Passive Delay Measurement for Fidelity Monitoring of Distributed Network Emulation

Houssam Elbouanani, Walid Dabbous, Chadi Barakat, Thierry Turletti

Passive Delay Measurement for Fidelity Monitoring of Distributed Network Emulation

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

***Abstract*— Emulation has become an excellent 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 its results. However, as the network components are only virtual, it 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 single-machine settings (e.g. with Mininet), in scenarios where the emulation is distributed over multiple physical machines (e.g. Distrinet) any monitoring is limited by their asynchrony. In this paper we tackle the case of packet delay monitoring, to which we propose a methodology for passively measuring delays with or without underlying assumptions about time synchronization. We implement and evaluate our proposed methodology in an open testbed and show that it can reach results within few microseconds of perfect accuracy.**

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. The ideal approach would be to run the study on a real *testbed*, i.e. one that is perfectly similar to the intended environment, in order for the experimentation to be as realistic as possible. However, this is neither easily reproducible nor scalable to scenarios with large size network topologies such as data center and ISP networks, or the whole Internet. It is also possible to use network simulators (e.g. ns2 and ns3 [1]), which abstract away enough irrelevant mechanisms by implementing a simplified model of the network as a system where state evolution is a sequence of discrete events, at the cost of less accuracy and realism.

Enter network emulators, such as Mininet [2], which go so far as to mimic the operation of network hardware (switches, routers, links, etc.) in real-time and run actual code –applications and/or operating systems– using software tools. The goal of which is to find a compromise between testbed implementation and network simulation in order to offer good accuracy and realism while allowing the study to be as reproducible and scalable as possible.

However, network emulators do not always provide perfectly accurate results. As they are designed for running on everyday laptops, their emulation of multiple events (running code in emulated hosts, switching and routing multiple packets in parallel, etc.) is very limited by the available resources [3]. This renders them practically unusable for emulating latency-sensitive scenarios or those that require packet-level precision. Researchers have thus proposed *fidelity monitoring* [4] 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. Other metrics can be monitored (e.g. the bandwidth, the queues' sizes, etc.) but 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 queue size.

Emulators do not scale well either. In computing-intensive scenarios, Mininet cannot emulate more than a certain number of hosts and network hardware devices due to resource limitations inherent to the physical machines intended to run it. Several researchers have thus worked on distributed versions of Mininet, ones that let users emulate large-scale networks over a geographically localized cluster of multiple physical machines. Distrinet [5] is one such iteration that particularly focuses on reproducibility by natively allowing users to run their emulations on public clouds such as Amazon's AWS.

While each of the aforementioned solutions can offer either more reproducibility and scalability, or more accuracy, combining the two is a very complicated task. In fact, packet delay monitoring requires measuring packets' delays between multiple nodes of the virtual network, but in a distributed setting, 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 in physical networks*. 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 mainly twofold: we present a methodology to passively measure the one-

way delay 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 also present an extension of our methodology to the monitoring of the round-trip delay in scenarios when accurately measuring the one-way delay is not possible. The methodology will be evaluated on a real testbed 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 sychronization. We then move on to present our experimental setup in Section III. In Sections IV and V we introduce methods to passively measure one-way and two-way delays respectively with high accuracy. We finally conclude and discuss our current and future work on high-fidelity distributed network emulation in Section VI.

II. Background

A. Delay Measurement

Unlike the throughput which is a flow-level measure, the network delay characterizes either an individual packet, or a pair of request-response packets. Researchers have therefore identified two types of network delay: the one-way delay (OWD) defined in [6] and the round-trip delay (RTD) in [7].

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. Three quantities contribute to the OWD:

- the equipment delay: which mainly consists of the amount of (absolute) time that the packet will spend in the queue waiting to be transmitted;
- the transmission delay: that is the amount of (absolute) time needed for the transmitting hardware (NIC, router interface, switch port, etc.) to write the packet on 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: i.e. the length of (absolute) 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 medium and does not depend on the size of the packet.

Accurately measuring packets' OWDs and successfully decomposing them into their three components can give useful information about the network: from the transmission delays of multiple packets the bandwidth of the medium can be inferred; a long equipment delay can signify congestion or saturation of computing resources; and a longer than anticipated end-to-end OWD can be evidence of poor routing policies. 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.) [8].

