Troubleshooting Distributed Network Emulation

Houssam Elbouanani, Chadi Barakat, Walid Dabbous, Thierry Turletti

Troubleshooting Distributed Network Emulation

Houssam ElBouanani, Chadi Barakat, Walid Dabbous, and Thierry Turletti
Inria, Université Côte d'Azur, France
first.last@inria.fr

***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, with these latter ones connected using overlay networking technologies. Mininet [10] and its forks (Mininet CE [1], Maxinet [16], 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., [11], [13]. 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 build upon 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 has thus contributed to stretching the network delay of emulated packets. In particular, we propose a network tomography [2], [8] algorithm[1] to infer the network delay of the infrastructure components from the delay measurements passively collected by Mininet [10] or its distributed variants [1], [3], [16]. Indeed, distributed network emulators are mainly designed to run on shared infrastructures (grids and clouds), which can be virtual, and whose topology might be known by the user but not directly accessible or measurable. 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 network links from end-to-end delay measurements. This problem has been formulated in [9], [12] 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 [7] 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 [14] 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 for the network delay of the infrastructure components.

This journal paper is an extension of a previously presented work in the *26th Conference on Innovation in Clouds, Internet, and Networks*, and published as a conference paper in its proceedings [6]. The remainder of the paper is organised as follows: the next section presents the problem in more details.

[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.

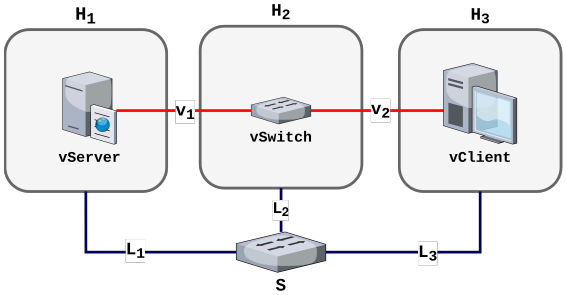


Fig. 1: Emulated and infrastructure topologies.

We 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 we then evaluate in Section IV. Finally, Section VI 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, estimate 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], and $d_i(P)$ 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 $\epsilon(P)$ and $\epsilon(Q)$ 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 $\overline{\epsilon}_j(T)$ as the mean delay error of all sampled packets during T, we have:

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

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 $(\overline{\epsilon}_j(T))_{T\in\mathcal{T}}$. According to how virtual links map to the infrastructure network, infrastructure and virtual links can then be related by linear equations of the form:

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

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:

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

[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.

where $\mathbf{A}(T)$ is defined as the *embedding matrix* whose coefficients are $(a_{i,j}(T))$, $\mathbf{X}(T)$ is a vector of unknown variables $(x_i(T))$ modeling infrastructure link delays, and $\mathbf{b}(T)$ is a vector of collected delay errors $(\overline{\epsilon}_j(T))$ on virtual links. For instance, the example scenario above can be described by the following 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 and estimate if they deviate from their expected values. 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 $\hat{\mathbf{X}}(T)$ 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 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 three heuristics with incremental complexity and comment on their signification.

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

a) Heuristic 0 - Lower and upper bounds of delay: This first heuristic aims at finding very loose lower and upper bounds on underlay link delays. The goal of its formulation is generally not to solve the problem but only to offer insight and a baseline against which the next heuristic can be compared. Concretely, the heuristic answers the question: *what are the minimum and maximum possible values of each individual underlay link delay given the mapping matrix and the overlay-level measurements?* This can be achieved by solving the formulated abstract problem 5 for the pair of functions

$$f_i^1(x_1, ..., x_n) = x_i \text{ and } f_i^2(x_1, ..., x_n) = -x_i,$$

for each underlay link i.

The lower and upper bounds are particularly interesting in our context of finding overloaded links for troubleshooting purposes. Indeed, by defining a delay threshold θ above which an underlay link is considered overloaded, the heuristic can classify with absolute certainty the links into three categories: normal-load, overload, and uncertain, following Algorithm 1.

Algorithm 1 Troubleshooting algorithm: lower and upper bounds

```
solve convex problem (4) and get values for ε*
for i = 1, ..., n do
    solve linear problem (5) with f(x_1, ..., x_n) = x_i and get x_i^m
    solve linear problem (5) with f(x_1, ..., x_n) = −x_i and get x_i^M
    if x_i^m > θ then
        consider link i as overloaded
    else if x_i^M < θ then
        consider link i as normal-load
    else
        consider link i as uncertain
    end if
end for
```

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

[5]We know from queuing theory that in practice, an overloaded link with a finite buffer 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.

