Distrinet: a Mininet Implementation for the Cloud

Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, Chidung Lac

Distrinet: a Mininet Implementation for the Cloud

Giuseppe di Lena*†, Andrea Tomassilli*, Damien Saucez, Frédéric Giroire*, Thierry Turletti*, and Chidung Lac†
*Université Côte d'Azur, Inria, CNRS, France
†Orange Labs, France

***Abstract*—Networks have become complex systems that combine various concepts, techniques, and technologies. As a consequence, modelling or simulating them now is extremely complicated and researchers massively resort to prototyping techniques. Mininet is the most popular tool when it comes to evaluate SDN propositions. Mininet allows to emulate SDN networks on a single computer but shows its limitations with resource intensive experiments as the emulating host may become overloaded. To tackle this issue, we propose *Distrinet*, a distributed implementation of Mininet over multiple hosts, based on LXD/LXC, Ansible, and VXLAN tunnels. Distrinet uses the same API than Mininet, meaning that it is compatible with Mininet programs. It is generic and can deploy experiments on Linux clusters (e.g., Grid'5000), as well as on the Amazon EC2 cloud platform.**

I. Introduction

Modern networks became so complex and implementation-dependent that it is now impossible to solely rely on models or simulations to study them. On one hand, models are particularly interesting to determine the limits of a system, potentially at very large scale or to reason in an abstract way to conceive efficient networks. On the other hand, simulations are pretty handy to study the general behavior of a network or to get high confidence about the applicability of new concepts. However, these methods do not faithfully account for implementation details. To this end, emulation is more and more used to evaluate new networking ideas. The advantage of emulation is that the exact same code as the production one can be used and tested in rather realistic cases helping to understand fine-grained interactions between software and hardware. However, emulation is not the reality and it often needs to deal with scalability issues for large and resource-intensive experiments.

When it comes to Software Defined Networking (SDN), Mininet [1] is by far the most popular emulator. The success of Mininet comes from its ability to emulate potentially large networks on one machine, thanks to lightweight virtualization techniques and a simple yet powerful API. Mininet was designed to run on one single machine, which can be a limiting factor for experiments with heavy memory or processing capacity needs. A solution to tackle this issue is to distribute the emulation over multiple machines. However, as Mininet was designed to run on a single machine, it assumes that all resources are shared and directly accessible from each component of an experiment. Unfortunately, when multiple machines are used to run an experiment, this assumption does not hold anymore and the way Mininet is implemented has to be revised.

In this paper, we present Distrinet [2], an extension of Mininet implemented to allow distributed Mininet experiments to leverage resources of multiple machines when needed.

Challenge. Mininet is used by a large community ranging from students to researchers and network professionals. This success of Mininet comes from the simplicity of the tool: it can work directly on a laptop and its installation is trivial. The challenge is to extend Mininet in such a way that these conditions still hold, while being distributed over multiple machines. Distrinet allows applications to run in isolated environments by using LXC to emulate virtual nodes and switches, avoiding the burden of virtual machine hypervisors. Distrinet also creates virtual links with bandwidth limits without any effort from the user.

Contributions. Mininet programs can be reused with minimal or even without any changes in Distrinet, but with a higher degree of confidence on the results in case of resource intensive experiments. Our main contributions to reach this objective can be summarized as follows.

- **Compatibility with Mininet.** Mininet experiments are compatible with Distrinet, either using the Mininet API or with the Mininet Command Line Interface (i.e., `mn`).
- **Architecture.** Distrinet is compatible with a large variety of infrastructures: it can be installed on a single computer, a Linux cluster, or the Amazon EC2 cloud. Distrinet relies on prominent open source projects (e.g., Ansible and LXD) to set up the physical environment, manage experiments, and guarantee isolation.
- **Comparison with other tools and link bandwidth.** Comparisons with Mininet Cluster Edition [3] and Maxinet [4] show that our tool *handles more efficiently link bandwidth limitation* which is a *fundamental basic brick* of network emulation.
- **Flexibility**. Thanks to the usage of LXC, Distrinet allows to run VNFs or generic containers on the emulated topology composed of virtual switches and hosts. Each virtual node is properly isolated and the virtual links can be capped natively.

