Kollaps: Decentralized and Dynamic Topology Emulation

Paulo Gouveia
U. Lisboa & INESC-ID
Lisbon, Portugal

João Neves
U. Lisboa & INESC-ID
Lisbon, Portugal

Carlos Segarra
University of Neuchâtel
Neuchâtel, Switzerland

Luca Liechti
University of Neuchâtel
Neuchâtel, Switzerland

Shady Issa
U. Lisboa & INESC-ID
Lisbon, Portugal

Valerio Schiavoni*
University of Neuchâtel
Neuchâtel, Switzerland
valerio.schiavoni@unine.ch

Miguel Matos†
U. Lisboa & INESC-ID
Lisbon, Portugal
miguel.marques.matos@tecnico.ulisboa.pt

Abstract

The performance and behavior of large-scale distributed applications is highly influenced by network properties such as latency, bandwidth, packet loss, and jitter. For instance, an engineer might need to answer questions such as: What is the impact of an increase in network latency in application response time? How does moving a cluster between geographical regions affect application throughput? What is the impact of network dynamics on application stability? Currently, answering these questions in a systematic and reproducible way is very hard due to the variability and lack of control over the underlying network. Unfortunately, state-of-the-art network emulation or testbed environments do not scale beyond a single machine or small cluster (*i.e.*, MiniNet), are focused exclusively on the control-plane (*i.e.*, CrystalNet) or lack support for network dynamics (*i.e.*, EmuLab).

In this paper, we address these limitations with Kollaps, a fully distributed network emulator. Kollaps hinges on two key observations. First, from an application's perspective, what matters are the emergent end-to-end properties (*e.g.*, latency, bandwidth, packet loss, and jitter) rather than the internal state of the routers and switches leading to those properties. This premise allows us to build a simpler, dynamically adaptable, emulation model that does not require maintaining the full network state. Second, this simplified model is amenable to be maintained in a fully decentralized way, allowing the emulation to scale with the number of machines required by the application.

Kollaps is fully decentralized, agnostic of the application language and transport protocol, scales to thousands of processes and is accurate when compared against a bare-metal deployment or state-of-the-art approaches that emulate the full state of the network. We showcase how Kollaps can accurately reproduce results from the literature and predict the behaviour of a complex unmodified distributed key-value store (*i.e.*, Cassandra) under different deployments.

CCS Concepts • **Networks → Network experimentation**.

Keywords distributed systems, emulation, dynamic network topology, containers, experimental reproducibility

ACM Reference Format:
Paulo Gouveia, João Neves, Carlos Segarra, Luca Liechti, Shady Issa, Valerio Schiavoni, and Miguel Matos. 2020. Kollaps: Decentralized and Dynamic Topology Emulation. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3342195.3387540

*Corresponding author
†Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys '20, April 27–30, 2020, Heraklion, Greece

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387540

1 Introduction

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

Such uncontrollable and poorly specified environment leads to a slow experimental cycle, results that are hard to

reproduce, and potential downstream costs worth millions of dollars [24]. It is therefore of the utmost importance to have tools that allow 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 [58], Linux LXC [18]) and container orchestration (*e.g.*, Docker Swarm [15], Kubernetes [17]) 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. Is the observed performance improvement really due to improvements in the application, or due to a *lucky* run when the network was lightly loaded? How is performance affected by common network dynamics, such as background traffic, or link flapping? The very same questions and issues also arise in the reproducibility crisis currently plaguing the system's community [29, 64]. 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 [46], CloudLab [72], or PlanetLab [35] used to conduct experiments tend to get overloaded right before system conference deadlines [51]. We therefore need tools to systematically assess and reproduce the evaluation of large-scale applications. Notably, similar tools could be used to test and (otherwise) prevent costly incidents due to mis-configurations, as in recent events [4, 20].

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 [25]. 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. In fact, there is a large gap between the simulated model and the real-world deployment, usually leading to several unforeseen behaviors not captured by the model [34, 40, 41, 63]. And even if the simulated model yields correct results, the real implementation is not guaranteed to faithfully follow the simulated model. Moreover, despite some efforts to model complex systems either through formal method analysis [60] or simulation, this is, to the best of our knowledge, seldom the case for large-scale systems.

The alternative is to resort to network emulation. In a network emulation, the real system is run against a model of the network that replicates real-world behavior by modeling a network topology together with its network elements, including switches, routers and their internal behavior. Emulation thus allows to reach conclusions about the behavior of the real system in a concrete scenario rather than its model. Unfortunately, state-of-the art network emulators

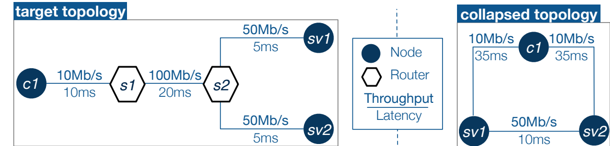

Figure 1. Left: 3 application containers (*c1*, *sv1*, *sv2*) and 2 network elements (*s1*, *s2*). Right: collapsed topology with links describing maximum bandwidth and minimum latency between each two nodes.

suffer from several limitations. MiniNet [44] is limited to a single physical machine and therefore cannot be used to emulate a large-scale resource-intensive system. MaxiNet [87] and the multi-host version of MiniNet support distributed clusters but scale poorly [56]. ModelNet [83] and alike rely on a dedicated cluster of nodes to maintain the emulation model to which the application nodes must connect. However, accuracy is dependent on application traffic patterns and can degrade with a modest increase in the number of application nodes [81]. CrystalNet [56] 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. While emulation testbeds (*e.g.*, Emulab [46]) 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 tools cannot systematically assess and reproduce the evaluation of large-scale applications subject to network dynamics.

In this paper we introduce Kollaps, a decentralized network emulator for large-scale applications. Kollaps overcomes the state-of-the-art limitations through three key insights. First, from the perspective of a distributed application, the observable end-to-end properties, such as 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) but provides 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. Finally, the simplified model lends itself to quick changes enabling us to emulate dynamic events such as link removals and additions or background traffic in a fraction of a second. To understand Kollaps's main principle, consider the topology in Figure 1 (left), with two network elements (*s1* and *s2*) and three containers (*c1*, *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 1 (right). Note that the bandwidth and

Table 1. Classification of network emulation tools. NetEm [45] uses a different queueing discipline to implement bandwidth shaping. Dockemu uses ns-3 [73] for link-level features. ↯: ability to dynamically change this property. P=process, V=virtual machine, C=container.

