Troubleshooting Distributed Network Emulation

Houssam ElBouanani, Chadi Barakat, Walid Dabbous, and Thierry Turletti
Inria Sophia Antipolis-Méditerrannée
Université Côte d'Azur

2023 26th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN) | 979-8-3503-9804-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICIN56760.2023.10073480

***Abstract*—Distributed network emulators allow users to perform network evaluation by running large-scale virtual networks over a cluster of fewer machines. While they offer accessible testing environments for researchers to evaluate their contributions and for the community to reproduce its results, their use of limited physical network and compute resources can silently and negatively impact the emulation results. In this paper, we present a methodology that uses linear optimization to extract information about the physical infrastructure from emulation-level packet delay measurements, in order to pinpoint the root causes of emulation inaccuracy with minimal hypotheses. We evaluate the precision of our methodology using numerical simulations, then show how its implementation performs in a real network scenario.**

***Index Terms*—network emulation, passive delay measurement, network tomography**

I. Introduction

Distributed network emulation is a relatively recent approach for network experimentation that uses containers and virtual switches to emulate the behaviour of real, physical networks for research and/or education purposes. In particular, it allows users to run a large-scale virtual network over a cluster of fewer machines, each running a subset of the virtual machines, and which are connected using overlay networking technologies. Mininet [8] and its forks (Mininet CE [1], Maxinet [14], and Distrinet [3]) implement this network emulation approach with a focus on accessibility and flexibility by providing users with a simple-to-use Python API.

A direct challenge of network emulation is resource contention: running multiple virtual components over fewer machines leads to concurrency in using the available physical resources. This aspect of network emulation has been extensively researched in previous studies, e.g., [9], [11]. In its extreme cases, this may lead to emulated packets getting scheduled for transmission later than normal when the packet rates exceed the speed of the hardware CPUs, or to virtual links getting throughputs lower than normal when the total packet rates exceed the capacity of the underlying network. In general, the underlying physical infrastructure adds delay to emulated packets which might lower emulation quality, in a way that tends to be silent and can bias the results unless a careful analysis is carried out.

To measure this delay, the authors in [4] have implemented a methodology that passively monitors the network delay of emulated packets to gauge its increase by the physical infrastructure, ultimately in order to detect eventual occurrences of contention failures. In this paper we use this passive delay measurement tool to infer information about the underlying infrastructure. Such information can be useful for troubleshooting purposes, i.e., to identify which resource –particularly network links– has not had enough available capacities to correctly host the emulated network, and have thus contributed to stretching the network delay of emulated packets. In particular, we propose a network tomography [2], [6] algorithm[1] to infer the network delay of the infrastructure components from the delay measurements passively collected by Mininet or its distributed variants. Indeed, distributed network emulators are mainly designed to run on shared infrastructures (grids and clouds), which can be virtual, and whose topology the user might know but cannot directly access or measure. These main assumptions define the crux of the problem. Using carefully tailored heuristics, our algorithm performs relatively well even in settings where the user's emulation scenario does not provide enough measurements to infer infrastructure delay.

Our troubleshooting methodology is heavily inspired from previous works on delay tomography, whose objective is inferring delays of internal links from end-to-end delay measurements. This problem has been formulated in [7], [10] and while it was historically solved using active measurement-based methods, more recent attempts have instead focused on internal delay inference from passive measurements. In [5] for instance, the authors have focused on monitoring optimisation: solving for the minimum set of vantage end-points in a network from which a statistically accurate estimation of the internal links' delay distributions can be achieved. A later study [12] assumed the impossibility of completely inferring the internal (physical) delays of a network infrastructure from the measurements collected at overlay virtual networks, and proposed to train a neural network from simulated traffic to fill in the missing information. This paper deals with similar assumptions in a context of network emulation, where emulated traffic can be very diverse and too short-lived to be learned by a model. Instead we propose optimisation heuristics to solve it.

The remainder of this paper is organised as follows: the next section presents the problem in more details. We particularly argue our choice of hypotheses and present our modeling framework. Section III describes our delay tomography algorithm along its implementation using existing tools, which

[1]The source code of the algorithm's implementation, as well as instructions to reproduce all the results in this paper are available at https://github.com/distrinet-hifi/tshoot.

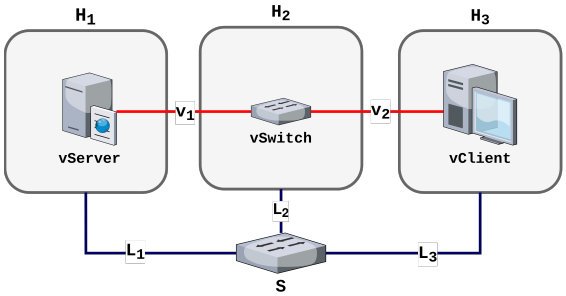


Fig. 1: Emulated and infrastructure topologies.

we then evaluate in Section IV. Finally, Section V concludes the paper with a summary and a discussion of possible future work in this direction.

