Introducing fidelity into network emulation

Houssam Elbouanani

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Introduire de la fidelité dans l'émulation de réseaux

Houssam ELBOUANANI

Centre Inria d'Université Côte d'Azur

Présentée en vue de l'obtention du grade de docteur en Informatique **d'**Université Côte d'Azur

Dirigée par : Walid DABBOUS, Directeur de Recherche, Centre Inria d'Université Côte d'Azur, Sophia Antipolis
Co-dirigée par : Chadi BARAKAT, Directeur de Recherche, Centre Inria d'Université Côte d'Azur, Sophia Antipolis
Co-encadrée par : Thierry TURLETTI, Directeur de Recherche, Centre Inria d'Université Côte d'Azur, Sophia Antipolis
Soutenue le : 2 mars 2023

Devant le jury, composé de :
Guillaume URVOY-KELLER, Professeur des Universités, I3S, Sophia Antipolis
André-Luc BEYLOT, Professeur des Universités, IRIT, Toulouse
Stefano SECCI, Professeur des Universités, CNAM, Paris
Christian ESTEVE ROTHENBERG, Associate Professor, Université d'État de Campinas, Brésil
Laurent MATHY, Professeur Ordinaire, Université de Liège, Belgique

INTRODUIRE DE LA FIDELITÉ DANS L'ÉMULATION DE RÉSEAUX

Introducing Fidelity into Network Emulation

Houssam ELBOUANANI

⋈

Jury :

Président du jury

Guillaume URVOY-KELLER, Professeur des Universités, I3S, Sophia Antipolis

Rapporteurs

André-Luc BEYLOT, Professeur des Universités, IRIT, Toulouse
Stefano SECCI, Professeur des Universités, CNAM, Paris

Examinateurs

Christian ESTEVE ROTHENBERG, Associate Professor, Université d'État de Campinas, Brésil
Laurent MATHY, Professeur Ordinaire, Université de Liège, Belgique

Directeur de thèse

Walid DABBOUS, Directeur de Recherche, Centre Inria d'Université Côte d'Azur, Sophia Antipolis

Co-directeur de thèse

Chadi BARAKAT, Directeur de Recherche, Centre Inria d'Université Côte d'Azur, Sophia Antipolis

Co-encadrant de thèse

Thierry TURLETTI, Directeur de Recherche, Centre Inria d'Université Côte d'Azur, Sophia Antipolis

---

Université Côte d'Azur

ABSTRACT

The design and development of new network protocols, architectures, and technologies requires an evaluation phase where the researcher must provide empirical evidence for the performance of their contributions, potentially in comparison to existing solutions. In this context, network emulation has proven to be an attractive approach as it offers more flexibility compared to traditional testing platforms, and more realism compared to simulation.

Network emulators provide contained, customisable, and scalable testing environments both for researchers to evaluate their contributions and for the community to reproduce their results. However, two limitations to network emulation have been identified and well documented in the literature: its scalability limits and its accuracy issues.

This dissertation documents our attempts to address these concerns. Our findings are distilled into Hifinet: a lightweight scalable and fidelity-aware distributed network emulator. We particularly show how Hifinet outperforms its state-of-the-art counterparts in terms of scalability and efficiency by working around the flaws of their design principles and the technological limitations of the tools they rely on. Hifinet is also fidelity-enhanced, in that it implements a fidelity monitoring framework we have theorised, which passively monitors emulated packet delays to evaluate realism of network emulation and accuracy of results. Another asset of Hifinet is its ability to infer underlying causes in case of erroneous emulation. This is achieved by using delay tomography algorithms and heuristics.

Keywords

reproducible research; network emulation; mininet; network measurements; delay tomography

RÉSUMÉ

La conception et le développement de nouveaux protocoles, architectures et technologies de réseau nécessitent une phase d'évaluation au cours de laquelle le/la chercheur·se doit fournir des preuves empiriques de la performance de ses contributions, potentiellement en la comparant avec des solutions existantes. Dans ce contexte, l'émulation de réseau s'est avérée être une approche attrayante car elle offre plus de flexibilité par rapport aux plateformes de test traditionnelles d'un côté, et plus de réalisme par rapport à la simulation d'un autre côté.

En effet, les émulateurs de réseau fournissent des environnements de test contenus, personnalisables et scalables, à la fois pour que les chercheur·se·s puissent évaluer leurs contributions et pour que la communauté puisse reproduire leurs résultats. Cependant, deux limites à l'émulation de réseau ont été identifiées et bien documentées dans la littérature : son incapacité à passer à l'échelle et ses problèmes de précision.

Cette thèse documente nos tentatives pour répondre à ces préoccupations. Nos résultats sont distillés dans Hifinet : un émulateur de réseau distribué léger, résistant aux facteurs d'échelle et sensible à la fidélité. Nous montrons en particulier comment Hifinet surpasse ses homologues en scalabilité et en efficacité en contournant les défauts de leurs principes de conception et les limites technologiques des outils sur lesquels ils reposent. Hifinet est également plus précis, car il met en œuvre un cadre de contrôle de la fidélité que nous avons théorisé, qui mesure passivement les latences des paquets émulés afin d'évaluer le réalisme de l'émulation du réseau et la précision des résultats. Un autre atout de Hifinet est sa capacité à déduire les causes sous-jacentes en cas d'émulation erronée. Ceci est possible grâce à l'utilisation d'algorithmes de tomographie de délais.

Mots-clés

recherche reproductible; émulation de réseaux; mininet; mesures réseaux; tomographie de délai

To my parents who have always supported me in pursuing my passion,

To the memory of my beloved grandmother who was proud to see me embark on this long journey but cannot share the happiness of its completion,

To everyone who has helped me carry on despite grief and illness,

I hope, at the least, that I have made you all proud.

I would be foolish to believe this work will expand the horizon of human knowledge, but it will forever remain a personal monument to resilience and self-reliance.

ACKNOWLEDGMENTS

I would like to give my warmest thanks to all my supervisors. In all difficult and frustrating moments you were understanding and considerate. This thesis would not have been possible without your help, guidance, and priceless comments and suggestions.

I would also like to give special thanks to –now PhDs– fellow colleagues Mamoutou and Othmane. I have learned a lot from our discussions and your friendship has made this journey much more enjoyable.

Finally, I would like to thank my dear Paolito. No words can fully describe how much your companionship has been valuable throughout this journey.

La vérité ne fait pas tant de bien dans le monde que ses apparences y font de mal.

–François de La Rochefoucauld

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Reproducibility is an essential criterion for scientific correctness. A scientific scholarship must be reproducible both *in principle*, in the sense that the experimental outcomes should not depend on uncontrollable circumstances, but also *in practice*, in the sense that anyone should be able to repeat the experimental processes that led to the scholarship's results and conclusions, provided they can acquire the necessary tools and environments. However, the modern –international, open, fast-paced[1]– academic production of scientific knowledge has shown this criterion to be most often overlooked, ultimately leading to the replication crisis that scientific research is currently facing.

Indeed, a 2016 Nature survey [14] of 1,576 researchers found that 70% of the surveyed have tried and failed at least once to reproduce a peer's research, and more than 50% to reproduce their own. Of the surveyed, only less than 31% believe that failure to reproduce some published results suggests that the results are wrong, and most would still trust irreproducible published research. Nevertheless, only 20% of the surveyed report having been contacted in the past by another researcher who could not reproduce their scholarship. These numbers not only show the shocking magnitude of the issue, but also the dangerous indifference of scientists and experts. This indifference varies by field: researchers in human sciences (medicine, sociology, and economics) show more attention to reproducibility, while their peers in *formal* sciences (physics, chemistry, and engineering) are generally overconfident in irreproducible results.

Many factors contribute to the replication crisis, some are social-institutional (fraud [83], pressure to publish [68], jockeying for competitive journals and conferences, race to private and public funding [63], etc.) and some technical (difficulty to acquire data and/or

[1] According to the sociologist Hartmut Rosa [74], acceleration of technical and scientific advancements is a key feature of modernity, with ongoing consequences on the quality and integrity of industry innovation and academic research.

tools to reproduce experiments, difficulty to package one's own experimental processes, etc.). However, the fact of the matter remains that making research reproducible is not properly incentivised, and failure to do so is not fairly punished. The amount of work and time required from an author to make their scholarship reproducible or from a reader to reproduce their results are not profitable, and therefore overlooking this aspect is a reasonable attitude.

1.1 Paradigms in Experimental Network Research

In the fields of computer science and information technology, most research amounts to building models and architectures and to writing algorithms. This design phase must generally be followed by an evaluation phase, where the researchers are to prove the implementability of their algorithm/model/architecture and compare its performance to existing solutions. Authors are pressured to conduct such evaluation *experimentally* (or *empirically*), instead of relying on mathematical models that formally prove asymptotic or lower bound metrics of performance. However, there is no clear consensus as to what constitutes an experimental or empirical evaluation. Ideally, in order to evaluate how algorithm or architecture A performs when implemented in environment E (or a set of environments $\mathcal{E}$), the researcher must implement A in E (or in a good sample from $\mathcal{E}$). For instance, to evaluate with maximum precision how a new topology improves the performance of a data centre network, it must be tested in a set of data centres with typical network traffic conditions. Unfortunately, this is not always possible, as a researcher might wish to design algorithms and architectures for environments they cannot test it on, and such highly achievable requirement ultimately serves to gatekeep the production of knowledge from scientists with limited resources, and highly limits the scopes of reproducibility in computer science research.

This is especially the case in distributed systems and networks research, where the target environments E are generally fairly large and complex systems which are not available to all researchers and/or stakeholders. Indeed, while it might not be an issue for a large company

or institution to experiment their innovations on production-similar networks, the general public can only rely on their produced datasets to repeat the last steps of their experimental processes: reinterpreting data and redrawing results and conclusions.

To mitigate these methodological obstacles, efforts have been made to facilitate both the experimental and empirical evaluation of scientific innovations and their reproducibility in the specific context of computer networks and systems research (and to a lesser extent other fields of computer science). For a long part of network research history, this was achieved by two experimental paradigms: testbeds and simulation.

1.1.1 Testbeds

Testbeds are platforms designed to run experiments. In the context of computer networking and distributed systems, these take the form of hardware infrastructures used to test novel ideas in different areas of the field. Though it is not necessary, testbeds are most often shared by multiple researchers with different affiliations, and funded by country- or continent-wide public programmes.