					Concurrent	Path	Link-Level emulation capabilities				Any	Topology	
Name	Year	Mode	HW ind.	Orchestration	deployments	congestion	Bandwidth	Delay	Packet loss	Jitter	Language	dynamics	Unit
DelayLine [47]	1994	User	✓	Centralized	✗	✗	✗	✓	✓	✗	✓	✗	P
ModelNet [81]	2002	Kernel	✓	Centralized	✗	✓	✓↯	✓↯	✓↯	✗	✓	✓	P
Nist NET [33]	2003	Kernel	✓	Centralized	✗	✗	✓↯	✓↯	✓↯	✓↯	✓	✗	P
NetEm [45]	2005	Kernel	✓	*(N/A: single link emulation only)*			✗	✓	✓	✓	✓	✗	P
Trickle [39]	2005	User	✓	*(N/A: single link emulation only)*			✓	✓	✗	✗	✓	✗	P
EmuSocket [23]	2006	User	✓	*(N/A: single link emulation only)*			✓↯	✓↯	✗	✗	✓	✗	P
ACIM/FlexLab [71]	2007	Kernel	✓	Centralized	✗	✓	✓↯	✓↯	✓↯	✓↯	✓	✓	V
NCTUns [85]	2007	Kernel	✓	Centralized	✗	✓	✓	✓	✓	✓	✓	✗	P
Emulab [46, 88]	2008	Kernel	✗	Centralized	✗	✓	✓↯	✓↯	✓↯	✗	✓	✓	V
IMUNES [70]	2008	Kernel	✗	Centralized	✗	✗	✓	✓	✓	✗	✓	✗	P
MyP2P-World [75]	2008	User	✓	Centralized	✗	✗	✓	✓	✓	✗	✗	✗	P
P2PLab [61]	2008	Kernel	✓	Centralized	✗	✗	✓	✓	✓	✗	✗	✗	P
Netkit [67]	2008	Kernel	✓	Centralized	✗	✓	✓↯	✓↯	✓↯	✗	✓	✗	V
DFS [79]	2009	User	✓	Centralized	✓	✗	✓↯	✓↯	✓	✓	✗	✓	P
Dummynet [32]	2010	Kernel	✓	Centralized	✗	✗	✓↯	✓↯	✓↯	✗	✓	✗	P
Mininet [53]	2010	Kernel	✓	Centralized	✗	✓	✓↯	✓↯	✓↯	✓↯	✓	✓	P
SliceTime [86]	2011	Kernel	✗	Centralized	✗	✓	✓	✓	✗	✗	✓	✓	V
Mininet-HiFi [44]	2012	Kernel	✓	Centralized	✗	✗	✓↯	✓↯	✓↯	✓↯	✓	✓	C
SplayNet [76]	2013	User	✓	Decentralized	✓	✓	✓	✓	✓	✗	✗	✓	P
MaxiNet [87]	2014	Kernel	✓	Centralized	✗	✓	✓↯	✓↯	✓↯	✓↯	✓	✓	P
Dockemu [80]	2015	User	✓	Centralized	✗	✗	✓	✓	✓	✓	✓	✗	C
EvalBox [77]	2015	Kernel	✓	Centralized	✗	✗	✓↯	✓↯	✓↯	✓↯	✓	✓	P
ContainerNet [65]	2016	Kernel	✓	Centralized	✗	✓	✓↯	✓↯	✓↯	✓↯	✓	✓	C,V
Kathará [30]	2018	Kernel	✓	Centralized	✗	✓	✓↯	✓↯	✓↯	✗	✓	✗	C
Kollaps	**2020**	**Kernel**	✓	**Decentralized**	✓	✓	✓↯	✓↯	✓↯	✓↯	✓	✓	**C,V**

latency values in the collapsed network depict the maximum available bandwidth and the minimum achievable latency between each two nodes, as if it is 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 processes with accuracy on par with centralized state-of-the-art emulators.

Contributions. Our main contributions are:

1. Kollaps, the first network topology emulator that allows to evaluate large-scale applications in dynamic networks;
2. We integrate Kollaps with Docker Swarm [15] and Kubernetes [17], to deploy and evaluate unmodified containerized (distributed) systems;
3. A comparison of Kollaps's emulation accuracy versus bare-metal deployments and state-of-the-art approaches. Our evaluation scenarios includes static and dynamic topologies, various workload patterns (*i.e.*, short and long-lived data flows) and different TCP congestion models, including TCP Reno and Cubic;
4. A showcase of the new types of experiments that Kollaps enables. Namely, we reproduce results from published papers [28, 78], and assess how Apache Cassandra [13, 52] is affected by different network characteristics *as-if* it were deployed on AWS.

Organization. This paper is organized as follows. We survey related work in §2. The design and system architecture of Kollaps are described in §3, with implementation details presented in §4. In §5 we present our in-depth evaluation. We discuss the current limitations and future work in §6, before concluding in §7.

2 Related Work

We categorize network emulators along several dimensions (Table 1): 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), ability to dynamically adjust such features on the fly as well as to change the topology itself (*i.e.*, add/remove links, switches and nodes), implementation-language restrictions for the programs under emulation, and the supported deployment unit (virtual machines, containers or native processes). Due to lack of space, we only detail how Kollaps compares with some representative systems, and we consider simulation-based tools (*e.g.*, ns-3 [73], PeerSim [59]) out of the scope.

Few recent works cover orthogonal aspects of network emulation and illustrate the relevance of controlled experiments. CrystalNet [56] focuses on large-scale emulation of the control-plane, enabling network engineers to evaluate changes to the control-plane before deploying in production. Kollaps only deals with data-plane, and it is hence complementary to CrystalNet. To enhance the efficiency of large-scale control-plane analysis, Bonsai [27] 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 [89] 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. The

work done in Pantheon provides evidence 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. In this paper, we rely on the same insight to provide a network emulator able to emulate large-scale topologies with accuracy.

User-space approaches. Trickle [39] 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-host system to emulate large networks involves tedious and error-prone manual configuration since there is no 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 [23] and DelayLine [47] are userspace 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 [75] is a Java-based application-level emulator aimed at peer-to-peer protocols. Applications must be implemented in Java and rely on Apache Mina [5] to intercept and emulate large-scale network conditions. While Kollaps can be used for Java applications, it can be used with any other language as well.

SplayNet [76] extends Splay [54] to allow emulation of arbitrary network topologies, deployed across several physical hosts 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 host to the destination host. 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 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 also including dynamic topologies.

Kernel-space approaches. Next, we survey network emulators that require explicit or specialized support from the underlying OS and kernel. Dummynet [74] operates directly on a specific network interface. It is used as a low-level tool to build full-fledged emulators, such as Modelnet [81]. 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`) [19] to offer the same low-level traffic shaping features, but (1) without requiring dedicated hosts and (2) at the same time providing a complete testbed integrating with large-scale container orchestration tools.

The Emulab [88] testbed supports the deployment of user-provided operating systems. As ModelNet and Kollaps, it leverage 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 [53] emulates network topologies on a single host. It relies on Linux's lightweight virtualization mechanisms (*i.e.*, `cgroups`) to emulate separated network hosts. Similarly to Docker, it creates virtual Ethernet pairs running in separated namespaces and it assigns processes to those. Mininet can emulate hundreds of networked hosts (instances) on a single physical host, 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-host deployment hence preventing its use for large-scale resource-intensive applications that cannot fit a single machine. Besides, even with a simple topology, Mininet's accuracy quickly degrades under certain workloads such as short-flows. Maxinet [87] extends Mininet to allow for cluster deployments of worker hosts and with native support for Docker containers. It does so by tunneling links that cross different workers. However, it requires all emulated hosts 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 [65, 66] 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 [80], a network emulation tool based on Docker containers.

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. Finally, it is worth noting that the decentralized design of Kollaps' metadata exchange is similar in spirit to the hose model of ElasticSwitch [68], which was used to provide bandwidth guarantees for virtual machines in cloud environments.

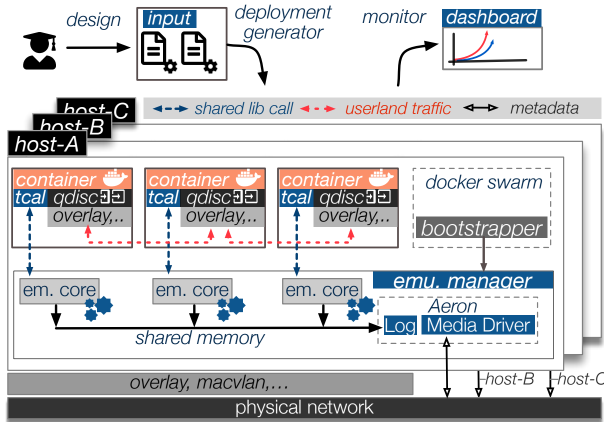

Figure 2. Kollaps system design. Note that the *bootstrapper* component is not required in a Kubernetes deployment (§4).