II. Problem Statement

The objective in this paper is troubleshooting emulation failures by inferring physical infrastructure load from network measurements collected at the virtual level. The main idea is to determine if there is any unexpected load in the virtual network, evidenced by higher packet delays, and identify which elements of the underlying infrastructure are responsible for it. As any underlying hardware (links and/or machines) can add undesirable network delay to packets proportional to its load, we can extract load information from the passively measured delay of emulated packets. In this section, we describe the problem in more details by presenting our working hypotheses, our mathematical modeling, and by discussing raised challenges.

A. Hypotheses

Consider for example the simple scenario in Figure 1: a virtual network (consisting of a virtual client and a virtual server connected to a virtual switch) is emulated on top of a physical network of three hosts H_1, H_2, and H_3 connected by a switch S. The virtual server sends a flow of packets to the virtual client. Using traffic control tools, the virtual links v_1 and v_2 are configured by the user to shape the traffic according to the scenario they wish to emulate: limiting link bandwidth, adding propagation delay, introducing packet loss, etc. Given these traffic shaping parameters, each packet P should experience a certain *normal* delay $d(P)$ depending on its size, its position in the virtual links' queues, etc. As this packet moves over the virtual network, the links L_1, L_2, and L_3 of the physical network that are crossed by the packet will also add a certain *error* delay $\epsilon(P)$ depending on the packet itself and on the current load of the infrastructure. When this error delay exceeds some tolerance value, it will negatively impact the results of the emulation. Unfortunately, the user does not have full control over the physical infrastructure to monitor the delay in all network nodes and links. However, using the measurement tool designed by the authors in [4], the user can monitor their own emulated network delays, and can easily get information about error delays ϵ and use them to infer infrastructure delays.

In our example, a packet P crossing the virtual link v_1 will experience a total measurable delay

$$\hat{d}(P) = d(P) + d_1(P) + d_2(P),$$

where $d(P)$ is the normal emulation delay[2], $d_i(P)$ is the error delay introduced by physical link L_i to the packet P. Likewise, a packet Q crossing the virtual link v_2 will experience a delay

$$\hat{d}(Q) = d(Q) + d_2(Q) + d_3(Q).$$

The total error delays $\epsilon(P), \epsilon(Q)$ experienced by packets P and Q respectively can be written as:

$$\epsilon(P) = d_1(P) + d_2(P) \text{ and } \epsilon(Q) = d_2(Q) + d_3(Q).$$

It follows that information about the delays experienced by the packet on each underlying infrastructure link is embedded in the measured delay of packets in the virtual network. However, it is impossible to extract that information by analysing each packet individually. Instead, we can resort to a statistical approach that analyses infrastructure link delays d_i on finite time intervals, and that examines a large number of packets from different emulated links (i.e., that pass over different infrastructure paths). Given some prior information on the mapping of the virtual network onto the infrastructure, statistics on the link delays of the infrastructure can thus be inferred. In our scenario for example, if we define $x_i(T)$ as the average delay on link L_i during a certain time interval $T \in \mathcal{T}$, and $\bar{\epsilon}_j(T)$ as the mean delay error of all sampled packets during T, we have:

$$\begin{cases} x_1(T) + x_2(T) = \bar{\epsilon}_1(T) \\ x_2(T) + x_3(T) = \bar{\epsilon}_2(T) \end{cases}$$

In the general case, to each physical link L_i corresponds a sequence of variables $(x_i(T))_{T\in\mathcal{T}}$, and to each virtual link[3] v_j corresponds a sequence of mean delay errors $(\bar{\epsilon}_j(T))_{T\in\mathcal{T}}$. According to how virtual links map to the infrastructure network, infrastructure and virtual links can then be related by linear equations of the form:

$$\sum_i a_{i,j}(T) \cdot x_i(T) = \bar{\epsilon}_j(T), \tag{1}$$

where $a_{i,j}(T)$ is a binary value equal to 1 if virtual link v_j crosses physical link L_i and 0 otherwise.

The above set of linear equations can be further rewritten into a more compact form:

$$\mathbf{A}(T) \cdot \mathbf{X}(T) = \mathbf{b}(T), \tag{2}$$

where $\mathbf{A}(T)$ is defined as the *embedding matrix* whose coefficients are $(a_{i,j}(T))$, $\mathbf{X}(T)$ is a vector of variables $(x_i(T))$,

[2]An emulated network can be congested due to a surge in emulated traffic. The delay of its packets $d(P)$ remains normal as long as the physical infrastructure does not interfere with the emulation.

[3]Without loss of generality, virtual links that cross the same path of infrastructure links can be aggregated into a single virtual link. The measurements from these virtual links are combined into one homogeneous set.

and $\mathbf{b}(T)$ is a vector of collected delay errors ($\overline{\epsilon}_j(T)$). For instance, the example networks above are described by the embedding matrix;

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

Our problem then translates into solving the set of equations in (2) under the following three main hypotheses:

- the underlying topology and the embedding of the emulated network are known, but the total load on the different links of the infrastructure is unknown and cannot be directly measured;
- through sampled passive delay measurement of emulated packets, we are given broad information about the added error delays, as well as the timestamps of packets to be able to assign them to time intervals T; *and*
- over time intervals of finite length, packets from different virtual links crossing the same infrastructure link experience more or less the same delay distribution.