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 need to 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 more infrastructure links always appear together in the embedding of virtual links, which translates into 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 the inverse (log-)probabilities of the overload of links L_i, estimated from past 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 2 summarises our methodology for estimating the delay of infrastructure links with either of the two heuristics presented above.

Algorithm 2 Troubleshooting algorithm: Occam's razor

$\alpha_i \leftarrow 1$
for $Tin\mathcal{T}$ **do**
  solve convex programme 4 and get values for ε^*
  solve linear programme 5 with $f(x_1, ..., x_n) = \sum_i \alpha_i x_i$
  update coefficients α_i
end for

B. Implementation

In real distributed emulators, our troubleshooting algorithm can be implemented by building on top of the delay measurement 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) [15] 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. Evaluation

A. Testbed

a) Underlay network (physical infrastructure): We run the simulations and the emulated experiment on a testbed which consists of a subset of 10 machines at the Rennes site of the Grid5000[6] shared infrastructure. Figure 2 provides details on the infrastructure topology. Each of the end-nodes is used to host a part of our emulated network using a distributed network emulator. Other end-nodes are used for generating external traffic to overload the links of the infrastructure. Furthermore, we only use the `eth0` interface of the machines, and we consider links `gw--c6509` and `bigdata-sw--c6509` as one single link. The reason behind this is that switch `c6509` acts here as a repeater between interfaces of equal bandwidth, and it is therefore impossible to single out one of the two links for overloading. The testbed thus involves 13 links (10 access and 3 inter-switch links), which amounts to 8192 configurations of overloading (any of the 13 links can be either overloaded or not). We will run simulations to cover all these cases, and run the following emulated experiment on a selected sample.

b) Overlay network (emulated scenario): On the physical testbed we run an experiment that emulates a near-national scale telco network where multiples ASes provide connectivity to clients and servers located in multiple regions (10) of a metropolitan France model. Figure 3 shows the telco network we consider for our emulation. In this scenario, each site hosts the same number of clients and servers, which are randomly matched at the country-level: a random server is assigned to each client in the network (1-to-1 matching), which may not belong to the same AS. The clients then synchronously download a file from the assigned servers, thus generating network traffic on all overlay links and in all directions.

[6]Detailed information about its topology and the hardware specifications of its nodes can be found at https://www.grid5000.fr/w/Rennes:Network.

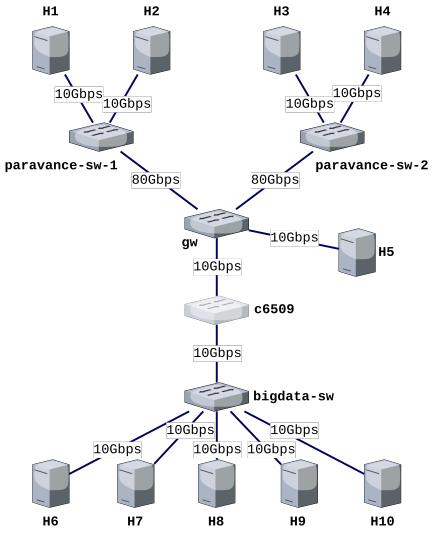


Fig. 2: Underlay infrastructure network.

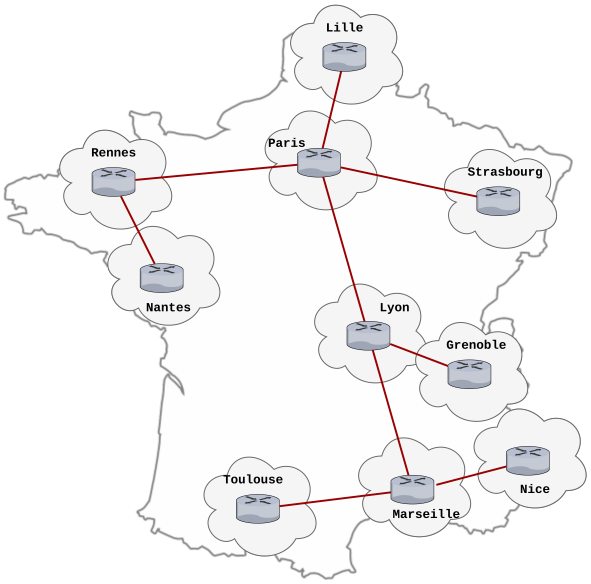


Fig. 3: Overlay emulated network.

c) Overlay-to-underlay mapping: The overlay network is emulated on the underlay infrastructure optimally and equally: all capacity constraints are satisfied and each physical host of the infrastructure runs the same number of virtual nodes. This is achieved by assigning an entire site from the overlay emulated network to its own unique physical host from the underlay infrastructure, see Table I.