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
```

3 The Kollaps System

In this section we describe the system design and architecture of Kollaps. Figure 2 depicts the main components and a deployment over several hosts. Kollaps consists of several components. The Emulation Manager has a single instance per physical machine and is responsible for maintaining and enforcing the emulation model. The TC Abstraction Layer (TCAL) is deployed once per application container, and it retrieves and sets the link properties. The Bootstrapper is started once per physical machine: it initiates Kollaps in Docker Swarm deployments (but is not needed under Kubernetes, as detailed in §4). The Dashboard exposes a web-based interface to monitor and control the experiments. Finally, the Deployment Generator converts an experiment description (*e.g.*, Listing 1 and Listing 2) into a deployment plan. Next, we detail these components and their interactions.

The **Deployment Generator** translates a topology description into an actual container deployment plan. Kollaps supports an XML Modelnet-like syntax [81] to facilitate porting of existing topology descriptions, as well as a lean YAML-based syntax that we show here. Listing 1 describes the network topology from Figure 1 (left). The topology description language supports `services`, `bridges`, `links`, and `dynamic` elements. 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). 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, properties (*i.e.*, latency, bandwidth, packet loss, jitter), and the name of the container network to attach to (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 a rich set of dynamic events, *e.g.*, 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 (containers) 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 [69]. 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 easily program more complex dynamic patterns on top of Kollaps is detailed in [55].

The **Emulation Manager** (EM) is the main component of Kollaps. An instance of EM is deployed on every physical server involved in the experiment. Each instance is responsible for providing containers collocated on the same physical host with their emulated 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 end hosts. This is achieved as follows. The EM starts by parsing the topology description (*e.g.*, Listing 1) into a graph structure, maintained throughout the emulation. Next, the EM computes the shortest paths between every pair of reachable containers. Each shortest path is composed of several links, whose properties are used to determine the end-to-end network properties. Formally, given a path $\mathcal{P}$ composed of links $\mathcal{P} = \{l_1, l_2, \ldots, l_n\}$, the end-to-end properties of the path 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 [49, 57], which gives a share to each flow that is proportional to its round-trip time and is inspired by TCP Reno [62], 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 is not consuming all its available share. Therefore, when the sum of shares of all active flows is less than the maximum bandwidth on the link, the EM performs a maximization step that increases the share of the other flows, proportionally to their original shares. Note that Kollaps enforces bandwidth sharing per destination, not per flow.

Congestion. The chosen model computes, per each flow, the maximum available bandwidth allowed at any given time. While this works well when bandwidth usage is below or at capacity, it produces unrealistic results when the cumulative bandwidth required by flows surpasses the maximum bandwidth capacity. The reason for this is a complex interplay between the Linux kernel Traffic Control's shaping action, the congestion algorithms of reliable transport protocols, and the implementation of such transport protocols in the kernel itself. 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. Unlike TCP, 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 [82, 84]. However, this approach does not work due to the differences in behavior between `tc` queues and those found in a switch or router. On one hand, when a buffer in a router or switch fills up, it drops further incoming packets.[1] On the other hand, when the `htb qdisc` queue (used by the kernel) is full, rather than dropping packets, it back-pressures the application. The reasons for these discrepancies between `tc` and router queues are because modeling packet loss is done in `netem` [1] and not in `htb qdiscs`, and also due to Linux's TCP Small Queues (TSQ) [3] (since kernel v3.6). TSQ reduces the number of packets in `qdiscs` and device queues with the goal of reducing the RTT, hence mitigating bufferbloat. It works by tracking the amount of data waiting to be transmitted, and when this surpasses a given limit, the socket is throttled down, preventing further packets to be enqueued. The impact of this depends on the application — when writing to a socket, an application using blocking I/O would block, while an application using non-blocking I/O would observe zero bytes written. From our perspective, this means that there is no packet loss upon congestion and thus congestion control algorithms sensitive to loss would behave differently when emulated in Kollaps than in practice. We address this limitation as follows. When we observe that the overall requested bandwidth surpasses the available bandwidth we leverage `netem` to drop packets per flow proportionally to the oversubscribed capacity. This dynamic adjustment of packet loss according to the excess bandwidth exposes packet loss to the congestion avoidance algorithm, allowing TCP connections to adjust their throughput.

For each container, the EM spawns an *Emulation Core* process attached to the network namespace of the respective container. Each Emulation Core is responsible for obtaining the container bandwidth usage, exchange this metadata with the neighboring Emulation Cores to update the emulation model described above, and enforce the topology constraints through the TCAL, described below. This design has several

[1]This is a simplification, as in practice packets already in the queue might be dropped to make room for incoming traffic with higher priority.

advantages. First, it allows Kollaps to support any containerized (including third-party, legacy, code-obfuscated) applications without modifications. Second, Emulation Cores on the same physical machine exchange emulation metadata via shared memory, reducing computational and networking overhead. Finally, it allows the EM to aggregate the data from the local Emulation Cores and exchange it directly with the EMs on the other physical machines. Note that having an EM per physical host allows metadata traffic to scale with the number of physical hosts rather than the number of application containers. Further, each EM only computes the part of the topology that affects the local containers thus reducing the computational overhead.

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 could take several seconds thus precluding accurate emulation of sub-second dynamics.

The **TC Abstraction Layer (TCAL)** interfaces with Linux's Traffic Control (`tc`). `tc` is a Linux user-space tool that allows manipulating the network properties (*i.e.*, latency, bandwidth, packet loss, jitter) and retrieving bandwidth usage of active connections. To this end, the `tc` exposes an interface that allows for applying filters to classify data and then manipulate the network properties of each class independently. To control the network properties for each class, `tc` supports a wide range of queuing disciplines (qdiscs) where the packets are enqueued before being sent. Kollaps leverages two different types of qdiscs:*(i)* hierarchical token bucket (htb qdisc), a type of qdisc to control the bandwidth of outgoing packets and *(ii)* netem qdisc, that allows to apply delay and packet loss. For each destination, Kollaps creates a htb qdisc that enforces the bandwidth allocated to flows towards that destination. Besides, a netem qdisc is also used to apply latency, jitter, and packet loss.