PlanetLab [20] is one of the most popular and longest-running testbeds. It was first launched in 2002 for collecting network measurements and is composed of more than 1000 nodes (1353 at its peak). These nodes are deployed at more than 700 sites spanning over 48 countries[2] from all continents, which makes it a realistic environment for testing research aimed at wide-area technologies (P2P networks and protocols, inter-AS routing, security models and attacks, etc.). In fact, research from hundreds of papers was validated using the PlanetLab testbed [6], although the number of new published papers has since been declining.

The particular limitation of PlanetLab as a testbed is its usage scope. It has been historically aimed toward planet-wide architectures and protocols, and thus most of its services

[2]PlanetLab has been discontinued in the USA as of may 2020. Its North American user base has migrated to MeasurementLab.

have become obsolete as research in their fields has declined or even stopped. It is also not up-to-date with new technologies, particularly SDN, 5G, and IoT architectures, which are not supported without simulation.

Similar projects have been developed and deployed in Europe. Fed4Fire+ [3,4] is a European programme that has funded and federated a large number of testbeds all over the European Union. These platforms span a wide range of technology domains: standard wired networking, wireless technologies, 5G hardware, IoT technologies, SDN research, and Cloud and Big Data architectures (Figure 1.1). Fed4Fire+ even offers a federated portal to all these platforms, where users can access from a single user account the resources that match their need. Fed4Fire+ also includes *associated testbeds*, which are experiment platforms from collaborating partners (both private companies and academic institutions) that are partially funded by Fed4Fire+ but which are not federated in its web portal. One special associated testbed is Antwerp, Belgium's Smart Highway [61] which is a unique facility designed for vehicle-to-everything communication research.

However, testbeds suffer from some inherent limitations that impact their flexibility and their potential to support reproducible experiments.

- Cost: as has been discussed, most testbeds are infrastructures maintained by large institutions that span entire countries or continents, as a single university or laboratory generally cannot fund its own platform. This issue of cost also makes the process of building a testbed difficult;

- Timeline: building a testbed is a long process. It typically takes several years of design, preparation (including institutional and legal delays), and implementation before a platform is fully operational. This, in contrast with the recent technologies' shorter

[3]Fed4Fire+ has been discontinued as of june 2022. A new project called SLICES-RI has been launched under the ESFRI programme (2020 to 2024) to fund a European-wide research instrument for the digital sciences.

[4]It must be disclosed that this PhD thesis was funded by Fed4Fire+ until june 2022 then by SLICES-SC.

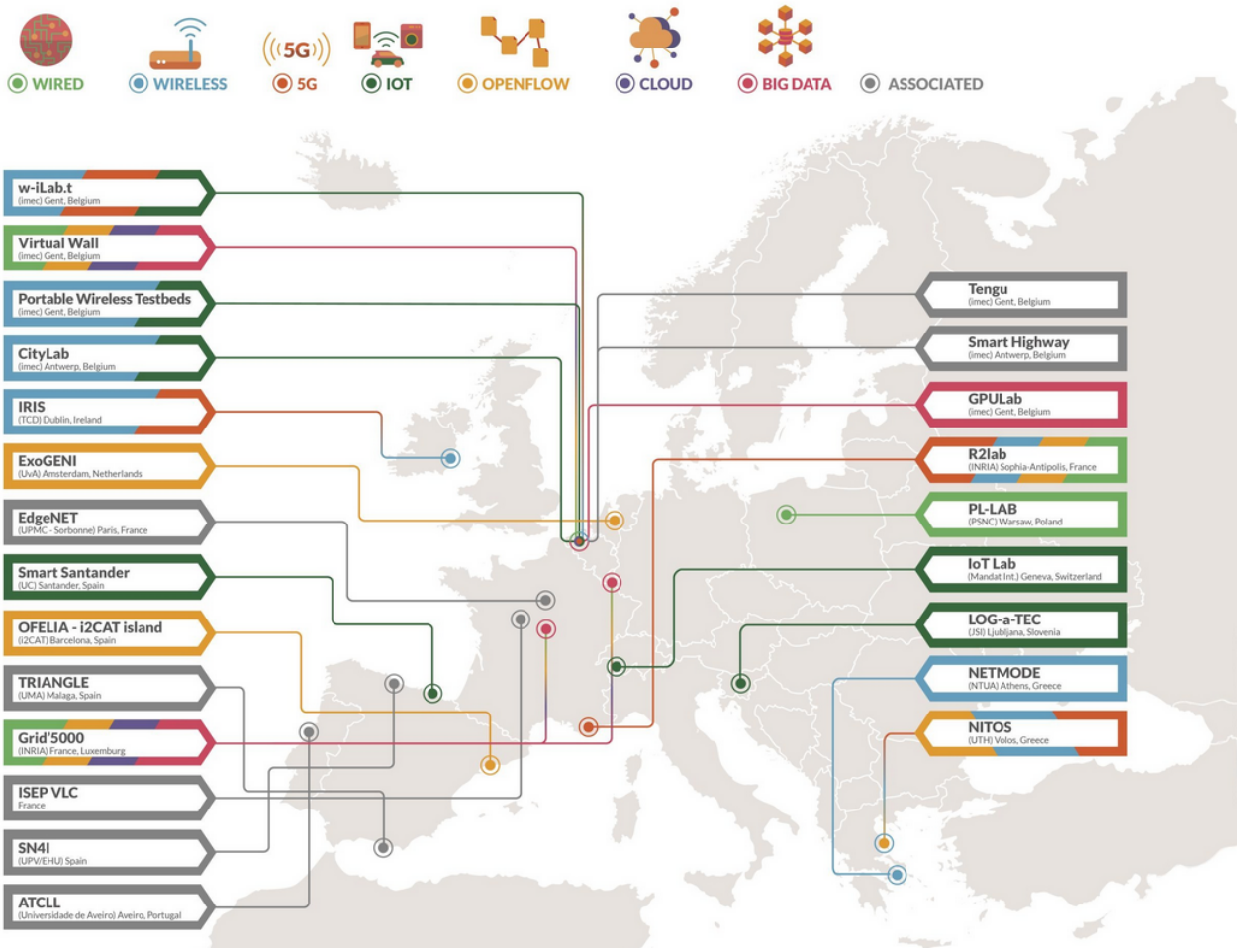


Figure 1.1: Federated (and associated) testbeds under the Fed4Fire+ programme. Source: `https://www.fed4fire.eu/testbeds/`

life cycles and the fast-changing networking paradigms, can make the deployment of experiment platforms for emerging technologies challenging;

- Sustainability: just as networking paradigms and technologies rapidly change, so too must testbeds, which ultimately need to be terminated once their intended technologies become obsolete, a process that also requires time and resources. Some testbeds can be transformed or updated to integrate new technologies, but that is not always possible;

- Recognition: deployed testbeds need to be properly advertised to the community to attract potentially interested researchers in order to justify their existence and prove to be profitable. Also, for research produced in a testbed to be recognised by the community as experimentally valid, the testbed's credentials must be proven, customarily through reputation which is built by repeated use from researchers;

- Inter-operability: testbeds ideally need to be inter-operable, in that a scholarship produced in one should be reproducible in all others that provide similar environments. In practice however, this is rarely the case, except for platforms federated under the same project or institution. This also makes it harder for potential users to switch between testbeds, as learning the specific API and architecture of each individual platform may discourage reproducibility;

- Governance: the creation, management, and termination of testbeds are decided by the central institutions that provide funding. This mode of governance goes against the ideals of open and democratic science, because whoever can decide which practical answers can be tested, retroactively decides which scientific questions deserve to be asked[5];

[5]The relationship between knowledge and power is a complex question in epistemology and sociology of scientific research. The general observation is that knowledge has a non-negligible social dimension that makes neither absolute nor politically neutral, but rather contingent on the power structures within societies. In concrete terms, the pursuit of knowledge is always impacted by *outside* effects, such as institutional policies and market forces, which are ever-changing never immune to agendas [38, 55].

- Openness: experiment platforms are not completely open for use to all interested researchers. Some clearly state that they are restricted to users from certain affiliated universities, other claim to be open but require a long registration and verification process that can discriminate against users without known affiliations and overall discourages reproducibility;

- Impact assessment: the return on investment of a testbed is often gauged by the number of active users, trained students, and published papers. These indicators might not always reflect the usefulness of a testbed but are used to determine its viability and whether it deserves to keep running.

These limitations of testbeds, particularly with regards to their accessibility and usability, have led to the development of other methods of experimentation.

1.1.2 Network Simulation

Network simulation is a class of experimentation techniques whereby a software and/or mathematical model of a given problem, environment, and solution is built to conduct empirical evaluation. Virtually any environment, architecture, and technology can be simulated by a more or less accurate model. This works by abstracting away all details deemed irrelevant about the operation of a system and only focusing on the key features that can be modeled. The model is then fed into a software which can run performance evaluation.

Depending on which components of the environment are modeled, and on how compact the models are, we can distinguish between multiple levels of simulation. The most basic technique is mathematical modeling and simulation: the environment and solution are modeled using abstract mathematical objects that distill its most important features and transforms the performance evaluation problem into a mathematical one. The problem is then solved analytically using mathematical tools when possible, or approximately using numerical simulation techniques. Common modelings of computer networks use queuing

theory, graph theory, optimisation and linear algebra, etc. Such mathematical techniques have been popular for a long part of network research history, mainly due to their simplicity and to a lack of access to more empirical approaches (software simulators and testbeds).

Another technique is the use of network simulators [77]. These software tools work by breaking down the simulated environment into distinct components whose interactions are modeled as software code that replicates their behaviour. Many network simulators have been developed since the 1980s to study the performance of network protocols, each with a different architecture and a different goal. Some are universal and allow the simulation of virtually any network (provided the user programs the necessary components), others focus on particular target environments, such as 5G networks, IoT technologies, and VANET architectures. Network simulators also differ in whether they are free (in all senses of the word) or closed-source and/or under commercial licenses.

The most popular network simulator is ns-3 [43]. It started in 2006 for the purpose of enhancing the limited functionality of the previous versions of ns (ns-1 [62] and ns-2 [45]) which had since then become obsolete. It is a discrete-event network simulator: the components of the simulated network are represented as `C++` classes (e.g., `UdpEchoServerApplication`, `WifiChannel`) and the operation of the simulated network is broken down into a sequence of events represented as `C++` functions (e.g., `StartApplication()`, `ScheduleTransmit()`). A simulated experiment then runs in a discrete time fashion: the passage of time is simulated as a succession of discrete small units while every step is logged for measurement purposes. Furthermore, modularity is a core design principle of ns-3. Each component (application, protocol, or network hardware) functions independently of the simulation engine and many libraries exist to provide users with collection of classes and functions useful for typical technologies and scenarios (WiFi, SDN, 5G, etc.). These aspects make ns-3 a powerful and near-universal simulator.

