THUNDERSTORM: a tool to evaluate dynamic network topologies on distributed systems

(Tool Paper)

Luca Liechti†, Paulo Gouveia*, João Neves*, Peter Kropf†, Miguel Matos*, Valerio Schiavoni†
*INESC ID & IST, University of Lisbon, Portugal. Email: miguel.marques.matos@tecnico.ulisboa.pt
†University of Neuchâtel, Switzerland. Email: valerio.schiavoni@unine.ch

***Abstract*—Network dynamics, such as sudden changes in latency or available bandwidth, have a significant impact on the performance of distributed systems. While such dynamics are common, especially in WAN deployments, existing tools lack the capabilities to systematically evaluate the impact of such changes in real systems. We present THUNDERSTORM, a tool to evaluate the impact of dynamic network topologies on the performance of large-scale distributed systems. THUNDERSTORM is a fully functional tool that integrates with Kubernetes and can be used to evaluate off-the-shelf applications. THUNDERSTORM defines an easy-to-use language to describe arbitrarily complex network topologies and dynamic events used to enrich the default container composition descriptors. Our evaluation, using micro- and macro-benchmarks, as well as off-the-shelf unmodified systems (*e.g.*, Apache Cassandra, MariaDB) shows that THUNDERSTORM is easy to use, accurate in reproducing dynamic behaviours and that it can help researchers uncover unexpected behaviours otherwise very costly to reproduce in real deployments typically captured only during malfunctioning periods.**

I. INTRODUCTION

Distributed systems are nowadays typically built and deployed as microservices [**?**], [**?**], [**?**], [**?**], [**?**] leveraging the flexibility offered by container technology such as Docker [**?**], LXC [**?**], or KataContainer [**?**]. In fact, engineers can fetch and deploy ready-to-use container images from public repositories (*e.g.*, DockerHub) or private ones, and all the major Infrastructure-as-a-Service providers (*e.g.*, Amazon Web Services [**?**], Google [**?**], Azure [**?**]) offer native support for containers. Once packaged as containers, these systems typically are deployed over wide-area networks, spanning intercontinental links (for instance when using different datacenter regions or availability zones). These links cross long Internet backbones and, therefore are subject to unpredictable network dynamics. Moreover, at large scale, such deployments must be resilient to churn[1] as failures become the norm rather than the exception.

While the deployment of such systems has been greatly streamlined by the aforementioned container technologies, understanding their behavior and performance evaluation is very challenging, in particular in the presence of churn and network dynamics. As a matter of fact, studying the performance of a deployed system when subject to varying network conditions, and the impact of these conditions on customer facing metrics such as response time, is specially difficult to measure in a systematic way, due to the large variance of the underlying network properties. This difficulty stems mainly from two phenomena which we discuss next.

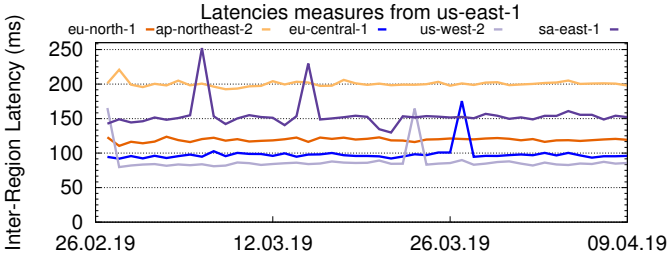


Fig. 1: Latency variability between five different AWS regions across the world over 45 days. Latencies vary on average between 90ms and 250ms, while spikes occur across all regions.

On the one hand network properties such jitter, packet loss, failures of middle-boxes (*i.e.*, switches, routers) are by definition difficult, if not impossible, to predict from the standpoint of a system developer, who has no control over the underlying network infrastructure. On the other hand, such conditions are the norm rather than the exception, in particular when considering large-scale wide-area networks that might cross several distinct administrative domains. As a motivating example, consider Figure 1 which shows the average latency between six AWS [**?**] regions over 45 days measured by https://www.cloudping.info. We observe that even in the infrastructure of a major cloud provider, there are significant and unpredictable variations in latency.

Variability is not limited to latency. We demonstrates this with a measurement experiment for two different cases. The first set of measures are taken between two stable endpoints inside university networks, respectively in Portugal and Switzerland, shown in Figure 2. The second case measures the network conditions between a remote AWS instance in the `ap-northeast-1a` zone (in Tokyo) and a server node in Switzerland (Figure 3). As we can observe, there are important variations both for bandwidth and latency. Such variability can have a dramatic effect not only on a system's performance but also on reliability as shown by recent post-mortem analysis of major cloud providers [**?**]. The challenge, therefore, is how to equip engineers and researchers with the tools that allow to systematically understand and evaluate how this variability

[1]The term *churn* is often used in peer-to-peer settings to indicate the unpredictable joining and leaving of nodes.

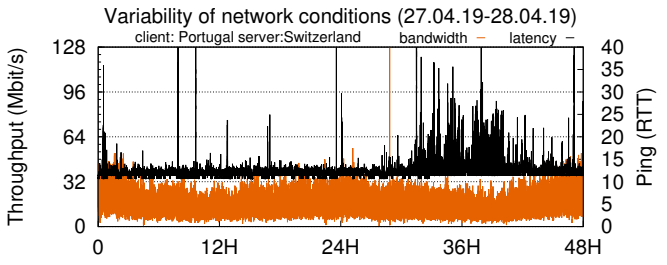


Fig. 2: Dynamic network conditions between two university campuses in Europe, in Portugal and Switzerland respectively.

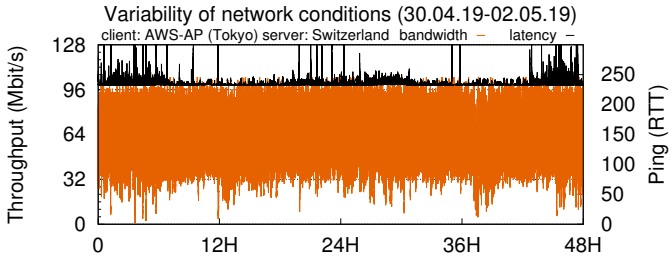


Fig. 3: Dynamic network conditions between a university node in Switzerland and a node in AWS `ap-northeast-1a` (Tokyo).

affects system's performance and behavior.

A wide range of existing network emulation tools allow to assess the behavior of an application in an arbitrary network with different scalability and accuracy trade-offs [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. Unfortunately, support for emulating network dynamics — which exist in practice and can severely affect application behavior and performance — is severely limited or completely lacking (see §II).