In this paper, we first discuss the advantages and limitations of existing emulation tools in Sec. II. We present the architecture of Distrinet in Sec. III and how it is integrated in an environment in Sec. III-B. We then evaluate Distrinet by comparing the emulation results using Mininet, Maxinet, Mininet Cluster Edition and Distrinet in Sec. IV. Last, we discuss the current and future work on Distrinet in Sec. V and conclude in Sec. VI.

	Distrinet	Mininet CE	Maxinet
Mininet compatibility			
Runnable with mn command	✔	✔	✘
Mininet API	✔	✔	●
Tunneling technologies			
VXLAN Tunnels	✔	✘	✘
GRE Tunnels	●	✔	✔
Emulation features			
Unlimited vLink	✔	✔	✔
Limited vLink (No tunneling)	✔	●	●
Limited vLink (Tunneling)	✔	✘	●
vNode isolation	✔	●	●
Automatic cloud provision	✔	✘	✘

✔=Yes, ●=Partial, ✘=No

Table I: Supported features in the different tools for distributed emulation.

II. Related Work

Emulation allows to test the performances of real applications over a virtual network. A first frequently used tool to emulate networks is the Open vSwitch (OVS) software switch [5]. To build a virtual network, virtual switches (vSwitches) can be connected with virtual interfaces, through GRE or VXLAN tunnels. To emulate virtual hosts (vHosts), one can use containerization tools (e.g., LXC [6] or Docker [7]) or full virtualization tools (e.g., Virtual Box [8]).

Graphical Network Simulator-3 (GNS3) [9] is a software emulating routers and switches in order to create virtual networks with a GUI. It can be used to emulate Cisco routers and supports a variety of virtualization tools such as QEMU, KVM, and Virtual Box to emulate the vHosts.

Mininet [1] is the most common Software Defined Networking (SDN) emulator. It allows to emulate an SDN network composed of hundreds of vHosts and vSwitches on a single host. Mininet is easy to use and its installation is trivial. As we show in Sec. IV, it is possible to create a network with dozens of vSwitches and vHosts in just a few seconds. Mininet is very efficient to emulate network topologies as long as the resources required for the experiments do not exceed the ones that a single machine can offer. If physical resources are exceeded, the results might be not aligned with the ones of a real scenario.

The tools closest to ours are Maxinet [4] and Mininet Cluster Edition (Mininet CE [3]). They allow to distribute Mininet on a cluster of nodes. Maxinet creates different Mininet instances in each physical node of the cluster, and connects the vSwitches between different physical hosts with GRE tunnels. Mininet CE extends directly Mininet in order to distribute the vNodes and the vLinks in a set of machines via GRE or SSH tunnels. Containernet [10] allows to extend Mininet to support Docker containers. By default, it is not able to distribute the emulation on different nodes, but it is possible to combine it with Maxinet or Mininet CE to support such an option and provide better vNodes isolation. While the Maxinet approach makes it possible to increase the scalability of Mininet and offers a speed-up in terms of virtual network creation time for certain topologies, its main drawback is that it is not directly compatible with Mininet. Moreover, even though it is straightforward to setup networks with unlimited vLinks (i.e., vLinks without explicit bandwidth limit or delay), Maxinet does not fully support limited vLinks (i.e., vLinks with explicit bandwidth limits or delay). The Mininet CE approach offers a full compatibility with Mininet, but like Maxinet, it has some limitations when it come to emulate vLinks with limited bandwidth or delay. It is not possible to add limitations on the vLink if it is connected between 2 vNodes in different physical machines [11]. We believe that automatic cloud provision offered by Distrinet, its flexibility, and its compatibility with Mininet give our tool an important added value as Mininet is by far the most used tool to emulate SDN networks. Table I summarises the main differences between the tools.

III. Architecture

Four key elements have to be considered in order to distribute Mininet experiments over multiple hosts. First, emulated nodes must be *isolated* to ensure the correctness of the experiments even when the hosts supporting the experiments are heterogeneous. To obtain these guarantees, virtualization techniques (full or container-based) have to be employed. Similarly, *traffic encapsulation* is needed such that the network of the experiment can run on any type of infrastructure.

To start and manage experiments, an *experimentation control plane* is necessary; this control plane allows to manage all the emulated nodes and links of the experiment, regardless of where they are physically hosted.

Finally, if the deployment of an experiment in Mininet is sequential and generally does not severely affect the overall experimental time, when the experiment is distributed, *parallelization* is required as the deployment of nodes can be slow because data may have to be moved over the network.