Outbound traffic is matched to netem qdiscs through an u32 universal 32bit [10] 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. The TCAL structures are queried and updated very frequently during each experiment, namely to retrieve bandwidth usage and enforce the dynamic properties. To minimize the overhead of these calls, we rely on netlink sockets [50] that communicate directly with the kernel, circumventing the need to periodically spawn a new `tc` process.

The **Dashboard** allows users to 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.

As shown in §5, the decentralized design and simplified emulation model allows Kollaps to achieve accuracy on par with state-of-the-art approaches, while scaling to thousands of containers.

4 Implementation

Kollaps components are implemented in Python (v3.6), C and C++, and available at https://angainor.science/kollaps. 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. Users can customize these files (as required by many real applications [31]) before starting an actual deployment.

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 [14]. 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 (the *bootstrapper*) on every Swarm node. Its job is to launch, on that machine and outside Swarm itself, the Emulation Manager (Em). The Em 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 Em launches an Emulation Core responsible for that container, as discussed in §3. 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 Em is indeed deployed automatically.

Integration with container orchestrators. The design of Kollaps facilitates the integration with existing Docker images and container orchestration tools (*e.g.*, Docker

Swarm, Kubernetes). In addition to producing ready-to-deploy Compose/Manifest files, the Kollaps deployment toolchain must configure three key resources managed in very different manners by the mentioned container orchestrators: *(1)* access to the orchestrator APIs, used at runtime for name resolution, *(2)* the topology descriptor file, read by each Em instance to setup the initial network state and compute the graph of the dynamic changes, and *(3)* the setup of multiple virtual networks.

4.1 Emulation Core and TCAL

The TCAL library provides an interface to setup the initial networking configuration, retrieve bandwidth usage, and modify the maximum available bandwidth on paths. It is implemented in C for performance reasons and consists of 2693 SLOC. The Emulation Core is implemented in Python and consist of 2963 SLOC.

The 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 via the internal Swarm discovering service or Kubernetes's API. Then, it runs an all-pairs shortest path graph traversal [38] between the local instance and all the other reachable instances. 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 flow. It works by periodically executing the following steps: *(1)* clear the state of all local active flows; *(2)* obtain the bandwidth usage by querying the TCAL; *(3)* disseminate the local bandwidth usage to the other instances; *(4)* compute bandwidth usage on each path and its constituent links; *(5)* enforce bandwidth restrictions. In parallel to the above algorithm, each Emulation Core instance collects the data sent at step (3) by the other Emulation Core instances. This is used at (4) to compute the global bandwidth usage on a path and link basis.

4.2 Metadata dissemination

All metadata is disseminated via Aeron [11], an open-source, efficient and reliable UDP and IPC message transport protocol. For containers on the same machine, the metadata is exchanged through shared memory. This is possible because all Emulation Core processes are running on the Emulation Manager's process namespace. Across remote machines, metadata is disseminated through UDP messages. For each Emulation Manager there is an *Aeron Media Driver* responsible for the dissemination of messages. Every process reads from and writes to the Media Driver using a C++ library. The metadata messages embed the following fields: *(i)* number of flows, 2 bytes; *(ii)* list of used bandwidth per flow, 4 bytes per flow; *(iii)* number of links; *(iv)* list of link identifiers. For emulated networks with $\leqslant$ 256 nodes, it is possible to pack the metadata information for links and identifiers in a single byte each (2 bytes are used for bigger emulated topologies). As shown in §5, this approach allows to fit into a single UDP datagram as much information as possible, reducing the metadata traffic.

Table 2. Bandwidth shaping accuracy for several emulated link capacities on a simple point-to-point client-server topology.

Link BW	Kollaps	Mininet	trickle (def.)	trickle (tuned)
Low (Kb/s)				
128 Kb/s	122 (-5%)	123 (-4%)	262 (+104%)	131 (+2%)
256 Kb/s	245 (-5%)	286 (+11%)	472 (+184%)	262 (+2%)
512 Kb/s	490 (-5%)	490 (-5%)	717 (+40%)	525 (+2%)
Mid (Mb/s)				
128 Mb/s	122 (-5%)	122 (-5%)	250 (+95%)	131 (+2%)
256 Mb/s	245 (-5%)	245 (-5%)	493 (-4%)	261 (+1%)
512 Mb/s	487 (-5%)	486 (-5%)	952 (+85%)	518 (+1%)
High (Gb/s)				
1 Gb/s	954 (-4%)	933 (-7%)	1.67 (+67%)	1.00 Gb/s
2 Gb/s	1.91 (-4%)	N/A	1.93 (-3%)	1.97 (-1.5%)
4 Gb/s	3.79 (-7%)	N/A	4.12 (+3%)	3.61 (-10%)

5 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 scales with the number of flows and containers, and has constant cost regardless of the emulated bandwidth usage; *(2)* running an application with Kollaps in a cluster or in Amazon EC2 yields similar results; *(3)* Kollaps emulation accuracy is comparable with, and in some scenarios better than, tools that emulate the full network state such as Mininet.

Evaluation settings. Our cluster is composed of 5 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 nodes run Ubuntu Linux 18.04 LTS, kernel v4.15.0-65-generic. 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 use the latest stable releases of Mininet (v2.2.2) and Maxinet (v1.2).

5.1 Link-level Emulation Capabilities

First we evaluate the accuracy of our bandwidth shaping mechanism under a topology that consists of two services running iPerf3 [16], connected by a single link. iPerf3 is a tool that measures the maximum bandwidth between its client and server instances. In this experiment, we use iPerf3 to assess the accuracy of Kollaps to emulate a range of different target bandwidths, and compare the results with Mininet [53] and Trickle [39], a userspace bandwidth shaper. During the experiment, we run iPerf3 for 60 seconds, and report the average bandwidth in Table 2. The values obtained with Kollaps and Mininet are similar since both systems rely on

Table 3. 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

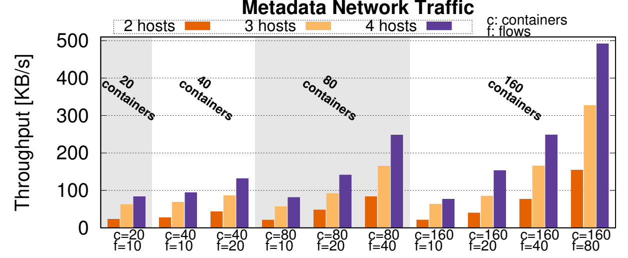

Figure 3. Kollaps metadata network usage with an increasing number of containers (C), flows (F), and hosts.

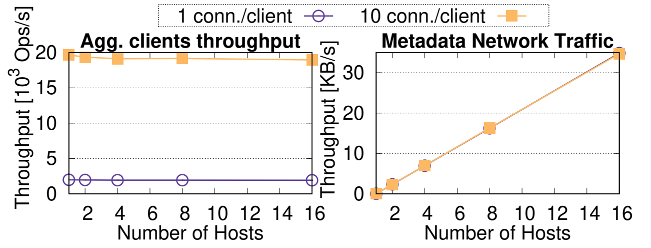

Figure 4. Aggregate throughput of twelve memcached clients (left) and the metadata traffic per host (right) for emulation on 1, 2, 4, 8 and 16 physical hosts. Note that the lines on the right figure overlap.

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 (≈ 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.

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 regions. Table 3 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. 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.

5.2 Scalability