This paper presents THUNDERSTORM, an efficient and accurate tool to evaluate the performance of unmodified distributed systems under dynamic network conditions. In a nutshell, researchers can use THUNDERSTORM to describe in the same deployment manifest three aspects of a system: (1) the actual services to be deployed — the business logic, (2) the target network topology to emulate, statically defined and (3) the dynamic network behavior to be injected into the emulated network using an human-readable domain specific language (DSL). The key idea behind THUNDERSTORM is to leverage existing traffic shaping features in the Linux kernel to build the network emulation and dynamic features. THUNDERSTORM will be released as open-source, together with the deployment manifests used in this paper in order to provide full reproducibility of our experiments.[2]

The rest of this paper is organized as follows. In §II we survey related work and highlight how existing tools either completely lack the desired features or only offer a very limited support for those. In §III we describe THUNDERSTORM architecture, THUNDERSTORM's topology description language and some implementation details for its integration in Kubernetes. We present the experimental evaluation of the THUNDERSTORM prototype in §IV, using several micro-benchmarks as well as real-world distributed systems. Finally, §V concludes the paper.

[2]The version used in this paper is available from the PC Chairs.

II. RELATED WORK

In this section, we survey existing network emulation tools. Due to space constraints, we focus mostly on tools that allow to play with the dynamism of the underlying emulated topology. We explicitly omit from our review tools such as Trickle [**?**], EmuSocket [**?**] or NetEm [**?**], since they solely support statically defined point-to-point connections without any support to modify its features through time. Similarly, DockEmu [**?**], IMUNES [**?**] and NCTUns [**?**] support multi-node topologies but no dynamic behaviour.

Several recent works cover orthogonal aspects of network emulation and illustrate the relevance of controlled experiments. **CrystalNet** [**?**] focuses on large-scale emulation of the control-plane, enabling network engineers to evaluate changes to the control-plane before deploying them in production. THUNDERSTORM is complementary to CrystalNet, as we are focused on the data-plane instead. **Pantheon** [**?**] allows 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 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. We rely on the same insight to provide a network emulator able to accurately emulate large-scale topologies. **Chaos Monkey** [**?**] is a tool used inhouse by Netflix to test the resiliency of their distributed system. Similar to THUNDERSTORM, it allows to terminate virtual machine instances, to inject latency between services, or even making entire Amazon regions unavailable. Furthermore, it sends malformed requests to endpoints and performs other application-specific potentially harmful actions. In contrast, THUNDERSTORM only targets the network state and it is completely application-agnostic.

Table I gives an overview of other network emulators with dynamic capabilities. We group these tools by virtualization of the application running atop the emulated network: processes §II-A, virtual machines §II-B and containers §II-C.

A. Process-based emulation

Dummynet [**?**] operates directly on a specific network interface. It is a low-level tool to build full-fledged research testbeds, such as Modelnet [**?**], and is available in FreeBSD and MacOS kernels.

Modelnet [**?**] is a network emulation testbed that 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. THUNDERSTORM relies on Linux's Traffic Control (`tc`) [**?**] to provide similar low-level traffic shaping features, but (1)

TABLE I: Classification of network emulation tools with support for dynamic topology features. DSL=domain-specific language to inject the topology dynamics in the emulated network. Column *unit*: support for processes, virtual machines (VM) or containers.

Name	Synchronization	Topology dynamics			Link-level dynamic features				DSL	Deployment Unit
		node	bridge	link	BW	latency	lossrate	jitter		
ModelNet [?]	Centralized	✓	✗	✗	✓	✓	✓	✗	✗	Process
NIST Net [?]	Centralized	✗	✗	✗	✓	✓	✓	✓	✗	Process
Emulab [?]	Centralized	✓	✗	✗	✓	✓	✓	✗	✗	VM
Netkit [?]	Centralized	✗	✗	✗	✓	✓	✓	✗	✗	VM
Dummynet [?]	Centralized	✗	✗	✗	✓	✓	✓	✗	✗	Process
Mininet [?]	Centralized	✓	✓	✓	✓	✓	✓	✓	✗	Process
MaxiNet [?]	Centralized	✓	✓	✓	✓	✓	✓	✓	✗	Process
Mininet-HiFi [?]	Centralized	✓	✓	✓	✓	✓	✓	✓	✗	Process
SliceTime [?]	Centralized	✓	✗	✗	✗	✗	✗	✗	✗	VM
EvalBox [?]	Centralized	✓	✗	✓	✗	✗	✗	✗	✗	Process
ContainerNet [?]	Centralized	✓	✓	✓	✓	✓	✓	✓	✗	Container,VM
SPLAYNET [?]	Decentralized	✓	✗	✗	✗	✗	✗	✗	✓	Process
Kathará [?]	Centralized	✓	✓	✗	✗	✗	✗	✗	✗	Container
THUNDERSTORM	Decentralized	✓	✓	✓	✓	✓	✓	✓	✓	Container,VM

without requiring dedicated hosts (2) while providing a complete testbed integrated with large-scale container orchestration tools.

NIST Net [?] is a Linux kernel module and two user interfaces (command-line and graphical) to interact with it via API. When the module is loaded, all IP packets are sent to the NIST Net core that matches it against a set of user-provided rules. Packets may be delayed or dropped, and jitter may be applied. These parameters are changed at runtime. However, it is not integrated with any data-plane applications. A topology is simply a set of rules to apply between defined IP addresses. Hence, all topology-level dynamism has to be provided manually by the users.

EvalBox [?] provides an interface to bridge together simulated, emulated and real networks. It includes an emulation plugin based on Crystal [?] to support topology-level dynamics at runtime, but not link-level ones. EvalBox further supports OMNeT++[?] to include a wider range of link-level dynamics similar to those supported by THUNDERSTORM. However, this is only available in simulated EvalBox networks, not emulated ones.

SPLAYNET [?] extends SPLAY [?] to allow emulation of arbitrary network topologies, deployed across several physical hosts in a fully decentralized manner. It allows deploying several experiments simultaneously, as long as the underlying physical infrastructure can provide enough resources. SPLAYNET is fully distributed and 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. Splay lacks support for link-level dynamics. THUNDERSTORM adopts a similar fully decentralized approach while completely overcoming its limitations. In fact, THUNDERSTORM can be used with unmodified, off-the-shelf applications and assess their performances under different network conditions also including dynamic topologies. Support for churn in Splay/SplayNet is limited to nodes (joining/leaving). The Splay churn generator [?] inspired the design of the THUNDERSTORM Description Language, which improves it by allowing the definition of services, static topology elements and the dynamic behaviour of all the involved elements, while Splay's support is only limited to the churn behaviour.

B. VM-based emulation

Emulab [?] is a network emulation testbed that supports the deployment of user-provided operating systems. Similar to ModelNet and THUNDERSTORM, it relies on Linux's `tc` or BSD's Dummynet to shape the traffic directly at the edge nodes. Emulab can deploy large topologies across shared clusters, maintaining the user's requested resource allocation and the ability to perform this scheduling optimally. It supports only basic dynamics, *e.g.*, starting and stopping hosts during the experiment.