1) Numerical simulations: We first conduct a set of numerical simulations to evaluate our troubleshooting algorithms on these specific overlay and underlay network structures. The objective of this series of simulations is to estimate the efficacy of our troubleshooting algorithms on all possible overload cases (∼8000), which for a lack of time and resources cannot all be run using emulation.

TABLE I: AS-Host mapping

AS	Host
Lille	H1
Nancy	H2
Rennes	H3
Nantes	H4
Paris	H5
Lyon	H6
Grenoble	H7
Toulouse	H8
Marseille	H9
Nice	H10

The simulation flow is as follows:

- First a binary vector $\mathbf{u}$ of size 13 is generated, where each element u_i indicates whether underlay link L_i is overloaded ($u_i = 1$) or not ($u_i = 0$);
- From the binary vector we generate exponential-random underlay link delays $\mathbf{X}$, where $x_i > 1\,ms$ if and only if link L_i is overloaded;
- Using the mapping matrix from the testbed we compute the overlay link delays $\mathbf{b}$;
- Then we estimate the underlay link delays $\hat{\mathbf{X}}$ from the overlay link delays $\mathbf{b}$;
- Finally, the overloaded links $\hat{\mathbf{u}}$ are determined from the delay estimations, and compared to the ground truth $\mathbf{u}$.

This is repeated for all possible vectors $\mathbf{u} \in \{0,1\}^{13}$. The estimations are evaluated using two metrics:

- *precision*: a very conservative metric which is either 1 if the estimation perfectly mirrors the truth ($\mathbf{u} = \hat{\mathbf{u}}$), and 0 otherwise;
- F_1*-score*: a looser metric that measures the similarity between ground truth and estimation by taking values in the interval $[0, 1]$, and which is defined as:

$$F_1 = \frac{2TP}{2TP + FP + FN},$$

where TP, FP, and FN are the numbers of true positives (overloaded links correctly labeled as such), of false positives (non-overloaded links labeled as overloaded), and of false negatives (overloaded links labeled as non-overloaded) respectively. The score is equal to 1 if and only if the estimation is perfect ($\mathbf{u} = \hat{\mathbf{u}}$).

Figure 4 shows the results. We see that our algorithm performs relatively well with regards to the F_1-score for all cases but its precision drops down the larger the number of overloaded links is. This is not surprising given that the main assumption motivating the heuristic is that events where many links are overloaded are unlikely to happen in practice. On the other hand, the basic heuristic that relies on upper and lower bounds often cannot fully troubleshoot congestion failures.

2) Sample runs: Since we cannot conduct all 8192 possible combinations of overloaded links, we present here only a selection –representative– sample runs and comment on potential reasons why the troubleshooting algorithm works or not make good predictions.

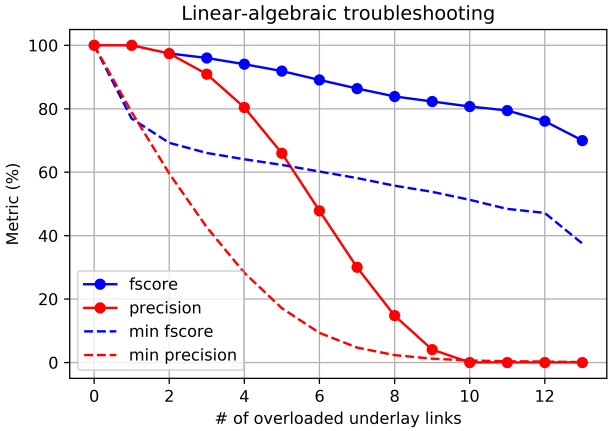


Fig. 4: Simulation results on all 8192 overloading cases. The continuous lines show the performance of linear-algebraic troubleshooting with Occam's razor heuristic (Heuristic 1); and the dotted lines by relying only on lower and upper bounds (Heuristic 0).