The first hypothesis essentially implies that the user knows how the nodes of the infrastructure are connected, but does not know their loads at all time instants, and cannot access them for direct monitoring. This hypothesis is the default scheme in shared infrastructures such as grids and clouds, where static information (topology, component characteristics, etc.) can be provided but the user cannot directly access networking nodes and/or measure dynamic information (load, delay, packet loss) as it is impacted by other users of the infrastructure.

The second hypothesis defines our source of data: the user has complete control of her emulation scenario and can implement a monitoring tool to passively measure the delays of emulated packets. Such tools essentially intercept a subset of the emulated packets (based on a preconfigured sampling rate) and use information available to the emulator (queue lengths, virtual link speed, etc.) to infer normal delays. The last hypothesis is to ensure that different emulated packets experience the same infrastructure network conditions when they pass by the same infrastructure link even if they are from different virtual links. In practice, this holds in all distributed network emulators forked from Mininet, independently of the emulated scenario, as they use typical tunneling protocols, e.g., Generic Routing Encapsulation (GRE) and Virtual Extensible LAN (VXLAN) to create virtual Ethernet links on top of an infrastructure network. Thus, neither differentiated treatment of virtual links nor QoS mechanisms are used.

B. Challenges

1) Time synchronization: Being an explicit measure of time, network delay measurement inevitably requires some degree of time synchronization. Previous works like [4] have discussed these limitations in passive delay measurement, and have demonstrated that in a geographically localised network, it is possible to achieve as few as 100 nanoseconds of clock drift using only regular time synchronization protocols without specialised hardware. In cases where this cannot be achieved, these works propose to measure the joint round-trip delay $d(P,Q)$ of pairs of packets (P,Q) instead of their individual one-way delays $d(P)$ and $d(Q)$. Whether we consider individual one-way delays or joint round-trip delays, our above model does not change: if $\overline{\epsilon}_j$ are measures of mean round-trip delays on virtual links, then x_i will also be measures of round-trip underlay link delays.

2) Time decomposition: In the previous subsection we have stated that infrastructure link delays x_i can be approximated by considering the mean absolute error of emulated packet delays on a certain virtual link at a certain time interval. The quality of such approximation heavily depends on the number of collected packets and the length of the time interval. The former can be improved by collecting packets using a higher sampling rate, but the latter requires a compromise: longer time intervals will contain more values but will challenge the assumption that the physical link delay distribution is stationary.

3) Problem dimension: The set of equations (2) have unique solutions $x_i(T)$ only if there are enough virtual links that cross the diverse set of infrastructure links, i.e., when the embedding matrices have more linearly independent rows than columns. In such cases, a solution can directly be obtained by discarding extra rows (those which are linear combinations of other rows), and inverting the embedding matrix:

$$\mathbf{X}(T) = \mathbf{A}^{-1}(T) \cdot \mathbf{b}(T).$$

However, one must be cautious of potential noise added to the measurements $\mathbf{b}(T)$, which is due to the inevitable lack of precision of any tool used to passively measure the delay. This noise can be large enough to cause negative solutions to the equations, which would correspond to negative values of infrastructure delay. Nonetheless, an invertible matrix can help control such errors: if instead of *precise* measurements $\mathbf{b}(T)$ the user provides approximations $\hat{\mathbf{b}}(T)$, then they can only hope to get an approximate solution $\hat{\mathbf{X}}(T)$ but which can be as close to the *real* solution as necessary, provided the measurements are precise enough. Indeed, it follows from the continuity of the matrix $\mathbf{A}^{-1}(T)$ that:

$$\forall\ \varepsilon > 0,\ \exists\ \delta > 0,\ \|\hat{\mathbf{b}} - \mathbf{b}\| < \delta \Rightarrow \|\hat{\mathbf{X}} - \mathbf{X}\| < \varepsilon\ .$$

In the general case however, we cannot assume to have an easily invertible embedding matrix. In the previous example (Figure 1), the system of equations in (1) transforms into 2 equations (corresponding to 2 virtual links) and 3 variables (corresponding to 3 physical links), or equivalently to a non-invertible matrix, which cannot yield a unique solution. The following section aims at working around all these constraints by solving the problem suboptimally with the minimum possible error.

III. Troubleshooting Algorithm

A. Methodology

Considering all discussed challenges, a resolution methodology necessarily requires controlling measurement imprecision and circumventing underdimensioned matrices. To deal with

the former, we add a vector $\varepsilon(T)$ of artificial variables $\varepsilon_j(T)$ that represent estimation and approximation errors for measurements on virtual links v_j. The system then has the form

$$\mathbf{A}(T) \cdot \mathbf{X}(T) = \mathbf{b}(T) + \varepsilon(T) \ . \tag{3}$$

While this mitigates measurement errors, it adds more unknown variables to an already underdimensioned problem. In practice, measurements tools designed for network emulation are implemented with high precision as an important specification, to thoroughly reduce these errors[4]. This observation can help us control those measurement errors $\varepsilon_j(T)$ by assigning them the smallest possible values in order to have a solvable set of equations.

