Version of Record: https://www.sciencedirect.com/science/article/pii/S0140366422002523
Manuscript_c8ed7c3fdfb50e2d767424b311f769a8

Passive Delay Measurement for Fidelity Monitoring of Distributed Network Emulation

Houssam ElBouanani, Chadi Barakat, Walid Dabbous, and Thierry Turletti
Inria, Université Côte d'Azur, France

***Abstract*— Emulation has become a popular approach for the validation and evaluation of network research. It provides researchers with a contained, customizable, and scalable testing environment, which can be easily packaged and published for potential readers to reproduce their results. However, as the network components are only virtual, emulation lacks the inherent realism of physical testbeds. In light of this, monitoring specific metrics of the emulated network has been proposed as a solution to mitigate to some degree inaccuracies caused by emulation. While this is not difficult to implement in a single-machine setting (e.g. with *Mininet*), monitoring is limited by the lack of time synchronization in scenarios where the emulation is distributed over multiple physical machines (e.g., *Distrinet*). In this paper we tackle the case of packet delay monitoring, to which we propose a methodology for passively measuring one-way delays with underlying assumptions about time synchronization, and round-trip delays otherwise. For an efficient implementation of our methodology, we propose an *eBPF*-based packet measurement tool that performs better than current packet sniffers under emulation-specific assumptions. We implement and evaluate our system in an open testbed and show that it can reach results within few microseconds of perfect accuracy and precision.**

I. Introduction

The design and engineering of new network protocols and architectures require rigorous functional testing and evaluation to finely examine their implementability and performance in practice. Network emulators, such as *Mininet* [1], are becoming popular means to conduct network experimentation. These tools mimic the operation of network hardware using software tools, on which they can run actual application and operating system code. As such, they allow users to create and reproduce lightweight network *testbeds* on their computers through an easy-to-use Python interface. As for computing-intensive scenarios, when *Mininet* cannot emulate more than a certain number of hosts and network hardware devices due to resource limitations inherent to the physical machine intended to run it, several researchers have worked on distributed versions of *Mininet*, ones that let users emulate large-scale networks over a geographically localized cluster of multiple physical machines. *Distrinet* [2], *Maxinet* [3], and *Mininet Cluster Edition* [4] are such iterations, some of which particularly focus on reproducibility by natively allowing users to run their emulations on public clouds such as Amazon's AWS.

However, researchers have shown that network emulators do not always provide perfectly accurate results [5]. In fact, as they are designed for running on everyday laptops, their emulation of multiple events (e.g., running code in emulated hosts, switching and routing multiple packets in parallel, etc.) is very limited by the available computing and network resources [6]. This renders them practically unusable for emulating latency-sensitive scenarios or those that require packet-level precision. Researchers have thus proposed *fidelity monitoring* [7] as a way to achieve more accuracy and precision by appropriately allocating computing and/or memory resources for emulated hosts and by finely monitoring the network packets throughout their journeys in the network. Essentially, as each packet at each hop of its path will experience multiple amounts of delay (propagation, transmission, queuing, switching, etc.), the experiment may be labeled "inaccurate" if an unacceptable fraction of the packets were not appropriately delayed on each of the emulated links. Even though other performance metrics can be also monitored (e.g., bandwidth, queues' sizes, etc.), the fine-grained monitoring of packet delays can ensure very high-fidelity emulation with good enough guarantee on accuracy, and can also be used to monitor overall performance and infer information about other metrics, especially the bandwidth and the queues' sizes.

In practice, packet delay monitoring inevitably requires measuring packets' network delays between multiple nodes of the virtual network. In a distributed setting, however, such virtual nodes can be hosted at different physical machines, which generally do not have the same perception of physical time even if geographically localized. Therefore, implementing fidelity monitoring on a distributed network emulator raises a complicated sub-problem: *accurate passive delay measurement between physical machines of a network*. In this paper we focus at tackling this problem in the particular context of distributed network emulation. Specifically, we answer the following questions: how can one accurately monitor packet delays in a distributed environment? and in particular, how can one passively measure delays of packets exchanged between physical machines in a cluster?

Our contributions in this work are manifold: we present a methodology to passively measure the one-way delay (OWD) of packets –with an accuracy of up to mere microseconds– between physical machines and/or virtual machines hosted on separate physical machines, to be used for monitoring purposes in the context of distributed network emulation. We further present an extension of our methodology to the monitoring of the round-trip delay (RTD) in scenarios when accurately measuring the OWD is not possible due to time synchronization assumptions. We also introduce a non-intrusive and distributed packet measurement tool based on the extended Berkeley Packet Filter (*eBPF*) [8], which is

highly compatible with network emulators. The new packet measurement system is then evaluated on a real use case to show that it can reach its objectives.

The remainder of this paper is organized as follows. In Section II we quickly present a background on delay measurement and time synchronization. We then move on to lay the ground for our passive delay measurement system in Section III with preliminary discussions on packet identification and packet timestamping. In Sections IV and V we introduce methods to passively and accurately measure one-way and two-way delays respectively, of which we present a distributed implementation. In Section VI we benchmark our measurement tool against standard packet sniffers, then we briefly evaluate our system in Section VII, before concluding and discussing our current and future work on high-fidelity distributed network emulation in Section VIII.

II. Background

A. Delay Measurement

Unlike the throughput which is a flow-level measure, the network delay is a value that characterizes either an individual packet or a pair of request-response packets. In general, the packet delay is the amount of time needed for one or a pair of packets to travel from one point to another in a path of one or multiple physical media and eventually one or multiple intermediate networking nodes (switches, routers, proxies, firewalls, etc.). From this general model, the network delay can be precisely defined along three axes:

- one-way vs round-trip: whether the delay is defined for single packets (one-way delay), or pairs of packets in opposite directions (round-trip delay);
- one-hop vs end-to-end: whether the delay is defined on a single transmission medium separating two layer 1 and above machines (one-hop delay), or on a whole path separating two layer 4 and above machines (end-to-end);
- application- vs system- vs hardware-level: whether the delay is considered at the point in time when the application creates the message, when the message is made into a network packet and then into a system data structure, or when the transmission hardware sends the packet as a stream of bytes.

For example, the classical definition [9] considers a one-hop, hardware-level model of the one-way delay. In this definition, the OWD of a packet P between two machines A and B (which can be user terminals, servers, routers, switches, etc.) separated by a communication medium (wired or wireless) is the duration of (absolute) time between the instant when A sent the first bit of P, and the instant when B received the last bit of P. While this can be deemed a *pure* model of the *network* delay, as it does not involve any system-level latency, it is very hard to accurately measure. In fact, it inevitably requires using specialized network hardware to timestamp packets in order to measure their delays.

