Scalable Network Emulation

Sebastião Amaro
Instituto Superior Técnico, Universidade de Lisboa

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

I. Introduction

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

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

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

The current state-of-the-art suffers from different limitations. Mininet [3] is limited to the use of a single host, and therefore cannot accurately emulate large-scale geo-replicated distributed systems. Maxinet [4] which expands upon Mininet, uses multiple Mininet instances to provide emulations with several hosts. However, it scales poorly due to its approach to workload management. Modelnet [5] introduces a separation between application nodes and nodes responsible for maintaining the emulation accuracy. But it still suffers from the same scalability limitations due to the centralization of the nodes responsible for managing the emulation. SplayNet [6] differs from these previous systems, introducing a decentralized network emulator. However, it is limited to the Splay [7] framework and the use of the Lua programming language. Another angle for networking evaluation is the control plane, which is responsible for routing packets. CrystalNet [8] accurately emulates it but cannot emulate the data-plane, and therefore it cannot emulate network properties such as bandwidth, latency. NEeaS [9] is an innovative cloud-based network emulation platform that aims at providing users with Network Emulation as a Service (NEaaS). However, it does not present results that confirm their platform works at large scale scenarios, and as a cloud-based platform, there are costs involved. Given all these limitations, we can conclude to the best of our knowledge that they are not suitable to systematically reproduce the evaluation of large scale distributed systems.

Kollaps [10] is a fully decentralized emulator. It emulates a network topology on containers, is agnostic to application language and transport protocol, and can scale to thousands of application nodes while maintaining an accuracy similar to the previously mentioned centralized solutions. And to our knowledge is the most suitable solution for network emulation.

However, it suffers from multiple limitations, for instance. (i) high CPU usage due to the communication mechanism between components. (ii) high CPU usage in large scale deployments due to the retrieval of information from the kernel. (iii) the model of sharing metadata causes an excessive amount of metadata to be shared between Kollaps components (iv) bare metal deployments are not supported, which could allow for use cases of applications that do not run in containers.

In this thesis, we propose Kollaps 2.0 as an improvement to Kollaps by addressing its main limitations. The ideas behind our solutions to solve the main ones were as follows. (i) to lower CPU usage, we will replace the existing mechanism of communication (Aeron [11]) with a new one. (ii) use eBPF [12] to provide a new way to retrieve information from the kernel. (iii) solve the problem of excessive amounts of metadata circulating in long-lived flows by moving to a new data dissemination model. (iv) introduce bare metal deployments by allowing the introduction of Kollaps to an already running system, and to integrate it with the already running network to do accurate emulation.

II. Related Work

In Section II-A we introduce some of the state-of-the-art network emulators and Kollaps in detail. Next, in Section II-B we summarize the state-of-the-art and explain Kollaps main limitations.

A. *State of the Art*

Mininet [3] is a prototyping tool for large networks, that run on a single machine with limited resources. Mininet accomplishes this by using lightweight OS-level virtualization techniques. These techniques consist of leveraging processes and virtual Ethernet pairs in separated network namespaces, allowing the user to launch networks with gigabit bandwidth and hundreds of nodes.

Maxinet [4] is a distributed emulator of software defined networks, based on Mininet, that makes emulation over several physical machines possible. Using this method, Maxinet can emulate networks with thousands of nodes on just a few physical machines.

ModelNet [5] is an Internet emulation environment that provides users a way to deploy unmodified software prototypes in configurable Internet-like environments. And to control the network conditions and subject them to faults.

Splaynet [6] is a user-space network emulation system, built on Splay [7] it allows users to deploy several topologies on shared physical nodes, with minimal setup complexity in a fully decentralized way.

NEeaS [9] is a cloud-based network emulation platform aiming at providing users with Network Emulation as a Service (NEaaS). NEeaS can deploy experiments on both public and private clouds. To emulate networks of larger scale and to reduce the hardware cost of the proposed platform, NEaaS uses light-weighted virtualization technology. Namely, it uses Docker containers to supplement virtual machines (VM) to emulate networking nodes in a hybrid manner.

1) Kollaps: Kollaps [10] is a fully distributed decentralized network emulator, capable of scaling to thousands of processes, while staying accurate when compared to the state-of-the-art centralized solutions, and bare-metal deployments.