That being said, our resolution methodology will operate in two steps. First, starting from an incomplete formulation and noisy measurements, we look for the smallest error vector $\varepsilon(T)$ to be accounted for to obtain a solvable system. The output of this step is a set of values for the $\varepsilon_j(T)$ vector that allow the system to be solved. In concrete terms, we first solve the convex optimization problem:

$$\begin{aligned} \text{minimize}_{\mathbf{X},\varepsilon} \quad & \|\varepsilon(T)\|^2 \\ \text{subject to} \quad & \mathbf{A}(T) \cdot \mathbf{X}(T) = \mathbf{b}(T) + \varepsilon(T) \\ & \mathbf{X}(T) \geq 0 \ . \end{aligned} \tag{4}$$

Solving this convex optimization problem yields one solution with values for variables $\varepsilon_j^*(T)$ as well as the variables of interest $x_i(T)$ (i.e., infrastructure link delays). However in this first step we are only interested in the solvability of the system and not in its entire resolution. In the case of Figure 1 for example, we would be dealing with a linear system of equations of dimension two and three unknowns, after measurements are corrected with $\varepsilon^*(T)$ values.

The objective of the second step of our algorithm is to reduce the set of possible solutions, and to select one of them based on a certain heuristic. One way to achieve this is, again taking inspiration from convex optimization, to choose the solution that minimizes an objective function f:

$$\begin{aligned} \text{minimize}_{\mathbf{X}} \quad & f(\mathbf{X}(T)) \\ \text{subject to} \quad & \mathbf{A}(T) \cdot \mathbf{X}(T) = \mathbf{b}(T) + \varepsilon^*(T) \\ & \mathbf{X}(T) \geq 0 \ . \end{aligned} \tag{5}$$

Next, we present two heuristics with incremental complexity and comment on their signification.

1) Heuristic 1: This formulation is based on the observation that the probability that a large number of infrastructure links exhibit high delay is relatively low. Indeed, in a complex physical network, only a small subset of links –generally those with the lowest capacities and/or that transport the most traffic volumes– can be overloaded at the same time. This means that among all solutions, we will select those that describe a situation where the least number of overloaded infrastructure links are the cause of delay emulation errors in the virtual network.

To achieve this, we first need to define a threshold delay value θ, above which an infrastructure link should be considered overloaded. The choice of such a threshold clearly depends on the situation at hand, but in general this should be in the order of few milliseconds[5]. We then define our function f as the number of x_i values that exceed the threshold θ, i.e.,

$$f(x_1, ..., x_n) = \sum_i \mathbb{1}(x_i > \theta) \ .$$

This formulation does not involve a convex function, but it can be rewritten into an equivalent form by adding new binary variables z_i, where $z_i = 1$ if and only if $x_i > \theta$. We can write:

$$f(x_1, ..., x_n) = \sum_i z_i \ .$$

We then add new constraints that link variables z_i and x_i together: $\theta - x_i \leq M \cdot (1 - z_i)$ and $x_i - \theta \leq M \cdot z_i$, where M is a very large constant. The problem is then formulated as:

$$\begin{aligned} \text{minimize}_{\mathbf{X},\mathbf{z}} \quad & \sum_i z_i(T) \\ \text{subject to} \quad & \mathbf{A}(T) \cdot \mathbf{X}(T) = \mathbf{b}(T) + \varepsilon^*(T) \\ & \mathbf{X}(T) \geq 0 \\ & \theta - x_i(T) \leq M(1 - z_i(T)), \quad \forall i \\ & x_i(T) - \theta \leq M z_i(T), \quad \forall i \ . \end{aligned}$$

While this effectively implements the described strategy, its main drawback is its computational difficulty. No algorithm to solve such a linear program in polynomial time exists, and thus the system can be computationally intractable for relatively large networks. An easier and more straightforward variant eliminates the z_i variables and minimizes instead the *total* physical delay:

$$f(x_1, ..., x_n) = \sum_i x_i.$$

This behaves similarly, but not always exactly, to the previous objective function but is continuous and does not involve integer variables:

$$\begin{aligned} \text{minimize}_{\mathbf{X}} \quad & \sum_i x_i(T) \\ \text{subject to} \quad & \mathbf{A}(T) \cdot \mathbf{X}(T) = \mathbf{b}(T) + \varepsilon^*(T) \\ & \mathbf{X}(T) \geq 0 \ . \end{aligned} \tag{6}$$

2) Heuristic 2: The above heuristic reduces the set of solutions by choosing those that have a certain special property, i.e., those that minimize the set of infrastructure links causing the emulation delay anomaly. However, in some cases, this may not be enough to select a good solution. One can find cases (see our evaluation setup in Section IV) where two or

[4]The precision of the measurement tool depends on its design and implementation. In this paper we use the tool from [4] which is guaranteed to achieve a precision of a few hundred nanoseconds.