In this paper, as we are mostly concerned with measurement, we consider a more relaxed model of the one-way network delay: we define it as the one-hop, system-level delay. This delay has the advantage of being measurable using simple software tools without the need for any additional hardware, and thus it can be easily measured in scenarios involving virtual and/or emulated machines and networking equipment. This delay can be decomposed into three contributing terms:

- The system (or queuing) delay: which mainly consists of the amount of time that the packet will spend in the system queues waiting to be transmitted;
- The transmission delay: the amount of time needed for the transmitting hardware (NIC, router interface, switch port, etc.) to write the packet onto the physical medium. This delay depends on the writing speed of the hardware, the transmission speed of the medium (also known as its bandwidth or capacity), as well as the size of the packet; *and*
- The propagation delay: the length of time needed for the signal to travel from A's transmission hardware to B's receiving hardware. It is mainly characterized by the propagation speed of the signal and the dimensions of the medium and does not depend on the size of the packet.

In the case of wired media, this decomposition can be summarized into the following formula:

$$d(P) = \frac{l}{v} + \frac{S_{Q(P)}}{B} + \frac{S_P}{B}, \tag{1}$$

where $d(P)$ is the total one-way delay of a packet P of size S_P between two machines separated by a link of length l, velocity factor v, and bandwidth B; and where $S_{Q(P)}$ is the size of the queue (including remaining bits of the head-of-line packet) at the instant when P arrived.

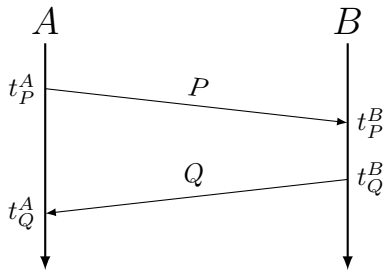
The measurement tool we will present can also be used to measure multiple-hop (and end-to-end) system-level one-way delay.

Note that in cases where A and/or B are virtual hosts, switches, or routers separated by a physical network (e.g., A is a virtual machine hosted in a physical machine, and B, a virtual switch hosted in a different physical machine), the delay needs to be measured between the virtual NICs of A and/or B, not the physical NICs of their hosting physical machines. Thus when virtualization is involved, the delay of a packet also accounts for the *system* delay between the virtual node's virtual NIC and the hosting machine's physical NIC.

Accurately measuring packets' OWDs and successfully decomposing them into their three components can give useful insights about the network: from the transmission delays of multiple packets one can infer the bandwidth of the medium; and a long system delay can signify congestion or saturation of computing resources. However, this is not always an easy task, and researchers have proposed many techniques to estimate the OWDs of probe packets up to varying degrees of accuracy, most of which require proper hardware (GPS systems, specialized NICs, etc.) [10].

An easier value to measure is the round-trip delay (RTD). The RFC 2681 [11] defines it for a pair of request-response packets P and Q as the duration of (absolute) time between the instant when A sent the first bit of P, and the instant when A received the last bit of Q. It is thus a round-trip, end-to-end, hardware-level model of the delay, and is in fact equal to the sum of the individual one-way hardware-level delays of packets P and Q, and the processing delay between the reception of the request packet by B and its sending of the response packet. Certainly, the information on the individual OWDs is lost when measuring the RTD.

This definition of the RTD can also be relaxed to make it measurement-friendly. In this paper, we rather define it from a system-level point of view, as well as extend its definition from simple request-response packets, to almost any pair of packets. For a couple of packets P and Q such that P was sent from A before Q was received by A (see below), we define the round-trip, one-hop, system-level delay as simply the sum of their individual one-way, one-hop, system-level delays, $t_P^B - t_P^A$ and $t_Q^A - t_Q^B$, without accounting for the "processing" time $t_Q^B - t_P^B$ by B between the reception of P and the sending of Q. The time elapsed between t_P^B and t_Q^B is not relevant in the general case since P and Q do not need necessarily to be related packets (unlike ICMP echo request-response, TCP SYN-ACK, etc.). Again, the measurement tool we will present can also be used to measure the round-trip, multiple-hop, system-level delay.



The use of ICMP echo probes is the de facto active method for measuring RTDs [12], [13]. It works by simply sending a probe "echo request" ICMP packet and waiting for the destination to answer with an equal size "echo response" ICMP packet. The source timestamps the instant when the request packet is sent and the instant when the response packet is received, and reports the round-trip time (RTT) as the difference between the two. It accurately measures the round-trip, end-to-end, application-level delay with no need for time synchronization, and thus can be used in all cases without relying on external hardware. Other more powerful tools[1,2] can be used to send upper-layers probes (UDP, TCP, or application-level protocols).

B. Clock Synchronization

One-way delay measurement is intricately tied to the problem of clock synchronization. Without specialized hardware to measure network delays, relying on software- or operating system-level mechanisms inevitably requires some degree of synchronization between clocks that ought to timestamp probe packets (or in the case of passive measurement, data packets) [14]. The problem particularly arises because the time dissimilarity between the clocks of different machines (called clock offset) changes over time. This is due to differences between the clock frequencies (called clock skew) which are sensitive to physical phenomena (such as hardware heating) that also change over time [15]. This problem has been extensively studied in the scientific literature, and numerous protocols based on different sets of assumptions have been proposed to continuously resynchronize clocks of machines connected by LANs or WANs.

The Network Time Protocol (NTP) is the most used solution for clock synchronization [16]. It organizes machines into a tree-like hierarchy, where the root node is the primary server which is generally connected to a highly reliable source of time (e.g., an atomic clock) and which will propagate its time to other nodes of the hierarchy through protocol messages; other nodes synchronize their clocks to the root server and eventually propagate the time to nodes in lesser levels of the hierarchy. The process reiterates as clocks naturally drift from each other. At the convergence of the algorithm, each node will be synchronized to its server with a precision on the order of the network jitter. Thus, in an Ethernet LAN, NTP can theoretically guarantee precision down to 100 or even 10 microseconds, provided it is given long enough time to converge.

As applications in distributed systems have become reliant on finer levels of time synchronization, a more powerful protocol was proposed: the Precision Time Protocol (PTP), also known as IEEE 1588 [17]. Just like NTP, PTP organizes nodes into a hierarchy of "masters" and "slaves" (where a node can be both a master and a slave) and uses protocol messages to exchange time information between nodes of the hierarchy. But unlike NTP, which can be implemented on any device with a Network Interface Card (NIC), PTP requires special NICs with integrated time clocks. This allows high-resolution synchronization by relying on the NIC clocks to timestamp protocol messages, thus avoiding all delays caused by software and operating system-level processing.