Kollaps relies on metadata dissemination to model and emulate bandwidth restrictions for competing flows. In this section, we assess the scalability of Kollaps by analysing metadata traffic growth and emulation accuracy with an increasing number of physical nodes.

First, we study the cost of metadata dissemination by deploying several dumbbell topologies, across a cluster of increasing size. The containers are distributed evenly among the physical nodes, with the client containers on one side and the server containers on the other side of the dumbbell topology. We use iPerf3 to generate steady TCP traffic of $50Mb/s$, the maximum capacity of the shared link. We denote each configuration by a tuple `(C,F)`, with `C` the total deployed containers and `F` the number of concurrent flows. Results with clusters of up to four physical machines are shown in Figure 3.

As discussed in §3 and §4, Kollaps uses shared memory on top of the network (through Aeron) to exchange metadata among the local Emulation Cores and the Emulation Manager components residing on each machine, respectively. As expected, running on one single machine leads to zero bandwidth usage, and increasing the number of physical hosts increases the metadata bandwidth. Interestingly, though, we observe that the metadata traffic is not affected by the number of containers. This is because: *(i)* only active flows require the exchange of metadata and *(ii)* bandwidth sharing is enforced per destination and not per flow, thus reducing the overall bandwidth.

Note that metadata traffic does not increase with the emulated application bandwidth, since the messages have a constant size. Overall, we observe that metadata traffic requirements are quite modest even for topologies with 160 containers deployed across 4 physical machines, *i.e.*, $493KB/s$. Despite growing linearly with the number of physical hosts, the overall metadata traffic is negligible when considering the available bandwidth of the dedicated clusters we target with Kollaps. We further study this in the next experiment.

To study how metadata grows with an increasing number of nodes, and how distribution affects the emulation, we design the following experiment. We deploy `memcached` [7], an in-memory key-value store, in a geo-distributed topology with 4 emulated Amazon AWS regions [78] (also used in the experiments of §5.6). We place a `memcached` server container on each region, collocated with three client containers. Each

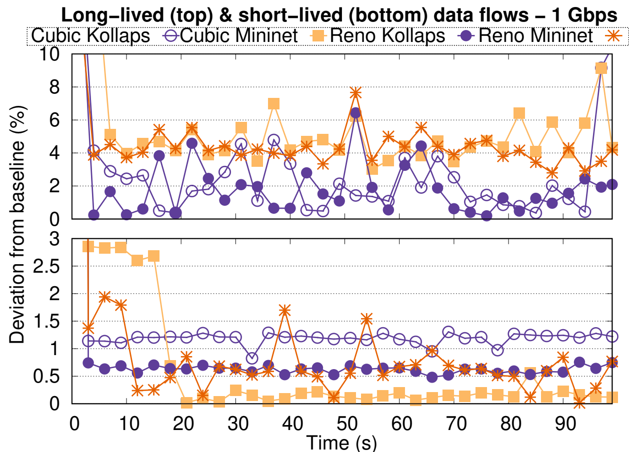

Figure 5. Error rate of bandwidth measures for long-lived (top) and short-lived (bottom) flows in Kollaps and Mininet when compared to a bare-metal deployment. Flows correspond to iPerf3 and wrk2 traffic respectively.

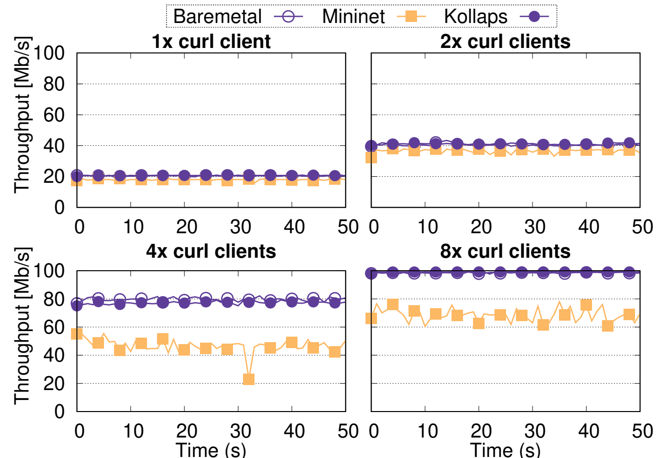

Figure 6. Throughput of a HTTP server serving different number of `curl` clients using Kollaps and Mininet when compared to a bare-metal deployment.

server handles two local clients and a remote one. Clients execute the `memtier` benchmark [22] for key-value stores. Figure 4 reports the results for two different configurations of the client, with 1 and 10 connections per client, which correspond to an increase in the number of the flows. The target load was selected to fit the experiment in a single physical machine thus allowing us to observe how the distribution across an increasing number of physical machines affects the application metrics. We deploy the experiment on top of 1, 2, 4, 8 and 16 physical machines.

We report the aggregate throughput of all the twelve clients and the metadata traffic consumed per physical hosts. The aggregate throughput (Figure 4, left) for both configurations (1 and 10 connections per clients) is consistent across the number of hosts. This results confirms that the decentralized nature of Kollaps allows to emulate the same behavior with a larger number of physical hosts. Looking at the metadata traffic (Figure 4, right), it indeed increases with the number of physical hosts but the overall value is negligible when compared to the available bandwidth in the target cluster.

5.3 Long- and short-lived flows

We now study the accuracy of Kollaps when handling short-lived and long-lived flows. To this end we set up a bare-metal experiment with one server and two clients interconnected through a 1Gb/s switch. For the long-lived flows we use iPerf3 and `tcpdump` (v4.9.2) to measure the throughput of Cubic [43] and Reno [48] congestion control algorithms over 100 seconds. We then repeat the experiment with Mininet and Kollaps and compute the observed deviation (error) from bare metal as $\left|1 - \frac{observed\ bandwidth}{baremetal\ bandwidth}\right|$. Results are depicted in Figure 5 (top). Note that due to the unpredictable nature of congestion control algorithms the overall deviation is more important than momentary fluctuations over time. Interestingly, we observe that in general Kollaps is closer to the bare metal observations than Mininet, despite the former fully modeling the full state of the network switch.

We repeat the same experiment for short-lived flows using `wrk2` [21], a popular HTTP benchmarking tool. This tool maintains a set of open connections (*i.e.*, 100 connections over 2 threads, the default configuration) and executes continuous HTTP requests (~64KB each) over them for a given duration. Results are depicted in Figure 5 (bottom). In this scenario, Kollaps is on-par with Mininet, both with a relative error smaller than 2% for most of the experiment.

Note that `wrk2` keeps a connection open and sends several small requests over the same connection. We now show the results for small requests sent over short connections, *i.e.*, each request starts a new TCP connection. To this end, we use the same request size used for `wrk2`, but instead rely on `curl` [2], a popular command-line HTTP client. In this experiment we use a 100Mb/s bandwidth link. We varied the number of concurrent clients from 1 to 8 to control the load on the network. Figure 6 shows the result for this setting. As expected, increasing the number of clients increases the load proportionally, as each client is independent and the server has sufficient capacity (*i.e.*, memory and cpu) to accomodate the increasing load. We can see that Kollaps provides similar throughput to bare-metal deployment across the different configurations. On the contrary, in such workload, Mininet fails to keep up as the client load increases. We explain this behaviour by the fact that Mininet needs to maintain the full state of the switches, which becomes a significant overhead when the number of (short) connections grows.

Lastly, we design an experiment involving both short- and long-lived data flows. We set up three hosts: host one running an `HTTP` webserver and an iPerf3 client, host two running a `wrk2` client querying host one, and host three running an iPerf3 server being queried by host one. We run this

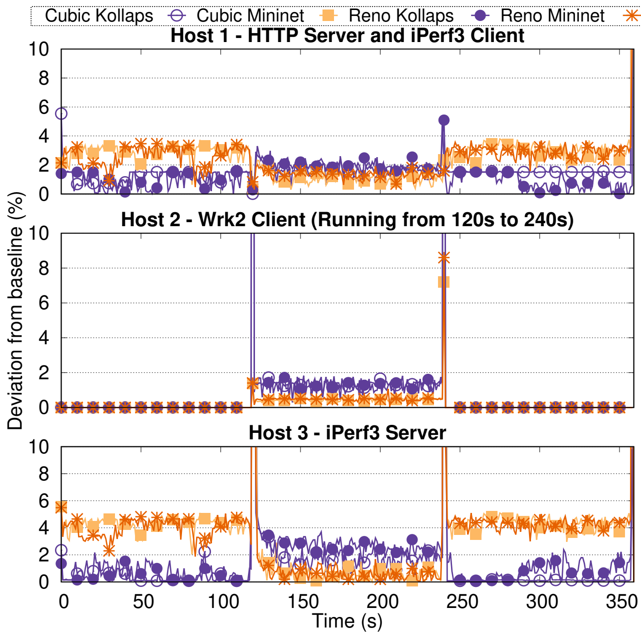

Figure 7. Error rate of bandwidth measurements in a mixed short- and long-lived flows deployment. We compare the measures recorded from host 1 (wrk2 server and iPerf3 client - top), host 2 (wrk2 client - middle) and host 3 (iPerf3 server) for Kollaps and Mininet against a bare-metal baseline.

setting for 6 minutes. Until minute two, only the long-lived (iPerf3) client is sending data. From minute two to minute four, the short lived (`wrk2`) client tries to utilize the link to its full capacity. For the remainder, only the long-lived client is sending data, like the first two minutes. Figure 7 presents the deviation from the bare-metal deployment, when reproducing this scenario with Kollaps and Mininet. We observe that the error rate for Mininet and Kollaps are comparable and mostly below 5% with only a spike in the transition period in both systems.