[5]We know from queuing theory that in practice, an overloaded link with a finite size will result in high loss rate, which translates to infinite delay. Thus the actual value of such threshold should not be of large concern.

more infrastructure links always appear together in the embedding of virtual links, which translates to a clique of variables x_i that either appear together or not at all in all equations. In such cases more information is needed to discriminate between the x_i variables and select a good candidate for a solution. Such information can be accounted for in the form of coefficients $\alpha_i \in \mathbb{R}$ for each link L_i, leading to an objective function of the form:

$$f(x_1, ..., x_n) = \sum_i \alpha_i x_i,$$

such that for any two links L_i and L_j, we have $\alpha_i > \alpha_j$ if link L_j is more likely to cause delay emulation error than link L_i. If direct information about the infrastructure links can be obtained (static characteristics such as type, length, or bandwidth, or dynamic information about the traffic such as load and queue backlog), the values of the α_i coefficients can be chosen to reflect this information. In the case this information is not available (lack of control on the infrastructure by the emulator), one can draw data from the *history* of the links: if a physical link has consistently been the cause of delay emulation error in previous time intervals $S \in \mathcal{T}$ (as concluded by the heuristic itself), then its coefficient $\alpha_i(T)$ at the current time interval T can be lowered to reflect this fact. An example implementation of this observation is by assigning the values $\alpha_i(T)$ as inverse (log-)probabilities of overload of links L_i, estimated from previous time intervals:

$$\alpha_i(T) = -\log\left[\frac{\sum_{S\in\mathcal{T}, S<T} \mathbb{1}\{x_i(S) > \theta\}}{|\{S \in \mathcal{T}, S < T\}|}\right].$$

The following algorithm summarises our methodology for estimating the delay of infrastructure links with either of the two heuristics presented above.

Algorithm 1 Physical delay inference methodology

$\alpha_i \leftarrow 1$
for $t \in T$ **do**
  Solve Convex Program (4) and get values for ε^*
  Solve Linear Program (5)
  Update coefficients α_i
end for

B. Numerical simulations

The objective of this preliminary series of simulations is to evaluate our delay inference methodology and to compare the performances of the proposed heuristics.

We consider for our simulations an abstraction of the physical network shown in Figure 2. This network is designed as a 4-ary fat tree topology typically used in data centers. The end servers are simulated to host a certain number of virtual nodes and links. In particular, a number of virtual links are hosted between randomly selected pairs of end servers. Then, a Markovian network traffic of random rate is simulated asynchronously on each of these virtual links. We timestamp each packet at both ends of the virtual link on which it is simulated, but also on both ends of each physical link it crosses. Finally, the first set of measurements (at the virtual level) is fed to our algorithm to infer the underlying physical delays at specific time intervals (of 10 ms), which are then compared to ground truth on the delay obtained from the second set of measurements (at the infrastructure level). The experiment is designed to last an average of 100 s.

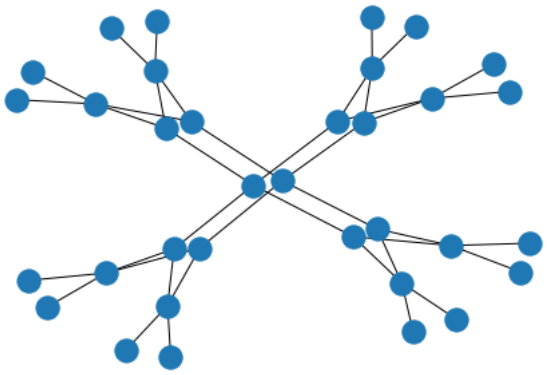


Fig. 2: Graph representation of a 4-ary fat tree topology. The central vertices represent the cores switches, which connect 4 pods of aggregate and edge switches; the end servers are represented by the vertices of degree 1.

The results are shown in Figure 3. Each dot represents the mean measured delay (x-axis) and estimated delay (y-axis) on a certain physical link for a certain time interval. We can see how well our algorithm performs overall, and how Heuristic 2 behaves relatively well (less underestimated values) compared to Heuristic 1. Knowing that the embedding matrices $\mathbf{A}(T)$ in more than 90% of all time intervals had ranks[6] less than half their number of columns, the results are an average error over all links and all time intervals of 2.50 ms (resp. 1.63 ms) between measured and estimated values for Heuristic 1 (resp. Heuristic 2). With the precision of an estimation algorithm defined as the fraction of links that are correctly identified as overloaded (with a tolerance threshold value of 10 ms), our methodology achieves an average precision of 81.5% with Heuristic 1, and 86.2% with Heuristic 2 over all time intervals. The measurements clustered in a vertical line at $(x = 8.10^{-4}s,\ y > x)$ correspond to time intervals when one of the core links had normal delay but was mistakenly designated as faulty by our algorithm. Indeed, our heuristics are designed to minimize the total delay in the infrastructure network (or a weighted sum over all links for Heuristic 2), and thus will tend to assume high delays in links that are part of multiple paths (in this case, core links). Nonetheless, this constitutes less than 10% of all measurements in this scenario.

C. Implementation

In real distributed emulators, our troubleshooting algorithm can be implemented by building on top of the delay mea-

[6]The rank of a matrix or of a linear system is defined as the number of linearly independent rows or equations.