In [19], the authors show that with proper configuration of NTP and PTP software in a local Ethernet network, it is possible to achieve precision on the order of 10 microseconds with NTP, and on the order of 100 nanoseconds with PTP, without incurring much overhead on the network. In fact, they show that by synchronizing clocks every 8 seconds with NTP, the total overhead of protocol messages is 23B/s per client and the one of computing resources is negligible; and by using PTP, the total network overhead is 186B/s per client, and the one of computing is also negligible. In our work and in settings where time synchronization is needed, we will use their findings to configure our testbed.

C. Packet Monitoring

Packet timestamping is another inevitable requirement for passive packet delay measurement. Both end hosts need to

[1] *hping*3: https://linux.die.net/man/8/hping3
[2] *tcpping*: http://www.vdberg.org/~richard/tcpping.html

record the instants each packet was seen by their NICs, and send that information to compute the delay from the individual timestamps. And as with clock synchronization, there are specialized hardware that can tap into NICs and extract information from data packets with minimal interference on the traffic. This solution, although most efficient in terms of performance, is not suitable for two main reasons:

- Firstly, it requires physical access to the machine on which the tap must be installed. This is especially restrictive in our context of network research where the user might be running her experiments on a remote grid or cloud; and
- Secondly, it cannot work in situations with virtualization as packets must be timestamped at the virtual NIC level. It is also particularly ineffective when system-level traffic control mechanisms are in place to add delay or bandwidth to physical or virtual links.

Thus, any packet timestamping tool needs to be implementable in virtual NICs and be compatible with traffic control mechanisms. To this end, using traffic sniffers (e.g., *libpcap*[3]) is the most straightforward solution. These tools simply capture packets as they go through the (physical or virtual) NICs for monitoring and analysis purposes. However, their intrusiveness in high-speed networks needs to be mitigated by intelligent sampling. They are also not naturally compatible with traffic control, as they capture outgoing traffic after being shaped, but this too can be mitigated by system-level packet redirection.

Another solution is to leverage kernel tracers and the recent advances in kernel programming. The *eBPF* is one such solution that allows users to run their code in kernel space through a secure and contained virtual machine with its own registers, memory space, and helper routines. More precisely, it allows users to attach pieces of code to certain kernel functions. Its use cases include monitoring and troubleshooting kernel operation, and high-performance packet processing (filtering, routing, etc.). It can also be integrated within the Linux Traffic Control suite [18] to perform powerful and flexible packet classification and traffic shaping with minimal overhead. We will use it in our work to implement a basic but efficient timestamping tool for passive delay measurements.

III. Preliminaries

A. Testbed

All of the following testing has been performed on the open testbed R2lab[4]. The platform includes a cluster of machines that are connected through Gigabit Ethernet wires and store-and-forward switches. In our tests, machines (equipped with a CPU Intel Core i7-2600 processor and 8 gigabytes of RAM) are running a Ubuntu 18.04 Linux distribution over a 4.15.0 kernel.

To evaluate how our delay measurement methodology behaves in different network settings, we leverage Linux Traffic Control [18], which is a standard tool used by network emulators to simulate links of different properties (bandwidth, propagation delay, packet loss, packet duplication, etc.). In parallel, we use *ping*, *hping*3, and *tcpping* as active round-trip, end-to-end, application-level delay measurement tools to provide reference performance metrics to compare against. These tools allow for controlled probe sizes and sending rates, and will thus also serve as packet generators throughout our experimentation. We will also use netsniff-ng's *trafgen*[5] to customize packet generation when it is necessary. As for timestamping, we will use an *eBPF* program to capture and timestamp incoming or outgoing packets, and also the *libpcap* Linux utility for comparison. The collected data is then analysed by a Python program. We provide scripts[6] to reproduce all our results.

B. Packet Identification

Identifying packets is necessary for passive delay measurement. In order to measure any type of delay, timestamps at the source and destination hosts have to be matched to compute the delay. Ideally, the hosts can simply identify packets by their order, i.e. the first packet sent from a source A to a destination B corresponds to the first packet received at the destination B from the source A. But as packets can be lost or arrive unordered due to several reasons, especially in cases of multiple-hop or end-to-end delay measurement, more sophisticated mechanisms have to be implemented. Another straightforward solution is to tag all packets, either by a unique packet ID, or even directly by adding the packet timestamp to its header at the source. However, this requires unnecessary modifications to the operating system's network module, and can incur non-negligible network overhead at scale.

In our context of distributed network emulation, all packets are encapsulated in UDP datagrams as soon as they leave the emulated host (Distrinet uses VXLAN while Mininet Cluster and Maxinet use GRE). We can therefore safely make the assumption that all packets are IPv4 packets, and for each flow of packets sent from a certain source to a certain destination, use the native ID field of IPv4 as identification tag. Unfortunately, this still has two major limitations: the ID field in IPv4 headers is shared between all fragments of a long packet and is encoded on 16 bits only which can lead to collisions. The first limitation can be managed by considering the pair *(ID, Fragment Offset)* as identification tag; the second limitation is trickier since packets with the same ID from the same source can arrive unordered. However this generally does not happen very often, but to make such assumption, we must ensure that packets take less time to get to their destination than it takes for their source to circle through the range of possible packet IDs. Formally, the assumption holds when $\Delta < 2^{16}\tau$, where Δ is an upper bound on the network delay, and τ is the average interarrival time of packets (equal to the average packet size over the

[3] *libpcap*: https://www.tcpdump.org/
[4] R2lab Anechoic Chamber: https://r2lab.inria.fr/
[5] netsniff-ng: http://netsniff-ng.org/
[6] See https://github.com/helllb/delay

bandwidth). It is generally the case because longer links (i.e. larger propagation delay) correlate with lower bandwidth (i.e. larger interarrival time). And even in our testbed with high bandwidth, low delay, and small packets, this condition holds as $2^{16}\tau = 30ms$ and $\Delta < 1ms$.

Thus in our delay measurement system, all packets are identified by a hash of the *(Source Address, Destination Address, Packet ID, Fragment Offset)* fields from their IPv4 header.

C. Workflow

The idea is simply to intercept and timestamp, by each machine and at each of its NICs, all sent packets as late as possible, and all received packets as soon as possible. This information can then be extracted and used to compute packet delays.

More specifically, the system we propose for the measurement and estimation of delay is built up from three main components.

