KOLLAPS: Decentralized and Dynamic Topology Emulation

Paulo Jorge Louseiro Gouveia
Instituto Superior Técnico, Universidade de Lisboa

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

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

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

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

I. INTRODUCTION

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

The advent of container technology (*e.g.*, Docker [1], Linux LXC [2]) and container orchestration (*e.g.*, Docker Swarm [3], Kubernetes [4]) greatly simplifies the deployment and partially addresses environment heterogeneity. The main challenge is thus how to systematically evaluate the system in a WAN environment. As a matter of fact, the inherent variability of WAN conditions (*i.e.*, failures, contention and reconfigurations), makes it hard to assess the impact of changes in the application logic. Can we actually be sure the observed performance improvement was really due to the changes made to the application, or was it just due to a *lucky* run when the network was lightly loaded? How is performance affected by common network dynamics, such as background traffic, or link flapping? The very same questions and issues also arise in the reproducibility crisis currently plaguing down the distributed systems' community [5], [6]. 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 [7], CloudLab [8], or PlanetLab [9] where such experiments are conducted tend to get overloaded right before system conference deadlines [10]. We therefore need tools to systematically assess and reproduce the evaluation of large-scale applications.

One approach to systematically evaluate a large-scale distributed system is to resort to simulation, which relies on models that capture key properties of the target system and environment [11]. Simulation provides full control of the system and environment — and hence it is fully reproducible — and allows to study the model of the system in a variety of scenarios. However, simulations suffer from several well-known problems. On one hand, there is a large gap between the simulated model and the real-world deployment, usually leading to several unforeseen behaviors not captured by the model [12]–[15]. On the other hand, even if the simulated model yields correct results, the real implementation is not guaranteed to faithfully follow the simulated model. Moreover, despite some efforts to model complex systems either for formal method analysis [16] or simulation, this is, to the best of our knowledge, seldom the case for large-scale systems.

The alternative is to resort to network emulation. In network emulation, the real system is run against a model of the network that replicates real-world behavior by modeling a network topology together with its network elements, including switches, routers and their internal behavior. Emulation thus allows to reach conclusions about the behavior of the real system in a concrete scenario rather than its model. Unfortunately, state-of-the art network emulators suffer from several limitations. MiniNet [17] is limited to a single physical machine and therefore cannot be used to emulate a large-scale system. MaxiNet [18] and the multi-host version of MiniNet support distributed clusters but scale poorly [19]. Approaches such as ModelNet [20] rely on a dedicated cluster of nodes to maintain the emulation model to which the application nodes must connect. However, accuracy is highly dependent on application traffic patterns and quickly degrades with a modest increase in the number of application nodes. CrystalNet [19] accurately emulates the *control-plane* of large-scale networks (*e.g.* routing tables, software switch versions or device firmwares) but cannot be used to emulate the data-plane (*e.g.* latency, bandwidth), and hence evaluate the behavior of large-scale distributed systems.

One could also resort to emulation testbeds (*e.g.*, Emulab [7]). While this provides a semi-controlled environment and network, it cannot model network dynamics and thus one cannot assess their impact on application behavior.

To summarize, to the best of our knowledge, existing tools do not allow to systematically assess and reproduce the evalu-

ation of large-scale applications subject to network dynamics. This is where KOLLAPS comes in. KOLLAPS is a decentralized network topology emulator. It emulates a network topology beneath unmodified containerized applications, is agnostic to the application language and transport protocol, and can scale to thousands of containers, maintaining an emulation accuracy that is on par with other state-of-the-art systems that emulate the full state of the network.

KOLLAPS builds on previous work (NEED [21]) and addresses the limitations of it, namely scalability and efficiency, which are the goal of this thesis to solve. First, the initial iteration of KOLLAPS requires a broadcast of information regarding the flows and bandwidths used by each container to every other container. This hurts the system's scalability because with a significant amount of application containers the impact of these packets on the network is no longer negligible. Second, it is possible for the computations to find the maximum bandwidths on paths to be repeated on different containers. The idea behind KOLLAPS is to share the flow information through shared memory for containers on the same physical host, reducing the network overhead, and to merge emulation processes into one, being the computations done only once for every physical host.