An easier value to measure is the round-trip time. The RFC 2681 [7] 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 equal to the sum of the individual OWDs of packets P and Q, and the processing delay between the reception of the request packet by B and the sending of the response packet Q. Certainly, the information on the individual OWDs is lost when measuring the RTD, especially when the two ends are multiple hops away and therefore when the paths in the two directions cannot be safely assumed to be symmetric.

The use of ICMP echo probes is the de facto active method for measuring RTDs [9], [10]. 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 RTD 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) [11]. The problem particularly arises because the time dissimilarity between the clocks of different machines (called the 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 [12]. 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 one for clock synchronization [13]. It organizes machines into a tree-like hierarchy, where the root node is the primary source 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. And at the convergence of the algorithm,

[1] hping3: https://linux.die.net/man/8/hping3

[2] tcpping: http://www.vdberg.org/ richard/tcpping.html

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 [14]. 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 the nodes. But unlike NTP which can be implemented by 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 [15], 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 this work and in settings where time synchronization is needed, we will use their findings to configure our testbed.

III. Experimental Setup

A. Testbed

All of the following testing will be performed on the open testbed R2lab [16]. It is a cluster of machines that are connected through Gigabit Ethernet wires to a store-and-forward switch. In our tests, they will be running a Ubuntu 18.04 linux distribution over a 4.15 kernel. Furthermore, we deactivate the machines' processors' C-states which puts them in low power consumption modes. As we will use multiple active delay measurement tools to compare and evaluate our method against, a constant processing frequency and the full-power mode let them receive and timestamp packets as soon as they arrive to the NIC, and makes their reported measurements much more accurate, and thus a good reference point for our evaluations.

Ping, hping3, and tcpping will be used as active RTD measurement tools. These tools allow for controlled probe sizes and sending rates, and will thus also serve as packet generators. We will also use netsniff-ng's trafgen[3] and Scapy[4] to customize packet generation when it is necessary. As for timestamping, we will use the simple tcpdump utility that captures and timestamps incoming or outgoing packets. The collected data is later analysed by a python program. We provide scripts[5] to reproduce all our results.

B. Packet Identification

Identifying packets is necessary for passive delay measurement. For both OWD and RTD measurement, 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, more sophisticated mechanisms have to be implemented. The second easiest way 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 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. 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 can arrive unordered or one of them can be lost and not received by the destination. We must therefore add another assumption, one that allows to distinguish between packets with the same ID in all cases and under all circumstances: we therefore assume that no two packets with the same ID can be received from the same source in the same time interval of certain length T. The choice of T must take into account the sending rate and the typical values of the delay in the network. More precisely, 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 bandwidth). Even in our testbed with high bandwidth-delay product and small packets, this condition holds as $2^{16}\tau = 30ms$ and $\Delta < 100\mu s$.

IV. Passive OWD measurement

In this section we study the extent to which it is possible to passively measure the OWD, i.e. the delay of data packets exchanged between the pair of physical machines from the previously described testbed.

The idea is simply to capture and timestamp using tcpdump all sent packets just before they enter the sending machine's NIC, and all received packets immediately as they leave the receiving machine's NIC. Then from those packet dumps a collector creates two tables of (packet ID,

[3]netsniff-ng: http://netsniff-ng.org/

[4]Scapy: https://scapy.net/

[5]See https://github.com/helllb/delay

timestamp) pairs that we match to compute the OWDs as the timestamp differences. The larger the number of captured packets, the likelier collisions become. However, in our setting and from the assumption we made earlier, such ID collisions are easily mitigated by using packet timestamps to avoid mismatches eventually caused by packet unordering and loss. The method is described in algorithm 1 below.

Data: Packet dumps from A and B: in_A, in_B, out_A, out_B
Result: Array of (packet_ID, owd) pairs
initialize array OWD;
foreach *(packet_ID, timestamp_A) in out_A* **do**
  lookup matching packet_ID in in_B with the closest timestamp_B;
  compute owd := timestamp_B - timestamp_A;
  add (packet_ID, owd) to OWD;
end
foreach *(packet_ID, timestamp_B) in out_B* **do**
  lookup matching packet_ID in in_A with the closest timestamp_A;
  compute owd := timestamp_A - timestamp_B;
  add (packet_ID, owd) to OWD;
end