1) Packet loggers: This component is a program written in eBPF specification and which therefore runs in kernel space. Its goal is to capture and log information about sampled packets (Figure 1a). It can be further split into two parts:

- One part of the program, in charge of outgoing packets, is plugged as a *k-probe* into the `qdisc_enqueue` routine in the TC egress queue of all probed interfaces in the machine. It runs its instructions whenever a packet is sent from the network stack for queuing, and logs general and enqueuing information about the packet: its ID, a timestamp of its transmission, and its size;
- A second part of the program, in charge of incoming packets, is plugged into the TC ingress queue of all emulated interfaces. It runs its instructions whenever a packet was received and can log its ID and reception timestamp.

2) Local monitoring agents: This component is a user space program that runs on each physical machine and whose goal is to receive the logs from the packets loggers and compile them into structured tables that can later be used for analysis (Figure 1b). There are two possible solutions to achieve this:

- Either the eBPF program logs all the information in files in persistent storage, then at the end of the monitoring period the monitoring agent gathers the files, parses the information out, and finally compiles it into structured tables; *or*
- The eBPF program sends the information by batches (of hundreds or thousands of packets) to the monitoring agent via user space-to-kernel space communication protocols (e.g. Linux Netlink). Alternatively eBPF also offers an interface for shared memory between eBPF programs and user space applications. The monitoring agent then compiles the packet information *online*.

In our current implementation, where we do not see a specific advantage to online monitoring, we will adopt the first solution.

3) A collector/analyzer: This component is the brain of the system. Its job is to collect and analyse packet information compiled by the monitoring agents (Figure 1c). It is logically unique and achieves its goal in three steps:

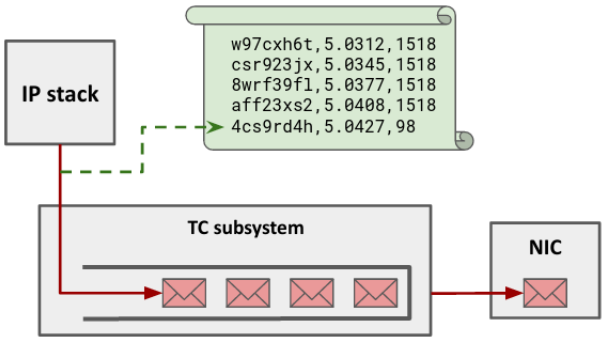
- First is the collection of data from monitoring agents;
- Then the individual tables received from the monitoring agents are cross-examined to match information about packets distributed over multiple tables. For instance, a table from one monitoring agent (and therefore from one machine of the cluster in the case of distributed emulation) can contain the sending timestamp of a packet, and another table from a different monitoring agent can contain the reception timestamp of the same packet. The output of this step is a unique large table where each entry corresponds to a packet and is uniquely identified by its ID, and which contains all information about it;
- Finally packets from the table are paired together according to a pairing rule and their joined RTD is measured (from the logged timestamps) and estimated (from other information such as packet sizes, queue lengths, etc.). These two values are then compared for all considered pairs of packets and an overall judgement about the emulation can be made.

The design we propose for the system is distributed but very hierarchical. It is distributed in that at the lower levels, the packet loggers and the local monitoring agents have many independent instances: each virtual interface runs a replica of the packet logger, and each machine that hosts part of the emulated network runs a local replica of the monitoring agent. The architecture is hierarchical in the sense that the collector/analyzer communicates with all the monitoring agents of the infrastructure, and each monitoring agent communicates with the packet loggers of the virtual interfaces. The advantages of such organization are mainly twofold: first is that it centralizes most of the intelligence and keeps the packet information logging part as lightweight as possible to not impact the normal operation of the emulated network and of the hosting system; second is that it perfectly integrates into distributed emulators' architectures, which also involve a *leader/workers* architecture.

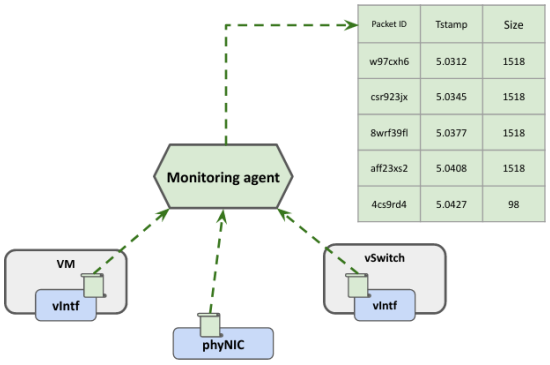
IV. Passive OWD measurement

In this section we study the extent to which it is possible to passively measure the one-hop and end-to-end OWD, i.e., the delay of data packets exchanged between a pair of machines from the testbed. From the packet dumps generated by the monitoring agents in accordance with the previously described workflow, we measure the OWDs of packets using the method described in Algorithm 1.

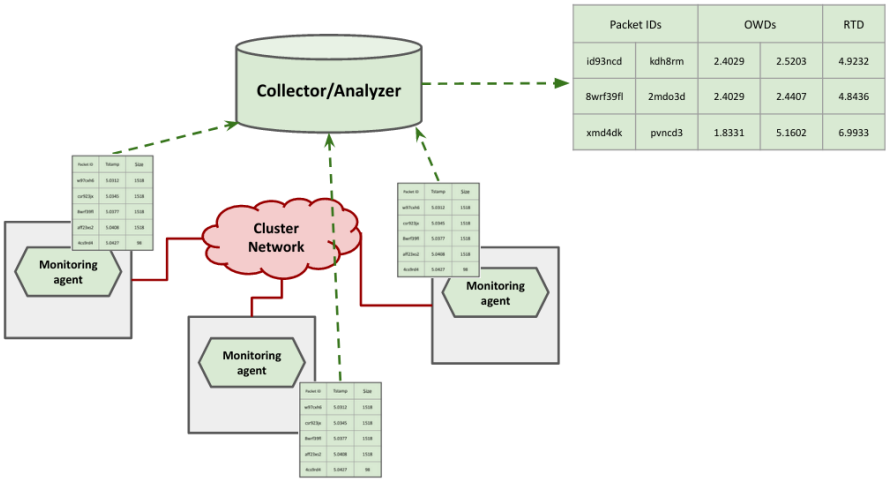
However, without proper time synchronization, it is practically impossible to accurately measure the OWD between two machines with different clocks. Consider for example the plots in Figure 2. We show the OWDs of generated ICMP packets measured by our method with no clock synchronization for a large number of ICMP packets sent with 1 a ms interval. We can clearly see how the two machines' clocks drift over time, how this drift affects the measurement of



(a) After a message is made into a packet and then into a data structure, it is enqueued by the TC subsystem before being dequeued and sent to the NIC for transmission. The packet logger logs in raw files information about the packet when it is enqueued (ID, enqueue timestamp, size).