5.4 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 up a topology with 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 8 shows the bandwidth of each of the established flows along the time. We use iPerf3 to establish continuous TCP flows between clients and servers, while the experiment

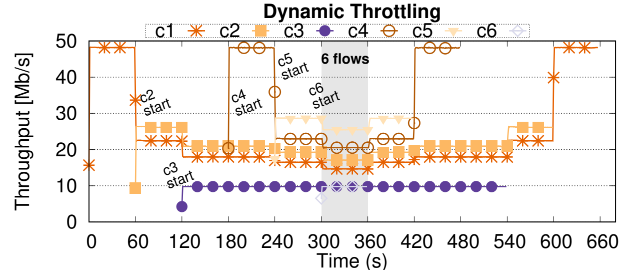

Figure 8. Decentralized bandwidth throttling: several clients compete on a shared link. Each gets a different share of bandwidth, adjusted at runtime.

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 §3, 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.

5.5 Large-scale topologies

We now compare Kollaps with Mininet [53] and Maxinet [87] in large-scale topologies.

We consider large-scale topologies generated using the preferential attachment algorithm [26]. This method yields scale-free networks, which are representative of the characteristics of Internet topologies. The experiment consists of end-nodes sending ICMP echo requests (`ping`) to other random end-nodes for 10 minutes. We compare the obtained

Table 4. 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	0.0261	0.0079	28.0779
2000	1344	656	0.0384	NA	347.5303
4000	2668	1332	0.0721	NA	NA

round-trip times (RTT) with the theoretical ones statically computed from the topology. The results are presented in Table 4 as a mean squared error between these two quantities for topologies with 1000, 2000, and 4000 elements. 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 $0.27ms$, $0.4ms$ and $0.55ms$ for the 1000, 2000, and 4000 topologies, respectively. For reference, the minimum theoretical RTTs in the three topologies are $10ms$, $22ms$ and $14ms$, respectively. Accordingly, the deviation values correspond to a MSE of 0.0261, 0.0384, and 0.0721, respectively, as depicted in the Table 4.

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 [9] modules, Floodlight [6], and Opendaylight [8] 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 (other configurations produced 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.

5.6 Geo-replicated systems

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

Reproducibility. In this experiment, we reproduce results obtained for two Byzantine fault tolerant state machine replication libraries: BFT-SMaRt [28], and its optimized version Wheat [78]. The authors of these systems evaluate and

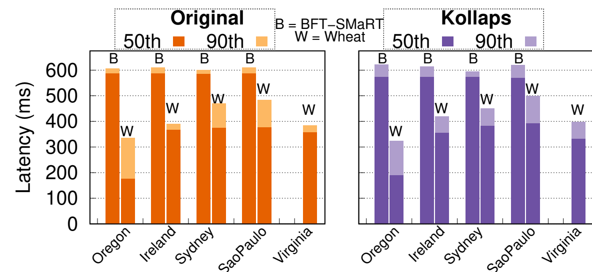

Figure 9. Reproduction of an experiment with a geo-replicated deployment of BFT-SMaRt and Wheat. The experiment measures the latencies of clients located in different Amazon EC2 regions (left: results from [78], right: same experiments with Kollaps).

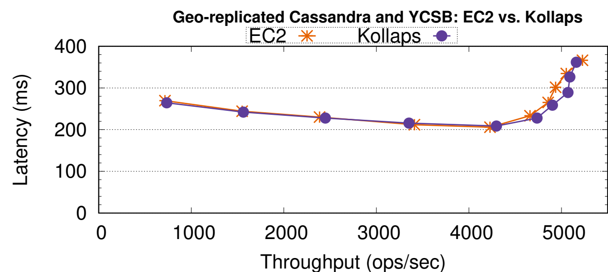

Figure 10. Throughput/latency of a geo-replicated Cassandra deployment on Amazon EC2 and Kollaps.

compare them through a geo-distributed deployment on Amazon EC2 instances spanning 5 regions [78]. 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 ([78], Table II), which we use to model a topology in Kollaps that mimics the one observed in their experiments.

Figure 9 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 [12], 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.

NoSQL evaluation. We now compare the results of benchmarking a geo-replicated Apache Cassandra [13, 52] deployment on Amazon EC2 and on Kollaps in our local dedicated cluster. The deployment consists of 4 replicas in

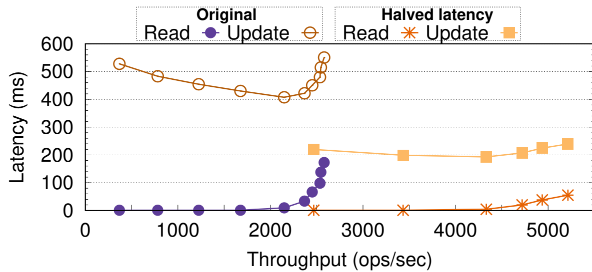

Figure 11. Throughput/latency of a geo-replicated Cassandra on Kollaps using a hypothetical topology answering the question: *what-if* nodes were moved from `ap-south` to `ap-northeast`?

Frankfurt, 4 replicas in Sydney and 4 YCSB [37] clients in Frankfurt. Cassandra is set up to active replication with a replication factor of 2. The YCSB operations are configured to require a quorum on updates and only one response on reads, with a 50/50 mix of reads and updates. YCSB will direct most requests at the replicas in Frankfurt which are closer, however, a reply from the replicas in Sydney must always be present for a write quorum to succeed. In order to model the network topology in Kollaps, we collected the average latency and overall jitter between all the Amazon EC2 instances used, prior to executing the experiment on Amazon. Figure 10 shows the throughput-latency curve obtained from the benchmark on both the real deployment on Amazon EC2 and on Kollaps in our local cluster. The results are a close match, showing only slight differences after the turning point where response latencies climb fast, as Cassandra replicas are under high stress. This experiment illustrates how Kollaps can be used to assess the behavior of real systems in a controlled environment without requiring expensive real-life deployments.

The what-if use-cases. Finally, we present a possible use-case for Kollaps, evaluating applications in an hypothetical *what-if* scenario. For instance, it might be useful to study the behavior of Cassandra if the latencies between EC2 regions were to be halved. In practice, this would correspond to the scenario of moving 4 Cassandra nodes from Sydney (`ap-south`) to Seoul (`ap-northeast`). Instead of relying on a costly and time-consuming real-life deployment, Kollaps enables this study with a simple change in the topology configuration file. Figure 11 shows the obtained results. For the sake of readability and to ease the comparison, we further split the previous results (from the Frankfurt and Sydney deployment) into read and write curves and show them alongside the hypothetical results (obtained from Kollaps emulation) with the halved latency scenario. In this case, Cassandra behaves as expected: request latencies drop by about half and Cassandra reaches higher throughputs.

6 Limitations

We identify the following limitations in Kollaps.

Interactivity. For dynamic topologies, we compute offline, 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. The experimenter can afterwards decide, based on the reported error, if the experiment is satisfactory or if the same sequence of steps should be converted into a statically defined experiment.

Multipath routing. 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 containers, effectively discarding any multipath routing [36] 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 take this into account.

Multicast. 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, the fact that the bandwidth sharing is updated upon each iteration of the Emulation Manager forces a lower bound on the minimum latency that Kollaps can emulate. For example, 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 is better suited for emulating WAN deployments rather than emulating data-center environments. Possible approaches to mitigate these limitations are discussed in Section 7.

7 Conclusion and Future Work

The present work stems from 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. We assessed the feasibility of this idea by designing, implementing and evaluating Kollaps a decentralized topology emulator. 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 [78]. Finally, Kollaps can also be used by engineers to predict application performance and correctness under hypothetical, but fully controlled, network conditions.

In future work, we plan to address several limitations of Kollaps. To emulate networks with capacities higher than the infrastructure upon which Kollaps is running, we plan to investigate time dilation [42] capabilities, both in Kollaps and on the containers themselves. This can also help in mitigating the limitation of flows shorter than a single iteration of the Emulation Manager, and therefore enable Kollaps to emulate data-center environments, as recently proven to help in the context of SDN emulation [90]. To further enhance efficiency of the Emulation Manager and lowering the lower bound of the minimum latency that can be emulated, one can switch from periodic metadata dissemination to sending updates only upon changes in the flows. This approach reduces the metadata traffic, especially in the presence of long-lived flows, and will allow the Emulation Manager to react more promptly to short-lived flows.