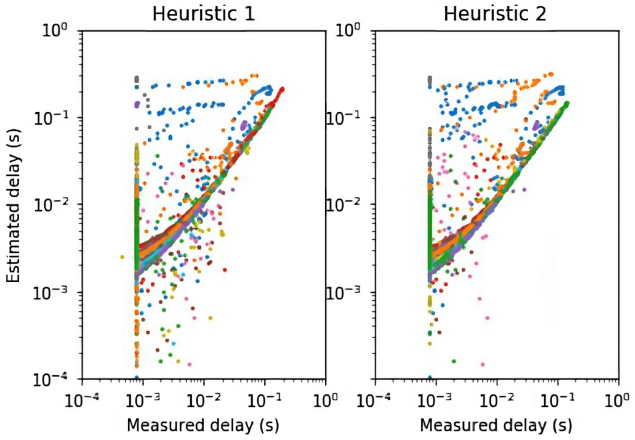


Fig. 3: Scatterplot of measured delays (x-axis) vs. estimated delays (y-axis) using Heuristic 1 (left) and Heuristic 2 (right). Each dot shows the measured and estimated delay of a certain link (colour) during a certain time interval.

surement tool designed in [4]. This tool allows the user to passively monitor the link-level delay of network packets in virtual and/or physical networks, by plugging an Extended Berkeley Packet Filter (eBPF) [13] program into both ends of each link. This lightweight program is a set of instructions added to the traffic control subsystem to log relevant information (packet hash, timestamp, size of queue and head-of-line packet at arrival) about all intercepted packets in persistent files, which are later parsed and analysed offline for delay measurement. However, this tool, when used in virtual links set up by the emulator, measures the link-level delay of emulated packets, which does not directly inform about the underlying infrastructure delay. In the example in Figure 1, by plugging the eBPF code on both ends of virtual link v_1, we can measure the delay $d(\hat{P})$ of every packet P, which is the sum of its emulated delay $d(P)$ and the physical (error) delay $\epsilon(P)$. Thus, we modify the program to also log the emulated delay $d(P)$ in order to evaluate the physical delay.

In short, our algorithm can be implemented as:

- *packet loggers*: eBPF code pluggable into the traffic control (Linux TC) subsystem of emulated links, which intercepts emulated packets and logs in files raw information about their transmission and reception (timestamps, packet hashes, emulated delay, etc.). As explained in [4], eBPF performs this task in a low-overhead manner, as it does not add more than a microsecond of delay to intercepted packets; and
- an *offline analyser* which gathers all logged information to estimate measured infrastructure delays.

As it uses the same design logic as the delay measurement tool presented in [4], the interception and logging of packets does not decrease the performance of the virtual network. The authors show that it only adds sub-microsecond delay to each intercepted packet. However, the information logging can be heavy in terms of storage. Nonetheless, as intercepting each and every packet is not necessary, using a sampling strategy (e.g. random packet sampling with a rate of 10%) reduces this overhead without decreasing the overall performance of our troubleshooting algorithm.

IV. Experimental evaluation

A. Testbed

We consider the network scenario described in Figure 4. In this scenario, 40 clients $(vc_1, vc_2, ..., vc_{40})$ are synchronously downloading a 100 MB file from a random server $(vs_1, vs_2, ..., vs_5)$ located on the same Ethernet segment. All clients are connected to the central switch by links of 100 Mbps bandwidth and 1 ms delay; the servers are connected by links with no traffic control, i.e., without limitation of bandwidth and practically without delay. The experiment is run using Distrinet [3], and is embedded on an private cloud infrastructure composed of four hosting machines connected with two switches as shown in Figure 4. Each machine runs an Ubuntu 18.04 Linux distribution with a 4.15.0 kernel, using a CPU Intel Core i7-2600 processor and 8 GB of RAM. Furthermore, the embedding algorithm used is configured in a way that all emulated file servers and the virtual switch are hosted in the same machine (host H_1); the emulated clients in their turn are distributed fairly over the four other machines.

Throughout the duration of the experiment, we collect passive measurements of emulated packet delays (with a random sampling of 10%), from which we extract the error delays added by the infrastructure network components. We use our methodology described earlier to infer the load on the infrastructure which we interpret and discuss in light of the conclusions made from a more situational analysis.

B. Results

Figure 5 summarizes our results. The figure shows the evolution during the experiment time of the host throughput and the virtual link delay error. We first note how the clients experience different throughput values depending on whether they are hosted in H_1 and H_2 (mean throughput around 90 Mbps and 85 Mbps respectively), or H_3 and H_4 (mean throughput around 42 Mbps) (Figure 5a), even though all clients are designed in the emulated experiment with equal bandwidths of 100 Mbps. This is also evident from the delay emulation errors: virtual links for clients in hosts H_3 and H_4 experience higher delay emulation error (Figure 5b).

These absolute error values, which correspond to the delays added by the underlying infrastructure, are then fed to our estimation algorithm. Given the size of the downloaded file and the achieved throughput, each experiment lasts approximately 20 seconds in total, which are divided into T intervals of 100 ms each. Figure 6 shows the inferred delays for each link of the infrastructure. With a tolerance threshold of 1 ms, we can conclude that links H_1-S_1 and S_1-S_2 are the main causes behind the emulation inaccuracy. However, this conclusion depends on which heuristic is used: while the results of Heuristic 2 show that link S_1-S_2 adds the largest delay throughout the entirety of the experiment, Heuristic 1