A. Multi-host Mininet implementation

In Mininet, network nodes are emulated as user-level processes isolated from each other by means of light virtualization. More precisely, a network node in Mininet is a shell subprocess spawned in a pseudo-tty and isolated from the rest by the means of Linux cgroups and network namespaces. Interactions between Mininet and the emulated nodes are then performed by writing bash commands to the standard input of the subprocess and reading the content at the standard output and error of that process. As Mininet runs on a single machine, every emulated node benefits from the same software and hardware environments (i.e., the one from the experimental hosts). This approach has proven to be adequate for single-machine experiments but cannot be directly applied when experiments are distributed, as it would push too much burden in preparing the different hosts involved in the experiments. As a consequence, we kept the principle of running a shell process but instead of isolating it using cgroups and network namespaces, we isolated it within an LXC container [6]. Ultimately, LXC realizes isolation in the same way than using kernel cgroups and namespaces, but it provides an effective

tool suite to set up any desired software environment within the container just by providing the desired image when launching the container. In this way, even when the machines used to run an experiment are set up differently, as long as they have LXC installed on them, it is possible to create identical software environments for all the network nodes, regardless of the machine that actually host them. In Distrinet, to start a network node, we first launch an LXC container and create a shell subprocess in that container.

As Mininet runs on a single machine, the experiment orchestrator and the actual emulated nodes run on the same machine, which allows to directly read and write on the file descriptors of the bash process of the network nodes to control them. In Distrinet, we allow to separate the node where the experiment orchestration is performed from the hosts where the network nodes are hosted, meaning that directly creating a process and interacting with its standard I/Os is not as straightforward as in Mininet. Indeed, Mininet uses the standard `Popen` Python class to create the bash process at the basis of network nodes. Unfortunately, `Popen` is a low-level call in Python that is limited to launching processes on the local machine. In our case, we then have to rely on another mechanism. As we are dealing with remote machines and want to minimize the required software on the hosts involved in experiments, we use SSH as a means to interact between the orchestrator and the different hosts and network nodes. SSH is used to launch containers and once the container has been launched, we directly connect through SSH to the containers and create shell processes via SSH calls. In parallel, we open pseudo-terminals (PTYs) locally on the experiment orchestrator, one per network node, and attach the standard input and outputs of the created remote processes to the local PTYs. As a result, the orchestrator can interact with the virtual nodes in the very same way as Mininet does by reading and writing in the file descriptors of the network nodes' PTY. This solution may look cumbersome and suboptimal but it maximizes the Mininet code reuse, and ultimately guarantees compatibility with Mininet. Indeed, Mininet heavily relies on the possibility to read and write via file descriptors, the standard input and outputs of the shell processes emulating the virtual nodes, and massively uses `select` and `poll` that are low-level Linux calls for local files and processes. Therefore, providing the ability to have local file descriptors for remote process standard input and outputs allowed us to directly use Mininet code as the only change needed was in the creation of the shell process (i.e., using an SSH process creation instead of `Popen`), with no impact on the rest of the Mininet implementation. Solutions that would not offer low-level Linux calls compatibility to interact with the remote shell would cause to re-implement most of the `Node` classes of Mininet.

In Mininet, network nodes and links are created sequentially. The sequential approach is not an issue in Mininet where interactions are virtually instantaneous. However, a sequential approach is not appropriate in Distrinet since nodes are deployed from LXC images and because every interaction with a node is subject to network delays. For this reason, in Distrinet the node deployment and setup calls are made concurrent with the Asynchronous I/O library of Python 3. However, as the compatibility with Mininet is a fundamental design choice, by default all calls are kept sequential and we added an optional flag parameter to specify the execution to run in concurrent mode. When the flag is set, the method launches the commands it is supposed to run and returns without waiting for them to terminate. The programmer then has to check if the command is actually finished when needed. To help in this, we have added a companion method to each method that has been adapted to be potentially non blocking. The role of the companion method is to block until the command calls made by the former are finished. This allows one to start a batch of long lasting commands (e.g., `startShell`) at once, then wait for all of them to finish. We have chosen to use this approach instead of relying to callback functions or multi-thread operations in order to keep the structure of the Mininet core implementation.