Other simulators are engineered to focus on specific use cases. For instance, NetSim

[32] and Cisco Packet Tracer [1] are focused on simulating Cisco hardware, particularly for educational purposes; REAL [51] (REalistic And Large, on which ns-1 is based) is focused on congestion control algorithms; and JiST-SWANS [3] (Java in Simulated Time/Scalable Wireless Networks Simulator) is made for wireless networks (and particularly MANETs).

Network simulators also suffer from a set of ontological limitations:

- Realism: the main limitation of network simulation is realism. The results of a simulated experiments are to be trusted only insofar as the modeling and simulation are accurate description of the environment, which requires an additional layer of interpretation and analysis. If the objective of simulation is to examine the feasibility of a researched solution in a desired environment, then perfect realism is not an issue; if, however, the objective is to challenge the solution and precisely evaluate its performance compared to existing solutions, then careful attention must be paid to numerical results;

- Flexibility: to evaluate an application or an architecture in a network simulator, it must first be programmed and integrated into its code. This adds an additional step in the experimentation phase which, depending on the familiarity of the user with the simulator's framework, might add complexity and exhaust time resources, which may sometimes be more time-consuming than using regular testbeds. In fact, it is much easier to run a newly developed solution in an available testbed than it is to model it and/or adapt it into the simulator;

- Scale: although a simulator transforms a real environment into a miniature version that replicates some of its behaviour, in practice the running time of a simulation increases proportionally to the duration of the experiment, to the scale of the simulated system, and to the complexity of the model, and inverse-proportionally to the resources of the machine used to run the simulation. Thus simulating a large-scale system while capturing its complexity is most often difficult.

1.1.3 Network Emulation

Network emulators are a particularly interesting type of software-based network experimentation. Their purpose is to let the user run complex network scenarios using actual service and application code and highly personalizable topologies without the need for physical hardware (links, switches, routers, firewalls, middleboxes, servers, etc.). Popular emulators (e.g. Mininet [4], Dockemu [79]) use virtualisation, containerisation, and network softwarisation technologies to achieve this with minimal need for hardware. For instance, Mininet uses Linux-native tools and software-defined networking mechanisms: Linux network namespaces to emulate isolated upper-layers nodes (clients, servers, firewalls, proxies, etc.); Linux bridges (or Open vSwitch swtiches) to emulate SDN-enabled layer-2 and layer-3 nodes (switches and routers); and Linux Virtual Ethernet and Traffic Control to emulate wired links. Its use of such lightweight and natively supported technologies, alongside a simple Python API, make it a great tool for easily repeatable and fairly large-scale network experimentation. Variants of Mininet have also been developed to add functionality or to increase effectiveness. Mininet-WiFi [37] adds wireless capabilities by simulating WiFi, MANETs, and mobility; while distributed forks of Mininet (e.g. Mininet CE [5], Maxinet [82], and Distrinet [26]) greatly increase the software's scalability by allowing the user to run scenarios using aggregated resources from a cluster of machines, a public cloud, or a grid infrastructure.

However, research has shown that network emulators do not always provide perfectly accurate results [65]. In fact, as they are designed for running on everyday laptops, their emulation of multiple events (e.g., running code in emulated hosts, or switching and routing multiple packets in parallel) is very limited by the available computing and network resources. This renders them practically unusable for emulating latency-sensitive scenarios or those that require packet-level precision.

1.2 Modern Problems Require Modern Solutions

This thesis explores the world of scientific experimentation, particularly reproducible, scalable, and accurate network experimentation. The goal of any researcher working with such questions is to develop an efficient formula for producing empirical knowledge about computer networks that checks all the boxes: reproducibility, accuracy, scalability, accessibility, flexibility, openness, and efficiency. The fundamental hypothesis we are looking to investigate is whether network emulation as a paradigm could be the answer. Indeed, emulation already satisfies most requirements by design, but struggles with limitations regarding scalability and accuracy. The objective thus is to investigate how these two challenges can be overcome using modern technologies:

- Scalability: how can emulators efficiently use the available resources to sustain fairly big networks, modern architectures and technologies, and heavy traffic?
- Accuracy: how can emulators realistically mimic the behaviour of the network and ultimately produce accurate and fidelitous results? and otherwise is it possible to predict anomalous behaviour and trace its origins?

Both questions will be tackled using modern tools and revisiting well-theorised traditional methodologies. In particular, this work presents a highly scalable and fidelity-focused distributed network emulator that uses recently developed system containerisation, network virtualisation, and service orchestration technologies.

The remainder of this dissertation is organised into two parts.

- The first half provides the necessary background for tackling the above challenges, with supplementary findings from the last two decades of research in network experimentation and measurements. In particular, Chapter 2 focuses on network emulation, especially virtual machine- and container-based emulators, by uncovering their core design principles and by presenting how said principles are implemented in popular

emulators. The second chapter, Chapter 3, introduces the state of the art in delay measurement and network tomography;

- The second half of the dissertation documents our contributions. We start by presenting our newly-developed lightweight distributed network emulator in Chapter 4, where we show its design and implementation and compare its performance to its state-of-the-art counterpart. The following two chapters present our framework for passive measurement-based fidelity monitoring of network emulation: Chapter 5 showcases the theory behind while Chapter 6 presents its implementation and its operation in practical scenarios. The last chapter of this second part, Chapter 7, demonstrates a preliminary attempt at using the measurements collected during emulation to troubleshoot its potential failures.

The dissertation is concluded and summarised in Chapter 8 where we also discuss limitations and suggest prospects for future research.

Part I

Background and State of the Art

CHAPTER 2

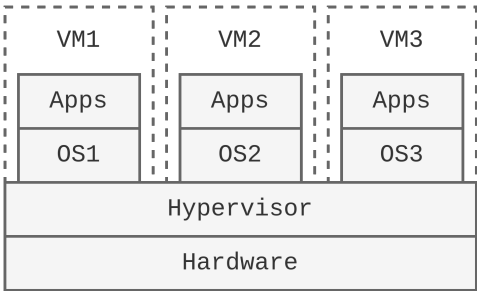
NETWORK EMULATION

Network emulation has been getting more and more attention from network researchers. Its history is mostly marked by three major points: the development of system and network virtualisation which permitted its birth; the publication and relatively widespread use of Mininet which made it accessible to experts and intermediate researchers alike; and the advance of distributed emulation which caused a paradigm shift from the traditional "network in a laptop" model, in the hope of answering users' concerns about scalability. This chapter is structured along the lines of these historical developments and lays the state-of-the-art in the matter. We will first give a brief background presentation on system virtualisation, the explain the machinery behind network emulation and distributed network emulation.

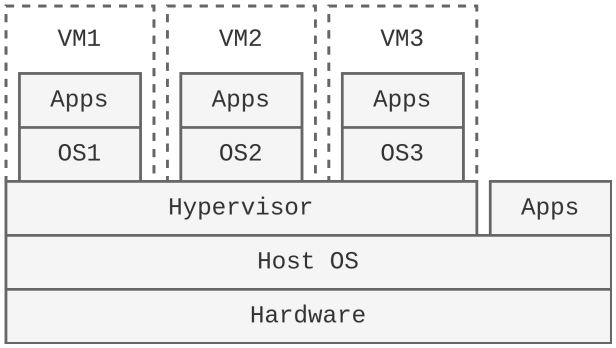
2.1 Virtualisation

Virtualisation is a major advance in computer networks and systems [75]. It was initially motivated by the inefficient use of resources and by the need to isolate different services running within the same machine. Indeed, in the early years of the Internet, network and web services had to be independent and isolated and thus would run on different physical hosts, which was highly inefficient [27]. With virtualisation, it is possible to run completely isolated environments, generally with their own operating systems and network stacks, in the same physical machine whose resources they share. Thus in theory, it is possible by using virtualisation to run operating systems with different kernels (e.g., a Linux and a Windows system) on the same hardware environment.

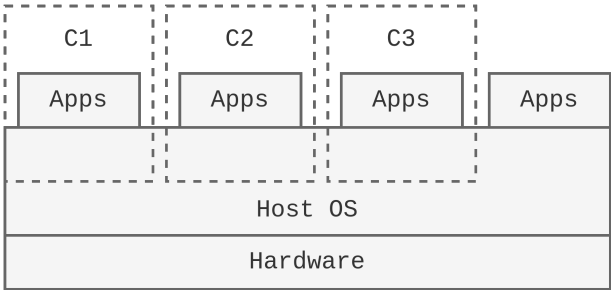
This is achieved through a software, called the *hypervisor*, that runs in a layer above the hardware. In this manner, each virtual machine (with its own operating system and applications) behaves as if it was directly running –alone– on the underlying hardware, while



(a) Bare-metal virtualisation. Three virtual machines (VM1, VM2, and VM3), each running its own operating system, share the same physical hardware through a hypervisor.



(b) Hosted virtualisation. The hypervisor runs as a regular software on the host machine, and relies on its operating system to manage the virtual machines' accesses to the hardware.



(c) Container-based virtualisation. The containers are logical separations of the applications running on the host machine. They share the kernel of the host machine, but run other parts of the operating system in an independent manner.

Figure 2.1: Bare-metal vs hosted vs container-based virtualisation.

all their requests to access its resources are intercepted and managed by the hypervisor to handle transparent sharing of the hardware. We can further distinguish between different types of virtualisation depending on how the hypervisor carries out this function and on whether or not the process involves the virtual machines.

Bare-metal virtualisation Bare-metal is the basic and most common way to deploy virtualisation, particularly in large-scale contexts such as data centres [60]. It refers to hypervisors that are installed and that run directly on the hardware and handle all instructions

to use its resources. Figure 2.1a visually shows how bare-metal virtualisation is designed and how it operates within a server with its interaction with other components.

Hosted virtualisation As opposed to bare-metal virtualisation, in hosted virtualisation the hypervisor is a software that runs as an application on the *main* operating system of the physical machine [81]. This technique of virtualisation is thus clearly less efficient than bare-metal virtualisation, and is mostly used on personal computers for educational purposes or to run software that requires specific operating systems. Figure 2.1b is a representation of this paradigm.