Algorithm 1: Passive OWD measurement algorithm

From the definition given in Section II, the OWD of a packet P between the two machines is the sum of the equipment and transmission delays introduced by A's NIC and the switch's egreess port, and the propagation delay on the two Ethernet cables connecting the two machines to the switch. By keeping the packet sending rate below the links' capacity and the NIC and switch's port transmission speed, we can eliminate packet queuing altogether. Thus, in this particular setting of perfect symmetry, i.e. two machines with the same hardware and operating system configuration connected through symmetric links to equal switch ports, the delay can be safely assumed to be equal in both directions. Thus, for a pair of request-response packets P and Q of equal size (e.g. a Ping echo), we have:

$$RTD(P,Q) = OWD(P) + OWD(Q) = 2 \cdot OWD(P)$$

We can thus use half of the round-trip time (RTT) reported by Ping as a ground truth, apply our method to passively measure the ping echo packets' OWD, and compare the two to evaluate the accuracy of our passive method.

However, without proper time synchronization, it is practically impossible to accurately measure the OWD between two physical machines of different clocks. Consider for example the plots in Figure 1. We compare the OWDs of ICMP echo request packets measured by our method with no clock synchronization (bottom), together with the halves of the RTTs as given by Ping (top), for a number of ICMP echo packets sent with 1 ms interval. We can clearly see how the two machines' clocks drift over time, and how this drift is difficult to predict as it changes from time to time. In effect, when there is variable clock skew, the clock offset changes non-linearly (or in this case piecewise-linearly). In fact, the clock offset depends on uncontrollable physical phenomena (e.g. hardware heating) that make it hard to predict. And even if inferring the slope of the linear term is feasible, simply delinearizing the results is not sustainable and cannot work in general cases. Also note how the clocks largely drift relative to the standard deviation of the Ping RTTs, making the noise caused by the clock offset hide all the information from the actual network delay.

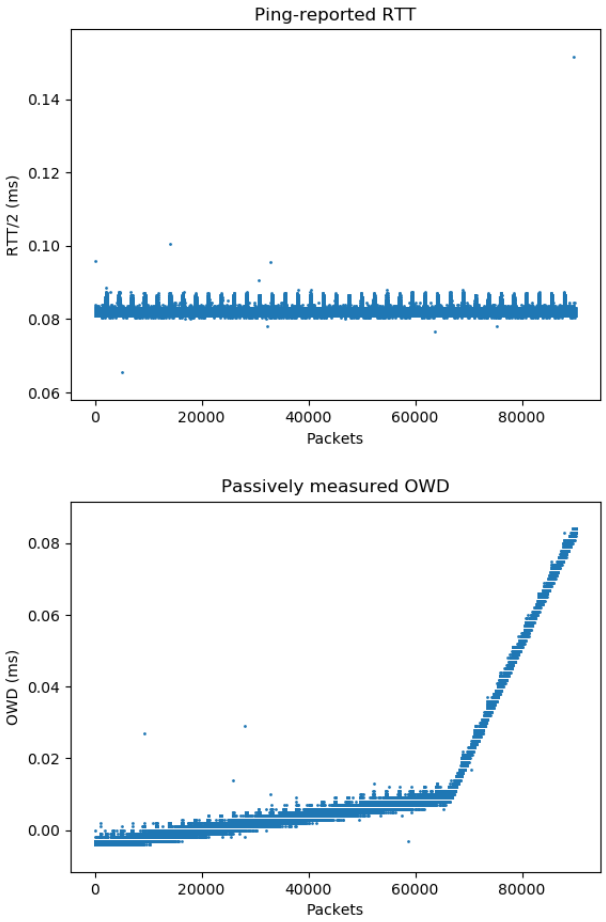


Fig. 1. Ping RTT (top) and estimated OWD (bottom).

Nevertheless, running NTP on the testbed almost perfectly solves the problem. At the convergence of NTP algorithm on clock synchronization and frequency stabilization, the clock offset and skew are neutralized and our method start reporting good results. We can see this in Figure 2, where we report on the results of our method after NTP has stabilized. We can notice how at convergence of NTP, the average measured OWD is only $2\mu s$ away from half the average reported RTT, and its standard variation is only $53ns$ larger. Also note how the pattern of variation is the same. This therefore confirms how our method can be as accurate as Ping (which does not rely on time synchronization).