To implement network links, Mininet uses virtual Ethernet interfaces and the traffic is contained within the virtual links thanks to network namespaces. When experiments are distributed, links may have to connect nodes located on different hosts, hence an additional mechanism is required. In Distrinet, we implement virtual links by using VXLAN tunnels (a prototype version with GRE tunnels also exists). The choice of VXLAN is guided by the need of transporting L2 traffic over the virtual links. In particular, we cannot rely on the default connection option provided directly with LXD. Indeed, the latter uses either multicast VXLAN tunnels or Fan networking [12] to interconnect containers hosted on different machines. However, cloud platforms such as Amazon EC2 do not allow the usage of multicast addresses and in some scenarios, a single physical machine may have to host hundreds of containers. Fan networking maps the addresses of a small network address space (i.e., /16 network) with a larger one (i.e., /8 network) and uses unicast tunnels to interconnect the different machines, but it does not allow to choose the IP addresses of the containers arbitrarily (the user cannot choose the IP of the virtual interfaces in the emulated network). In Distrinet, each link is implemented with a unicast VXLAN tunnel having its own virtual identifier. Also, since we are compatible with Mininet, to limit the capacity of the links, we simply use the Mininet implementation that relies on Linux Traffic Control (`tc`). SSH is used to send commands and retrieve data from the network nodes in the experiments, and each virtual node is reachable with an IP address. To do so, a bridge, called *admin bridge*, is setup on every machine that hosts emulated nodes. An interface, called *admin interface*, is also created on each node and bridged to the admin bridge and is assigned a unique IP address picked up from the same subnet. All these admin bridges are connected to the admin bridge of the master node. The machine running the script is then hooked with an SSH tunnel to the master host and can then directly access any machine connected to the admin bridge. The general architecture of Distrinet is presented in Fig. 1.

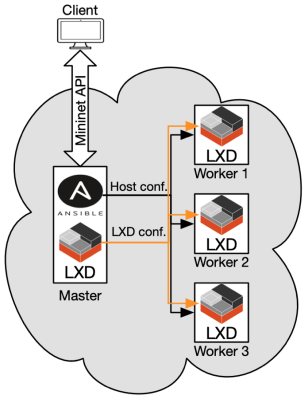

Figure 1: Distrinet general architecture.

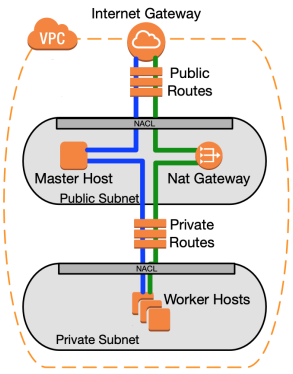

Figure 2: Amazon VPC configuration.

B. Infrastructure provisioning

Distrinet provides an infrastructure provisioning mechanism that uses Ansible to automatically install and configure LXD and SSH on each machine to be used during the experiment.

If the experimental infrastructure is Amazon EC2, Distrinet first instantiates a Virtual Private Cloud (VPC) configured as depicted in Fig. 2 in which the virtual instances running the experiment will be deployed. A NAT gateway is automatically created to provide Internet access to the Worker host. Access to the Worker nodes from the experimenter machine is ensured by a Master node acting as an SSH relay. The deployment on Amazon EC2 only requires an active Amazon AWS account.

The Distrinet environment (cloud or physical) includes the three following entities as shown in Fig. 1:

- *Client*: host in which the Distrinet script is running and decides where to place the vNodes around the physical infrastructure (round-robin by default). The Client must be able to connect via SSH to the Master host.
- *Master*: host that acts as a relay to interconnect the Client with all the Worker hosts. It communicates with the Client and the different Workers via SSH. Note that the Master can also be configured as a Worker.
- *Worker(s)*: host(s) where all the vNodes (vSwitches and vHosts) are running. vNodes are managed by the Master and the Client, via the admin network.

Distrinet can then automatically install the remaining requirements. In particular, it installs and configures LXD/LXC and OpenVSwitch in the Master and Worker hosts. After that, Distrinet downloads two images: an Ubuntu:18.04 image to emulate the vHosts, and a modified version of that image with OVS installed in order to save time during the configuration process. A default configuration setup is provided, but the user – by following the tutorial we provide [13] – can easily create a personalized image and distribute it in the environment using Ansible from the Master Node. After the configuration step, the user can start the emulation from the Distrinet Client.