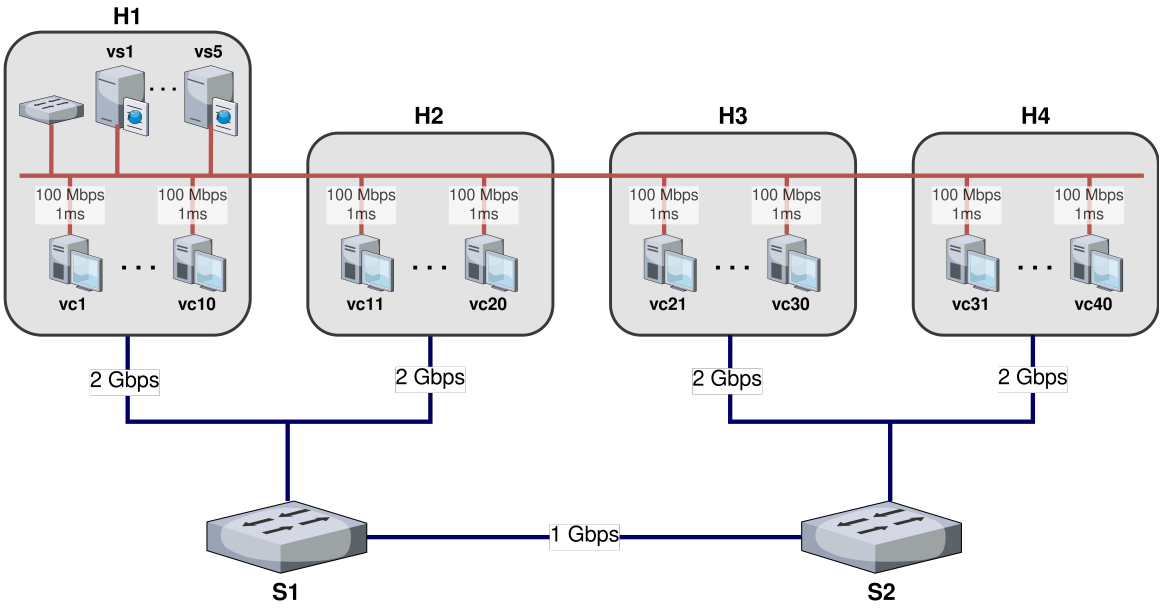
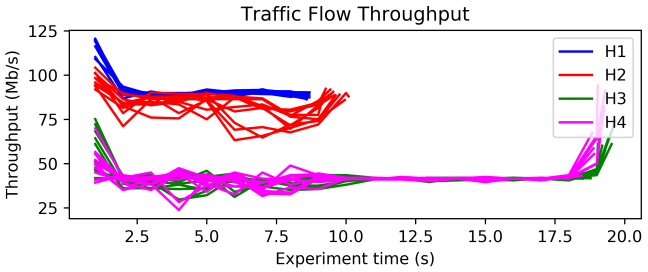
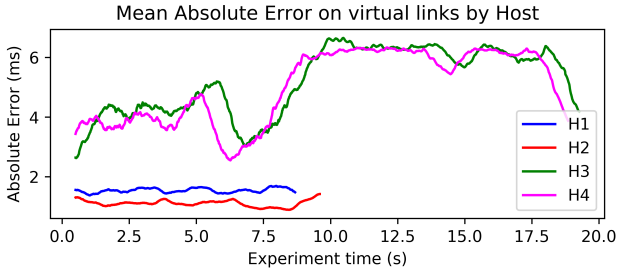


Fig. 4: Emulated network (red) and underlying cluster network.



(a) Application-level throughput achieved by each emulated client: clients hosted in H_1 in blue; clients hosted in H_2 in red; clients hosted in H_3 in green; and clients hosted in H_4 in magenta.



(b) Mean Absolute Errors of virtual links emulated in H_1 (blue), between H_1 and H_2 (red), between H_1 and H_3 (green), and between H_1 and H_4 (magenta).

Fig. 5: High-level (a) and low-level (b) evidence of emulation inaccuracy caused by infrastructure overload.

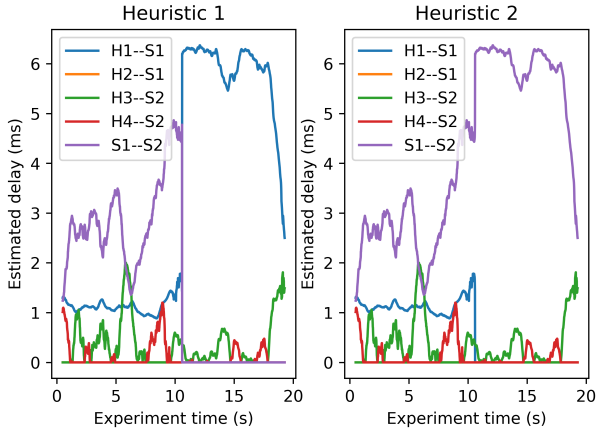


Fig. 6: Network delay of each link of the infrastructure estimated using Heuristic 1 (left) and Heuristic 2 (right).

switches from S_1-S_2 to H_1-S_1 mid-run. We discuss this difference in the next subsection, and explain why Heuristic 2 better reflects the reality in our particular scenario.