Acknowledgments

We are grateful to Allyson Bessani for having shared with us the datasets of BFT-Smart and Wheat. We thank João Leitão and Pedro Fouto for their help in deploying the Cassandra experiments. We would also like to thank our shepherd KyoungSoo Park, and the anonymous referees for their valuable comments and helpful suggestions who led to this version of the paper. This work was partially supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 and project Lisboa-01-0145-FEDER- 031456 (Angainor).

References

[1] 1997. Linux Network Emulator. https://www.linux.org/docs/man8/tc-netem.html. Accessed: 2020-03-12.
[2] 2011. Curl. https://curl.haxx.se/. Accessed: 2020-03-12.
[3] 2012. Linux TCP Small Queues. https://lwn.net/Articles/507065/. Accessed: 2020-03-12.
[4] 2015. Summary of the Amazon DynamoDB Service. https://aws.amazon.com/message/5467D2/. Accessed: 2020-03-12.
[5] 2018. Apache Mina. https://mina.apache.org/. Accessed: 2020-03-12.
[6] 2018. Floodlight. http://www.projectfloodlight.org/floodlight/. Accessed: 2020-03-12.
[7] 2018. Memcached. https://memcached.org/. Accessed: 2020-03-12.
[8] 2018. Opendaylight. https://www.opendaylight.org/. Accessed: 2020-03-12.
[9] 2018. POX. https://github.com/noxrepo/pox. Accessed: 2020-03-12.
[10] 2018. u32 Universal Identifiers. http://man7.org/linux/man-pages/man8/tc-u32.8.html. Accessed: 2020-03-12.
[11] 2019. Aeron. https://github.com/real-logic/aeron. Accessed: 2020-03-12.
[12] 2019. Amazon EC2 - T1 micro instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts_micro_instances.html. Accessed: 2020-03-12.
[13] 2019. Apache Cassandra. https://cassandra.apache.org/. Accessed: 2020-03-12.
[14] 2019. Docker Security Capabilities. https://docs.docker.com/engine/security/. Accessed: 2020-03-12.
[15] 2019. Docker Swarm. https://docs.docker.com/engine/swarm/. Accessed: 2020-03-12.
[16] 2019. iPerf3. https://github.com/esnet/iperf. Accessed: 2020-03-12.
[17] 2019. Kubernetes. https://kubernetes.io/. Accessed: 2020-03-12.
[18] 2019. Linux LXC. https://linuxcontainers.org/. Accessed: 2020-03-12.
[19] 2019. Linux Traffic Control. https://linux.die.net/man/8/tc.
[20] 2019. Post-mortem Google Cloud Storage Incident. https://status.cloud.google.com/incident/storage/19002. Accessed: 2020-03-12.
[21] 2019. wrk2: A constant throughput, correct latency recording variant of wrk . https://github.com/giltene/wrk2. Accessed: 2020-03-12.
[22] 2020. NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark. Accessed: 2020-03-12.
[23] M. Avvenuti and A. Vecchio. 2006. Application-level network emulation: the EmuSocket toolkit. *Journal of network and computer applications* 29, 4 (2006), 343–360.
[24] Peter Bailis and Kyle Kingsbury. 2014. The Network is Reliable. *Queue* 12, 7 (2014), 20.
[25] J. Banks, J.S. Carson, and B.L. Nelson. 2010. *Discrete-event System Simulation.* Prentice Hall. https://books.google.pt/books?id=cqSNnmrqqbQC
[26] Albert-László Barabási and Réka Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999), 509–512. https://doi.org/10.1126/science.286.5439.509 arXiv:https://science.sciencemag.org/content/286/5439/509.full.pdf
[27] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control Plane Compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 476–489. https://doi.org/10.1145/3230543.3230583
[28] Alysson Bessani, Joao Sousa, and Eduardo E.P. Alchieri. 2014. State machine replication for the masses with BFT-SMART. *Proceedings - 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014*, 355–362. https://doi.org/10.1109/DSN.2014.43
[29] Ronald F. Boisvert. 2016. Incentivizing Reproducibility. *Commun. ACM* 59, 10 (Sept. 2016), 5–5. https://doi.org/10.1145/2994031
[30] Gaetano Bonofiglio, Veronica Iovinella, Gabriele Lospoto, and Giuseppe Di Battista. 2018. Kathará: A container-based framework for implementing network function virtualization and software defined networks. In *2018 IEEE/IFIP Network Operations and Management Symposium (NOMS'18)*. 1–9. https://doi.org/10.1109/NOMS.2018.8406267
[31] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Queue* 14, 1 (2016), 10.
[32] Marta Carbone and Luigi Rizzo. 2009. Dummynet revisited. *ACM SIGCOMM Computer Communication Review* 40, 2 (2009), 12. https://doi.org/10.1145/1764873.1764876
[33] Mark Carson and Darrin Santay. 2003. NIST Net: A Linux-Based Network Emulation Tool. *ACM SIGCOMM Computer Communication Review* 33, 3 (July 2003), 111–126. https://doi.org/10.1145/956993.957007
[34] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed*

Computing (PODC '07). ACM, New York, NY, USA, 398–407. https://doi.org/10.1145/1281100.1281103

[35] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. 2003. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review* 33, 3 (2003), 3–12.

[36] I. Cidon, R. Rom, and Y. Shavitt. 1999. Analysis of multi-path routing. *IEEE/ACM Transactions on Networking* 7, 6 (Dec 1999), 885–896. https://doi.org/10.1109/90.811453

[37] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[38] E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (Dec. 1959), 269–271. https://doi.org/10.1007/BF01386390

[39] Marius A. Eriksen. 2005. Trickle: A Userland Bandwidth Shaper for Unix-like Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, USA, 43.

[40] Sally Floyd and Eddie Kohler. 2003. Internet research needs better models. *ACM SIGCOMM Computer Communication Review* 33, 1 (2003), 29–34.

[41] S. Floyd and V. Paxson. 2001. Difficulties in simulating the Internet. *IEEE/ACM Transactions on Networking* 9, 4 (Aug 2001), 392–403. https://doi.org/10.1109/90.944338

[42] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. 2005. To Infinity and Beyond: Time Warped Network Emulation. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. 1–2. https://doi.org/10.1145/1095810.1118605

[43] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-friendly High-speed TCP Variant. *ACM SIGOPS Operating Systems Review* 42, 5 (July 2008), 64–74. https://doi.org/10.1145/1400097.1400105

[44] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. 2012. Reproducible Network Experiments Using Container-based Emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12)*. ACM, New York, NY, USA, 253–264. https://doi.org/10.1145/2413176.2413206

[45] Stephen Hemminger. 2005. Network emulation with NetEm. In *Proceedings of the Linux Conference*.

[46] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. 2008. Large-Scale Virtualization in the Emulab Network Testbed. In *USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association, USA, 113–128.

[47] David B Ingham and Graham D Parrington. 1994. Delayline: a wide-area network emulation tool. *Computing Systems* 7, 3 (1994), 313–332.

[48] V. Jacobson. 1988. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols (SIGCOMM '88)*. Association for Computing Machinery, New York, NY, USA, 314–329. https://doi.org/10.1145/52324.52356

[49] Frank Kelly. 1997. Charging and rate control for elastic traffic. *European transactions on Telecommunications* 8, 1 (1997), 33–37.

[50] Hormuzd M. Khosravi, Alexey Kuznetsov, Andi Kleen, and Jamal Hadi Salim. 2003. Linux Netlink as an IP Services Protocol. RFC 3549. https://doi.org/10.17487/RFC3549

[51] Wonho Kim, Ajay Roopakalu, Katherine Y. Li, and Vivek S. Pai. 2011. Understanding and Characterizing PlanetLab Resource Usage for Federated Network Testbeds. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC '11)*. ACM, New York, NY, USA, 515–532. https://doi.org/10.1145/2068816.2068864