IV. Evaluation

To evaluate Distrinet, we considered 3 types of experiments. The first one measures the overhead in term of execution time that the tools introduce to allow the distribution of the emulations. The second experiment compares the network capabilities of Mininet CE, Maxinet, and Distrinet. The last experiment shows the behaviors of a resource intensive emulation inside a single physical host and in a distributed environment. For the evaluation, we used Amazon Web Service (AWS) [14] and Grid'5000 [15]. In Grid'5000, we used the *Gros* cluster where the hosts are all equipped with one Intel Xeon Gold 5220 (18 multi-threaded cores per CPU at 2.20 GHz, i.e., 36 vCores) and 96 GB of RAM.

A. Distrinet core performance assessment

We first measure the overhead of distributing an experiment by timing the most fundamental operations of the tool in Grid'5000 and compare it with the other tools. The results are reported in Table II. Mininet is running on a single host while the others are running on two hosts. We observe that the creation of a vNode or of a vLink is much longer with Distrinet because it uses LXC containers that are slower to start, but are more isolated. This implies that setting up an experiment with Distrinet is slower. As examples, we provide the time to create different classic topologies with both tools. A Fat Tree 4 is built in 73.72 s with Distrinet to be compared to around 2.6 s with Mininet. Mininet Cluster Edition requires 59.68 s while Maxinet requires around 8.4 s. This is due to the fact that the *Link creation* implementation in Distrinet and Mininet Cluster Edition are similar, while in Maxinet it is completely different. Maxinet first sets up the virtual sub-networks in the different physical machines without tunneling; after this step, it creates the tunnels between the vNodes placed in different hosts. For this reason, in Maxinet, there are different lines for *link creation* and *tunnel creation*, while in Mininet CE and Distrinet, the *tunnel creation* time is included in the *link creation*. In Mininet, as explained before, vNodes are light but are not completely isolated. This can be problematic for some types of experiments. With our LXD/LXC approach, the vNode creation is slower, but the container provides a better isolation and the vNodes can be distributed.

The difference in setup time is significant with Maxinet, but for networks with a higher density of links (e.g., FatTree $k = 4$), the gap between Distrinet and Mininet CE is reduced. However, we believe that, for a large subset of experiments (e.g., running a 24 hour trace), the setup time represents a negligible part of the total experiment time. In any case, computation intensive experiments – such as the one presented in Sec. IV-C – cannot be performed with a single physical machine. We thus believe that Distrinet is a useful tool in many different use cases.

B. Tools comparison

In this section, we compare Distrinet, Mininet, Mininet CE, and Maxinet network capabilities. The experiment compares the maximum network throughput measured with iperf (TCP traffic, for one minute) between the first and the last hosts for different linear topologies. For the star topology, there is an iperf connection between 5 pairs of hosts. Table III

Action	Mininet	Distrinet	Mininet CE	Maxinet
Link Creation	7.5 ± 0.19 ms	495 ± 18 ms	430 ± 840 ms	13 ± 0.3 ms
Link Deletion	48.4 ± 8.3 ms	495 ± 18 ms	40.4 ± 7.3 ms	—
Node Creation	13.2 ± 0.1 ms	2.09 ± 0.04 s	329.9 ± 161.4 ms	108.4 ± 5.4 ms
Running a Command	3.45 ± 0.1 ms	4.8 ± 0.2 ms	3.35 ± 0.07 ms	1.19 ± 0.1 ms
Tunnel Creation	—	—	—	184.8 ± 1.0ms
Topology Creation				
Linear ($n = 2$)	0.368 ± 0.01 s	13.44 ± 0.25 s	4.09 ± 0.05 s	1.60 ± 0.06 s
Linear ($n = 10$)	1.412 ± 0.03 s	37.5 ± 0.55 s	9.7 ± 0.08 s	5.22 ± 0.26 s
Binary Tree ($h = 4$)	2.051 ± 0.05 s	54.9 ± 0.69 s	13.51 ± 0.28s	6.36 ± 0.08 s
Fat Tree ($k = 4$)	2.605 ± 0.04 s	73.72 ± 0.48 s	59.685 ± 1.23 s	8.40 ± 0.27 s