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

Kollaps has six main components as seen in Figure 1:

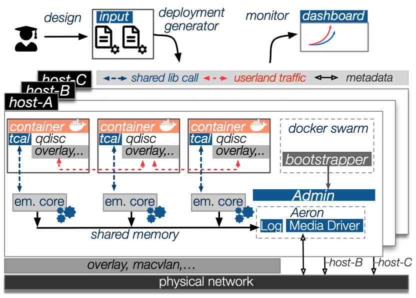

Fig. 1: Kollaps Architecture

The Deployment Generator consists of a Python program that receives as input an XML file containing the system topology. It then builds a graph of the given topology and outputs either a Docker Swarm file or a Kubernetes [13] manifest file, depending on the user's needs. This XML structure is specific to Kollaps and based on the one used by ModelNet [5]. Kollaps supports static topologies specifications and provides a rich set of dynamic events, such as changing the properties of links, removing and adding links, bridges, and services.

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

The Emulation Core (EC) is the main component of Kollaps. It runs on the Admin pid namespace and in the network

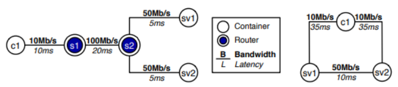

Fig. 2: Example of a topology collapsed into an equivalent one

namespace of the container. Kollaps does not directly emulate internal network devices or their state, instead, it collapses the topology as can be seen in Figure 2, on the left we can see the initial topology, and on the right, the collapsed one without internal network devices and with the links adapted to the network values. The process necessary to achieve this consists of first parsing the topology into a graph structure, then the EC calculates all the shortest paths between every two reachable containers. Each path contains multiple links, whose properties the EC uses to determine the end-to-end network properties. In the case of dynamic events, the EC precomputes the graphs offline. During the emulation, the EC changes the graph each time an event occurs.

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

TCP manages the competition for bandwidth in a link with its congestion control mechanism. There are many TCP congestion control mechanisms such as TCP Reno [14], TCP Vegas, [15] etc. These mechanisms are responsible for adjusting the throughput. To allow all of the competing flows to get a fair share of the bandwidth.

The RTT-Aware Min-Max model [16] receives a flow and returns the share of the link, which is inversely proportional to its round-trip time. The formula for calculating the fair share of a flow f is:

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

where

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

are the active flows on a link.

The ECs are responsible for calculating the network properties of their paths. They do this by maintaining a data structure with the bandwidth of each flow in the topology. The ECs leverage the RTT model to calculate the share for each flow. The EC obtains the bandwidth of the container and sends this metadata to the other ECs via the Aeron Media Driver. The Aeron Media Driver [11] is an open-source UDP and IPC message transport protocol. There is a single instance of Aeron in every physical machine. Shared memory assures communication between ECs on the same machine and UDP messages communication between ECs on different physical machines.

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

Fig. 3: Comparison of Network Emulators. P= Process, V = Virtual Machine, C = container, B = bare metal.

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

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

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

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

B. Discussion

Table 3 summarizes the comparison of the state of the art systems with Kollaps 2.0 and shows how it enhances it. Orchestration alludes to how the emulators structure themselves. Centralized emulators are more likely to have problems in terms of scalability. Application Agnostic refers to the type of application running in the system. An emulator is agnostic when any application can run in the emulator. An agnostic emulator provides significantly more use cases. Application Deployment mentions what type of use cases the emulators serve. The more different types of deployments, the more use cases they can cover. Efficiency is a qualitative measure that represents the resource usage of the network emulators. The centralized emulators, due to their nature, have low efficiency on the central components. SplayNet does not mention resource usage. NEeaS mentions that it still faces challenges against limited hardware resources. While Kollaps uses all of the available CPU.

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

Kollaps, to the best of our knowledge, is the only system with a decentralized architecture while using container technology, allowing Kollaps to be application-agnostic while maintaining emulation accuracy and scalability when compared to the state of the art. However, Kollaps has several limitations, such as high CPU usage, due to the constant polling of information done by Aeron, causing the CPU usage to always be close to 100%.

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

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

III. Approach

In this Section, we will explain in detail the main limitations of Kollaps [10] and discuss how we solved them in Kollaps 2.0. Kollaps provides a way of doing emulation in a fully decentralized way, it uses Docker containers to provide lightweight deployment of applications and `tc` to emulate network properties. Figure 4 depicts the Kollaps 2.0 architecture.