Containerisation Containerisation (also referred to as container-based virtualisation) is a more recent technique for running multiple isolated environments on the same physical hardware [84]. Its main and most important feature compared to regular virtualisation is that the virtual environments, called containers instead of virtual machines, share the kernel of the hosting machine. These containers are logical divisions of the software space so that applications and services can run independently, with their own file systems (and file hierarchies), their own network stack (and a virtual hostname), their own users and groups, and a limited and isolated access to hardware resources. Therefore, there is no additional component to act as interface between the containers and the host system and hardware, and instead isolation is performed logically by tools already available to the operating system (any Unix-like or BSD) (see Figure 2.1c for a visual diagram).

2.2 A Deeper Look into Network Emulation

Outside of computer networking, emulation is clearly defined as the process by which a hardware environment is substituted using software tools (called the emulators) to run on a different hardware environment. For example, an HP printer could be emulated on a non-HP printer so that it runs software written for HP printers. Historically, emulation has been

popular in video-gaming communities where many emulators were developed to run –on a regular personal computer– video-games designed for consoles and arcade systems with completely different operating systems and hardware architectures [23]. In such cases the emulated systems run in real-time or near-real-time, which is precisely what makes system emulation stand out from system simulation.

In computer networking research, however, the lines between emulation and simulation are more blurry, and there is no consensus yet that clearly defines its frontiers. For instance, it is common to find software tools that are labeled as both emulators and simulators (e.g., IMUNES [2]), not so much because they can perform both, but rather because of the inability to classify themselves into a clear-cut category. This classification difficulty comes from the inherent complexity of networks compared to single-unit hardware systems. The latter can generally be seen as a stack of services running on an operating system software that manages a set of physical resources, and thus defining emulation as *replicating the hardware part, and the hardware part only, using software* comes at no cost; but a network of very diverse –and very diversely connected– nodes cannot be reasoned about as easily.

To better understand the complexity of separating between network simulation and emulation, and to help us attempt a divide, a network environment can be modeled by breaking it down into multiple components in the manner of Figure 2.2. In this model, the environment is composed of multiple nodes that communicate with each other through a set of communication media and whose operation is impacted by features of the physical world (namely time and space):

- Application: is the software component that runs in the nodes of the environment. In general, an application is the highest layer in a node's stack, generating data packets processed by the network protocols, but it may also directly interact with the network stack (e.g. a proxy or a firewall) or the operating system;

- Data packets: are the messages and the data traffic that user applications generate and

which are to be processed by the network stack and encapsulated into communication units to be transferred through the communication media of the network environment;

- Network protocols: generally the TCP/IP suite of protocols designed for data communication in typical networks. They customarily process application messages and are implemented in the operating system of the node. The nature of such protocols also depends on the experimented scenario: whether it is a 5G environment, or an SDN architecture, etc.;

- Operating system: manages the access of the user applications and network protocols to the hardware resources of the nodes;

- Hardware: is the physical part of a node, which is the set of resources on which its logic runs: computing units, memory, storage, network devices, sensors and detectors, etc.;

- Communication media: are the elements of a network that transport the data in its raw physical form (electromagnetic waves, electric signals, etc.). They typically fall under two categories: wired (links) or wireless, each with its own characteristics and transmission capabilities;

- Time and space: are two environment features that can impact the experiment in certain settings (time passage and clock synchronisation, spatial mobility of nodes, transmission range, etc.).

For example, NetEm [42] and radio channel emulators are network emulation tools that allow the user to replicate the behaviour of certain special networks using a LAN testbed. Looking at the proposed model, these tools use *real* end-systems (with *real* software, network stack, operating system, and hardware) operating in *real*-time, but simulate the behaviour of wide-area networks (by adding artificial latency and bandwidth limitation to wired links)

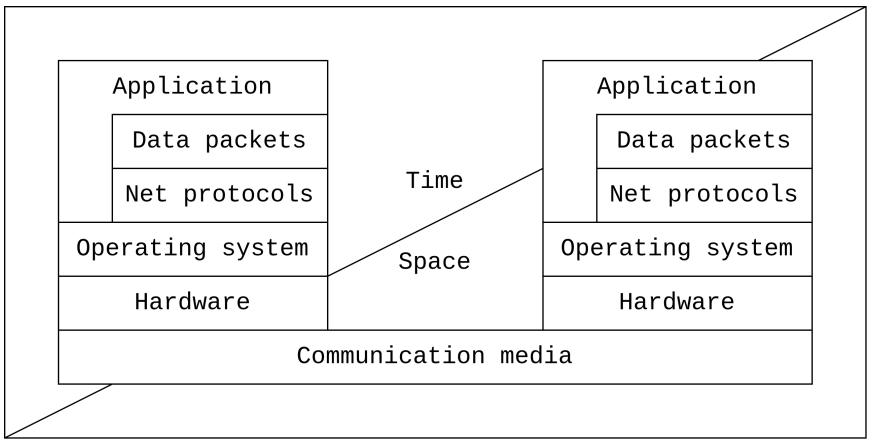


Figure 2.2: Model of a network environment.

and radio networks respectively. Thus the *communication media* component is simulated. Radio channel emulators additionally simulate spatial features of radio networks, such as node mobility and signal range, and thus simulates the *space* component as well.

On the other hand, a discrete-event simulator (such as ns-3) also replicates the nodes' *hardware*, *operating systems*, *network protocols*, *data packets*, and even the *applications* that they run. It also, and most importantly, simulates the passage of *time*.

The rise in popularity of system virtualisation and containerisation technologies, as well as software-defined networking paradigms, inspired the development of network emulators that rely on these technologies to create fairly large-scale networks with perfectly isolated and controlled nodes on a single computer. DockEmu [79] uses Docker containers and Linux bridges to run multiple end-systems and SDN-enabled switches and routers, who can run their own *real network protocols*, *operating systems*, and *applications* that generate *real data packets*, but whose *hardware* and underlying *communication media* are simulated using software tools.

For a lack of a more precise definition of network emulation, we can infer a working one from these examples and using our model. It will serve academic accuracy in this dissertation, but we believe it can be a good definition outside of it as well. In practice, we will say that a computer networking experiment is a *network emulation* if it was run in an environment

that satisfies three conditions (in at least one host):

- at least one component is simulated;
- the *applications*, generated *data packets*, *networking protocols*, and *operating systems* of nodes are *real*; and if
- the experiment runs in real-time, i.e. *time* is not simulated.

A *network emulator* is then a tool that can produce network emulations. The first condition essentially requires a part of simulation, which should not be at the level of the applications, the data traffic, or the system, but rather at the level of the hardware and communication media, according to the second condition. The last condition is an essential requirement that differentiates emulation from discrete-event simulation. Note that a network emulation may be run with simulation of spatial features, as is the case with radio channel emulation and Mininet-WiFi [37].

2.3 Mininet

Mininet [4] is the most popular network emulator. It was born at Stanford University from the need to experiment with SDN [15] technologies in a reproducible manner and to learn networking by practice. Its main selling point is its powerful but easy to use Python API.

2.3.1 Design Principles

Limited isolation Mininet is designed to emulate end-systems with a minimal constraint of isolation. Indeed, only the applications, network stacks, and resources are isolated between machines emulated with Mininet; while they share the rest of the operating system (file system and kernel) with the hosting computer. Thus by not having to run a kernel for each end-system, this principle increases the scalability of the emulator.

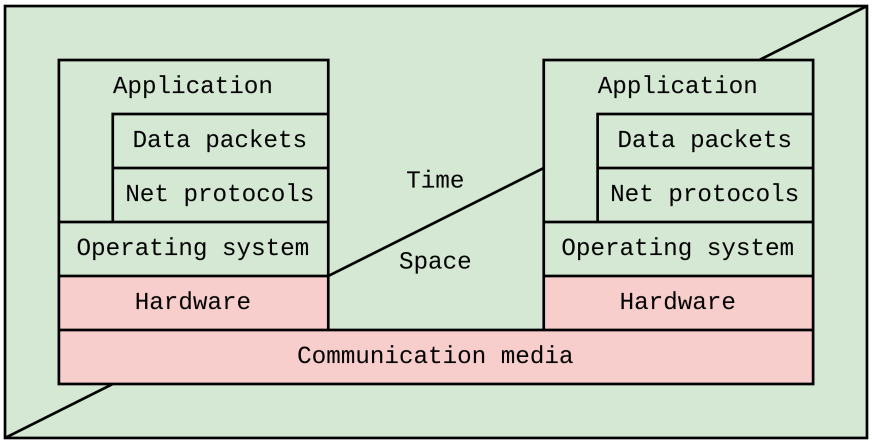


Figure 2.3: Modeling of network environments emulated using Mininet. Components in green are *real*; while red signifies that a component is only simulated.

Software-defined networks A second design principle of Mininet, which constitutes its distinguishing feature compared to other emulators, is the native support for SDN technologies. By default, switches emulated using Mininet are deeply programmable and can interact with an SDN controller to operate. This makes Mininet the natural choice for running experiments for technologies in the emerging SDN ecosystem.

Traffic shaping To simulate the communication media of a network, Mininet uses traffic shaping techniques to attribute special, user-defined characteristics to emulated wired links. By artificially delaying and dropping packets, Mininet can simulate links with specific parameters: propagation delay, link capacity, transmission speed, queuing, packet loss, packet corruption, packet duplication, etc.

APIs To perform network emulation, Mininet provides the user with two interfaces:

- A simple command-line interactive interface through which the user can emulate a network and interact with it by running commands and programs on emulated nodes;
- A programming interface for more sophisticated use cases. The user can describe in full details the elements of the network they wish to emulate (topology, link parameters,

etc.), as well as the scenario they intend to run on the network (synchronously or asynchronously running applications and services on emulated hosts, generating traffic, etc.). The scripts are then run by the Mininet engine without further interaction from the user.

Educational use Mininet is also designed for educational use. Its lightweight architecture allows it to run networking scenarios (ranging from simple client-server interactions, to complex data centre SDN networking) on personal laptops, which makes it ideal for learning very diverse networking concepts, particularly regarding SDN and NFV technologies. In fact, many online networking courses [35] rely on it for teaching computer networks to newcomers, and the well-known Stanford's advanced networking course [85] uses Mininet to teach graduate students about reproducing famous papers from prestigious networking conferences and journals.