As the underlay network is a geographically localised high-speed cluster of hardware, one should not expect network delays exceeding few tens or hundred microseconds. As such, we will consider any estimated underlay delays higher than one millisecond to be alarming, and henceforth conclude failure. The threshold delay θ is therefore fixed at 1 ms.

The runs were conducted using HifiNet[7] on a cluster of machines running a 18.04 Ubuntu distribution with 4.15.0 Linux kernel[8].

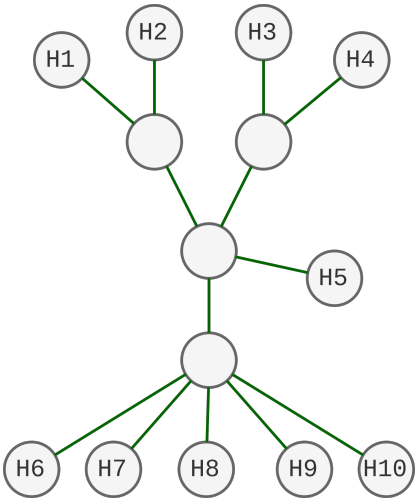
a) Run 0 - clean infrastructure: In this first run, apart from few control and management packets, no heavy traffic external to the emulation is running on the underlay infrastructure network. The user has exclusive access and exploitation of the cluster. Therefore, this run serves as a control experiment and will be considered as baseline truth for following runs.

Using the emulation measurements, the fidelity monitoring tool does not observe irregular delays and the emulation is recognised as non-faulty. The (very low) delays in the underlay links can then be approximated and the algorithm can correctly identify (Figure 5) that the infrastructure is not saturated and therefore that no link is overloaded. The results, showing an average flow completion time of 7.80 seconds over all clients of the emulation, are thus to be trusted.

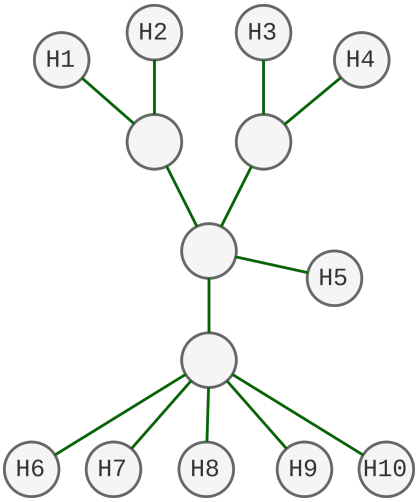
b) Run 1 - inter-cluster bottleneck: In this second run, an artificial external traffic is generated by unused machines to overload host 5's access link as well as the inter-switch link (`gw--(c6509--)bigdata-sw`). This is experienced at the emulation-level as unwanted delay in all core links connected to the Paris site (hosted in `H5`), as well as an additional

[7]HifiNet is a distributed network emulator powered with a fidelity monitoring plug-in that passively collects delay measurement on the emulated packets to detect emulation failures [5]. Its code can be found at: https://github.com/distrinet-hifi/hifinet

[8]Full description of the hardware can be found at https://www.grid5000.fr/w/Rennes:Hardware.

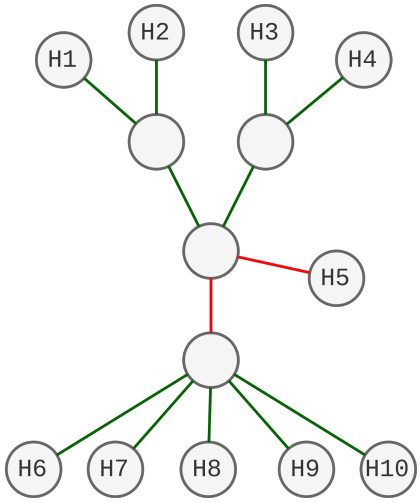


(a) Ground truth.

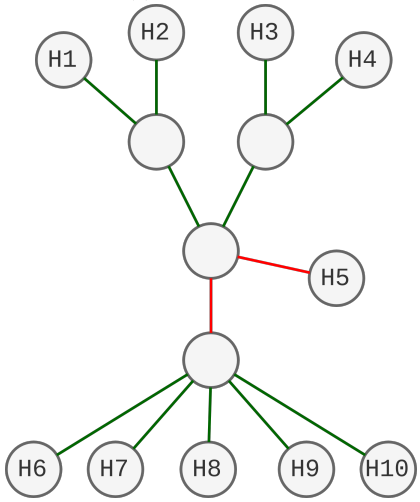


(b) Algorithm output.