II. RELATED WORK

In this section, we detail how KOLLAPS compares with some representative network emulation systems, and we consider simulation-based tools (*e.g.*, NS-3 [22], PeerSim [23]) out of the scope.

Two recent works cover orthogonal aspects of network emulation and illustrate the relevance of controlled experiments. CrystalNet [19] focuses on large-scale emulation of the control-plane, enabling network engineers to evaluate changes to the control-plane before deploying them in production. KOLLAPS is complementary to CrystalNet, as we only cope with the data-plane. Pantheon [24] is a system 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 [25] 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 a lot of 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 [26] and DelayLine [27] are userspace tools, similar in design and features to Trickle.

SPLAYNET [28] extends SPLAY [29] 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 [30] operates directly on a specific network interface. It is used as a low-level tool to build full-fledged research testbeds, such as Modelnet [31]. 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. KOLLAPS relies on Linux's Traffic Control (`tc`) [32] to provide similar 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.

Emulab [33] is a network emulation testbed that supports the deployment of user-provided operating systems. Similar to ModelNet and KOLLAPS, 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 while maintaining the user requested resource allocation, and the ability to perform this scheduling optimally. Its graph coarsening technique is similar in principle to the KOLLAPS approach for collapsing the topology.

Container-based approaches. Finally, we look at emulation tools used with containers. Mininet [34] 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. Maxinet [18] 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 [35], [36] 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 [37], a network emulation tool based on Docker containers. It supports L1 and L2 link layer types, leveraging NS-3 [22], a network simulation tool to emulate wireless network topologies, which we intend to support in KOLLAPS in future work.

To the best of our knowledge, KOLLAPS is the only system that can be used to deploy unmodified applications over emulated topologies without any centralized node, supporting a richer set of emulation features and still providing emulation accuracy on par with existing state-of-the-art systems.

III. ARCHITECTURE

In this section we describe the architecture of KOLLAPS and discuss the design of its components. KOLLAPS is a network emulation system that takes advantage of Docker to emulate arbitrary network topologies on a cluster of physical machines. The main standout feature of KOLLAPS is the combination of containers, for lightweight deployment of applications on a cluster, with techniques for performing point-to-point emulation in a decentralized way. This work was based on an already existing project, NEED [21]. The design employed by NEED suffers from two problems that limit its scalability. The first is the loss of the metadata UDP packets containing information with the flows. The second is the delay of the metadata packets that can be caused by heavy networking load or by the fact that windows of time between cycles of the emulation loop are not guaranteed to be constant across all instances. In this section, we give an overview of the KOLLAPS architecture and how its components where changed from the previous iteration.

Overall architecture. The major hindrance to KOLLAPS is the huge amounts of exchanged metadata from every container to every other container, which severely impacts its scalability and response time. Further developments by the original author over the initial prototype of NEED, have made all `Emulation Managers` accessible from a separate container, as opposed to strictly running alongside applications in their respective container. A special container, named `God Container`, now has access to all emulation cores running on a given physical host. This makes it possible to replace the exchange of UDP messages between instances running on the same physical host with shared process memory.

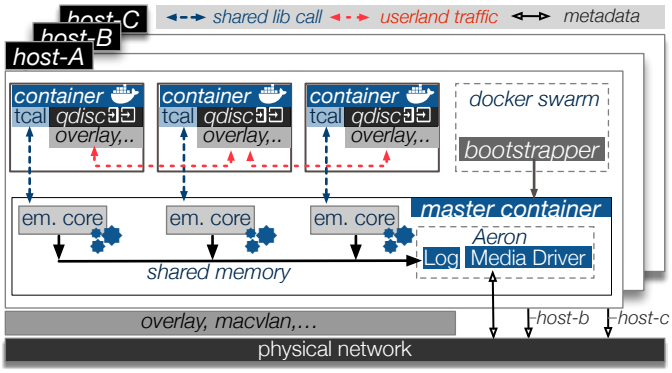

Fig. 1: KOLLAPS architecture.

A general overview of the KOLLAPS architecture can be seen in Figure 1. KOLLAPS is made up of six main components: the `Deployment Generator`, the `Bootstrapper`, the `God Container`, the `Emulation Manager`, the `tc Abstraction Layer` and the `Dashboard`.