2.3.2 Implementation

Mininet's implementation is limited to Linux, and thus its implementation relies on the operating system's toolset. The current version is available as a package for most Linux distributions and can also be complied from the source code. For inexperienced users who are anxious to run software that may break their system or that may use libraries with incompatibility issues, the developers also provide a virtual machine image with a ready-to-use Mininet installation and which can be deployed using any (hosted virtualisation-based) virtual machine manager. This is the safest way to start using Mininet as it would run on a controlled and isolated environment.

Network namespaces and cgroups To implement its limited isolation constraint, Mininet uses Linux's network namespaces to emulate virtual nodes. These allow for the creation of virtual domains running alongside the *root namespace* with their own network stacks: their

own (virtual) interfaces, IP addresses, hostnames, routing tables, firewall rules, etc. These network namespaces' interfaces are connected to each other using virtual Ethernet pairs (veth pairs) which are essentially simulated wired links that connect two interfaces.

For resource isolation, Mininet uses Linux's controlled groups (cgroups). This feature allows the limiting and isolation of a group of processes' resource consumption. In practice, an emulated end-system is assigned a control group with limited share of CPU utilisation, memory, disk I/O throughput, etc.

Open vSwitch and SDN controllers Open vSwitch [69] is an implementation of an SDN-enabled multilayer virtual switch. In general, this virtual switch can run on a switching hardware, but Mininet uses it as the default software to emulate networking nodes (regular switches, hubs, routers, etc.). OVS supports all standard layer-2 protocols: Spanning Tree Protocol (STP), VLANS, etc. and can also emulate other features of hardware switches, such as monitoring and management protocols (Netflow, sFlow, port mirroring, etc.).

But the main feature of OVS is its native support for SDN. All its versions implement the openflow protocol which lets OVS communicate and be programmed by an SDN controller, such as its own OVSController, Nox, and Ryu. These are available by default with Mininet, but any other controller can be used provided it goes through Mininet's API.

Traffic control Linux's traffic control (TC) subsystem [10] is a very powerful and versatile set of tools for granular traffic shaping, QoS management, and flow prioritisation. In its intended usage, it gives the user the possibility to separate traffic flows to accommodate multiple QoS requirements and avoid volatile contention for resources. It does so with a very sophisticated architecture whose main elements are queuing disciplines, classes, filters, classifiers, and actions. The arrangement of such elements allows users to implement a variety of traffic shaping, policing, and prioritising policies, which are very useful for system and network administrators, but can also be used on a regular personal computer.

Currently integrated into Linux TC, NetEm (network emulation) [42] is another set of tools for traffic management which is focused on network emulation: its main difference compared to regular TC is that it is explicitly focused on simulating traffic phenomena and not on optimising network resources usage. In particular, its addition to TC is artificially delaying, dropping, and reordering packets.

Mininet uses TC and NetEm to transform a simple virtual Ethernet pair connecting two virtual interfaces into a virtual link that behaves close to a real one:

- using TC queuing disciplines, particarly Hierarchical Token Bucket [24], Token Bucket Filter [52], and Hierarchical Fair Service Curve [78], it is possible to attribute a certain bandwidth to a virtual link;
- using TC classes, it is possible to add an artifical finite-size buffer to a virtual link;
- using network emulation, it is possible to add latency, lossiness, bit corruption, and reordering to a virtual link.

Mininet also allows the user to use other TC and NetEm features to program a variety of links with specific characteristics.

Python API Though a user can run an emulated network by combining all previously described tools that are readily available on most Linux installations, Mininet offers multiple abstractions to facilitate emulation. The user can describe the desired network topology and experimental scenario using Python objects and abstractions, then Mininet translates the description into commands and system calls to netns, veth, OVS, and TC. The main components of such abstractions are the following:

- the topology (`Topo` class in `topo.py`): is an object that describes a high-level abstraction of the network, viewed as a graph whose vertices are the nodes and whose arcs are the (unidirectional) links. Each vertex and each arc are labeled with a set of parameters that are interpreted by lower-level components of Mininet;

- the nodes (`Node` class and its `Host` and `Switch` children classes in `node.py`): are objects that represent the virtual nodes (end-systems and switches/routers) in an emulated network. They possess attributes that represent characteristics of the virtual nodes (hostname, CPU and memory limits, list of interfaces and their MAC and IP addresses, etc.) and methods that implement their behaviour (running a command, adding an interface, setting a routing table, connecting to an SDN controller, etc.);

- the links (`Link` class in `link.py`): in parallel to nodes, links are objects that represent virtual links in the emulated network. They also possess attributes that represent characteristics of the virtual links (propagation delay, queue lengths, bandwidth, loss rate, etc.) and methods that implement certain actions on them (attaching to a virtual node's interface, deletion, etc.);

- the controller (`Controller` class in `node.py`): is an object that represents an SDN controller for scenarios involving SDN. It contains all information about the controller (brand, IP address, etc.) and methods to implement actions on it (to start listening, etc.);

- the network (`Network` class in `net.py`): is the main object in Mininet. It represents the entire emulated network, and contains in data structures references to all its components (virtual nodes and links and, when relevant, SDN controller). It also implements many routines for acting on the emulated network and for interacting with its virtual components.

A Mininet script typically consists of creating a topology class that inherits from the abstract `Topo` class, and which overwrites its `build` method, where the user clearly describes the topology of the network they wish to emulate: adding a controller, and a set of hosts and switches with the desired parameters that are connected in a specific way. The user also writes a Python function describing the scenario (a certain host runs a service and a

number of hosts connect to it while their communication is monitored, for example). The Mininet "engine" then launches the necessary virtual components and runs the set of events described by the user.

The openness of Mininet, the simplicity of its use, and the freely available underlying tools it works with make it a good solution to design reproducible experiments. In principle, a researcher only needs their personal computer running an installation of Mininet to repeat any result produced by the emulator, provided the original researchers share the Python scripts. This has motivated the promotion of runnable papers [41]: scholarships published alongside the Mininet scripts used to produce their results, that any interested reader can reproduce in their own and without much struggle.

2.3.3 Limitations

There are two major obstacles constraining Mininet's performance and reliability as a network emulator, both are intrinsic limitations imposed by Mininet's design and implementation, and both have been extensively documented in the literature. On the one hand, as Mininet runs multiple virtual nodes in the same machine, the size of the network is heavily limited by the amount of physical resources the machine has; and on the other hand, the use of software tools (that also use computing resources to function) to simulate communication media lowers the accuracy of any obtained results. In a sense, Mininet essentially suffers two limitations: scalability and realism.

One study on the interplay between Mininet's performance and resource consumption empirically proved Mininet's scalability issues [67]. The authors have shown that although end-systems can be isolated using Mininet's control groups interface, their kernel threads and the OvS processes are impacted by the number of emulated hosts and their utilisation of the underlying resources. In particular, they conclude that the user should isolate the emulated hosts but should save at least two cores for the kernel and the emulated switches

to function properly. They have also studied the impact of scaling link characteristics on CPU usage. They have shown that a higher packet loss decreases the amount of CPU load because, though it adds a small amount of processing to randomly drop emulated packets, it decreases the total number of packets to be processed and forwarded by the switching elements. On the other hand, a higher emulated delay does increase the CPU load as it needs more processing time without necessarily decreasing the number of processed packets. Finally, without extensive and deep investigation, they have observed that the maximum aggregated throughput of all emulated flows in a Mininet scenario is roughly equal to the memory bandwidth measured by standard Linux tools (e.g., `mbw`). They have also shown that this maximum aggregated throughput decreases with a high number of hosts, switching elements, or links. The overall conclusion is that while Mininet scales well, it has its limits, which may bias the experiment results of an unprepared user.

To this end, the original creators of Mininet [41] proposed a set of new functionalities –some of which are implemented in the current version of the tool[1]– to mitigate to some degree these issues of fidelity. For instance, they suggested carrying the emulation while logging specific events, then checking that these events follow certain timing properties, called *network invariants*. For example, the link capacity is a network invariant that packet transmission events must satisfy: if a packet P of size $|P|$ is transmitted at t_1 through an emulated link with a capacity (or transmission speed) b, and received by the other end at t_2, then it must satisfy the link capacity inequality:

$$\frac{|P|}{t_2 - t_1} \leq b.$$

Another such invariant is switching speed. If a packet P is received by a switch from a certain port at t_1, it must be forwarded through another port at t_2, with a constant delay k that only depends on the forwarding speed of the virtual switch and not on the packets'

[1]Mininet v2.3.0: `https://github.com/mininet/mininet/releases/tag/2.3.0`

characteristics:

$$t_2 - t_1 = k.$$

However, no universal approach to fidelity monitoring that extensively reflects emulation accuracy and that relies on scenario-agnostic network invariants has been proposed.

2.4 Distributed Network Emulation

Distributed emulation attempts to overcome Mininet's limitations in terms of scalability (and reliability to some degree) essentially by running network emulations on top of a cluster of machines instead of a single personal computer. This allows more physical resources to be combined in order to sustain a much larger emulated network. Distributed emulation draws from the current broad paradigms in computer systems and networks: distribution, overlay networking, virtualised resources, and resource optimisation. It is built on the basic assumption that an emulated network can be cut into multiple slices (or subnetworks) that can run on independent physical machines, and which then can be stuck back together using overlay network technologies. In principle, this can go beyond the hardware limits of individual machines to support emulated networks infinitely large, provided enough physical machines are used. Again, the technologies and protocols to implement this are already supported by most operating systems but the challenge of optimisation and efficiency still remains broadly unsolved. This section will present the general principles of distributed network emulation and how each distributed network emulator has solved (or at least mitigated) the efficiency challenge.

2.4.1 Design Principles

Overlay virtual networks Overlay networking [21] is the core feature behind distributed network emulation. It is a concept that has been extensively studied in telecommunications

and computer networking literature, and that has been implemented in a variety of ways combined with very diverse technologies for different use cases. In the most general case, an overlay network assumes the existence of an underlying structure called the underlay network. The overlay is constructed by connecting nodes with logical links that span whole paths of the underlay network, and which define a completely different structure. By corollary, if the underlay network is connected (i.e., if any node can reach any other node), then it is possible to construct overlay networks with any topology. Otherwise, the set of possible topologies is limited by the connectivity of the underlying structure. To achieve this, many implementations rely on packet encapsulation: the overlay-level frames exchanged between two nodes connected by a logical link are encapsulated into protocol-specific packets that are used to transport them through the underlay network. The nodes then interpret and decapsulate them accordingly. In the example (Figure 2.4), the two switches communicate as if they are part of the same Ethernet segment, and the frames they exchange are encapsulated into IP packets that are routed by the underlay routers.