Table II: Time overhead $\pm$ standard deviation over 10 experiments

	Single Host (Grid'5000)		Multiple Hosts (Grid'5000)				Amazon EC2
Topology	**Mininet**	**Distrinet**	**Mininet CE (SSH)**	**Mininet CE (GRE)**	**Maxinet**	**Distrinet**	**Distrinet**
Linear 2	957.0 ± 0.0	957.0 ± 0.0	731.2 ± 14.6	955.0 ± 0.0	953.0 ± 1.9	957.0 ± 0.0	947.2 ± 5.5
Linear 10	957.0 ± 0.0	957.0 ± 0.0	640.7 ± 8.8	955.0 ± 0.0	897.3 ± 8.3	957.0 ± 0.0	947.2 ± 4.8
Linear 20	957.0 ± 0.0	956.9 ± 0.3	582.3 ± 14.5	955.0 ± 0.0	841.9 ± 21.8	957.0 ± 0.0	938.3 ± 4.3
Linear 50	957.0 ± 0.0	955.0 ± 2.2	391.0 ± 12.3	955.0 ± 0.0	641.3 ± 44.0	957.0 ± 0.0	916.4 ± 7.0
Star 10	957.0 ± 0.0	957.0 ± 0.0	729.1 ± 15.4	955.0 ± 0.0	927.6 ± 2.76	957.0 ± 0.0	946.4 ± 5.3

Table III: Maximum Throughput (in Mbps) $\pm$ standard deviation over 10 experiments. Virtual links bounded at $1\,\mathrm{Gbps}$.

summarizes the results over 10 experiments for each virtual topology. The physical environment is composed by one or two machines completely dedicated to the experiments in Grid'5000. The vLink capacities are set to 1 Gbps. The second and the third columns compare Mininet and Distrinet while running on a single machine. We can see that the maximum throughput measured with iperf is very close between our tool and Mininet. The next set of experiments compares Mininet CE, Maxinet, and Distrinet while distributing the virtual network on two physical hosts (the MTU for the physical interfaces has been set at 1600 bytes to avoid fragmentation). To be fair in the comparison, the distribution of the virtual nodes is done using the Maxinet placement algorithm with all the tools, since Maxinet is the only one that has restrictions on the placement (i.e., Maxinet does not allow to place a vSwitch and a vHost on different physical machines if they are connected through a vLink). As mentioned in Sec. II, Mininet CE does not support TCLinks for vLinks connecting vNodes in different physical machines. Therefore, it is not possible to limit the capacity of the virtual link between two physical hosts. In Table III, we observe that Mininet CE using SSH tunnels does not manage to run at Mininet speed even for a simple topology such as *Linear 2*. The explanation is that an SSH tunnel consumes lots of processing resources, which is confirmed by the fact that Mininet CE obtains results comparable to the ones of Mininet and Distrinet when lightweight GRE tunnels are used instead. Maxinet obtains good results when the topology is not too large and when the traffic is not passing through many switches (e.g., *Linear 2* and *Star 10*). However, performances drop when topologies get larger. Distrinet results, obtained in a distributed environment, are comparable to the ones of Mininet and Distrinet on a single host. In the last column, we show Distrinet on Amazon AWS using two instances (m5.8xlarge [14]). As shown in the table, the performances are similar to the ones obtained in Grid'5000. The difference in terms of performances can be explained by the additional virtualization layer needed on the AWS instance and the network resources shared with other users' instances in the cloud.

C. Experiments with high load

To observe the behavior of Distrinet under high load, we set up an experiment running Hadoop Apache [16]. We compare two sets of runs: one set distributed between one and 100 physical machines, the second set running different topologies on a single physical machine.

Distrinet distributed setup. Fig. 3 shows a simplified overview of the Distrinet setup. It is important to distinguish between the *Distrinet Master/Workers* and the *Hadoop Master/Workers*. Distrinet Master/Workers are the physical machines in which the environment emulation is running, whereas Hadoop Master/Workers are the virtual hosts created inside the physical machines. In the example of Fig. 3, physical Host 1 is the Distrinet Master, while the other physical hosts are the Distrinet Workers. *Physical Host 1* deploys 2 virtual hosts: *h1* (Hadoop Master) and *h2* (Hadoop Worker).

The remaining Hadoop Workers are virtualized inside the different Distrinet Workers.

