professeur Titulaire and Maître d'enseignment et recherche at the University of Neuchâtel, Switzerland. His research interests lie at the intersection of systems, security, and data management. He served on more than 50 program committes and published his work in JPDC, TPDS, TDSC, Eurosys, Middleware, DNS, ICDCS, IPDPS, PPoPP, SRDS, and more.