Fig. 5: Run 0. Perfect prediction: 100% precision and F_1-score.



(a) Ground truth.



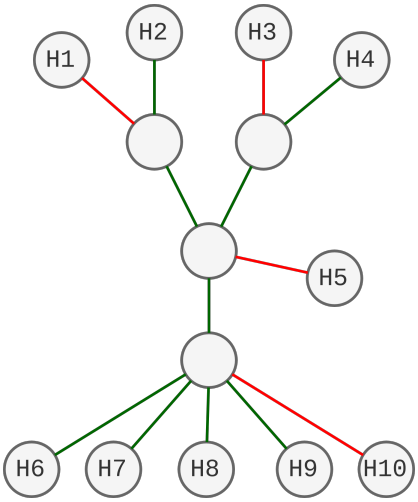
(b) Algorithm output.

Fig. 6: Run 1. Perfect prediction: 100% precision and F_1-score.

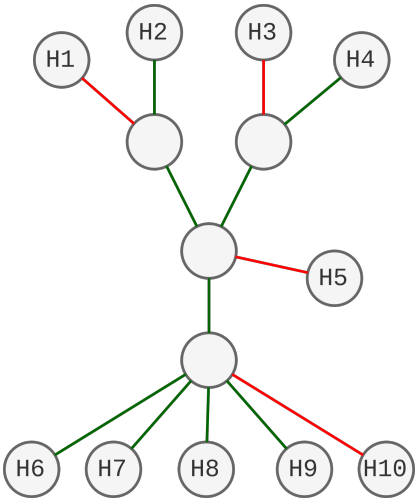
delay between Paris and Lyon. This delay is high enough to signal a break in emulation fidelity, and the troubleshooting algorithm correctly attributes its source to the overloaded links (Figure 6). From the perspective of the users, this has translated into inaccurate results and poorer experience: an average flow completion time of 10.20 seconds, mostly between clients and servers hosted in different sides of the country (North-to-South and South-to-North traffic).

c) Run 2 - uncorrelated bottlenecks: In this run, we artificially overload certain random links in the infrastructure network with multiple uncorrelated traffic flows using external machines (see Figure 7). As the number of overloaded links is relatively low, the proposed heuristics can still correctly troubleshoot the failures with perfect precision.

d) Run 3 - heavy rain in the South: In this run, we generate external traffic from and to hosts 6 through 10, and over the interswitch link. This creates congestion on the involved underlay links which in its turn incurs high delays on the emulated packets. The fidelity monitoring tool captures this delay increase and raises the alarm for emulation failure. Subsequently, the troubleshooting algorithm analyses the overlay packet delays to estimate the underlay delays, using the presented linear algebraic methods and relying on the assumption that a minimal number of links is responsible for emulation failure. In particular, our algorithm decides (wrongly) that congestion of links `H7--gw`, `H9--gw`, and `gw--(c6509--)bigdata-sw` is behind the measured high delays in the overlay, simply because these constitute a sufficient explanation. However, other links are also overloaded and their congestion contributes to the measured delay errors in the overlay. We are here in a scenario of multiple overloaded underlay links, where our algorithm has difficulty to perform as long as it finds a *simpler* explanation for the anomaly with fewer loaded links.

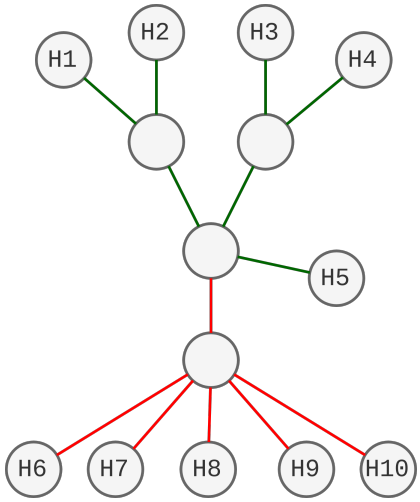


(a) Ground truth.