NetKit [?] runs applications within instances of the User-Mode Linux kernel (UML kernel), a fork of the Linux kernel that runs as a (user space) process within another instance of Linux. The VMs are connected according to a defined topology by means of *virtual hubs*. These are user-space processes that mimic the behavior of network switches, providing the notion of separate networks. In contrast to THUNDERSTORM, NetKit does not scale beyond single-host deployments.

SliceTime [?] provides a synchronisation framework for a software prototype within a simulated network. The prototype is any application wrapped inside a virtual machine running on top of the Xen hypervisor. Thus, it can be an application running in any x86-compatible OS. The network simulation runs on top of the ns-3 network simulator [?]. SliceTime supports the dynamic addition and removal of nodes to the experiment, but as its main focus is on time synchronisation, it offers no dynamic changing of link properties.

C. Container-based emulation

Finally, we survey emulation systems that support lightweight virtualization such as Docker containers or *e.g.* leveraging Linux' `cgroups`. Specifically, groups of processes share the same kernel (on the host), but have an isolated view of system resources, such as processes, network interfaces, or the network stack.

Mininet [?] emulates network topologies on a single host. It relies on Linux isolation mechanisms (*i.e.*, `cgroups`) to emulate separated network hosts. Similarly to Docker, it creates virtual Ethernet pairs inside separated namespaces and 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, THUNDERSTORM does not require these additional network instances, as the state of the emulation is maintained at each container. Mininet is limited to single-host deployments, preventing its use for large-scale resource-intensive systems not fitting in a single machine.

Mininet-HiFi [?] extends Mininet, adding mechanisms for performance isolation, resource provisioning and monitoring for performance fidelity.

Maxinet [?] allow for cluster deployments of worker hosts with native support for Docker containers. It creates tunnels for links that cross different workers. Maxinet natively supports large-scale Software Defined Networks (SDN). However, it requires all emulated hosts that connect to the same switch to be deployed on its same worker, a limitation that does not exist in THUNDERSTORM.

Similarly, **ContainerNet** [?], [?] extends Mininet to add native support for Docker containers and dynamic topologies. Still, it is limited to single-node deployments.

Kathará [?] leverages NetKit, but relies on Docker containers rather than virtual machines. Similarly to Emulab, Kathará has only limited support for dynamics, such as stopping and pausing of containers, and it lacks THUNDERSTORM's rich set of dynamic link manipulation primitives.

D. Dynamic Capabilities

Table I shows the topology- and link-level dynamic support of the mentioned systems. Some do not have a notion of 'topology' and simply apply rules to end-to-end connections based on IP addresses. Others offer a full set of link-level dynamics, but assume a static topology. Mininet, Mininet HiFi, Maxinet and ContainerNet offer the best support for dynamic topologies. While the available dynamic primitives can be used as building blocks for more complex scenarios, these systems do not provide a declarative specification language to drive such experiments. Besides, their design places restrictions on the possible deployed topologies: typically, they only support single-host deployments, or force switches and application nodes to be deployed in the same physical machine.

In contrast, THUNDERSTORM is agnostic of the deployment unit, application, transport protocol or topology and can scale to hundreds of containers by leveraging a multi-host deployment. We describe how this is achieved in the next section.

III. THUNDERSTORM

In this section, we describe the workflow and design of THUNDERSTORM. As a tool paper, we start by describing, in §III-A, the workflow that a user of THUNDERSTORM should follow, and then present the architecture of the Emulation Manager in §III-B. The THUNDERSTORM Description Language (TDL) is detailed in §III-C.

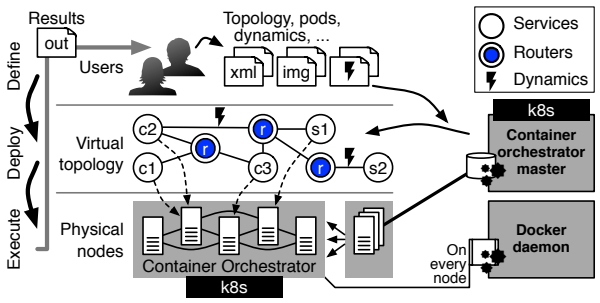


Fig. 4: THUNDERSTORM: workflow. Due to scheduling constraints, some containers (`c1` and `c2`) can end-up being co-hosted on the same physical node.

A. Workflow

The general workflow and architecture of THUNDERSTORM are depicted in Figure 4. There are two main steps required from the end-user in order to run a THUNDERSTORM experiment, which we detail next.

First, the user must provide a TDL file, describing: the network topology; its dynamics, if any; and the Docker images of the distributed application being evaluated (Figure 4, *define* step). These images can come from either private repositories or public ones such as Docker Hub [?]. We describe in detail the syntax and the language features available in the THUNDERSTORM Description Language in §III-C.

With the TDL instance defined, the user invokes the *deployment generator*, a Python library shipped with THUNDERSTORM. This transforms the TDL into a Kubernetes Manifest file, compatible with Kubernetes v1.14, ready to deploy.

The user can then use this file to deploy the experiment in any Kubernetes cluster (Figure 4, *deploy* step). Setting up the deployment involves a series of steps. First, we deploy the *bootstrapper* component as a `DaemonSet` [?]. When a container image is deployed as a `DaemonSet`, Kubernetes enforces that there will be exactly one container instance per node in the cluster. The goal of the bootstrapper is to check, in each node, for containers belonging to this experiment. This is achieved through the Kubernetes API, and leverages the `tags` [?] attached to each container by the deployment generator. The reason for this is to enable running multiple independent experiments in parallel.

When such a container is found, the bootstrapper starts a process called *Emulation Manager* — described in more detail in §III-B — which will be responsible for enforcing the topology constrains and dynamics of that container. Note that even though the Emulation Manager is started inside the bootstrapper container, it requires access to, and control of, the network namespace of the application container. We achieve this as follows. First, the bootstrapper is started with the `CAP_NET_ADMIN` capability [?]. In Kubernetes this is natively supported via the `securityContext.capabilities` Manifest element which is injected automatically by the deployment generator. Next, we use the `nsenter` syscall [?] to start the Emulation Manager in the namespace of the application

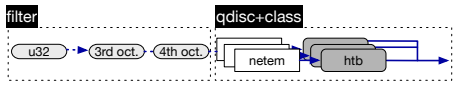


Fig. 5: TCAL: THUNDERSTORM's qdisc and filter hierarchy

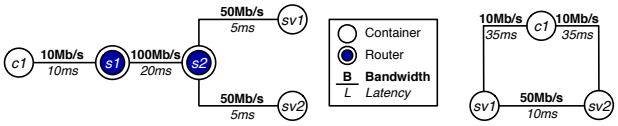


Fig. 6: Example of collapsing a simple topology

container. After these steps, the experiment starts executing (Figure 4, *execute* step) and the Emulation Manager takes control of the experiment and enforces the network topology and dynamics.