(b) The monitoring agent retrieves the log files from the eBPF programs plugged into the interfaces of the network's nodes, and compiles the parsed information into structured tables.



(c) The tables sent by the monitoring agents in the machines of the cluster are collected and made into a final table that summarizes the data and computes measured delays.

Fig. 1: Overview of the system.

the OWD, and how it is difficult to predict it as it itself changes over time. In general, the clock skew depends on uncontrollable physical phenomena (e.g., hardware heating) which cause clock offset between the machines that changes in a non-linear fashion. Note also how the clocks largely drift over a relatively short period of time (17 milliseconds in a 100 seconds-long run), making the noise caused by the clock offset hide all the information from the actual network delay.

Nevertheless, running NTP on the testbed almost perfectly solves the problem. At the convergence of the NTP process for clock synchronization and frequency stabilization, the clock offset and skew are almost neutralized and our method starts reporting *good* results. We can see this in Figure 3, where we report on the results of our method after NTP has stabilized. We can notice how at convergence of NTP, the standard deviation of the measured OWD is less than $10\mu s$.

V. Passive RTD Measurement

The OWD measurement method gives accurate results only if the end hosts' clocks are highly synchronized. While this is not impossible in practice thanks to NTP, it requires that the machines be in a local network with reasonably low delay and jitter values to be able to reach high-resolution time synchronization. Furthermore, the NTP algorithm can take long time to converge. In our setting, the convergence of NTP was observed two hours after NTP had started. This makes OWD measurement difficult and inflexible. In this section we propose a new method to passively measure the RTD that does not require such strong assumptions.

The method for passively measuring the RTD and OWD follows a similar approach: a program that captures and timestamps packets between the NIC and the upper layers is installed on the machines, then the packet dumps are sent to a collector which is in charge of computing the RTDs from the information in the packets (namely their IDs) and their timestamps. In the case of the RTD, for each pair of

Data: Packet dumps from A and B: dump_A, dump_B
Result: Arrays of (packet_ID, owd) pairs
initialize arrays OWD_AB and OWD_BA;
foreach *(packet_ID, timestamp_A) in dump_A[outgoing]* **do**
  lookup matching packet_ID in dump_B[incoming] with the closest timestamp_B;
  compute owd := | timestamp_B - timestamp_A | ;
  add (packet_ID, owd) to OWD_AB;
end
foreach *(packet_ID, timestamp_B) in dump_B[outgoing]* **do**
  lookup matching packet_ID in dump_A[incoming] with the closest timestamp_B;
  compute owd := | timestamp_A - timestamp_B | ;
  add (packet_ID, owd) to OWD_BA;
end

Algorithm 1: Passive OWD measurement algorithm

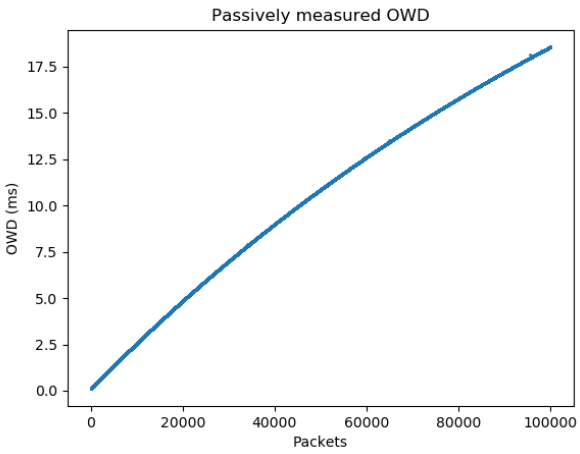


Fig. 2: Measured OWD between two machines before clock synchronization.

machines A and B, and for each packet P sent from A at time t_P^A (in A's clock) and received on B at time t_P^B (in B's clock), and Q sent by B at time t_Q^B (in B's clock) and received on A at time t_Q^A (in A's clock), such that $t_Q^A > t_P^A$, the collector will report the RTD of packets P and Q as:

$$\widehat{RTD}(P,Q) = (t_Q^A - t_P^A) - (t_Q^B - t_P^B).$$

Similar to the previous passive OWD measurement method, this does not always give perfectly accurate estimations of the RTD. In fact, while it does eliminate any inaccuracy due to constant clock drift between the two machines, (i.e., the clock drift at time $t = 0$) it is still vulnerable to its variation. In fact, the longer the time interval between the two packets P and Q, the more the clocks might have drifted during that interval, and the larger the error that

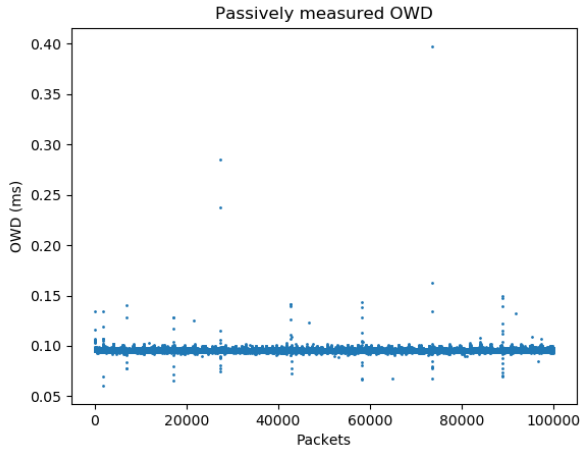


Fig. 3: Measured OWD between two machines after clock synchronization.

Data: Packet dumps from A and B: dump_A, dump_B
Result: Array of (packet1_ID, packet2_ID, rtd) tuples
initialize array RTD;
compute arrays OWD_AB and OWD_BA;
foreach *(packet1_ID, owd1) in OWD_AB* **do**
  lookup first (packet2_ID, owd2) in OWD_BA;
  if *timestamps of packet1 and packet2 are close enough* **then**
    compute rtd := owd1 + owd2;
    add (packet1_ID, packet2_ID, rtd) to RTD;
  end
end

Algorithm 2: Passive RTD measurement algorithm.

will be induced. Thus, in practice, the collector should only stick to pairs of packets sent and received within a small enough time interval τ so that the error caused by clock drifts on the estimation of RTD is no larger than a tolerance value δ. This ensures that whenever P and Q are such that $t_Q^A - t_P^A \leq \tau$, we have:

$$|\widehat{RTD}(P,Q) - RTD(P,Q)| \leq \delta.$$

Note that when NTP is active, it will periodically correct the clocks which could cause sudden drifting that can affect the accuracy of the method. However, as NTP is limited to one resynchronization every 8 seconds, the error is insignificant.