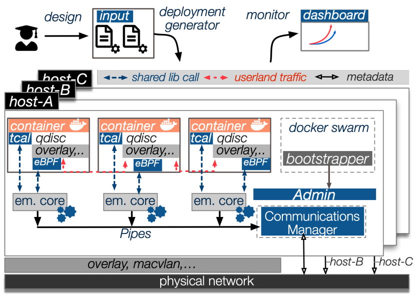

Fig. 4: Kollaps 2.0 architecture

A. Metadata Dissemination Mechanisms

In Kollaps, CPU usage is always close to 100%, because of the polling of metadata from the Aeron Media Driver [11]. Kollaps polls with a thread in an infinite loop in busy waiting until new messages appear. If no other processes are running, the scheduler will always pick up the Aeron thread causing high CPU usage. Therefore Aeron is unfit with Kollaps and will be removed. With the removal of Aeron, there was a need for new communication mechanisms, these mechanisms need to provide communication between Emulation Cores (ECs) in the same machine(inter node communication), and in different physical machines (intra node communication). To implement these mechanisms, we must understand the flow of metadata in Kollaps.

The current flow of metadata in Kollaps is complex. The flow starts in the TCAL reading from the Linux `tc` (see II-A1), to the EC then, to Aeron, then to all the other ECs.

Figure 5 shows a depiction of the flow of metadata in Kollaps.

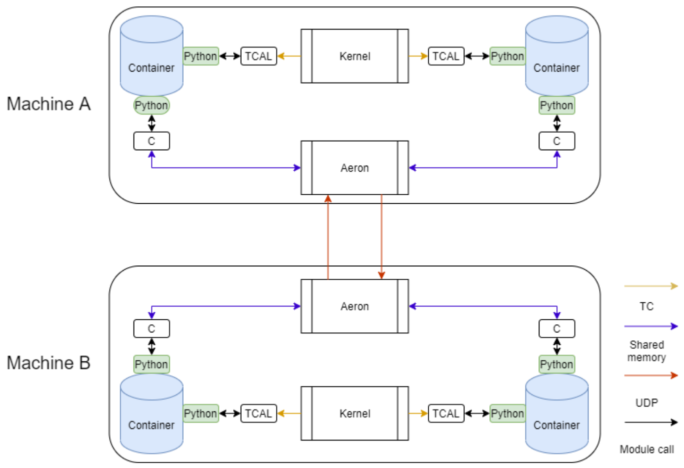

Fig. 5: The flow of metadata in Kollaps.

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

1) Communication Manager: Aeron is an independent C++ component, therefore we could either use Python to implement

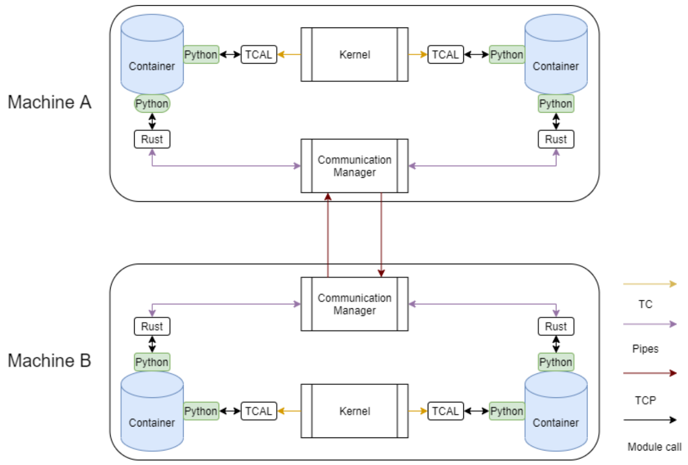

Fig. 6: The flow of metadata in Kollaps 2.0 .

the component that would replace Aeron, since most of Kollaps is written in Python, however, this is not optimal, since Python is considered a language not fitted for systems where performance is a major metric [17], or we could use another language more fit to Kollaps needs. Rust is a multi-paradigm programming language designed for performance and reliability [18]. Aeron worked as a component that receives metadata and shares this metadata with the system, so, we decided to implement the Communication Manager that would work similarly to Aeron but implemented in Rust.

The Admin will start the CM, and a single CM exists per physical machine. The CM will have two tasks. The first is for inter node communication, which means that it must share the metadata among ECs in its machine. The second one is for intra node communication, to achieve this, it must share the information of its ECs with other CMs, as well as the information it receives from other CMs to its ECs.

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

For inter node communication, the CMs use TCP sockets [19]. For each other CM in the deployment, there is a socket connecting them. Figure 6 shows a depiction of the flow of metadata in Kollaps 2.0.