A special instance of overlay networking is in the contexts of virtualisation and cloud computing. In such settings, the overlay network is the structure that connects the virtual elements running on the physical machines which are connected by the underlay network. This enables the setup of a connected virtual infrastructure that spans multiple machines in the physical infrastructure.

Mapping The main challenge when distributing an emulation over multiple machines is that of the *mapping* (or *virtual network embedding (VNE)*) problem [36]: considering resource and connectivity limitations on the physical infrastructure, how should the emulated network be divided? and how to distribute its parts on the physical machines? This is a well-studied question in optimisation and graph theory, which can either be formulated as an optimisation problem (minimising a cost function or maximising a utility function under constraints of capacity) or as a satisfiability problem (finding one or more solutions that

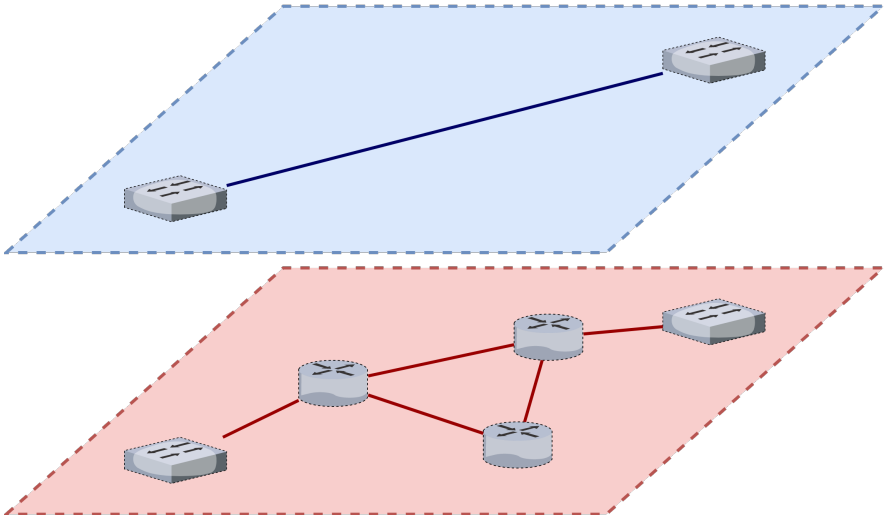


Figure 2.4: Example of an underlay (red) network and an overlay (blue) network. The overlay is one single Ethernet segment that connects the two switches and the underlay spans multiple subnetworks.

satisfy all constraints). This problem is proven to be NP-complete, which suggests that there is no known algorithm to solve it in a number of steps polynomial in the size of the virtual network[2]. Thus, finding an exact solution can be impossible even for fairly small virtual networks (less than a hundred nodes). However, multiple algorithms exist that compromise exactness for solvability: instead of finding an exact embedding, they would instead settle for a suboptimal solution, one that either satisfies most constraints or only approximately minimises/maximises the cost/utility function [50, 19].

2.4.2 Implementations

Mininet Cluster Edition Mininet CE [5] is the *official* distributed version of Mininet. Without changing the design nor redefining the core principles of Mininet, it simply adds the possibility of distribution, so that a network might be emulated on multiple physical machines. It is still in prototype phase and no complete implementation has been developed yet. In principle, it is designed to use basic tunneling protocols to connect nodes residing

[2]This does not prove, in the strictest sense, that no such algorithm exists (unless P=NP). Instead it suggests that the problem is as hard as the hardest problems in theoretical computer science.

on different physical hosts. As for virtual network embedding, it only offers infrastructure-agnostic solution: that is, Mininet CE does not distribute the emulated network on the physical resources by considering the available amount of resources. Instead, it only offers *dummy* placement strategies: random and round-robin embedding algorithms.

Maxinet Maxinet [82] is the fruit of one of the earliest attempts to develop a distributed network emulator. It is heavily inspired from Mininet, in its easy-to-use Python API, the native support for SDN, and the use of standard Linux tools. Not unlike Mininet, it allows the creation of an emulated network of end-systems, switches, and SDN controller(s). The main difference is its capability to be run using a cluster of physical machines to overcome the resources limitation of a single one.

Maxinet uses overlay networking to achieve customised connectivity of emulated nodes. In particular, any link that connects two virtual nodes hosted on different physical machines is emulated as a GRE [34] tunnel, which entails that all emulated packets are encapsulated in physical-level IP packets with GRE protocol headers containing identification and other optional information about the overlay logical link. This creates an overlay emulated network that connects the emulated nodes with the user-defined topology.

To embed the overlay emulated network on the physical infrastructure, Maxinet offers multiple placement strategies, none of which are exact solutions but rather simple heuristics that do not take infrastructure resources into account. In practice, Maxinet relies on *network* or *graph partitioning*: it divides the emulated network into a number of partitions with *equal* loads (measured by the number of end-systems, switches, and configured link capacities) and distributes them into the physical hosts. This implies that all infrastructure machines are considered identical and no attention is given to whether or not the underlay network can handle the aggregate traffic loads between the partitions.

Distrinet Distrinet [26] is one of the most powerful and comprehensive implementations of distributed emulation . It combines the good aspects of both Mininet CE (compatibility with Mininet's code and API) and Maxinet (intelligent partitioning and placement of emulated nodes and proper emulation of overlay links), while mitigating their flaws and limitations.

In particular, Distrinet uses the same Python API as Mininet. Thus any emulated network designed as a Mininet script can be ported to Distrinet with no adaptation: the user need only indicate the infrastructure layout in a separate file, as well as the number and IP addresses of the physical machines involved in the distributed emulation. To achieve this, Distrinet uses a distributed architecture with three components (Figure 2.5):

- worker nodes that run the emulated network's components: LXC containers [46], virtual switches and routers, and overlay links;
- a master node that manages the infrastructure and the emulated network through an overlay management network; and
- a client to which the user connects to submit the Mininet script and/or interact with the emulated network.

In Distrinet, the overlay virtual network is created using virtual Ethernet pairs for links emulated between nodes hosted by the same machine, and VXLAN [57] tunnels to connect interfaces of nodes hosted in different machines. Packets that cross the infrastructure network are thus encapsulated in VXLAN on UDP. Distrinet also creates a management network for the master to reach each emulated node individually, for running commands and for providing a transparent CLI to the user.

Distrinet's main strength is that it implements VNE algorithms specifically designed for distributed network emulation in a private cluster of machines or a public cloud [25]. When it is known, the user can feed the layout of the infrastructure (topology, link capacities, and machines' resources) as input to the algorithm which will dispatch the virtual elements ac-

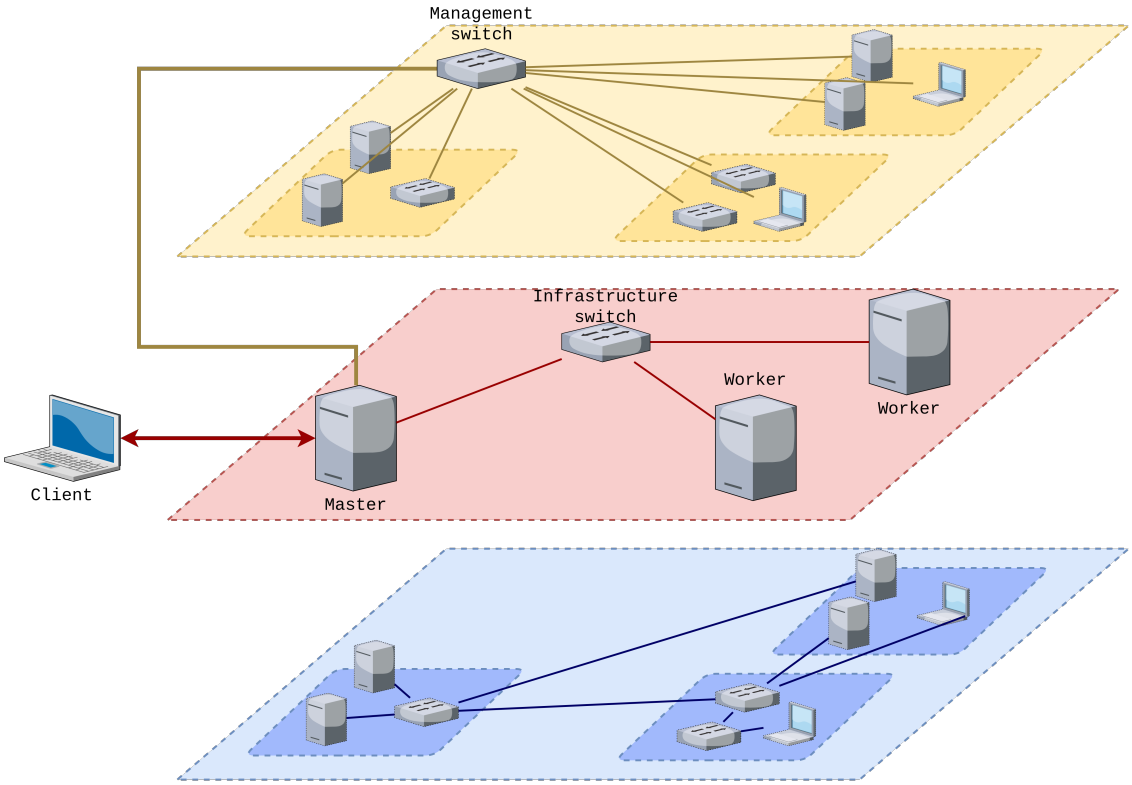


Figure 2.5: Distrinet architecture with an example infrastructure (red) and an example emulated network (blue). Each worker (including the master) runs a partition of the emulated network. The master orchestrates the emulation and the emulated nodes through the management virtual network (yellow).

cordingly; otherwise, as in the case of public clouds, the algorithm will minimise the number of elastic instances needed to run the emulated network, so as to minimise the total expenditure. The embedding algorithm relies on relaxations of the NP-complete VNE problem and uses heuristics that perform well in the context of distributed network emulation, as has been shown by its authors. It scales well and can achieve embedding of large-size emulated networks.