To evaluate this passive RTD measurement method, we conduct the same experiments as earlier, where we passively measure the delays of generated packets. However, to provide a baseline to compare our method against, we use the *ping* tool to generate ICMP echo packets and measure their round-trip, application-level, end-to-end delay. Figure 4 shows how the RTDs measured by our method, in the

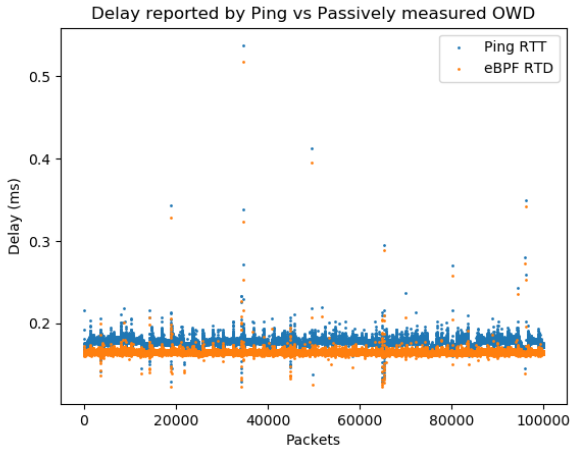


Fig. 4: *ping* RTT and Passively measured RTD.

absence of time synchronization by NTP, compare to the RTT reported by *ping*. Since time synchronization –or rather time asynchronization– does not impact this RTD measurement method, we can safely give explanations as to the difference between the two measured values:

- Firstly and most importantly, the two methods (active measurement with *ping* vs our passive RTD measurement method) do not exactly measure the same thing. As mentioned earlier, the former measures the delay between emission of echo request packets and reception of their corresponding echo response packets, which includes the processing time at the destination. The latter only attempts to measure the network delay without accounting for system-added delays when possible, which makes it more accurate in our context of passive measurement of network delay; and
- Secondly, as our measurement solution runs in kernel space instead of user space, it does not suffer from any delay variation caused by random process scheduling and user space-to-kernel space communications.

VI. Other Measurement Tools

In the previous sections we have used our methodology with an *eBPF*-based packet logger. Its main strength compared to standard packet sniffing tools (such as the Linux *libpcap* library used by *tcpdump* and *wireshark*) is in its flexibility. In effect, as *eBPF* allows users to run code in kernel space, through kernel routines, and in parallel with kernel operations, much more can be achieved beyond simple timestamping of packets. For instance, and unlike with *libpcap*, it is possible to get information about the context surrounding the passage of each packet (e.g., NIC, system, or socket queues lengths) and correlate it with its delay for better analysis.

A second advantage of using *eBPF* for timestamping is that it is perfectly compatible with Linux Traffic Control (*tc*). In fact, we have chosen in our testbed to timestamp packets as they pass through the *tc* subsystem: our timestamping program is run each time *tc* runs its `qdisc_enqueue` routine, unlike *libpcap* that captures and timestamps packets when they pass through the network device (see Figure 5). This choice is not arbitrary as in the context of network emulation, emulators use *tc* to configure link parameters such as network delay, which cannot be captured by a measurement program if the packets are not timestamped *before* any emulated delay is added. To see this, consider Figure 6 that plots the passively measured RTD using both our *eBPF* packet logger and *libpcap* as timestamping tools, in the same testbed as before but with an added 1 ms of delay in both ways. We can clearly see how *libpcap* only measures the propagation delay of the physical medium (around $170\mu s$) and not the emulated delay (2 ms), unlike what *ping* and our *eBPF*-based method report.

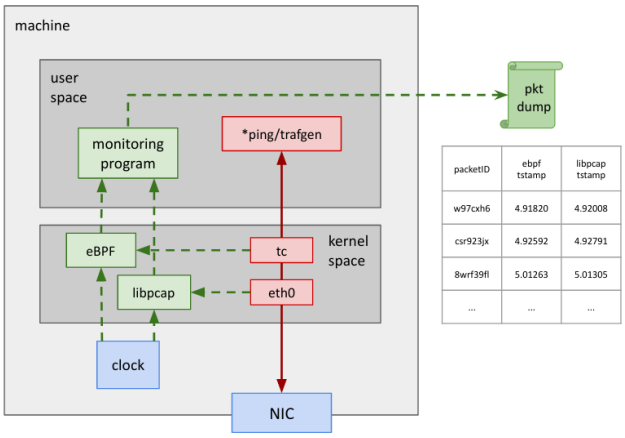


Fig. 5: Implementation with *eBPF* and *libpcap*.

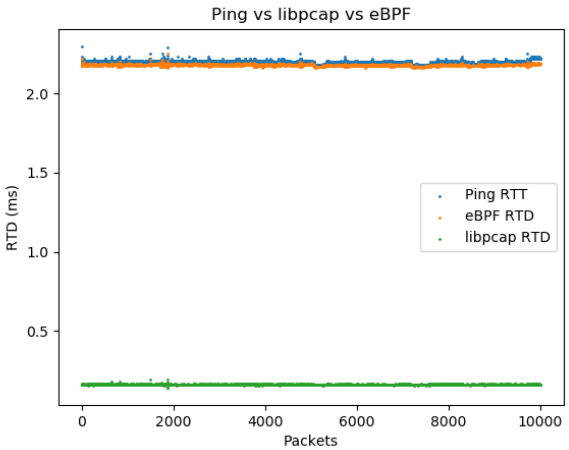


Fig. 6: *ping* RTT and passively measured RTD with emulated delay.

Having said that, one can mitigate this issue with *libpcap* by creating a virtual network device to intercept all packets before they go through *tc* (see Figure 7). The downside is that this solution will add system delay and jitter to the packets

that will be accounted for in their delay measurements. Figure 8 shows this: while the measured RTD using *libpcap* is close to what our *eBPF* program measures (sum of the physical and the emulated delays), the complexity of the setup adds delay (up to $50\mu s$), and jitter ($12\mu s$) to the packets.

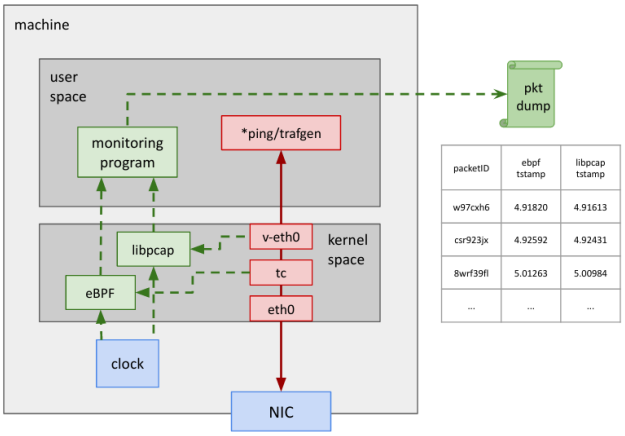


Fig. 7: Measurement system with *eBPF* and *libpcap*: modified setup.