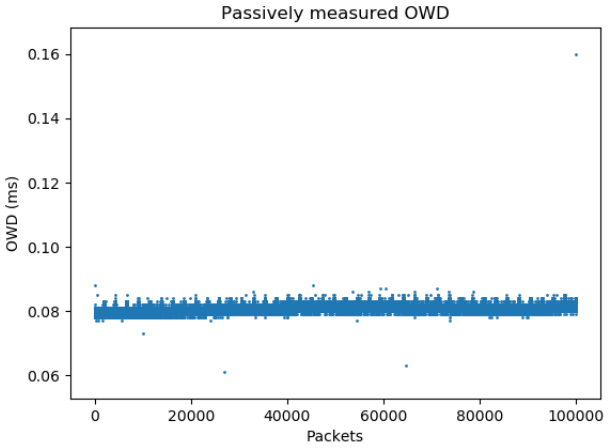
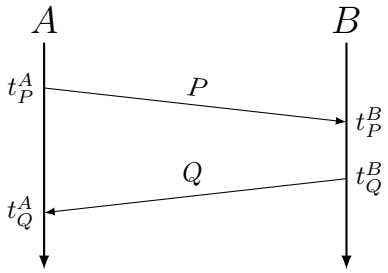


Fig. 2. Estimated OWD under NTP.

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 will propose a new method to passively measure the RTD that does not require such strong assumptions.

For the RTD to make sense in the case of passive measurement, we will extend its definition to almost any pair of packets, not just request-response pairs. For a pair of packets P and Q such that P was sent before Q was received, we define the RTD as simply the sum of their individual OWDs $t_P^B - t_P^A$ 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 as it is not relevant in the general case where P and Q are not necessarily related packets (unlike ICMP echo request-response, TCP SYN-ACK, etc.).



The method for passively measuring the RTD and OWD use similar settings: a program that captures and timestamps packets between the NIC and the upper layers is installed on the machines (tcpdump plays this role in our testbed), then the packets 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. However, in the case of the RTD, for each pair of machines A and B, and for each packet P sent from A at time t_P^A (in A's clock) and received at time t_P^B (in B's clock), and Q sent at time t_Q^B (in B's clock) and received 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. While it does eliminate any inaccuracies due to constant clock drift between the 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 that will induce. Thus, in practice, the collector should only stick to pairs of packets sent and received within a small enough time interval τ where the error caused by clock drifts is guaranteed to be 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 can 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 manageable.

To evaluate this passive RTD measurement method, we conduct the same experiments as earlier, where we passively measure the delays of generated packets and compare the results to what is reported. Figure 3 shows the RTDs measured by our method with no need for time synchronization. We reach nearly as much accuracy as Ping on average, around which our method varies even less than Ping's reported RTTs.

Table V compares the RTDs measured with our method to the ones reported by delay measurement tools, of 10000 packets of the same size sent with 1 ms interval. The "avg" column shows the average measured/reported RTD and "std" their standard deviations, both in miliseconds. Note that trafgen does not report any information about the delays (as it is only a traffic generator that does not natively act as active delay measurement). We can see how hping3 reports values far outside the range of typical values reported by the other measurement tools. And while tcpping shows more realistic measurement values, they are nonetheless very different from the classical Ping. In all cases, our passive measurement method gives very stable results (microsecond-level precision) across all tools, which is the expected behaviour as all the packets behave the same way from a network perspective, independently of the tool or protocol that generates them.

Furthermore, our method is precise enough that the transmission term (which corresponds to 100 ns/B in our high-bandwidth testbed) can be easily extracted from the measured delay, and then be used to infer information about the bandwidth of the underlying transmission media. To show

		reported		passively measured	
		avg (ms)	std (ms)	avg (ms)	std (ms)
hping	ICMP	7.5	0.9	0.160	0.001
	TCP	7.5	1.6	0.160	0.002
	UDP	4.9	2.2	0.162	0.003
ping		0.164	0.002	0.160	0.001
tcpping		0.193	0.028	0.165	0.015
trafgen				0.161	0.004

TABLE I

COMPARISON OF MULTIPLE DELAY MEASUREMENT TOOLS.