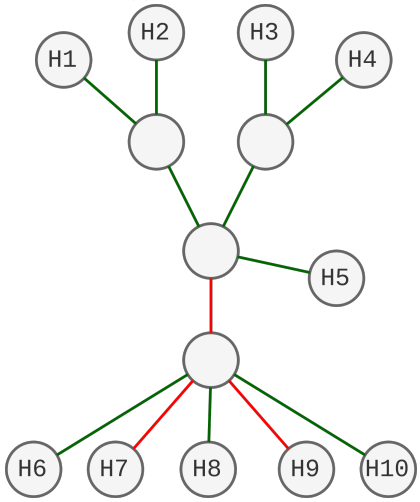


(b) Algorithm output.

Fig. 7: Run 2. Perfect prediction: 100% precision and F_1-score.



(a) Ground truth.



(b) Algorithm output.

Fig. 8: Run 3. Erroneous prediction: 0% precision and $66.\overline{6}\%$ F_1-score.

V. Emulation Remapping

The final step in emulation fidelity monitoring is using the troubleshooting predictions to help the user remake the experiment with a better mapping and potentially less errors. This can be done through a compromise by reevaluating the capacities of the infrastructure's components: by feeding the mapping algorithm artificially inflated information about the amount of compute resources and deflated information about the bandwidths of the underlay links. Indeed, the currently implemented mapping algorithms within distributed network emulators can be understood as black boxes that take static information about the underlay (underlay topology, and link and node capacities) and overlay networks (overlay topology, and link bandwidth and node compute requirements) as input, and produces a mapping of the latter over the former. The idea behind our remapping strategy is then to force the mapping algorithm to circumvent overloaded infrastructure links and instead localise as many links as possible inside the physical hosts. For this we propose an algorithm based on the following principles:

- If an underlay infrastructure link $l \in L$ is overloaded, it signifies that the total load $\lambda(l)$ of all the emulated links overlaid on top of it exceeds its *available* bandwidth which is unknown. The user has thus overestimated its capacity $\gamma(l)$ and it should be decreased accordingly;
- By increasing the compute capacities $\gamma(n)$ of the physical host $n \in N$, the mapping algorithm will be incited to redistribute the emulated nodes so as to decrease the total load on the infrastructure network;
- Priority should be given to links for which higher delays $d(l)$ have been estimated, which generally correlate with more stress.

Following the described principles, the algorithm operates as follows:

Algorithm 3 Distributed emulation remapping

```
sort L by decreasing average delay
for l ∈ L do
  if d(l) > θ then
    γ(l) ← λ(l)/2
    while m ← embed(γ) is not None and N is not empty
    do
      n ← pop(N)
      γ(n) ← 2 · γ(n)
    end while
  end if
end for
return m
```

- First, the set of underlay links are sorted by decreasing delay estimated by the troubleshooting algorithm. This will help give priority to links for which higher average loads have been observed;
- Then, for each underlay link whose average delay exceeds the overload threshold, its capacity (estimated available bandwidth) is set to be half the aggregate bandwidth of all links that were emulated over it (its load $\lambda(l)$). Indeed, if overload was observed, it necessarily means that the link could not handle the maximum emulated traffic throughput and therefore that its available bandwidth was less than the aggregate emulated bandwidths;
- Anytime an underlay link's bandwidth is decreased, the algorithm tries to find a new mapping by artificially inflating the compute resources of nodes n. The algorithm allows the inflation of each physical host by a factor of 2 at most; *and*
- Finally, if a better remapping m with updated capacity information γ is found then it is returned, otherwise the user is notified with a $None$ value.

VI. 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 established literature 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. Another opportunity for improvement is in the mapping and remapping algorithms: these can be redesigned to allow for a better tomography. For instance, constraints can be added to the mapping algorithm to force it to produce an embedding where multiple paths of the underlay network are crossed by emulated links, in order to allow for a high-precision delay tomography.

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.

Conflicts of Interest

The authors have no conflicts of interest to declare.

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] Houssam Elbouanani, Chadi Barakat, Walid Dabbous, and Thierry Turletti. Delay-based fidelity monitoring of distributed network emulation. In *Proceedings of the TASIR Workshop: Testbeds for Advanced Systems Implementation and Research*, 2023.

[6] Houssam ElBouanani, Chadi Barakat, Walid Dabbous, and Thierry Turletti. Troubleshooting distributed network emulation. In *2023 26th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 145–152. IEEE, 2023.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.