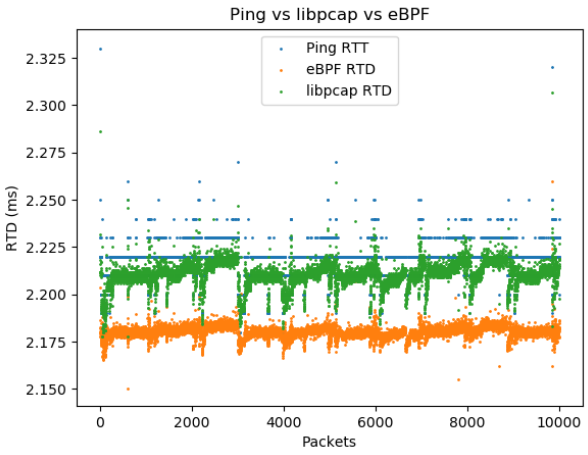


Fig. 8: *ping* RTT and passively measured RTD: modified setup.

VII. Use Cases

In addition to measuring network delays for latency-centered performance evaluation, our passive delay measurement methodology can be used to indirectly measure and/or estimate other network variables. In this section we focus on the bandwidth (or capacity), and provide two examples of how our measurement methodology can be used to infer links bandwidths.

A. Testbed

For the following experiments, we emulate a simple network consisting of two hosts connected by three cascading

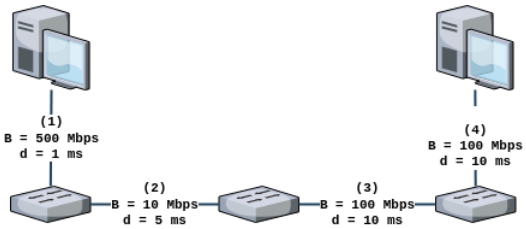


Fig. 9: Emulated testbed for bandwidth estimations. Each link is a full duplex wired link of bandwidth B and propagation delay d.

switches (Figure 9). This scenario is emulated using Distrinet in a single node of the R2Lab cluster, which is equipped with a CPU Intel Core i7-2600 processor and 8 gigabytes of RAM, and runs a Ubuntu 18.04 Linux distribution (kernel v4.15.0) with basic functionalities and no particular application running in the background. Each host then generates a flow of random-sized packets to the other. No other traffic runs between the two emulated hosts.

B. One-hop Link Bandwidth

As stated earlier, the round-trip, one-hop, system-level delay on a wired link is equal to the sum of its propagation delay along the link, its transmission delay by the hardware and the medium, and its waiting time in the queue, according to the formula:

$$d(P) = \frac{l}{v} + \frac{S_{Q(P)}}{B} + \frac{S_P}{B}, \tag{2}$$

When enough variables are known, this formula can be used for the estimation of the bandwidth B. In fact, according to the method famously described by the authors in [20], by generating probe packets of varying sizes and then measuring their delays, it is possible to infer the bandwidths of each link along the path. However, thanks to our passive measurement methodology, it is possible to achieve this without injecting packets into the network but rather only from the passively measured delays of data packets.

Consider a wired network link connecting two (physical or virtual) interfaces A and B. For each packet P going from A to B, and each packet Q going from B to A, their round-trip delay is equal to:

$$RTD(P,Q) = 2 \cdot \frac{l}{v} + \frac{S_{Q(P)}}{B_1} + \frac{S_{Q(Q)}}{B_2} + \frac{S_P}{B_1} + \frac{S_Q}{B_2},$$

where B_1 and B_2 are the bandwidths in both directions of the link. Thus for packets that are not queued, the round-trip delay is a simple linear function of their sizes.

In the above described testbed, we use the measurements collected at each end of the links to estimate their bandwidths based on the previous formula. We use a simple linear regression model to fit all RTD measurements against packet sizes (Figure 10). With just few hundred pairs of passively collected packets, we obtain good enough estimations of bandwidths: 439.434 Mbps for Link (1); 9.907 Mbps for Link (2); 129.793 Mbps for Link (3); and 111.329 Mbps for

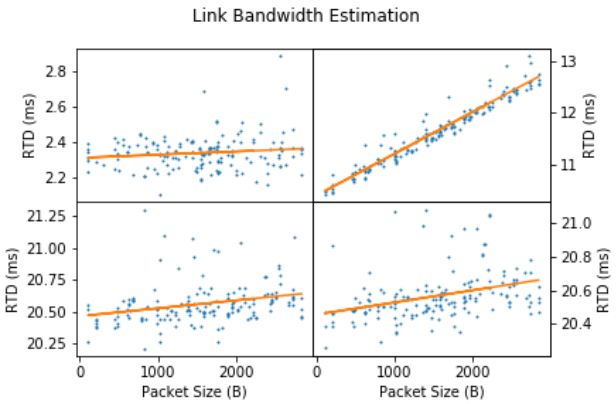


Fig. 10: Transmission speed estimation from passive measurement of RTD. Each data point corresponds to the RTD measurement (y-axis) of a pair of packets of a certain total size (x-axis); the orange lines plot the above formula using the estimated transmission speed. Clockwise from top-left: Link (1), Link (2), Link (3), and Link (4).

Link (4)[7]. The small inaccuracy of these estimations is due to the imperfection of the emulator (which adds small processing delay to emulated packets) and the measurement tools. These imperfections can be seen in Figure 10 where they manifest as small stationary noise added to all packets, which causes a constant drift to the measured delay (captured as an intercept by the linear regression algorithm) and as deviations around the regression line. The estimation accuracy can be made arbitrarily better, provided enough measurements are collected. In general, higher bandwidths cause lower transmission delays, which require more measurements to be distinguished from added noise and captured by the linear regression algorithm.

C. End-to-end Bottleneck Capacity

Another known method to estimate network bandwidth is *packet pair* [21]. It consists in sending pairs of packets back-to-back while timestamping them both at the source and at the destination, which are generally end-user machines and/or servers, and measuring their spacing difference. Intuitively, two packets sent back-to-back will get spaced along the path each time they cross a link of lower bandwidth. As such, the difference in their timestamps at the destination will be a function of their sizes and of the transmission speed of the slowest link, i.e. the bottleneck capacity of the path. This method has been extensively studied in the scientific literature. In this paper, however, we only implement it in a *passive measurement* framework using only the tools we have proposed.