B. Emulation Manager

The Emulation Manager is a key element in the THUNDERSTORM architecture. It comprises two components: the *Emulation Core* which maintains the emulation model during the experiment, and the *Traffic Control Abstraction Layer* (TCAL), which is responsible for enforcing the topology constraints computed by the *core*.

The TCAL implements a library on top of Linux's Traffic Control (`tc`) [**?**]. It sets up the initial networking configuration, retrieves bandwidth usage, and modifies the maximum available bandwidth on paths. Figure 5 shows TCAL's architecture. It maintains a hierarchy of queuing disciplines (`qdiscs`) for enforcing the topology restrictions for each destination. For each destination, we create a `netem qdisc` [**?**] and a `htb qdisc` (hierarchical token bucket) [**?**]. The former is used to apply latency, jitter, and packet loss rate, while the latter enforces bandwidth constraints. The TCAL library is implemented in C and leverages the low-level `netlink` sockets [**?**] for performance reasons. It consists of ~1000 SLOC.

The *Emulation Core* maintains the emulation model and enforces the topology constraints. A key insight of our approach is that the emulation model is maintained in a fully decentralized fashion. Rather than directly emulating the network elements such as routers and switches and their internal state we instead emulate an equivalent topology at the end hosts.

The intuition behind our approach is to collapse the target topology into a set of shortest paths starting at one host and ending at another, leaving out all intermediaries (*i.e.* switches, routers), and exposing to the application just the compounded properties of the original links. Figure 6 gives an example of topology collapsing. Consider the full topology in the left side of the figure, with one client (*c1*) and two servers (*sv1* and *sv2*). The path between host $c1$ and server $sv1$ is composed of three different links, whose properties are collapsed into a single virtual link with properties equivalent to the end-to-end properties of the three original links. The resulting collapsed topology is depicted in Figure 6, right side.

More precisely, for latency, packet loss and jitter, we sum or multiply the properties of the links (variance for the jitter case) to compute the properties of the virtual link. The maximum bandwidth in the path is determined by the link with the least bandwidth. However, the bandwidth that a given link can use at any given point in time depends not only on the maximum bandwidth given by the physical capacity of the path but also by all ongoing active flows on each link. In particular, 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 there is congestion the network elements such as routers and switches start to drop packets. In unreliable transport protocols, such as UDP, packet loss is ignored, however in reliable protocols such as TCP, packet loss serves as a signal to adjust the throughput of the application, hence allowing competing flows to get a fair bandwidth share. In THUNDERSTORM, because we do not model network elements such as routers or switches, we rely instead on a model to compute a fair share of the bandwidth available for each competing flow. Specifically, we leverage the RTT-Aware Min-Max model [**?**], [**?**], which gives a share to each flow that is inversely proportional to its round-trip time. This model was inspired by TCP Reno [**?**], a widely adopted implementation of the TCP.

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 its whole available share. Therefore, when the sum of shares of all active flows is less than the maximum bandwidth on the link, we perform a maximization step that increases the share of the other flows, proportionally to their original shares.

The Emulation Core is implemented in Python v3.7. The execution is split into two stages: initialization and emulation loop. The initialization consists in building an emulation graph that represents the collapsed topology, determining the IP of the other containers via the Kubernetes API, and initializing the TCAL. This emulation graph is the main data structure, upon which the Emulation Core computes, among others, bandwidth shares and constraints, and buffers occupancy. The emulation loop maintains an additional data structure with the bandwidth usage of each instance and periodically executes the following steps: *(i)* clear the state of all local active flows; *(ii)* obtain the bandwidth usage by querying the TCAL; *(iii)* disseminate the local bandwidth usage to the other instances; *(iv)* compute bandwidth usage on each path and its constituent links using the data sent by the other instances; *(v)* enforce bandwidth restrictions.

Metadata dissemination is a critical aspect of the system, as we want it to be efficient and fast, in particular for large-scale deployments. We disseminate metadata using Aeron [**?**], an open-source, efficient and reliable UDP and IPC message transport protocol. For containers on the same machine, the metadata is exchanged via shared memory since it has no impact on the network. Metadata dissemination between

physical machines is shared through unicast reliable UDP messages provided by Aeron. Each metadata message carries the following information: *(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 <= 256 nodes, this implementation allows to pack all metadata for links and identifiers in a single byte each (2 bytes are used for bigger emulated topologies).

C. THUNDERSTORM *Description Language*

Experiments can be described using a text-based domain-specific language, named THUNDERSTORM Description Language (TDL). Our main goal when designing TDL was to provide easy-to-use primitives that allow to define large and complex topologies as well as dynamic events in a human-readable format. Listing 1 showcases all the main features of the language that we currently support.

The language describes the services (lines 3-6), the static topology (lines 10-14) and the dynamics in the topology (lines 16-26). The `service`, `bridge` and `link` elements can get an arbitrary number of `tags`. In the example, the `api` and `db` services belong to the backend, while the `server` belongs to the frontend. These three are grouped into the same application together, expressed by the `app` tag. The `client` is not part of the application. We use tags to group services and links together based on real-world criteria. For example, one of the most common causes for network "failures" is the distributed roll-out of software upgrades [**?**], *e.g.* for routers. Tags help to capture groups of devices sharing network status-relevant characteristics, *e.g.*, driver versions that could be updated at the same time. Tags could also be used to map services to data centers (*i.e.*, what if one's connection suddenly changes?) or to logical parts of a distributed sytem (frontend, backend). Bridges (line 8) must have unique names. Links must specify the source, destination and the properties (*e.g.*, latency, bandwidth, jitter, etc.). The `symmetric` keyword allow to easily create bidirectional links with the same specified properties.

The dynamic events can be expressed in a concise yet rich manner. In our example, we first start all application services (3 replicas for the `api` and the `db`, and 5 replicas for the `server`, lines 16-19), and after 30 seconds the clients (line 19). After 30 minutes, we inject several faults into the topology. The `churn` keyword crashes either an absolute number of services, or a certain share of all instances of that service. The `replace` keyword then specifies the probability of such a service to immediately re-join the cluster. At line 20, we specify that the `server` replicas will be subject to churn over a 3 hour period. In particular, 40% of the servers will crash uniformly at random over this period, and of those, 50% will be replaced immediately. Although the language allows to define events with a degree of randomness, such as the churn event above, it is possible to systematically reproduce the same order of events by setting a fixed random seed. We can also specify the dynamic behavior for a specific container. In the example, one server instance leaves the system at four hours and twenty, and joins 5 minutes later (lines 21-22).

```
1  bootstrapper thunderstorm:2.0
2
3  service server img=nginx:latest tags=frontend;app
4  service api img=api:latest tags=backend;app
5  service client img=client:1.0 command=['80']
6  service db img=postgres:latest tags=backend;app
7
8  bridges s1 s2
9
10 link server--s1 latency=9.1 up=1Gb down=800Mb
11 link api--s1 latency=5.1 up=1Gb symmetric
12 link s1--s2 latency=0.11 up=1Gb symmetric
13 link client--s1 latency=23.4 up=50Mb down=1Gb
14 link db--s2 latency=8.0 up=1Gb symmetric
15
16 at 0s api join 3
17 at 0s db join 3
18 at 0s server join 5
19 at 30s client join
20 from 30m to 3h30m server churn 40% replace 50%
21 at 4h20m server--s1 leave
22 at 4h25m server--s1 join
23 from 10h2m to 10h6m api--s1 flap 0.93s
24 from 12h to 24h tags=be leave 60%
25 from 15h to 15h20s server disconnect 1
26 at 18h20m api--s1 set latency=10.2 jitter=1.2
```

Listing 1: Example of experiment descriptor using the THUNDERSTORM description language. Link rates are given in 'per second'.