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

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

Meanwhile, in the main thread, the CM starts the loop with two steps. First, the CM does select call with all of the pipes open in read mode. Then the main will block until a single pipe contains a message to be read. After unblocking, we go to the second step, where we iterate over the pipes that contain metadata. We read the metadata from the pipes and write to all of the pipes, previously opened in write mode. The CM uses the select call instead of an iterator, due to the blocking nature of the read call of the pipes we created [20]. If we used an iterator, the system would block if a single container stopped writing metadata.

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

To do this, we used PyO3 [21] which enables the user to generate a native Python module from a Rust library. So we created the Communication Core(CC) module. The EC will use this module as a Linux shared library.

The CC module provides a way for the EC to write metadata to the CM and to read from it. To achieve this, when the CC starts it creates pipes and opens one in write mode and one in read mode. Then it starts a thread with the task of reading from the pipe the CM writes to. When the thread reads metadata from the pipe, it hands it over to the EC. The module also provides ways for the EC to write its metadata to the pipe, allowing it to share metadata with the CM. Figure 7 shows a depiction of this part of the flow of metadata in Kollaps 2.0.

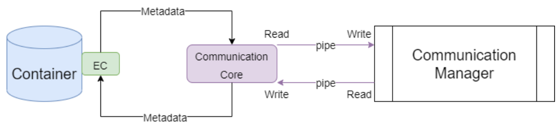

Fig. 7: The flow of metadata in a container.

B. Kernel Information Retrieval

With the removal of Aeron, we significantly reduced the CPU usage, as we later show in Section IV-A. However, when it came to large-scale topologies, CPU usage was still at 100%. Using pyspy, [22] we were able to see that the TCAL update_usage function was causing the high CPU usage.

From the perspective of an EC, the function does the following for every other EC in the deployment. First, it does rtnl_dump_request [23] to know how many bytes it sent to the other EC. Second, it uses rtnl_dump_filter [23] passing a callback as an argument. The kernel will then call the callback with the number of bytes associated. After receiving the information, the TCAL calls the EC with the number of bytes sent.

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

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

Extended Berkeley Packet Filter(eBPF) [12] is a highly flexible and efficient construct in the Linux kernel. That allows the execution of bytecode at various hook points in a safe manner.

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

Linux 2.2 for cBPF and 3.19 for `eBPF`, added functions to provide socket filtering [24]. They provide a way to attach BPF or `eBPF` programs respectively to filter data on a chosen socket. As in `tc`, we handle a `sk_buff` structure and can access its content to make traffic decisions and monitor traffic.

We will attach a filter to sockets. However, we do not want to attach a filter to every socket the container creates. RedBPF [25] allows attaching to a raw_socket [26] allowing us to filter all the traffic of the container.

We decided to use perf events, which allows us to notify userspace of events in kernel space. Specifically, we use a RedBPF PerfMap [27], this provides us a way to get the hashmap values from kernelspace, in userspace.

The solution, Figure 8 shows a depiction of the solution we designed. And works as follows, the `eBPF` program still maintains the same map, however, it sends its updates via the PerfMap to userspace. We do not want to generate an event each time the container sends a packet since this causes CPU usage to be high, and the EC only needs new values every pool_period (see Section II-A1. So we define that we only send events every perf_event period. Perf_event has a default value of 0.025 seconds, half of the pool_period value.

In userspace, we have to read these events, therefore at startup, the Communication Core will start a new thread. This thread is responsible for reading the events the `eBPF` program will generate. To save the values in userspace, we create a map similar to the one in the kernel. The map updates each time the `eBPF` program sends an event.

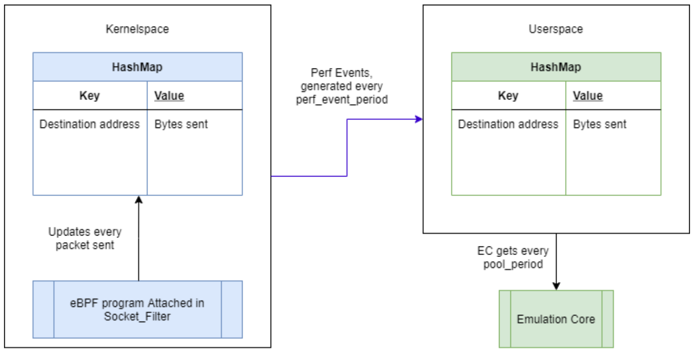
Kernelspace
HashMap
Key
Value
Destination address
Bytes sent
Updates every packet sent
eBPF program Attached in Socket_Filter
Perf Events, generated every perf_event_period
Userspace
HashMap
Key
Value
Destination address
Bytes sent
EC gets every pool_period
Emulation Core

Fig. 8: Mechanism to retrieve information from the Kernel .