Consider two packets P and Q sent from one host A at instants t_P^A and t_Q^A respectively, and received by a host B at instants t_P^B and t_Q^B respectively, through a path with n links of bandwidths $B_1, B_2, ..., B_n$. If all the links of the path are fast enough, the packets will not be further spaced by transmission delay, i.e. $t_Q^B - t_P^B \approx t_Q^A - t_P^A$. However, each slow link i will try to impose its transmission delay on the packet spacing, and we would have $t_Q^B - t_P^B \geq \frac{S_Q}{B_i}$ where S_Q is the size of packet Q. In fact, the authors in [21] argue that if the packets are sent with small enough interpacket time, then their spacing at the destination will be equal to the transmission delay of the second packet on the slowest link, according to the formula:

$$t_Q^B - t_P^B = \max(t_Q^A - t_P^A, \frac{S_Q}{B_l}),$$

where l is the bottleneck link of the path.

In the same scenario as above, we leverage the timestamps collected at the end hosts to apply this method. For each pair of packets sent *and* received successively, we compute their spacing $t_Q^A - t_P^A$ at the source and $t_Q^B - t_P^B$ at the destination, and use it to estimate the bottleneck bandwidth according to the previous formula. Then from the estimations gotten from each pair of packet we select the one with the maximum likelihood. Figure 11 shows the results, from which we obtain an estimated bottleneck bandwidth between 9.997 and 10.013 Mbps.

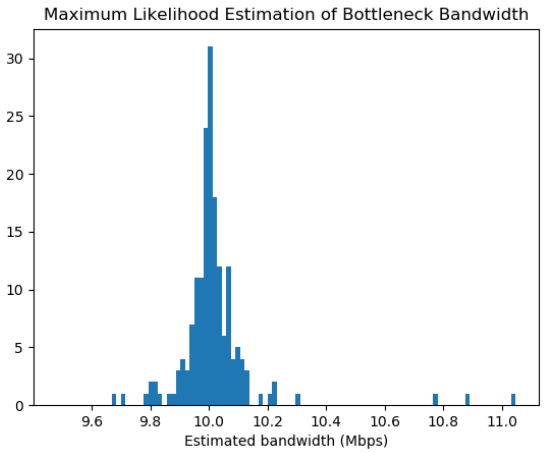


Fig. 11: Estimated bandwidth from different pairs of packets.

VIII. Conclusion

Fine-grained fidelity monitoring is essential for reinforcing realism in network emulation. It relies on the accurate measurement of the emulated packet delay, which in distributed scenarios is limited by clock offsets of the machines within the cluster. We have presented in this paper a new methodology for passively measuring delay of packets exchanged between physical machines and/or virtual machines hosted by separate physical hosts. We have implemented this methodology within a delay monitoring system that relies on the extended Berkeley Packet Filter's (*eBPF*) network and packet processing capabilities to extract information and timestamps from packets in an accurate, precise, and low-overhead manner, and which naturally integrates alongside

[7]Datasets and a Python notebook to reproduce these results can be found at `https://github.com/distrinet-hifi/delaymon/`.

existing network emulation tools. This system allows the passive measurement of packets' one-way delays when assumptions about time synchronization can be made, and their two-way delays otherwise. In both cases, it can reach microsecond-levels of accuracy and precision, which are necessary in our goal of monitoring data packets for fidelity purposes in distributed emulation scenarios.

Our current and future work is centered around the design of a lightweight fidelity monitoring system that uses the presented delay measurement methodology in large-scale emulated networks in distributed testbeds, to ensure that emulated experiments are carried out accurately. We will also import tools from statistics and signal processing to eliminate noise from passive delay measurements, in order to drop further assumptions about time synchronization.

REFERENCES

[1] Lantz, B., et al. (2010, October). A network in a Laptop: Rapid Prototyping for Software-defined Networks. In Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, 1-6.

[2] Di Lena, G., et al. (2021). Distrinet: a Mininet Implementation for the Cloud. ACM Computer Communication Review.

[3] P. Wette, et al. (2014). MaxiNet: Distributed emulation of software-defined networks. 2014 IFIP Networking Conference, 2014, pp. 1-9.

[4] The Mininet Project. Cluster Edition Prototype. `https://github.com/mininet/mininet/wiki/Cluster-Edition-Prototype`. Accessed: 2021-10-14.

[5] Ortiz, J., et al. (2016, October). Evaluation of Performance and Scalability of Mininet in Scenarios with Large Data Centers. In 2016 IEEE Ecuador Technical Chapters Meeting (ETCM) (pp. 1-6). IEEE.

[6] Muelas, D., et al. (2018). Assessing the Limits of Mininet-Based Environments for Network Experimentation. IEEE Network, 32(6), 168-176.

[7] Heller, B. (2013). Reproducible Network Research with High-fidelity Emulation (Doctoral dissertation, Stanford University).

[8] Vieira, M. A., et al. (2020). Fast Packet Processing With eBPF and XDP: Concepts, Code, Challenges, and Applications. ACM Computing Surveys (CSUR), 53(1), 1-36.

[9] Almes, G., et al. (1999, September). A One-way Delay Metric for IPPM. RFC 2679.

[10] De Vito, L., et al. (2008). One-way Delay Measurement: State of the Art. IEEE TIM, 57(12), 2742-2750.

[11] Almes, G., et al. (1999, September). A Round-trip Delay Metric for IPPM. RFC 2681.

[12] Postel, J. (1981). Internet Control Message Protocol DARPA Internet Program Protocol Specification. RFC 792.

[13] Muss, M. The Story of the PING Program. `https://ftp.arl.army.mil/~mike/ping.html`

[14] Spirent Communications. Remote Distributed Testing. `http://www.spirentcom.com/documents/185.pdf`

[15] Schmid, T., et al. (2009). Temperature Compensated Time Synchronization. IEEE Embedded Systems Letters, 1(2), 37-41.

[16] Mills, D., et al. (2010). Network Time Protocol Version 4: Protocol and Algorithms Specification.

[17] Eidson, J. C., et al. (2002, December). IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. In Proc. of the 34th PTTI meeting, 243-254.

[18] Almesberger, W. (1999). Linux Network traffic control—Implementation Overview.

[19] Libri, A., et al. (2016, July). Evaluation of Synchronization Protocols for Fine-grain HPC Sensor Data Time-stamping and Collection. In Proc. of HPCS, 818-825.

[20] Downey, A. B. (1999). Using pathchar to estimate Internet link characteristics. SIGCOMM Comput. Commun. Rev. 29, 4 (Oct. 1999), 241–250.

[21] C. Dovrolis, et al. What do packet dispersion techniques measure? Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213), 2001, pp. 905-914 vol.2.