Distrinet single host setup. The incomplete isolation of Mininet nodes prevents Hadoop from running properly. Hence, we use Distrinet in a single host, since the network performances are very similar with respect to the Mininet's ones (Tab. III).

Experiment. The experiment consists in running a standard Hadoop benchmark function (`hadoop-mapreduce-examples.jar`) [17] in a virtual linear topology. It calculates π using a quasi-Monte

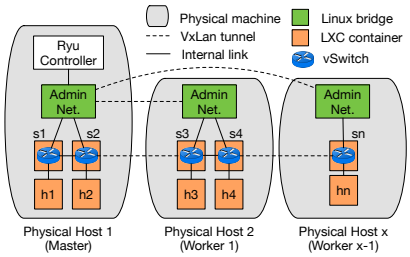

Figure 3: Distrinet setup example

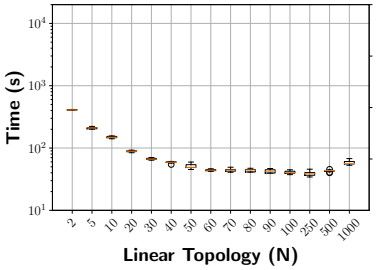

Figure 4: Distrinet running in multiple hosts, placing 10 vHosts and 10 vSwitches on each physical host.

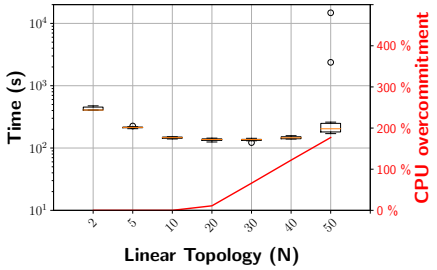

Figure 5: Distrinet running in a single host, with the CPU overcommitment percentage for each topology.

Carlo method; we used 400 maps and 400 samples for each map. Experiments are run for 2 to 1000 Hadoop hosts. Execution times of the experiments are reported in Fig. 4, and Fig. 5, each experiment being repeated 10 times for each topology size. The expected behavior is that the execution time decreases when Hadoop nodes can be executed on an increasing pool of physical resources (cores and memory). Then, adding Hadoop nodes should not change the execution time. We set each vHost with 2 vCores and 6 GB of RAM, while a vSwitch requires 1 vCPU and 3.5 GB of RAM. With these parameters, a single physical host is able to virtualize without **CPU overcommitment** 10 vHosts and 10 vSwitches. We call CPU overcommitment the estimation (in %) of how much CPU is assigned in excess to a physical node, i.e., if a physical machine has 36 cores and the vNodes assigned in the machine require 54 vCores at full speed, the machine has 50% overcommitment.

The behavior of the experiments using Distrinet with each host running 10 vHosts and 10 vSwitches (except for linear 2 and linear 5) is the one expected from Hadoop (Fig. 4). The computation time decreases while the number of vHosts increases, until adding new workers does not decrease anymore the completion time (in this case between 50 and 60 vHosts). When the linear topology is composed of 1000 vHosts, there is a slight increment in terms of the execution time, which probably depends on two factors: (i) the distance between the Hadoop Master host and the last Hadoop Worker (the connection has to cross 1000 vSwitches), (ii) and the large amount of Hadoop Workers that the single Hadoop Master has to manage.

We observe that the behavior using a single machine is hardly predictable (Fig. 5) when the resources in the single machine are not sufficient to satisfy the requests of the virtual instances. For this reason, we need to distribute the load of the emulation in different hosts. Comparing Fig. 4 to Fig. 5, we can observe in the single host emulation scenario that we obtain the same results as in the multiple hosts one when experiments do not exceed the physical resources of the Host (until linear 10). When the emulation requires additional resources, we can observe that the single host emulation needs more time to complete the execution. The red line is an estimation of the CPU overcommitment inside the physical host. Virtualizing the linear 50 network on a single host requires more time than a linear 10. With linear 50, we experienced some abnormal behaviors in two experiments (one run required 2370s and another one 14797s). This is due to an increasing overcommitment of the physical machine.