The language supports *link flapping*, where a single link connects and disconnects in quick succession [**?**]. In the experiment, the link between service `api` and bridge `s1` flaps every 0.93s during a period of 4 minutes (line 23). The `leave` action, used to define which entities should leave the emulation, takes as a parameter an absolute number or a share of all selected instances. At line 24, 60% of all nodes with the *backend* tag, chosen uniformly at random, will leave the experiment. Internally, when the language is translated into the lower level format used by the THUNDERSTORM engine, we keep track of all nodes that have joined, left, connected, or disconnected. Thus, if a percentage rather than an absolute number is provided, that is always relative to the amount of legal targets in the cluster *at that moment*.

The TDL parser is implemented in Python, leveraging Python Lex-Yacc (PLY) [**?**]. The output of the parser is a XML file, ready to be consumed by the deployment generator and starting the experiment workflow discussed in § III-A. Recall that upon initialization the Emulation Core builds an emulation graph representing the collapsed topology. The `dynamic` topology elements we just introduced correspond to modifications to this graph. Rather than computing modifications to the graph on the fly while the experiment executes, we pre-compute at initialization time all the modifications. This produces a sequence of graphs, whose paths' end-to-end-properties can be applied at runtime. We resort to this approach because, while computing all the required metadata 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.

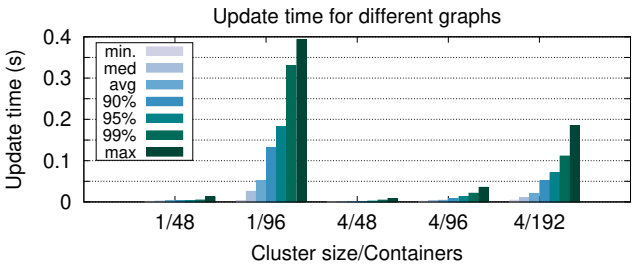


Fig. 7: Update time for graph objects of different cluster and graph sizes. We show the minimum, median, average, 90th, 95th and 99th percentile as well as the maximum time.

IV. Evaluation

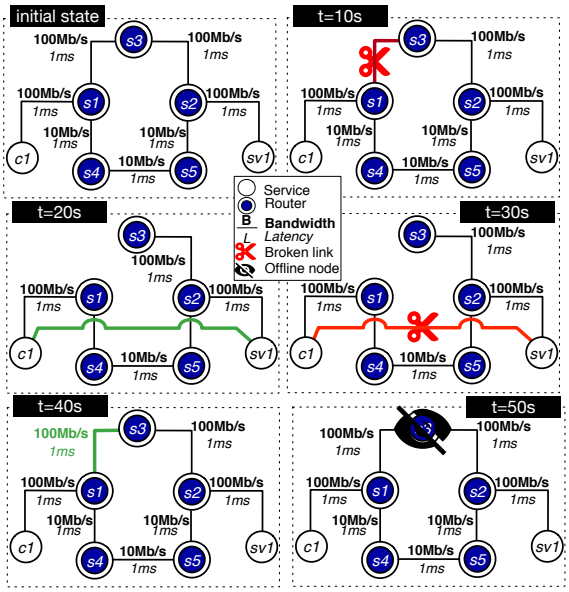
In this section, we evaluate ThunderStorm through a set of micro- and macro-benchmark, as well as real-world systems (*e.g.*, Cassandra, MariaDB). Our results show that ThunderStorm gives results close to real-world deployments and allows users to dynamically manipulate a chosen topology in many ways. Also, we show that ThunderStorm's support for background traffic is very close to bare metal deployments.

We start the evaluation with a series of micro-benchmarks that highlighting the individual features of ThunderStorm and justify the major design decisions. Next, we put ThunderStorm to the test with macro-benchmarks and real-world systems. First, we analyze the throughput-latency curve for a geo-replicated Cassandra experiment and obtain very similar results to the values obtained in the real world. Next, we observe in real time how the end-to-end latency in such a geo-replicated cluster changes when some path properties– *i.e.*, latency– are subject to change. Finally, we show how ThunderStorm can be used to look at the performance of MariaDB under lossy network conditions.

A. Evaluation Settings

The evaluation cluster is composed of 4 Dell PowerEdge R330 servers where each machine has an Intel Xeon E3-1270 v6 CPU and 64 GB of RAM. The single node cluster experiment shown in Figure 7 uses a Dell PowerEdge R630 with an 64-cores Intel Xeon E5-2683v4 with 128 GB of RAM. All nodes run Ubuntu Linux 18.04.2 LTS, kernel v4.15.0-47-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.

Graph update time. We begin by studying the time required to update the emulation graph for different cluster and graph sizes. Results are depicted in Figure 7. Updating the graph implies switching from the old graph to the new one, and invoking the necessary TCAL calls to adjust the emulation model to the new topology (§III-B). This experiment executes on a single machine with 64 cores (up to 96 containers) and a 4-node cluster (up to 192 containers). Note that this is very close to the 100 containers per machine limit imposed by Kubernetes [**?**]. Even in that extreme scenario, the update takes less than half a second. If the graph changes were done at run time, the time would be an order of magnitude larger



(a) Initial state: top corner left. Links disappear (at t=10s), a new link appears (t=20s), old links re-appear (t=40s), and bridges go offline (t=50s) and re-appear (not shown).

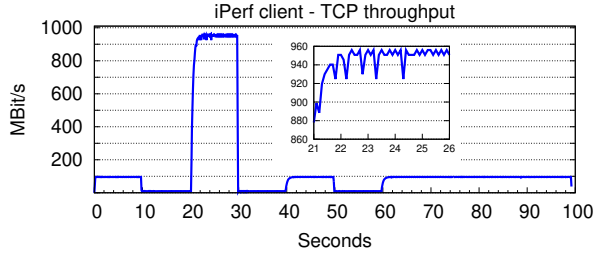
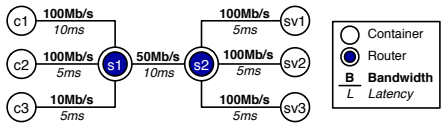


Fig. 8: Measured throughput at the server for the experiment depicted in Figure 8a.