C. Metadata Dissemination Model

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

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

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

To explain these problems, first, we must revisit the emulation loop explained in Section II-A1. Firstly in step (iv), after computation, the EC would delete the metadata related to flows of other ECs. Since we are reactive, the ECs must maintain the metadata, and assume that if the stored flows did not change, then nothing changed.

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

Thirdly, as explained before, we maintain metadata sent by other ECs. The metadata in Kollaps specifies a bandwidth value and the links in the graph it transversed. Integer values represent links. The integer values of the links are specific to a specific state of the graph. However, as explained in Section II-A1 the graph changes every time a dynamic event happens. Therefore, when a graph change happens, its configuration changes and an integer value that the EC assigned to a specific link between two nodes might not represent the same link anymore. Therefore, every time a dynamic event happens, the metadata stored becomes obsolete. Which means we must delete it.

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

D. Baremetal Deployments

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

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

To run bare metal deployments we assume that each machine has the Kollaps code in a specific directory. To interact with the machines, we use ssh, specifically a Python ssh library [28]. We also introduced two new commands and changed the start and stop commands. The install command that has the task of installing the Kollaps components on the machine, the initialize command initializes the emulation and data dissemination mechanisms. The start command will now start a script given by the user. The stop command stops the emulation and data dissemination mechanisms, after, it runs a script to stop the experiment if the user provides one.

IV. Evaluation

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

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

The first benchmark presented is the comparisons in CPU usage in differents scales of deployments, presented in Section IV-A. Then we present a comparison between metadata shared in Kollaps and Kollaps 2.0 in Section IV-B. In the next three sections, we compare Kollaps and Kollaps 2.0 in metrics related to the accuracy of the emulation. In Section IV-C we present bandwidth emulation accuracy, in Section IV-D emulation reaction time, and finally in Section IV-E scalability of latency emulation accuracy.

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

A. CPU Usage

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

TABLE I: Comparison of CPU usage per physical machine in smallscale deployments running iperf3.

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

TABLE II: Comparison of CPU usage per physical machine in largescale deployments running ping.

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

To test for the improvements, we set up various deployments with n-clients/n-servers dumbbell topology. The clients execute an iperf3 client, and the servers execute an iperf3 server. The CPU usage was retrieved using dstat [29], we separate results into user (`usr`)the percentage of time spent running user processes, and system (`sys`) percentage of time spent running system processes. Table I presents the results.

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

However, as explained in Section III-B the mechanism to retrieve the information from the kernel, did not fit Kollaps since it scaled quadratically. To measure the CPU usage of large scale topologies, we did different experiments using scale-free network topologies generated with [30]. This method constructs scale-free networks, which are representative of the characteristics of Internet topologies. The experiment consists of end-nodes running ping [31] to another end-node for 10 minutes. We did this tests with 500 nodes(332 services, 168 bridges),1000 nodes(666 services, 334 bridges), 2000 nodes (1344 services,656 bridges) and 4000 nodes(2668 services, 1332 bridges). During this test, we again run dstat.

Table II presents the results for Kollaps 2.0, Kollaps without Aeron, and Kollaps. Kollaps originally had the Aeron problem which, caused a lot of `usr` CPU usage. However, after being removed, revealed the problem with the mechanism to retrieve information from the kernel, which caused the high `sys` CPU usage. After being substituted by the mechanism combining both `eBPF` and the `socket` subsystem. We can see that the usage is negligible. Although we can expect significant CPU usage from these experiments, we must not forget pings

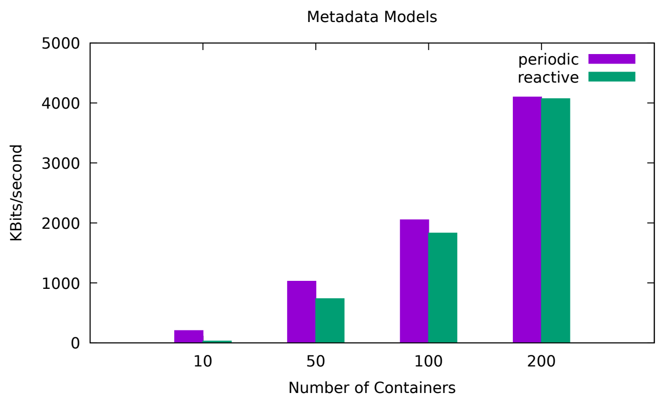

Fig. 9: Comparison between metadata generated in a periodic and reactive model.

do not cause significant bandwidth values. Therefore, Python calculations are not done, and this is the reason why we don't have significant values in Kollaps 2.0.