2.4.3 Limitations

The presented distributed network emulators run into some limitations only by virtue of their design. They also inherit some of the limitations of network emulation as a paradigm

(and container-based SDN-focused network emulation in particular). This section exhibits some of these limitations, which will be given more in-depth discussions in the next chapters.

Scalability The main purpose of using a distributed network emulator is to overcome the limitations of single-machine network emulation, namely the finite amount of resources for which multiple emulated components need to compete. Distributed emulation mitigates this issue by aggregating the computing resources of multiple physical machines. However, it also adds a new element to the stack: the infrastructure network which, depending on its capacity, can only handle a certain finite amount of emulated traffic. While the total traffic that a single machine can transfer in locally emulated links can reach few tens of gigabits per second, a physical network is generally limited to capacities an order (or few orders) of magnitude smaller. This constraint on traffic in the overlay emulated network generally limits the number of communicating nodes and amount of traffic that the distributed emulator can run on the given infrastructure, and may particularly limit how an emulated network can be embedded on the underlay infrastructure.

Fidelity Distributed network emulators answer fidelity issues to a certain degree, by eliminating all inaccuracies caused by overloading the finite computing resources of a single machine. As the emulator dispatches the virtual nodes over multiple hosts according to their capabilities, the possibility of failure as a result of strain is allegedly less likely.

However, no implemented distributed emulator has addressed the inherent lack of realism in simulating the communication media. Not unlike Mininet, all distributed emulators use traffic shaping tools and techniques to simulate the operation of real links, and as a result the accuracy of any distributed emulation is contingent on the realism of such traffic shaping techniques. In addition, the infrastructure network element also negatively impacts the accuracy of emulations. As emulated packets cross the physical network, they experience all its features and failures: queuing and delaying, packet loss, reordering, etc. Implementations

either encapsulate emulated packets in connection oriented (TCP) based tunneling protocols which mitigate packet loss through retransmission but add abnormal delay, or stick to connectionless (UDP or IP) tunneling which do not protect the overlay network from such infrastructure-level failures. We will identify and address these issues in extensive details in Chapters 5 and 6.

Reproducibility While a regular single-machine emulation, for example using Mininet, can easily be reproduced by anyone with a laptop and access to the scripts, a distributed emulation demands stricter requirements. Any interested researcher needs to have access to a cluster of machines of their own, perhaps with the same number of hosts, and the same infrastructure network capacity.

2.5 Summary

As a paradigm for network experimentation, emulation –and particularly virtualisation-based network emulation– is marketed as being reproducible, accurate, and compatible with recent technologies. It is the culmination of many advances in system and network virtualisation, which it relies on to achieve its objectives of hardware simulation that is transparent to upper layers.

Mininet is the most popular network emulator. It is built with few design principles (minimal host isolation, traffic shaping, easy-to-use APIs, SDN compatibility, and educational use) which it implements using already developed technologies natively supported by Linux (and other Unix-like operating systems). And while it does answer its own specification of reproducibility and flexibility, it nonetheless suffers from limitations of scalability and realism.

Distributed emulation enhances the scale capabilities of regular single-machine emulation by dispatching the task to multiple physical hosts. It offers all features of network emulation

but also inherits (and escalates) its lack of accuracy. This thesis is dedicated to mitigating distributed emulation's limitations using network measurements tools. Before presenting those contributions in Part II, the next chapter will discuss the necessary theoretical and practical background.

CHAPTER 3

DELAY MEASUREMENT AND NETWORK TOMOGRAPHY

Since the advent of the Internet, the number of connected devices and the amount of network traffic have massively increased both locally and globally [80]: local campus networks have been sustaining larger and larger traffic volumes as corporations and government agencies are more reliant on digital services; and the global Internet has become a complex web that connects almost every living person's many devices. This has made network and Internet measurements a critical task for traffic engineering, network management, and resource optimisation, which has created interest among network researchers for measurements and pushed the production of a large corpus of knowledge that assembles the expertise of many independent disciplines: system engineering, protocol hacking, classical and modern statistics, graph and optimisation theories, and many more.

In this chapter, we focus on two specific subproblems of network measurement whose solutions have proved to be helpful for our emulation fidelity monitoring and troubleshooting objectives: delay measurement and network tomography. The former is the practical problem of measuring network delay (or latency) in a precise and low-overhead manner, while the latter is a more theoretical question regarding what knowledge can be drawn about the internal components of a network from data collected at its edges. We will formulate each question around our specific context, and present the currently available solutions to address it.

3.1 Delay Measurement

3.1.1 Definitions and Modeling

Unlike the throughput which is a flow-level measure, the network delay is a value that characterizes either an individual packet or a pair of request-response packets. In general,

the packet delay is the amount of time needed for one or a pair of packets to travel from one point to another in a path of one or multiple physical media and eventually one or multiple intermediate nodes. From this general model, the network delay can be precisely defined along three axes:

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

For example, the classical definition considers a one-hop, hardware-level model of the one-way delay (OWD) [8]. In this definition, the OWD of a packet P between two machines A and B (which can be user terminals, servers, routers, switches, etc.) separated by a communication medium (wired or wireless) is the duration of (absolute) time between the instant when A sent the first bit of P, and the instant when B received the last bit of P. While this can be deemed a *pure* model of the *network* delay, as it does not involve any system-level latency, it is very hard to accurately measure. In fact, it inevitably requires using specialized network hardware to timestamp packets in order to measure their delays.

It is also possible to consider a more relaxed model of the one-way network delay, by defining it as the one-hop, system-level delay. This delay has the advantage of being measurable using simple software tools without the need for any additional hardware, and thus it can be easily measured in scenarios involving virtual and/or emulated machines and networking

equipment. This delay can be decomposed into three contributing terms:

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

In the case of wired media using FIFO queuing disciplines[1], this decomposition can be distilled into the following formula:

$$d(P) = \frac{|Q(P)|}{B} + \frac{|P|}{B} + \frac{l}{v}, \tag{3.1}$$

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

Note that in cases where A and/or B are virtual hosts, switches, or routers separated by a physical network (e.g., A is a virtual machine hosted in a physical machine, and B, a virtual switch hosted in a different physical machine), the delay needs to be measured between the

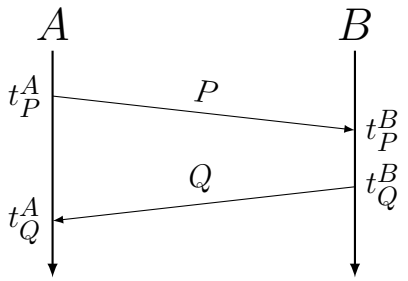
[1]This formula can be reshaped to model other queuing disciplines. For example, in a multiclass FIFO queuing discipline, $Q(P)$ can be redefined to be the subset of packets ahead of P which are from the same class.

virtual NICs of A and/or B, not the physical NICs of their hosting physical machines. Thus when virtualization is involved, the delay of a packet also accounts for the *system* delay between the virtual node's virtual NIC and the hosting machine's physical NIC.

An easier value to measure is the round-trip delay (RTD). The RFC 2681 [9] defines it for a pair of request-response packets P and Q as the duration of (absolute) time between the instant when A sent the first bit of P, and the instant when A received the last bit of Q. It is thus a round-trip, end-to-end, hardware-level model of the delay, and is in fact equal to the sum of the individual one-way hardware-level delays of packets P and Q, and the processing delay between the reception of the request packet by B and its sending of the response packet. Certainly, the information on the individual OWDs is lost when measuring the RTD.

This definition of the RTD can also be relaxed to make it measurement-friendly. It can instead be defined from a system-level point of view, and extended from simple request-response packets to almost any pair of packets. For a couple of packets P and Q such that P was sent from A before Q was received by A[2] (see below), the round-trip, one-hop, system-level delay is simply defined the sum of their individual one-way, one-hop, system-level delays, $t_P^B - t_P^A$ and $t_Q^A - t_Q^B$, without accounting for the *processing* time $t_Q^B - t_P^B$ by B between the reception of P and the sending of Q. The time elapsed between t_P^B and t_Q^B is not relevant in the general case since P and Q may not be correlated (unlike ICMP echo request-response, TCP SYN-ACK, etc. where the response is to be sent as soon as the request is received, and thus the in-between processing time is relevant as it informs about the response time of the destination host).

[2]For this definition of the RTD, only chronological order is required when considering a pair of packets P and Q: no causal dependency (e.g., Q being a response packet to P) is assumed.



3.1.2 Practical Delay Measurement

The use of ICMP echo probes is the de facto active method for measuring RTDs [70]. It works by sending a probe *echo request* ICMP packet and waiting for the destination to answer with an equal size *echo response* ICMP packet. The source timestamps the instant when the request packet is sent and the instant when the response packet is received, and reports the round-trip time (RTT) as the difference between the two. It accurately measures the round-trip, end-to-end, application-level delay with no need for time synchronization, and thus can be used in all cases without relying on external hardware. Other more powerful tools[3][4] can be used to send upper-layers probes (UDP, TCP, or application-level protocols).

But while actively measuring the round-trip delay does not entail any conceptual difficulty, one-way delay measurement between independent machines is intricately tied to the problem of clock synchronization [86]. Indeed, in a distributed network, each node has its own clock and therefore its own perception of time. To measure the one-way delay in such context, the source and the destination must exchange their timestamps of transmission and reception in order to compute the appropriate amount of *absolute* time. We know from theoretical physics that this is impossible to achieve with perfect precision[5]. For network

[3]hping: `https://linux.die.net/man/8/hping3`

[4]tcpping: `http://www.vdberg.org/~{}richard/tcpping.html`

[5]In his inaugural paper of special relativity [31], Albert Einstein made the observation that only the round-trip speed of light (and thus speed of any electric or electromagnetic signal) can be measured accurately, and postulated that in a symmetric trajectory, its value is the same in both directions. Many papers [7] have later tried (and failed) to find empirical evidence for that assumption. It is essentially the P=NP question of special relativity.

delay measurement, this is particularly challenging because the time dissimilarity between the clocks of different machines (called clock offset) also changes over time, i.e. even if perfect clock synchronisation is achieved at some instant t, it will be lost at $t + \delta t$ despite the clocks not moving relative to each other. This is due to differences between the clock frequencies (called clock skew) which are sensitive to physical phenomena (such as hardware heating [18]) that also change over time. This problem has been extensively studied in the scientific literature [86], and numerous protocols based on different sets of assumptions have been proposed to continuously resynchronize clocks of machines connected by LANs or WANs.