(not shown) hence precluding our goal of having sub-second dynamics. Moreover, and as expected, for a given number of containers the update times decrease with the number of machines as the load becomes evenly spread across machines.

Topology dynamics. In this experiment, we inject simple dynamic behaviours into a small topology made of 1 client, 1 server and 5 bridges. The client opens a TCP connection towards the server and starts streaming data as fast as possible using `iPerf` [**?**]. The initial topology is depicted in Figure 8a (top-left corner). Then, every 10 seconds the topology changes, either by failing links and bridges or by adding new links. Figure 8 shows the observed throughput. As the link between `s1` and `s3` is removed, the shortest path from `c1` to `sv1` now goes through `s4` and `s5`. However, that link carries a mere 10Mbps, which significantly affects throughput. At 20s, a new direct 1Gbps link is inserted into the topology, from `c1` to `sv1` directly, and thus is the new shortest path. This has a significant impact in throughput which quickly raises to be close to 1Gbps. Ten seconds later that link is removed and throughput again quickly adjusts. Between 60s and 70s, switch `s3` is temporarily removed from the topology, forcing traffic to go through the 10Mbps link again. ThunderStorm allow to observe TCP's *slow start* behavior whenever available



(a) Dumbbell topology used to validate link-level dynamics and bandwidth throttling reaction time.

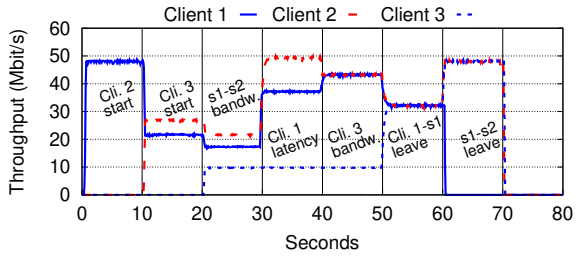


Fig. 9: Decentralized bandwidth throttling under a) changing flows, b) changing link properties, and c) changing topology.

bandwidth increases.

Link dynamics. This experiments has two goals: (1) demonstrate the accuracy of the RTT-aware min-max-model in computing bandwidth shares on a given link, and (2) evaluate the link-level dynamic capabilities of THUNDERSTORM. We use a 3-clients/3-server dumbbell topology, shown in Figure 9a. The link between the switches has a bandwidth of 50Mbps. The three clients (`c1`, `c2`, `c3`) have a bandwidth of 50Mbps, 50Mbps and 10Mbps, and an RTT of 50ms, 40ms, and 40ms, respectively. Note that clients `c1` and `c2` have the same bandwidth but different RTT.

The experiment proceeds as follows. Initially, only `c1` has an active flow, and hence it uses all the available bandwidth. At 10s, `c2` joins and thus it will compete for bandwidth over the shared link. At this point, since `c2` has a smaller RTT than `c1`, it gets a proportionally higher share of bandwidth, following the model described in §III. The reaction time, *i.e.*, how long it takes for THUNDERSTORM to throttle down the bandwidth available to `c1` due to the competing flow from `c2`, is around 0.3s. At 20s, `c3` starts and quickly saturates its outgoing link at $10Mbit/s$. The bandwidth of the other two clients gets proportionally adjusted to cope with this new competing flow, again in 0.3s.

Note that bandwidth shares are a function of bandwidth and RTT (*i.e.* 2 times the latency). To observe the dynamics behind these bandwidth shares, we change some of the underlying properties of the graph. At 30s, we double the bandwidth of the link connecting the switches. Since `c3` is already sending at full capacity, the extra bandwidth is shared between `c1` and `c2`, again inversely proportionally to their respective RTTs. At 40s, the link between `c1` and `s1` is set to have latency of 5ms instead of 10ms. Since on the server side of the topology, every link is identical, that makes the properties of paths from `c1` and `c2` to any server identical, reflected in their actual throughput becoming equal. At 50s, the link between `c3` and `s1` has its bandwidth set to 100Mbps as well. All links from clients to `s1` are now identical, resulting in three identical flows. At 60s, we remove the link between `c1` and `s1` from the

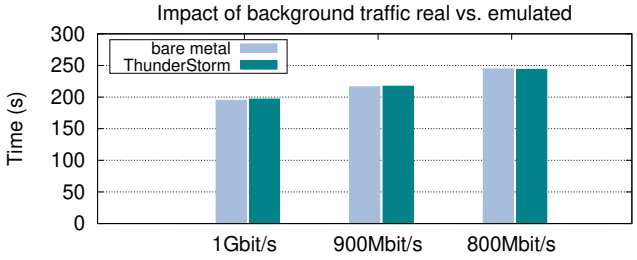


Fig. 10: Effect of background traffic on a 22GB secure file-transfer with bare-metal deployment and THUNDERSTORM.

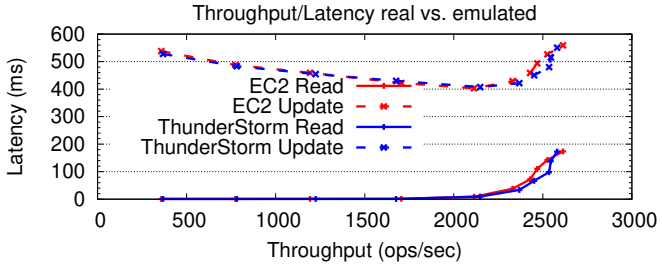


Fig. 11: Throughput/latency of a geo-replicated Cassandra deployment on Amazon EC2 and THUNDERSTORM.

topology. However, now the graph is not connected anymore, and `c1` cannot put any new data in the network. Since `c2` and `c3` have identical properties, the freed up bandwidth is equally shared between them. Finally, at 70s, the link connecting the switches is removed, and hence no traffic is observed.

Background traffic. In real-world scenarios, applications are rarely deployed on a dedicated network. Rather, a network infrastructure is typically shared among several competing systems, inducing *background traffic* to each other. We show THUNDERSTORM accuracy in emulating this behaviour with the following experiment. We start by measuring, on bare metal, the time it takes to secure file copy (*i.e.* using `scp`) a 22 gigabyte file over two hosts connected through a 1Gbps link in three scenarios: no backgroung traffic (*i.e.* in an isolated network), and with 100Mpbs and 200Mbps background traffic. Background traffic is inject with iPerf at the target rates. We then emulate the same scenario in THUNDERSTORM. However, rather than emulating real traffic, we dynamically adjust the available bandwidth on the link by the volume of the emulated background traffic. Results are depicted in Figure 10. As is it possible to observe, the resulting time are very similar (with a maximum variation of 1%), demonstrating the accuracy of our approach.