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

Bootstrapper and Master Container. In order for an application running inside a Docker container to use `tc` (as we do via the TCAL), it must be executed with the CAP_NET_ADMIN capability [38]. Although Docker allows executing applications in standalone containers with user-specified capabilities, this feature is currently unavailable in Docker Swarm.

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

releases of Docker Swarm to allow for a simplified mechanism. In the case of Kubernetes, such restrictions do not hold and the `God Container` can be started immediately and operate by itself, without the need for the `Bootstrapper`.

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

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

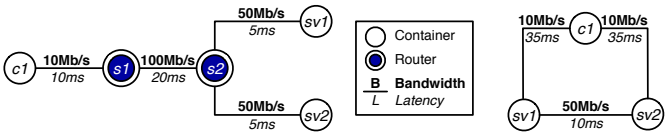

Fig. 2: Example of a topology and an equivalent collapsed topology.

Bridges do not really exist in the emulation so the `Emulation Manager` executes only on services. The specified topology is collapsed, into a different topology where the services are directly connected to all the other reachable services from the original topology, see Figure 2. However, the properties of the original topology still need to be preserved.

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

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

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

$$Loss(\mathcal{P}) = 1.0 - \prod_{i=1}^{n}(1.0 - Loss(l_i)) \tag{3}$$

$$\max Bandwidth(\mathcal{P}) = \min_{\forall l_i \in \mathcal{P}} Bandwidth(l_i) \tag{4}$$

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

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

After these steps, the `Emulation Managers` uses all the collected metadata to decide on new bandwidth limits for each path. These calculations are also offloaded to another process, which computes the new bandwidth limits according to the RTT Aware Min-Max model [39], [40].

tc abstraction layer (TCAL). The `tc abstraction layer` is implemented as a library, written in C, and provides a concise high-level API to setup initial networking conditions, retrieve bandwidth usage, and modify the maximum available bandwidth on paths.

The exposed API is the same from NEED except for the `init` function used to initialize the `tc` infrastructure, which no longer requires the control port since exchange of emulation metadata is done in a different manner. The remaining functions work as follows. The `initDestination` function sets up the `tc` infrastructure for enforcing delay, packet loss and bandwidth throttling on all traffic directed to the specified IP address. The `changeBandwidth` function modifies the `tc` infrastructure changing the maximum allowed bandwidth to the specified IP address. The `updateUsage` function takes a snapshot of the number of bytes sent to each previously setup destination. The `queryUsage` function retrieves this information for a specific destination. Finally the `tearDown` function destroys all the created `tc` infrastructure.

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

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

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

IV. EVALUATION

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

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

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

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

Link-level emulation accuracy. We begin by evaluating the accuracy of our bandwidth shaping mechanism under a simple scenario that consists of only two services connected by one link. One of the services executes an iPerf [41] server and the other executes an iPerf client. We access accuracy across a range of different target bandwidths, and compare the results with the same experiment executed with NEED, Mininet and Trickle. On all cases the iPerf client is configured to execute for 60 seconds before terminating, the average throughput reported at the server can be seen in Table I.

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

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

Emulation reaction time. The `pool_period` impacts the reaction time for traffic throttling, in particular when the number of competing flows varies over time. To measure this, we set up a simple 3-clients/3-servers dumbbell topology, depicted in Figure 3. The link connecting both switches has a maximum bandwidth of $50Mbit/s$. This results in connections between the clients and servers with bandwidths of $50MBit/s$, $50Mbit/s$ and $10Mbit/s$, and RTTs of $50ms$, $40ms$, and $40ms$, respectively. Note that clients `C1` and `C2` have the same bandwidth but different RTTs. Given our bandwidth sharing model, under network contention client `C2` will get a higher share than client `C1`, inversely proportional to their respective RTTs.

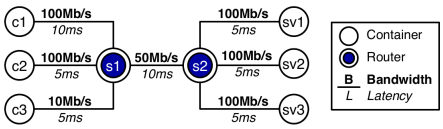

Fig. 3: Dumbbell topology with 3 clients, 3 servers, where the links connecting the clients to switch s1 all have different properties.