B. Metadata Generation

Kollaps RTT bandwidth model, as explained in Section II-A1 relies on having all the bandwidth values of competing flows. To meet this requirement, Kollaps relies on metadata sharing. However, the model used to share the metadata did not fit Kollaps, since metadata was always shared across every other EC in the system, even if nothing changed.

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

As explained in Section III-C in Kollaps, we have a periodic metadata sharing model. Thus every time the EC has new information, it will share it with others. With the new model, the EC only shares metadata if it changes by at least 5%. Figure 9 presents the results of these changes. To calculate the amount of metadata sent, we multiply the number of messages each EC sent by the size of the default Kollaps message which, is 256 bytes.

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

C. Bandwidth Emulation Accuracy

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

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

Fig. 10: Study of the accuracy bandwidth shapping accuracy for different link values on a client-server topology.

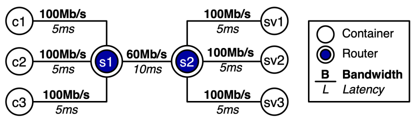

Fig. 11: Dumbbell topology with 3 clients, 3 servers.

To benchmark this accuracy, we run a test composed of two services connected by a single link. One service will execute an iperf3 [32] server in one machine, and another service, an iperf3 client in a different physical machine. We access bandwidth at different values, from low to high. The test consists of having the client execute iperf3 over a period of 60 seconds.

Observing the results presented in Table 10, we can see the bandwidth values of Kollaps 2.0, in both containerized and bare metal deployments, are always in a 5% error across all ranges. Showing that Kollaps 2.0 maintains the precise accuracy reported in Kollaps, we expect this because the information we retrieve using `eBPF` and with the `rtnetlink` library is the same.

D. Emulation Reaction Time

Given the changes previously mentioned in Section III, mainly, the change in the Communication Mechanisms and the change in the information retrieval from the kernel, we must test the reaction time of Kollaps 2.0. To assure that the new mechanisms are as fast at reporting correct values as the ones in the original Kollaps. To this goal, we set up a simple 3-clients/3-servers dumbbell topology, depicted in Figure 11. The link connecting both switches has a maximum bandwidth of 60Mbps. All the clients have the same bandwidth, and latency so they should get a similar share of the link.

The experiment, as seen in Figure 12 proceeds as follows. At the start, client1 be the only one with an active flow. So it will use all of the available bandwidth. After 10 seconds, client2 starts, and both the nodes will compete for a share of the link. Since they have the same latency and therefore RTT, the model as explained in Section II-A1, will award them with the same bandwidth. At second 20, client3 enters the experiment, and following the same logic, the three clients will have similar bandwidths. In both cases, we can see that

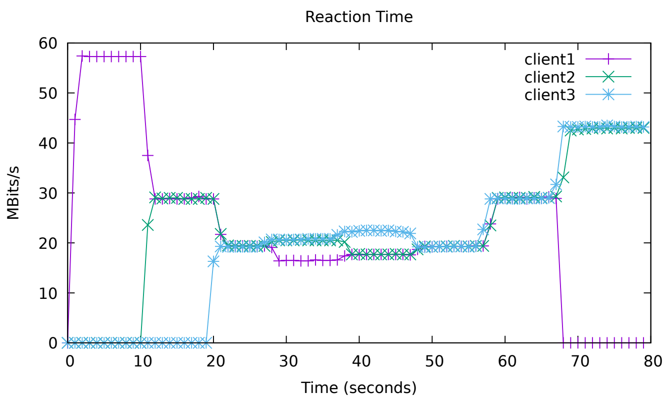

Fig. 12: Reaction time in a container deployment.

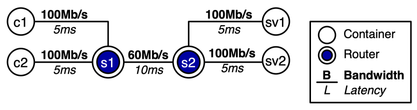

Fig. 13: Dumbbell topology with 2 clients, 2 servers.