Cassandra. We now compare the results of benchmarking a geo-replicated Apache Cassandra [**?**] deployment on Amazon EC2 and on THUNDERSTORM. The deployment consists of 4 replicas in Frankfurt, 4 replicas in Sydney and 4 YCSB [**?**] clients in Frankfurt. Cassandra is set up to active replication with a replication factor of 2. YCSB is configured to require a quorum on updates and only one response on reads, with a 50/50 mix of reads and updates. This means that YCSB will direct most requests towards replicas in Frankfurt which are closer, however, a reply from the replicas in Sydney

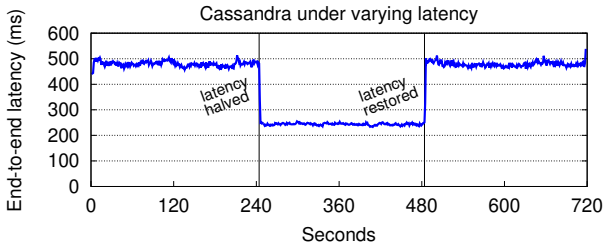


Fig. 12: Latency variations measured by YCSB during a transitory period: one of the replicas is moved to a far away region.

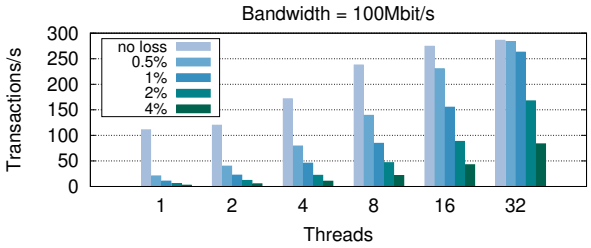


Fig. 13: MariaDB transactions per second over a 100Mbps link with different numbers of threads and levels of packet loss.

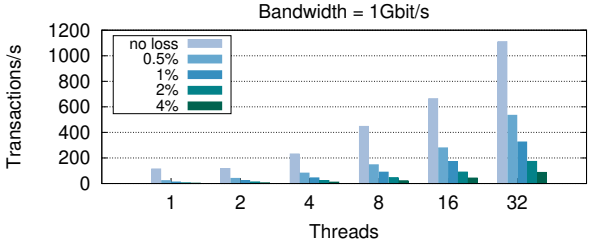


Fig. 14: MariaDB transactions per second over a 1Gbps link with different numbers of threads and levels of packet loss.

must always be present for a write quorum to succeed. In order to model the network topology in THUNDERSTORM, we collected the average latency and jitter between all the Amazon EC2 instances used, prior to executing the experiment. Figure 11 shows the throughput-latency curve obtained from the benchmark on both the real deployment on Amazon and on THUNDERSTORM. The curves for both reads and updates are a close match, showing only slight differences after the turning point where response latencies climb fast, as Cassandra replicas are under high stress. Although we found it surprising that the latencies on the update curve decrease slightly as throughput increases (before the turning point), this behavior occurs both on the real deployment and with THUNDERSTORM. This experiment demonstrates how such issues can be identified, debugged and eliminated with THUNDERSTORM before expensive real-life deployments.

Cassandra under a (thunder)storm. In this scenario, we show how THUNDERSTORM's unique support for dynamic topologies allows to easily uncover the behaviour of complex systems, such as NoSQL databases. In this experiment, the intercontinental link from `EU` to `AP` used for the Cassandra experiment in Figure 11 suddenly changes its latency to half (at 240s), and later on (at 480s) the original latency is restored. In Figure 12 we report the update latency observed by YCSB. Note that read operations do not use the intercontinental link and hence are not affected (not shown). This shows that the network dynamics imposed by THUNDERSTORM have a direct impact in client-facing metrics. Engineers and researchers can therefore use THUNDERSTORM to conduct controlled and reproducible experiments to assess the behavior of real system under a wide range of network dynamics and devise the best strategies to adopt when such events happen in production.

MariaDB over a lossy connection. We conclude our evaluation by studying the effect of lossy network connections on the performance of MariaDB [**?**], an open-source fork of MySQL. The experiments uses the official unmodified MariaDB Docker image, and the standard *sysbench* [**?**] image to execute the standard `oltp_read_write` benchmark. We vary the number of client threads in the benchmark to control the offered load. The topology is a simple point-to-point topology connected with a 100Mbps (Figure 13) or 1Gbps (Figure 14) link. We conducted a series of experiments with loss rates of 0, 0.5, 1, 2, or 4 percent. To establish a baseline of emulation accuracy, we ran the same benchmark on bare metal, registering only around 1% difference in performance.

This experiment shows that a database connection over TCP is very sensitive to packet loss. For a small number of threads, regardless of the connection bandwidth, as much as 0.5% packet loss can cause the number of transactions per second to be halved, or worse. Moreover, for loss rates of 2% and 4%, the difference in transactions per second between a 100Mbit connection an a 1Gbit one is just 1.12%. Interestingly, approaching the link throughput limit of the 100Mbps (as estimated from the lossless benchmark), the relative performance degradation due to packet drop decreases. We assume this performance degradation to be due to TCP throttling its throughput upon packet loss, as it would in a congested network. Finally, a packet loss rate of 2%, while certainly not negligible in nature, can cause a 1Gbps link to behave as if it had just a tenth of its actual bandwidth.

V. CONCLUSION

In this paper, we have shown that network dynamics, which are often unpredictable, have a large impact on the performance of distributed applications. Therefore, it is crucial for engineers, researchers and practitioners to understand how their systems behave under these conditions such that the appropriate designs, trade-offs and contingency plans can be selected. However, state-of-the-art tools limit the scale and scope of such experiments.

THUNDERSTORM is a tool acting as a decentralized topology emulator that can scale to hundrends of containers deployed across a cluster of commodity machines, and allows to inject a wide-range of network dynamic events throuh a compact human-readable language. Our experiments with static and dynamic settings show that THUNDERSTORM is able to accurately reproduce real-world deployments of off-the-shelf systems, such as Cassandra, and to expose the effect of faulty networks on the observable performances of DBMS systems such as MariaDB. The reproducibility of results is becoming increasingly important and we believe THUNDERSTORM can

be a useful tool to achieve this goal. Finally, THUNDERSTORM can also be used to predict application performance and correctness under hypothetical, but fully controlled, network conditions. Our THUNDERSTORM prototype is integrated in the Kubernetes container orchestrator and will be released under an open-source license.

ACKNOWLEDGMENTS

This work was partially supported by Fundo Europeu de Desenvolvimento Regional (FEDER) through Programa Operacional Regional de Lisboa and by Fundação para a Ciência e Tecnologia (FCT) through projects with reference UID/ CEC/ 50021/ 2019, Lisboa-01-0145-FEDER- 031456 (Angainor) and Lisboa-01-0145-FEDER-029271 (Cosmos).