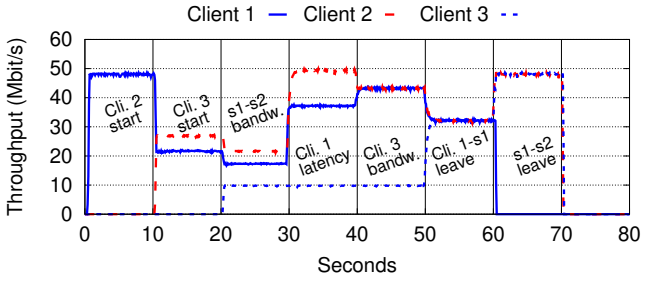

Fig. 4: Reaction time to throttle flows that vary in time.

The experiment seen in Figure 4 proceeds as follows. Initially, only `C1` has an active flow, and hence it uses all the available bandwidth. After 10 seconds, `C2` starts and thus it

will compete for bandwidth over the shared link. At this point, since C2 has a smaller RTT than C1, it gets a proportionally higher share of bandwidth, following the model we describe in Section III. The reaction time, *i.e.*, how long it takes for KOLLAPS to throttle down the bandwidth available to C1 due to the competing flow from C2, is around one second. At second 20, C3 starts and quickly reaches its bandwidth limit of $10Mbit/s$. The bandwidth of the other two clients gets proportionally adjusted to cope with this new competing flow also, also in around one second.

Reaction time is similarly around one second when dynamically changing properties of links or the topology itself. At second 30, we double the bandwidth of the link connecting the switches. Since C3 is already sending at full capacity, the extra bandwidth is shared between C1 and C2, again inversely proportionally to their respective RTTs. At second 40, the link between C1 and s1 is set to have latency of 5ms instead of 10ms. Since on the server side of the topology, every link is identical, that makes the properties of paths from C1 and C2 to any server identical, which is reflected in their bandwidth becoming equal. At second 50, the link between C3 and s1 has its bandwidth set to 100Mbps as well. All links from clients to s1 are now identical, resulting in three identical flows. At second 60, we remove the link between C1 and s1 from the topology. In other words, the graph is not connected anymore and C1 is no longer able to put flows on the network anymore. The freed up bandwidth is equally shared between C2 and C3. Finally, at second 70, also the link connecting the switches is removed. Naturally, we do not measure any traffic anymore.

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

Metadata bandwidth usage. KOLLAPS relies on metadata dissemination to accurately model and emulate bandwidth restrictions for competing flows. In this section, we study the cost of this metadata dissemination by deploying a simple dumbbell topology. We use iPerf [41] to generate steady TCP traffic at $50Mbit/s$, the maximum capacity of the shared link. We denote each configuration by a tuple M/C/F with M number of physical machines in the cluster, C total deployed containers and F number of concurrent data flows. Results are shown in Figure 5.

Every container submits the metadata to the Aeron Media Drivers in the system and every subscriber receives the metadata from the local Aeron Media Driver. For experiments running on a single machine, all messages are exchanged through shared memory and no network bandwidth is used. From Figure 5, we can see that the bandwidth used by the metadata is not related to the amount of containers as the messages are only exchanged between Aeron Media Drivers. Metadata bandwidth grows with the number of flows since increasing the number of flows means that there is more information to be shared. It also grows with the number

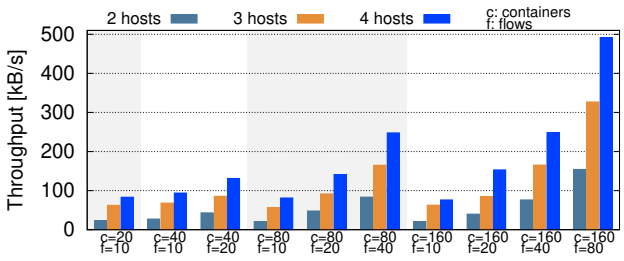

Fig. 5: KOLLAPS metadata bandwidth with an increasing number of flows and containers, 1 source and 1 destination per flow, F: concurrent data flows, C total deployed containers.

of physical machines since this means there are more Aeron Media Drivers to share the information with. However, metadata bandwidth does not increase with the emulated application bandwidth because the messages will still have the same size.