V. Optimization, Limitations and Future Work

From version v1.2, Distrinet manages to run up to 3200 vNodes using 120 physical machines, but for such large cases parameters tuning is needed on the physical hosts (i.e., increase the maximum SSH sessions and configure LXD in production setup [18]). By default, Distrinet uses the round-robin placement, but it provides the option to use placement algorithms that take into account resources of the testing infrastructure and virtual network requirements. It minimizes the resource required for the experiment without overcommitment (in terms of CPU, memory or network). This avoids to have situations like in Fig. 5. Up to now, Distrinet is able to create containers for the vNodes and VXLAN for the vLinks, but a prototype using GRE tunnels is also available. We are planning to migrate to LXD v4.0.0, with which VMs can also be created using QEMU and KVM, providing the users the possibility to choose to use either containers or VMs. The Distrinet client is available for all platforms (for Windows, it is possible to use Docker); we have extensively tested Distrinet Master and Workers on Ubuntu 18.04, Ubuntu 20.04, and Debian 10. We are also planning to create an automatic provision module for Google Cloud and Microsoft Azure.

VI. Conclusion

To overcome the limitations of resource-intensive experiments being run on a single machine, we proposed Distrinet that extends Mininet capabilities to make it able to distribute experiments on an arbitrary number of hosts. We have shown the cost to adapt an application designed to run on a single machine to run in multiple hosts, while remaining compatible with it. Our implementation is flexible and experiments can be run on a single Linux host, a cluster of Linux hosts, or even in the Amazon EC2 cloud. Distrinet automatically provisions hosts and launches Amazon instances such that experimenters do not need to know the details of the infrastructure used for their experiment. It is compatible with Mininet and its source code is available on https://distrinet-emu.github.io.

REFERENCES

[1] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *ACM SIGCOMM Workshop HotNets*, New York, NY, USA, 2010. ACM.

[2] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, and Chidung Lac. Mininet on steroids: exploiting the cloud for mininet performance. In *IEEE CloudNet*, 2019.

[3] The Mininet Project. Cluster edition prototype. https://github.com/mininet/mininet/wiki/Cluster-Edition-Prototype. Accessed: 2019-01-02.

[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*, pages 1–9, June 2014.

[5] Open vswitch. https://www.openvswitch.org.

[6] Canonical Ltd. Linux containers. https://linuxcontainers.org. Accessed: 2019-05-10.

[7] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[8] Jon Watson. Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1, 2008.

[9] Rodrigo Emiliano and Mário Antunes. Automatic network configuration in virtualized environment using gns3. In *2015 10th International Conference on Computer Science & Education (ICCSE)*, pages 25–30. IEEE, 2015.

[10] M. Peuster, H. Karl, and S. van Rossem. Medicine: Rapid prototyping of production-ready network services in multi-pop environments. In *IEEE NFV-SDN*, pages 148–153, Nov 2016.

[11] Mininet cluster edition tclink discussion. https://mailman.stanford.edu/pipermail/mininet-discuss/2016-July/007005.html.

[12] Ubuntu fan networking. https://wiki.ubuntu.com/FanNetworking.

[13] Create a personalized host image. https://distrinet-emu.github.io/personalize_vhost.html.

[14] M5 instances aws. https://aws.amazon.com/ec2/pricing/on-demand/?nc1=h_ls.

[15] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[16] Apache hadoop documentation. http://hadoop.apache.org.

[17] Running hadoop examples. http://www.informit.com/articles/article.aspx?p=2190194&seqNum=3.

[18] Setup lxd in production. https://github.com/lxc/lxd/blob/master/doc/production-setup.md.

APPENDIX

A. Distrinet experiments

It is possible to reproduce the experiments shown in the paper by installing Distrinet.

Requirements: an Amazon AWS account or at least 2 physical machines. We suggest to use Ubuntu:18.04 and install Distrinet as root user. The machines have to be connected in the same network.

The first thing to install is **Distrinet client** on your machine.
A step by step tutorial for the client is at :
https://distrinet-emu.github.io/installation.html

After the installation, you can use AWS or a general environment for your Distrinet Master and Workers.

If you want to use **AWS**, you have to follow this tutorial:
https://distrinet-emu.github.io/amazonEC2.html

If you are using a **general cluster**, you can follow this tutorial:
https://distrinet-emu.github.io/general_environment.html

After the installation, you can follow the **Hadoop Benchmark** tutorial at:
https://distrinet-emu.github.io/hadoop.html

B. Experiments plots

It is possible to plot the results, using the Jupyter notebook and the data included in the submission.