C. Discussion

For each time interval, our algorithm solves a linear program modeling the virtual-to-infrastructure embedding and taking into account the measured emulation delay errors. The solution to this linear program consists of an estimation of the infrastructure link delays during the considered time interval. The form of this linear program depends on the set of available measurements. In particular, during the first half of the experiment, as all virtual links are active, the linear

program is written as:

$$\begin{aligned}
&\text{minimize } \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 + \alpha_4 x_4 + \alpha_5 x_5 \\
&\text{subject to} \quad x_1 + x_2 = MAE_1 \\
&\qquad\qquad x_1 + x_3 + x_4 = MAE_2 \\
&\qquad\qquad x_1 + x_3 + x_5 = MAE_3 \ .
\end{aligned}$$

Once clients in H_1 and H_2 finish downloading the file, traffic stops on their links, and so variable x_2 and Equation (2) are no longer part of the program:

$$\text{minimize} \quad \alpha_1 x_1 + \alpha_3 x_3 + \alpha_4 x_4 + \alpha_5 x_5 \tag{7}$$

$$\text{subject to} \quad x_1 + x_3 + x_4 = MAE_2 \tag{8}$$

$$x_1 + x_3 + x_5 = MAE_3 \ . \tag{9}$$

In this new formulation, variables x_1 and x_3 essentially play the same role if $\alpha_1 = \alpha_3$ as with the case of Heuristic 1 (where all coefficients α_i are equal to 1). Thus, any solution (x_1, x_2, x_4, x_5) is valid for Heuristic 1 as long as $x_1 + x_3 = c$ for some constant c. However, using Heuristic 2, coefficients α_i are dynamically chosen based on the solutions in previous time intervals. And as link S_1-S_2 is more often designated faulty at the first half of the experiment, we have $\alpha_3 < \alpha_1$. Under these circumstances, variable x_3 is preferred, which makes S_1-S_2 more likely to be designated faulty in the second half of the experiment as well, consistent with the fact that the link is purposely underdimensioned.

V. Conclusion

Network emulation requires delicate fidelity monitoring to assess the accuracy of obtained results and avoid incorrect conclusions. But once the failure is acknowledged, an important next step is to troubleshoot the potential root causes and identify which parts of the infrastructure could not handle the emulation load. In this paper, we have presented a methodology inspired by past studies on network tomography, that uses passive measurements collected in an overlay emulated network to infer the delay of the underlay infrastructure network. This methodology models the two networks and the embedding of the former over the latter as a linear optimization problem, whose solution tries to capture the information on the delay values in each component of the underlay network. While we have shown that this modeling can yield good results with fair precision, some of its aspects can be further developed: the choice of the objective function (see Section III) and how to dynamically update its coefficients, for instance, can be improved to better quantify the likelihood of each component being faulty. Additionally, our troubleshooting algorithm only aims to perform root cause analysis of the underlying network, but does not offer solutions to overcome possible failures. Such a solution can be a better embedding of the emulated network, or a better routing policy. These aspects are to be developed in future work.

Acknowledgement

This work was carried out with the support of the SLICES-SC project, funded by the European Union's Horizon 2020 programme (grant 101008468). This work has received partial funding from the Fed4FIRE+ project under grant agreement No 732638 from the Horizon 2020 Research and Innovation Programme.

References

[1] Mininet cluster edition, 2016.

[2] Mark J Coates and Robert D Nowak. Network tomography for internal delay estimation. In *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No. 01CH37221)*, volume 6, pages 3409–3412. IEEE, 2001.

[3] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, and Chidung Lac. Distrinet: A mininet implementation for the cloud. *ACM SIGCOMM Computer Communication Review*, 51(1):2–9, 2021.

[4] Houssam Elbouanani, Chadi Barakat, Walid Dabbous, and Thierry Turletti. Passive delay measurement for fidelity monitoring of distributed network emulation. *Computer Communications*, 195:40–48, 2022.

[5] Ting He, Liang Ma, Athanasios Gkelias, Kin K Leung, Ananthram Swami, and Don Towsley. Robust monitor placement for network tomography in dynamic networks. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[6] Ting He, Liang Ma, Ananthram Swami, and Don Towsley. *Network Tomography: Identifiability, Measurement Design, and Network State Inference*. Cambridge University Press, 2021.

[7] Grigorios Kakkavas, Despoina Gkatzioura, Vasileios Karyotis, and Symeon Papavassiliou. A review of advanced algebraic approaches enabling network tomography for future network infrastructures. *Future Internet*, 12(2):20, 2020.

[8] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[9] David Muelas, Javier Ramos, and Jorge E Lopez de Vergara. Assessing the limits of mininet-based environments for network experimentation. *IEEE Network*, 32(6):168–176, 2018.

[10] Jian Ni and Sekhar Tatikonda. Network tomography based on additive metrics. *IEEE Transactions on Information Theory*, 57(12):7798–7809, 2011.

[11] Javier Ortiz, Jorge Londoño, and Francisco Novillo. Evaluation of performance and scalability of mininet in scenarios with large data centers. In *2016 IEEE Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6. IEEE, 2016.

[12] Mohamed Rahali, Jean-Michel Sanner, and Gerardo Rubino. Tom: a self-trained tomography solution for overlay networks monitoring. In *2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6. IEEE, 2020.

[13] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[14] Philip Wette, Martin Dräxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl. Maxinet: Distributed emulation of software-defined networks. In *2014 IFIP Networking Conference*, pages 1–9. IEEE, 2014.