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

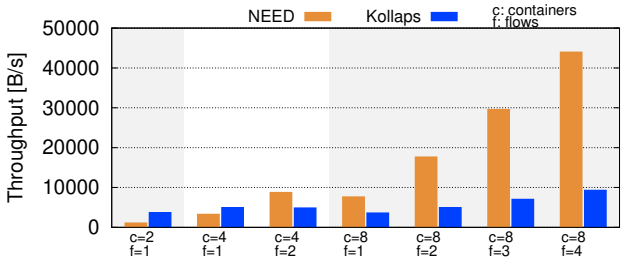

Fig. 6: Comparison of generated metadata between KOLLAPS and NEED, 1 source and 1 destination per flow. F: concurrent data flows, C total deployed containers.

On Figure 6 we can see a comparison between the metadata generated by KOLLAPS and NEED for the same topologies and workload. In this experiment, we run both systems on a cluster with only two nodes. In NEED every container sends the metadata to every other container. This means the metadata circulating in the system will scale with the number of containers with $O(c^2)$, c being the number of containers. With KOLLAPS, the metadata disseminated scales with the number of nodes in the cluster (i.e. Aeron Media Drivers). In this example, with only two physical hosts, every packet of metadata travels to only one destination: the other host. All other exchanges of metadata are done through shared memory.

Scalability of bandwidth emulation accuracy. The pool_period affects metadata bandwidth but it can also

affect, if configured incorrectly, emulation accuracy. Shorter periods increase the responsiveness of the system but also shorten the time window available to receive and process metadata packets. We study the impact of the period in the next experiment, shown in Figure 7. The topology is a dumbbell with the containers evenly split across two switches (clients on one side, servers on the other). Each client runs iPerf [41] without a throughput limit which will saturate the inter-switch link. This link is configured such that, for each topology, clients get a bandwidth share of $20Mbit/s$.

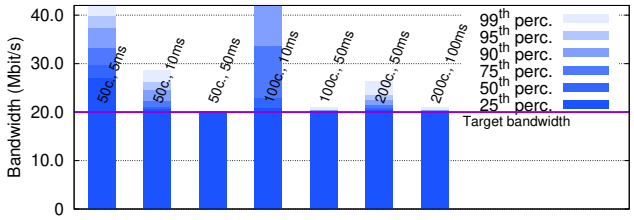

Fig. 7: Emulation accuracy for different update periods and total number containers. The topology allows at most 20 Mbit/s per flow.

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

Scalability of latency emulation. So far our experiments have focused on simple dumbbell topologies. In this section, experiments use scale-free topologies generated though the preferential attachment method described in [42]. This method yields scale-free networks, which are representative of the characteristics of Internet topologies. The experiment consists of end-nodes sending ICMP echo requests (`ping`) to other random end-nodes for 10 minutes. We compare the obtained round-trip times (`RTT`) with the theoretical ones statically computed from the topology. The results are presented in Table II as a mean squared error between these two quantities. We consider three topologies, respectively with 1000 elements (666 services, 334 bridges), 2000 elements (1344 services, 656 bridges) and 4000 elements (2668 services, 1332 bridges).

Results for NEED, Mininet and Maxinet were taken from NEED's evaluation [43]. KOLLAPS, NEED and Maxinet are deployed on four machines while Mininet is deployed in a single machine as a multiple machine deployment is not supported. We observe that Mininet produces smaller errors than both KOLLAPS and NEED. We attribute this to two factors. First, the container networking in Docker introduces small yet measurable delays. Second, because KOLLAPS and NEED are running on different physical machines, there is also a small but measurable delay when packets need to traverse the physical links. Despite these two factors, the largest deviation from the theoretical `RTT` observed with KOLLAPS was $0.55ms$, on the third topology. Due to the limitations with Mininet, it was not possible to gather results for the larger topologies.

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

V. CONCLUSION

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

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

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

and reduces this overhead through the use of shared memory to share this data.

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

REFERENCES

[1] D. Merkel, "Docker: lightweight Linux containers for consistent development and deployment," p. 2, 2014. [Online]. Available: http://dl.acm.org/ft_gateway.cfm?id=2600241&type=html%5Cnhttp://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment

[2] "Linux LXC," https://linuxcontainers.org/, 2019.

[3] "Docker Swarm," https://docs.docker.com/engine/swarm/, 2019.

[4] "Kubernetes," https://kubernetes.io/, 2019.