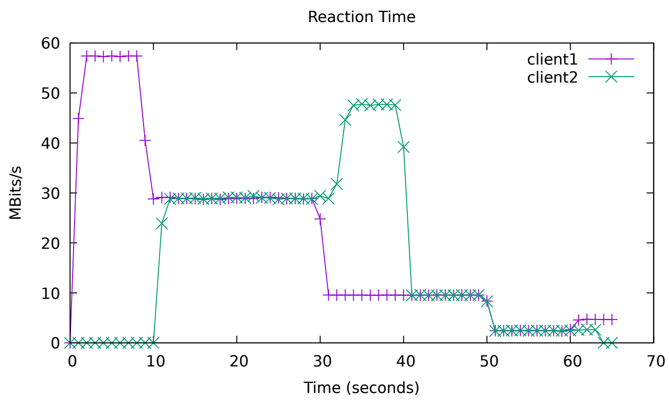

Fig. 14: Reaction time in a baremetal deployment.

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

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

Kollaps 2.0 took around a second to adjust the bandwidth of the clients. At second 30, the link between client1 and the s1 switch will change in latency from 5 to 10 since its RTT went up. It will have a smaller share of bandwidth. Causing his bandwidth to drop, since there is now available bandwidth to be used, both client2 and client3 will start using that bandwidth. At 40 seconds, the link between client2 and the s1 switch will have the same increase in latency. Therefore the bandwidth for client2 will be lower, causing the clients to pick up the available bandwidth. Finally, at 50 seconds, the same happens to client3, and the three clients will now have the same bandwidth since the links are identical. At second 60, the link connecting the switches s1 and s2 increases in bandwidth from 60 to 90 Mbs causing the available bandwidth to increase. And therefore, the bandwidth of the three clients will increase similarly. In second 70, client1 crashes leaving his share of the bandwidth available which, will be picked up by both client1 and client2 in a similar manner.

Given these results, we can conclude that we retained the ability of Kollaps to react under a second to changes in the topology in Kollaps 2.0. While also validating that the kernel information we are retrieving is correct and updated at a fast pace.

In Kollaps 2.0, we introduced bare metal deployments. In the previous section, we showed how we retain the link-level emulation capabilities of Kollaps. Now we must take a look at the validation of the RTT model, but also check if we maintained the reaction time accuracy. For this, we created a simple 2-clients/2-servers dumbbell topology, depicted in Figure 13. The clients have the same bandwidth values, and the link connecting both switches has a 60 Mbps bandwidth.

This experiment, as depicted in Figure 14 and starts with client1, using all of the available bandwidth. At second 10, client 2 joins, and since they have similar links, they will obtain similar shares of bandwidth. At second 30, the link between client1 and s1 changes from 100Mbps to 10Mbps. This constraint in the bandwidth causes client1 bandwidth usage to drop to 10Mbps. A second 40 a similar thing happens to client2. At second 50, the link between the switches changes from 60 Mbps to 5 Mbps causing the available bandwidth of the clients to only be 2.5Mbps for each. Finally, at second 60, the link between client2 and s1 disappears, causing the 5Mbps of available bandwidth to be allocated only to client1.

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

E. Scalability of Latency Emulation Accuracy

With the changes to the mechanism used to retrieve information from the kernel, there was a massive drop in CPU usage in large scale deployments as mentioned in Section IV-A. We now assess whether if Kollaps 2.0 can still scale while maintaining accurate latency values. To this avail, we did different experiments using scale-free network topologies generated using [30]. The experiment consists of end-nodes running ping [31] to another end-node for 10 minutes. We did these tests with the same large scale topologies as mentioned in Section IV-A.

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

V. Conclusion

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

In this document, we analyze the current state-of-the-art and leverage it against our network emulator Kollaps [10]. Kollaps is a fully decentralized distributed emulator agnostic to application and transport protocols, that can scale to thousands of nodes while maintaining an accuracy similar to the current centralized solutions and bare-metal deployments. Nevertheless, like all systems, Kollaps has its limitations. Due to its decentralized nature, Kollaps needs to share information, the current communication framework and communication model have many problems. On the engineering level, Aeron [11] causes the CPU usage to be at 100%. On an algorithmic level, the current model causes Kollaps to have a high amount of metadata in the system in long-lived flows. Finally, Kollaps does not support bare metal deployments, therefore, leaving a lot of use cases unexplored.

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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