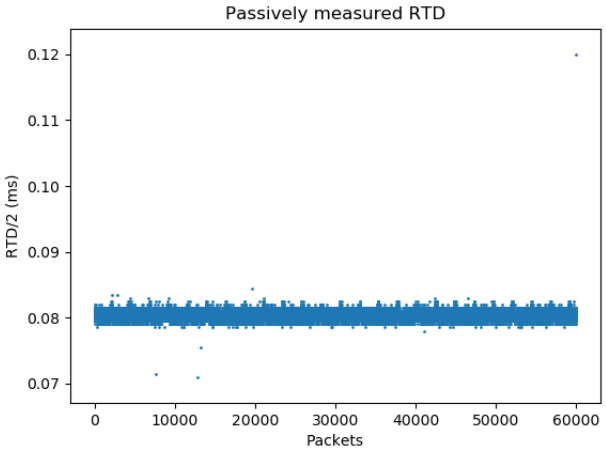


Fig. 3. Estimated RTD.

this, we generate with 1 ms interval using trafgen 2500 TCP probe packets of random sizes and measure the RTDs of each request-response pair $\widehat{RTD}(P,Q)$, that we plot against the sum of their sizes $s = |P| + |Q|$ (blue data points). We also plot the average RTD and a confidence interval for each joint size (orange curve). The result is figure 4, which shows how the passively measured RTD is linear in the packet size due to the transmission term, with a high coefficient of determination ($R^2 = 99.6\%$).

VI. CONCLUSION

Fine-grained fidelity monitoring is essential for providing network emulation with more realism. This relies on the accurate measurement of the emulated packets' delays, which in distributed scenarios is limited by clock offsets of the machines within the cluster. We have presented in this paper a methodology for passively measuring delay of packets exchanged between physical machines and/or virtual machines hosted by separate physical hosts. It allows the passive measurement of packets' one-way delays when assumptions about time sychronization can be made, and their two-way delays otherwise. In both cases, this can reach microsecond-levels of accuracy, which are necessary in our goal of monitoring data packets for fidelity purposes in distributed emulation scenarios.

Our current and future works are centered around the design of a lightweight fidelity monitoring system for local and distributed network emulation, that implements the presented methodology with good precision and low overhead.

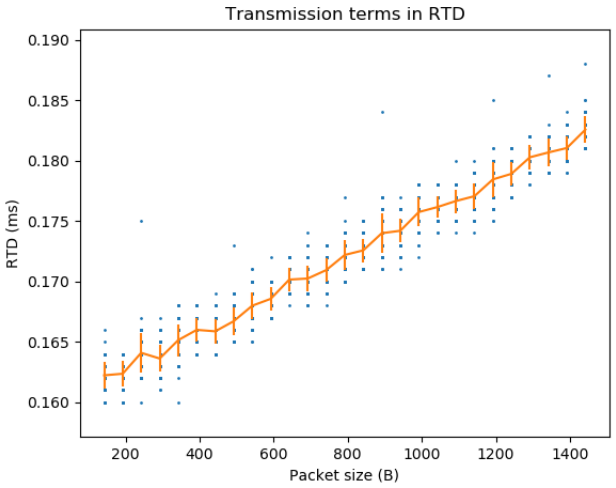


Fig. 4. Estimated RTD vs packet size.

In particular, we will present a replacement for tcpdump that is less intrusive, more precise, and more powerful (giving information about packets and network metrics other than just timestamps). Future works will also import tools from statistics and signal processing to eliminate noise from delay measurements, in order to drop some more assumptions about time synchronization.

REFERENCES

[1] Henderson, T. R., et al. (2008). Network Simulations with the ns-3 Simulator. SIGCOMM demonstration, 14(14), 527.

[2] 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 (pp. 1-6).

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

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

[5] Di Lena, G., et al. (2019). Distributed Network Experiment Emulation.

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

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

[8] De Vito, L., et al. (2008). One-way Delay Measurement: State of the Art. IEEE Transactions on Instrumentation and Measurement, 57(12), 2742-2750.

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

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

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

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

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

[14] Eidson, J. C., et al. (2002, December). IEEE-1588™ Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. In Proceedings of the 34th Annual Precise Time and Time Interval Systems and Applications Meeting (pp. 243-254).

[15] Libri, A., et al. (2016, July). Evaluation of Synchronization Protocols for Fine-grain HPC Sensor Data Time-stamping and Collection. In 2016 International Conference on High Performance Computing & Simulation (HPCS) (pp. 818-825).

[16] R2lab Anechoic Chamber. https://r2lab.inria.fr/