[52] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[53] B. Lantz, B. Heller, and N. McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Ninth Workshop on Hot Topics in Networks (HotNets'10)*. ACM.

[54] Lorenzo Leonini, Étienne Rivière, and Pascal Felber. 2009. SPLAY: Distributed Systems Evaluation Made Simple (or How to Turn Ideas into Live Systems in a Breeze). In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. USENIX Association, Berkeley, CA, USA, 185–198. http://dl.acm.org/citation.cfm?id=1558977.1558990

[55] Luca Liechti, Paulo Gouveia, João Neves, Peter Kropf, Miguel Matos, and Valerio Schiavoni. 2019. THUNDERSTORM: a tool to evaluate dynamic network topologies on distributed systems. In *2019 IEEE 38th International Symposium on Reliable Distributed Systems (SRDS'19)*.

[56] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. CrystalNet: Faithfully Emulating Large Production Networks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 599–613. https://doi.org/10.1145/3132747.3132759

[57] L. Massoulie and J. Roberts. 2002. Bandwidth sharing: objectives and algorithms. *IEEE/ACM Transactions on Networking* 10, 3 (June 2002), 320–328. https://doi.org/10.1109/TNET.2002.1012364

[58] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article Article 2 (March 2014), 1 pages.

[59] Alberto Montresor and Márk Jelasity. 2009. PeerSim: A scalable P2P simulator. In *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*. IEEE, 99–100.

[60] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (March 2015), 66–73. https://doi.org/10.1145/2699417

[61] L. Nussbaum and O. Richard. 2008. Lightweight emulation to study peer-to-peer systems. *Concurrency and Computation: Practice and Experience* 20, 6 (2008), 735–749.

[62] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose. 2000. Modeling TCP Reno performance: a simple model and its empirical validation. *IEEE/ACM Transactions on Networking* 8, 2 (April 2000), 133–145. https://doi.org/10.1109/90.842137

[63] Vern Paxson and Sally Floyd. 1997. Why We Don't Know How To Simulate The Internet. In *In Proceedings of the 1997 Winter Simulation Conference*. 1037–1044.

[64] Roger D Peng. 2011. Reproducible research in computational science. *Science* 334, 6060 (2011), 1226–1227.

[65] M. Peuster, J. Kampmeyer, and H. Karl. 2018. Containernet 2.0: A Rapid Prototyping Platform for Hybrid Service Function Chains. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. 335–337. https://doi.org/10.1109/NETSOFT.2018.8459905

[66] Manuel Peuster, Holger Karl, and Steven Van Rossem. 2016. 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, 148–153.

[67] Maurizio Pizzonia and Massimo Rimondini. 2008. Netkit: Easy Emulation of Complex Networks on Inexpensive Hardware. In *Proceedings of the 4th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom '08)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL, Article Article 7, 10 pages.

[68] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. 2013. ElasticSwitch: Practical

Work-Conserving Bandwidth Guarantees for Cloud Computing. *ACM SIGCOMM Computer Communication Review* 43, 4, 351–362. https://doi.org/10.1145/2534169.2486027

[69] Rahul Potharaju and Navendu Jain. 2013. When the Network Crumbles: An Empirical Study of Cloud Network Failures and Their Impact on Services. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 15, 17 pages. https://doi.org/10.1145/2523616.2523638

[70] Z. Puljiz, R. Penco, and M. Mikuc. 2008. Performance analysis of a decentralized network simulator based on IMUNES. In *2008 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*. 519–525.

[71] Robert Ricci, Jonathon Duerig, Pramod Sanaga, Daniel Gebhardt, Mike Hibler, Kevin Atkinson, Junxing Zhang, Sneha Kasera, and Jay Lepreau. 2007. The Flexlab Approach to Realistic Evaluation of Networked Systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI'07)*. USENIX Association, USA, 15.

[72] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE* 39, 6 (2014), 36–38.

[73] George F. Riley and Thomas R. Henderson. 2010. The ns-3 Network Simulator. In *Modeling and Tools for Network Simulation*, Klaus Wehrle, Mesut Güneş, and James Gross (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 15–34. https://doi.org/10.1007/978-3-642-12331-3_2

[74] Luigi Rizzo. 1997. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review* 27, 1 (1997), 31–41. https://doi.org/10.1017/CBO9781107415324.004

[75] Roberto Roverso, Mohammed Al-Aggan, Amgad Naiem, Andreas Dahlstrom, Sameh El-Ansary, Mohammed El-Beltagy, and Seif Haridi. 2008. MyP2PWorld: Highly Reproducible Application-Level Emulation of P2P Systems. In *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*.

[76] Valerio Schiavoni, Etienne Rivière, and Pascal Felber. 2013. SplayNet: Distributed User-Space Topology Emulation. In *Middleware 2013*, David Eyers and Karsten Schwan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 62–81.

[77] V. Sinha and M. Wang. 2015. evalBox: A Cross-Platform Evaluation Framework for Network Systems. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 15–18. https://doi.org/10.1109/MASCOTS.2015.17

[78] Joao Sousa and Alysson Bessani. 2016. Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines. *Proceedings of the IEEE Symposium on Reliable Distributed Systems* 2016-Janua, 146–155. https://doi.org/10.1109/SRDS.2015.40

[79] Chunqiang Tang. 2009. DSF: A Common Platform for Distributed Systems Research and Development. In *Middleware 2009*, Jean M. Bacon and Brian F. Cooper (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 414–436.

[80] Marco Antonio To, Marcos Cano, and Preng Biba. 2015. DOCKEMU - A Network Emulation Tool. *Proceedings - IEEE 29th International Conference on Advanced Information Networking and Applications Workshops, WAINA 2015* (2015), 593–598. https://doi.org/10.1109/WAINA.2015.107

[81] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. 2002. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 271–284.

[82] Arun Vishwanath, Vijay Sivaraman, and Marina Thottan. 2009. Perspectives on router buffer sizing: Recent results and open problems. *ACM SIGCOMM Computer Communication Review* 39, 2 (2009), 34–39.

[83] K. V. Vishwanath, D. Gupta, A. Vahdat, and K. Yocum. 2009. ModelNet: Towards a datacenter emulation environment. In *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*. 81–82. https://doi.org/10.1109/P2P.2009.5284497

[84] G. Vu-Brugier, R. S. Stanojevic, D. J. Leith, and R. N. Shorten. 2007. A Critique of Recently Proposed Buffer-sizing Strategies. *ACM SIGCOMM Computer Communication Review* 37, 1 (Jan. 2007), 43–48. https://doi.org/10.1145/1198255.1198262

[85] SY Wang, CL Chou, and CC Lin. 2007. The design and implementation of the NCTUns network simulation engine. *Simulation Modelling Practice and Theory* 15, 1 (2007), 57–81.

[86] Elias Weingärtner, Florian Schmidt, Hendrik Vom Lehn, Tobias Heer, and Klaus Wehrle. 2011. SliceTime: A Platform for Scalable and Accurate Network Emulation. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, USA, 253–266.

[87] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, and H. Karl. 2014. MaxiNet: Distributed emulation of software-defined networks. In *2014 IFIP Networking Conference*. 1–9. https://doi.org/10.1109/IFIPNetworking.2014.6857078

[88] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 255–270.

[89] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. 2018. Pantheon: The Training Ground for Internet Congestion-control Research. In *USENIX 2018 Annual Technical Conference (ATC '18)*. USENIX Association, Berkeley, CA, USA, 731–743. http://dl.acm.org/citation.cfm?id=3277355.3277426

[90] Jiaqi Yan and Dong Jin. 2017. A lightweight container-based virtual time system for software-defined network emulation. *Journal of Simulation* 11, 3 (2017), 253–266. https://doi.org/10.1057/s41273-016-0043-8