[5] R. F. Boisvert, "Incentivizing Reproducibility," *Commun. ACM*, vol. 59, no. 10, pp. 5–5, Sep. 2016. [Online]. Available: http://doi.acm.org/10.1145/2994031

[6] R. D. Peng, "Reproducible research in computational science," *Science*, vol. 334, no. 6060, pp. 1226–1227, 2011.

[7] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau, "Large-scale virtualization in the emulab network testbed," in *USENIX Annual Technical Conference*, 2008.

[8] R. Ricci, E. Eide, and C. Team, "Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications," *; login:: the magazine of USENIX & SAGE*, vol. 39, no. 6, pp. 36–38, 2014.

[9] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[10] W. Kim, A. Roopakalu, K. Y. Li, and V. S. Pai, "Understanding and Characterizing PlanetLab Resource Usage for Federated Network Testbeds," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC '11. New York, NY, USA: ACM, 2011, pp. 515–532. [Online]. Available: http://doi.acm.org/10.1145/2068816.2068864

[11] J. Banks, J. Carson, and B. Nelson, *Discrete-event System Simulation*. Prentice Hall, 2010. [Online]. Available: https://books.google.pt/books?id=cqSNnmrqqbQC

[12] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '07. New York, NY, USA: ACM, 2007, pp. 398–407. [Online]. Available: http://doi.acm.org/10.1145/1281100.1281103

[13] S. Floyd and E. Kohler, "Internet research needs better models," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 29–34, 2003.

[14] S. Floyd and V. Paxson, "Difficulties in simulating the internet," *IEEE/ACM Transactions on Networking (ToN)*, vol. 9, no. 4, pp. 392–403, 2001.

[15] V. Paxson and S. Floyd, "Why we don't know how to simulate the internet," in *In Proceedings of the 1997 Winter Simulation Conference*, 1997, pp. 1037–1044.

[16] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How amazon web services uses formal methods," *Commun. ACM*, vol. 58, no. 4, pp. 66–73, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2699417

[17] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 253–264. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413206

[18] P. Wette, M. Dräxler, and A. Schwabe, "MaxiNet: Distributed emulation of software-defined networks," *2014 IFIP Networking Conference, IFIP Networking 2014*, 2014.

[19] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "Crystalnet: Faithfully emulating large production networks," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 599–613.

[20] K. V. Vishwanath, D. Gupta, A. Vahdat, and K. Yocum, "ModelNet: Towards a datacenter emulation environment," in *IEEE P2P'09 - 9th International Conference on Peer-to-Peer Computing*, no. May, 2009, pp. 81–82.

[21] J. Neves, "Container network topology modelling," Master's thesis, Instituto Superior Técnico, UTL, Lisbon, Portugal, October 2018.

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

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

[24] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein, "Pantheon: the training ground for internet congestion-control research," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 731–743.

[25] M. a. Eriksen, "Trickle: A Userland Bandwidth Shaper for Unix-like Systems," *Usenix*, pp. 61–70, 2005.

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

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

[28] V. Schiavoni, E. Rivière, and P. Felber, "Splaynet: Distributed user-space topology emulation," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2013, pp. 62–81.

[29] L. Leonini, E. Rivière, and P. Felber, "Splay: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze)," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 185–198. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1558990

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

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

[32] "Linux Traffic Control," https://linux.die.net/man/8/tc, 2019.

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

[34] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop," *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets '10*, pp. 1–6, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1868447.1868466

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

[36] M. Peuster, J. Kampmeyer, and H. Karl, "ContainerNet 2.0: A Rapid Prototyping Platform for Hybrid Service Function Chains," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, 2018, pp. 335–337.

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

[38] "Docker Security Capabilities," https://docs.docker.com/engine/security/, 2019.

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

[40] L. Massoulié and J. Roberts, "Bandwidth sharing: objectives and algorithms," in *INFOCOM'99. Eighteenth Annual Joint Conference of*

the IEEE Computer and Communications Societies. Proceedings. IEEE, vol. 3. IEEE, 1999, pp. 1395–1403.
[41] "iPerf," https://github.com/esnet/iperf, 2019.
[42] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.
[43] J. Neves, "Need : Container-based decentralized topology emulation," 2018.