REFERENCES

[1] "A Microscope on Microservices by Netflix Technology Blog," https://link.medium.com/AnOLr9crdW, 2015.
[2] Majors and Kromhout, "Keep Calm and Carry On: Scaling Your Org with Microservices." San Francisco, CA: USENIX Association, 2017.
[3] https://blog.twitter.com/engineering/en_us/a/2015/all-about-apache-aurora.html, 2015.
[4] https://segment.com/blog/why-microservices/, 2016.
[5] Taibi, Lenarduzzi, and Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.
[6] Merkel, "Docker: lightweight Linux containers for consistent development and deployment," p. 2, 2014. [Online]. Available: https://bit.ly/2IuhKBv
[7] https://linuxcontainers.org/.
[8] https://katacontainers.io/.
[9] https://aws.amazon.com/ecs/.
[10] https://cloud.google.com/kubernetes-engine/.
[11] https://azure.microsoft.com/en-us/services/kubernetes-service/.
[12] https://status.cloud.google.com/incident/cloud-networking/18012?m=1.
[13] Vahdat *et al.*, "Scalability and accuracy in a large-scale network emulator," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 271–284, 2002.
[14] Hibler *et al.*, "Large-scale virtualization in the Emulab network testbed," in *USENIX ATC'08*.
[15] Handigol, Heller, Jeyakumar, Lantz, and McKeown, "Reproducible network experiments using container-based emulation," in *CoNEXT '12*.
[16] Weingärtner, Schmidt, Vom Lehn, Heer, and Wehrle, "SliceTime: a platform for scalable and accurate network emulation," in *NSDI'11*.
[17] To, Cano, and Biba, "DOCKEMU - A Network Emulation Tool," *Proceedings of the 29th IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2015*, pp. 593–598, 2015.
[18] Sinha and Wang, "evalBox: A cross-platform evaluation framework for network systems," in *IEEE MASCOTS'15*.
[19] Peuster, Karl, and Van Rossem, "Medicine: Rapid prototyping of production-ready network services in multi-pop environments," in *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN),*. IEEE, 2016, pp. 148–153.
[20] Schiavoni, Riviere, and Felber, "SplayNet: Distributed User-Space Topology Emulation," in *ACM/IFIP/USENIX Middleware'13*.
[21] Eriksen, "Trickle: A userland bandwidth shaper for unix-like systems." in *USENIX ATC'05*.
[22] Avvenuti and Vecchio, "Application-level network emulation: the emusocket toolkit," *Journal of network and computer applications*, vol. 29, no. 4, pp. 343–360, 2006.
[23] Hemminger, "Network emulation with NetEm," in *Proceedings of the Linux Conference*, 2005.
[24] Puljiz, Penco, and Mikuc, "Performance analysis of a decentralized network simulator based on IMUNES," in *International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, ser. SPECTS, 2008.
[25] Wang, Chou, and Lin, "The design and implementation of the nctuns network simulation engine," *Simulation Modelling Practice and Theory*, vol. 15, no. 1, pp. 57–81, 2007.
[26] Liu *et al.*, "Crystalnet: Faithfully emulating large production networks," in *SOSP'17*.
[27] Yan *et al.*, "Pantheon: the training ground for Internet congestion-control research," in *USENIX ATC'18*.
[28] Basiri *et al.*, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, May 2016.
[29] Carson and Santay, "NIST Net-a linux-based network emulation tool," *ACM SIGCOMM Com. Comm. Rev.*, vol. 33, no. 3, pp. 111–126, 2003.
[30] Pizzonia and Rimondini, "Netkit: easy emulation of complex networks on inexpensive hardware," in *TridentCom'08*.
[31] Carbone and Rizzo, "Dummynet Revisited," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 2, p. 12, 2009.
[32] Lantz, Heller, and McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *ACM HotNets'10*.
[33] Wette, Draxler, Schwabe, Wallaschek, Zahraee, and Karl, "Maxinet: Distributed emulation of software-defined networks," in *IFIP Networking'14*.
[34] Peuster, Kampmeyer, and Karl, "ContainerNet 2.0: A Rapid Prototyping Platform for Hybrid Service Function Chains," in *IEEE NetSoft'18*.
[35] Bonofiglio, Iovinella, Lospoto, and Di Battista, "Kathará: A container-based framework for implementing network function virtualization and software defined networks," in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2018, pp. 1–9.
[36] Hubert, *tc - show / manipulate traffic control settings*, 2001. [Online]. Available: http://man7.org/linux/man-pages/man8/tc.8.html
[37] Wang, Shojania, and Li, "Crystal: An emulation framework for practical peer-to-peer multimedia streaming systems," in *ICDCS'08*.
[38] https://omnetpp.org/, 2019.
[39] Leonini, Rivière, and Felber, "Splay: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze)," in *NSDI'09*.
[40] Riley and Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.
[41] "Docker hub," https://hub.docker.com/, 2019.
[42] "Kubernetes: Daemonset," https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/, 2019.
[43] "Kubernetes: Labels and selectors," https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/, 2019.
[44] "Docker Security Capabilities," https://docs.docker.com/engine/security/, 2019.
[45] "nsenter man page," http://man7.org/linux/man-pages/man1/nsenter.1.html, 2019.
[46] Fabio Ludovici, *NETEM(8)*, 2011. [Online]. Available: http://man7.org/linux/man-pages/man8/tc-netem.8.html
[47] Martin Devera, *HTB - Hierarchy Token Bucket*, 2002. [Online]. Available: http://man7.org/linux/man-pages/man8/tc-htb.8.html
[48] Khosravi, Kuznetsov, Kleen, and Salim, "Linux Netlink as an IP Services Protocol," RFC 3549, Jul. 2003. [Online]. Available: https://rfc-editor.org/rfc/rfc3549.txt
[49] Kelly, "Charging and rate control for elastic traffic," *European transactions on Telecommunications*, vol. 8, no. 1, pp. 33–37, 1997.
[50] Massoulié and Roberts, "Bandwidth sharing: Objectives and algorithms," *IEEE/ACM ToN*, vol. 10, no. 3, pp. 320–328, 2002.
[51] Padhye, Firoiu, Towsley, and Kurose, "Modeling TCP Reno performance: a simple model and its empirical validation," *IEEE/ACM ToN*, vol. 8, no. 2, pp. 133–145, 2000.
[52] "Aeron," https://github.com/real-logic/aeron, 2019.
[53] Potharaju and Jain, "When the Network Crumbles: An Empirical Study of Cloud Network Failures and Their Impact on Services," in *ACM SoCC'13*.
[54] "Ply (python lex-yacc)," https://www.dabeaz.com/ply/, 2019.
[55] "Building large clusters with kubernetes," https://kubernetes.io/docs/setup/cluster-large/, 2019.
[56] https://github.com/esnet/iperf, 2019.
[57] Lakshman and Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922
[58] Cooper, Silberstein, Tam, Ramakrishnan, and Sears, "Benchmarking Cloud Serving Systems with YCSB," in *ACM SoCC '10*, 2010.
[59] https://mariadb.org/, 2019.
[60] "Sysbench: scriptable database and system performance benchmark," https://github.com/akopytov/sysbench, 2019.