The Network Time Protocol (NTP) [64] is the most popular solution for clock synchronization. It organizes machines into a tree-like hierarchy, where the root node is the primary server which is generally connected to a highly reliable source of time (e.g., an atomic clock) and which will propagate its time to other nodes of the hierarchy through protocol messages; other nodes synchronize their clocks to the root server and eventually propagate the time to nodes in lower levels of the hierarchy. The process reiterates as clocks naturally drift from each other. At the convergence of the algorithm, each node will be synchronized to its server with a precision on the order of the network jitter. Thus, in an Ethernet LAN, NTP can theoretically guarantee precision down to 100 or even 10 microseconds, provided it is given long enough time to converge.

As applications in distributed systems have become reliant on finer levels of time synchronization, a more powerful protocol was proposed: the Precision Time Protocol (PTP) [30], also known as IEEE 1588. Just like NTP, PTP organizes nodes into a hierarchy of *masters* and *slaves* (where a node can be both a master and a slave) and uses protocol messages to exchange time information between nodes of the hierarchy. But unlike NTP, which can be implemented on any device with a Network Interface Card (NIC), PTP requires special NICs with integrated time clocks. This allows high-resolution synchronization by relying on the NIC clocks to timestamp protocol messages, thus avoiding all delays caused by software

and operating system-level processing.

In [54], the authors show that with proper configuration of NTP and PTP software in a local Ethernet network, it is possible to achieve precision on the order of 10 microseconds with NTP, and on the order of 100 nanoseconds with PTP, without incurring much overhead on the network. In fact, they show that by synchronizing clocks every 8 seconds with NTP, the total overhead of protocol messages is 23B/s per client and the one of computing resources is negligible; and by using PTP, the total network overhead is 186B/s per client, and the one of computing is also negligible.

The main drawback of active delay measurement methods is the use of probe packets: the measurement tool injects control packets into the network in order to estimate its delay. In addition to potentially disturbing the network, these tools only measure the delays of the probe packets. In Chapter 6 we will propose algorithms and tools for passive delay measurement of data packets.

3.2 Network Tomography

Network tomography [40] is a class of network measurement problems where the objective is to infer internal performance and characteristics from end-to-end measurements. Such characteristics can be either static (physical topology, link capacities and lengths, etc.) or dynamic information about the network (routing, network delay, link loss and usage, etc.). Depending on the assumptions made about the structure of the network, about the possible use of network probes, and about the arrangement of vantage points, these characteristics can be particularly difficult to (completely or partially) infer. This section will present the different theories underlying network tomography, with a particular focus on the tomography of delay.

3.2.1 Topology Inference

In computer networks, topology inference designates the problem of determining the structure and connectivity of a certain network, constituted by a set of end-hosts, internal nodes, and connecting links. For example, an ISP (or more generally an AS) can wish to infer the topology of the network of ASes of which it is part from the information received by its BGP routers [11]. However, in the context of network tomography, the available information is constrained to the point of view of the end-hosts: i.e., internal topology must be inferred using only information collectible in all or a subset of the end-hosts. This additional constraint makes the problem intractable, and many attempts have been made to propose active-measurement algorithms to solve it, eventually with minimal overhead to the network and in a reasonable amount of time.

Traceroute-based *Traceroute* [59, 12] is a network diagnostics tool that can determine the path (defined by the intermediate routers) between a source and a destination. It operates by sending increasing-TTL ICMP (or UDP/TCP) packets to the destination, which get dropped

at internal routers who respond with an *ICMP Time Exceeded* protocol message, effectively confirming their presence (by their IP address) on the path to the destination. Later works have critiqued the algorithm for its incorrectness in situations where the path between the source and the destination is asymmetric, and in the presence of load balancers [13]. These limitations have motivated the development of more sophisticated approaches, categorised under the umbrella term of *reverse traceroute*, the most famous [49] uses IP spoofing to kickstart the path discovery from the destination. From a network tomography perspective, Traceroute and derivatives can infer part of the internal topology when used between two end-hosts.

Large-scale projects have tried to map the Internet using Traceroute from and to multiple sources and destinations. In the early days of the Internet, the Mercator project [39] has been successful in carrying that out by augmenting Traceroute with a few heuristics in the selection of the source and destinations (informed random address probing), and by incorporating *source routing* to overcome Traceroute's bias in only reporting the shortest path. Skitter [17] is another project that uses Traceroute between multiple source and destination pairs to determine the overall structure of the network.

The main limitation of Traceroute-based (and more generally ICMP-based) approaches is that it assumes the cooperation of internal nodes, which must respond to end-hosts' requests and messages. As the Internet is growing in size and complexity, internal routers are less inclined to process certain messages [73]. Many researchers have made this observation and have worked on new approaches for topology inference. In [58], the authors build on the previous paradigm but present a heuristic, called *Max-Delta*, which selects a minimal set of pairs of end-hosts so as to avoid uncooperative, *anonymous* routers that do not respond to Traceroute messages.

Delay-based A later paradigm for topology inference relies on delay measurements between end-hosts in a network. A very recent study [29] aimed at determining the connectiv-

ity of internal links by using higher-delay statistics of end-to-end delay measurements. The authors describe *Möbius inference algorithm* (MIA), an algorithm that infers network structure, defined by a *routing matrix* (i.e., a correspondence between end-to-end paths and the underlying links that make them), from *cumulants*[6] calculated on the collected end-to-end delay measurements.

3.2.2 Delay Tomography

Along with topology inference and bandwidth estimation, many studies have worked on the problem of measuring the network delay on internal links of a network [22]. The delay has the particular property of being an additive measure: the end-to-end delay between two vantage points in a network is the sum of link-level delays over all links of the path [66]. This is in contrast to other metrics, such as the bandwidth, which cannot always be inferred from the measured end-to-end throughput. This property has motivated researchers to develop methodologies and systems to measure internal link delays under certain conditions and assumptions.

Active delay tomography In [71] and [76], the authors proposed an active measurement technique based on sending and receiving packets to and from vantage points in a network whose topology is known. The former study uses multicast probes sent from an end point (the root of the network's logical multicast tree) and received by other end points (the leaves of the tree). Each probe will disseminate in the tree and encounter a delay D_k in any link k it crosses, where such delays are assumed to be independent. When a packet arrives at a destination, a measurement of its end-to-end delay is made. Finally, by merging these measurements it is possible to reconstruct statistically good estimations of the link-level

[6] In probability theory, the cumulants $(\kappa_n)_{n\geq 1}$ of a probability distribution are values, similar to its moments, that fully describe it, in the sense that any two variables having equal cumulants are drawn from the same probability distribution. The advantage of cumulants over regular moments is their linearity on independent variables, i.e., the nth cumulant of a sum of two independent variables is equal to the sum of their nth cumulants.

delay distributions D_k.

On the other hand, in [76] the authors used unicast instead of multicast probing. While they concede that multicast measurements provide more correlated end-to-end measurements that can produce better estimations of internal link-level delays, the main drawback of such method is that multicast was in the way of becoming obsolete and more and more routers simply drop multicast packets, lowering the implementability of any multicast-based solution. A hybrid approach using multicast and unicast probes was proposed in [53] where unicast packets were used to fill in information about link delays that are not reachable by multicast packets.

Passive delay tomography More recent attempts at delay tomography leave out injecting probes into the network and instead rely on purely passive measurements on data packets from different vantage points, called monitors. The major challenge under such assumptions is the *identifiability problem*, i.e. the algebraic difficulty of finding a unique solution (the delay distributions of all internal links) from a set of end-to-end measurements (data packet delays between monitors) when such set does not span the entire space of possible paths [40]. In [56] the authors have focused on the optimal placement of such monitors, i.e. the selection of a minimum set of vantage points in a network from which passive measurements can completely and most accurately determine the internal links' delays.

Delay tomography is also a typical interesting problem in data centres where multiple virtual overlay networks controlled by different users share the same physical network resources, that the users may wish to infer in order to audit their SLAs. In such setting, the user can measure overlay-level delays that are mapped over a succession of physical-level link delays which they wish to estimate. This problem has been formulated in [72] where the authors have proposed a neural networks-based solution to work around the identifiability problem. They propose to feed their neural networks with simulated data at the learning phase, and prove in an emulated scenario that it can reach accuracy down to 10% error of

estimation.

Our work is built on similar assumptions: an emulated network which is overlaid on top of a shared physical network whose links' loads are to be inferred from delay estimations; only passive delay measurements are allowed in order not to interfere with the emulation; and the physical delays must be estimated under the identifiability constraint. However, as our delay tomography solution is intended for network emulation scenarios, the network traffics are typically too short to be learned by a learner or a statistical model. We will instead propose heuristics to work around the identifiability problem.

3.3 Conclusion

The following chapters will present our approach to the monitoring of emulation fidelity based on the measurement of packet delay, and to the troubleshooting of distributed emulation failures using network tomography. In this chapter we have given a coarse background on these problems and presented a few popular and relevant solutions. We will show how they can be applied to our specific scenarios and present our own adaptations and improvements.

Part II

Contributions

Posters and Demos

- Houssam ElBouanani, Chadi Barakat, Walid Dabbous, and Thierry Turletti, *Fidelity-aware distributed network emulation*, In proceedings of IEEE Conference on Standards for Communications and Networking (CSCN 2022).

Conference Proceedings

- Houssam ElBouanani, Chadi Barakat, Walid Dabbous, and Thierry Turletti, *Troubleshooting distributed network emulation*, 26th Conference on Innovation in Clouds, Internet, and Networks (ICIN 2023).

- Houssam ElBouanani, Chadi Barakat, Walid Dabbous, and Thierry Turletti, *Delay-based fidelity monitoring of network emulation*, Testbeds for Advanced Systems Implementation and Research (TASIR workshop), IEEE 15th International Conference on Communication Systems and Networks (COMSNETS 2023).

- Houssam ElBouanani, Chadi Barakat, Walid Dabbous, Thierry Turletti, *Passive delay measurement for fidelity monitoring of distributed network emulation*, In Proceedings of IEEE 20th Mediterranean Communication and Computer Networking Conference (MedComNet), Ibiza, Spain. Jun. 2021.

Journal Papers

- Houssam ElBouanani, Chadi Barakat, Walid Dabbous, and Thierry Turletti, *Passive delay measurement for fidelity monitoring of distributed network emulation*, Elsevier Computer Communications Journal (COMCOM), volume 195, pages 40-48. Nov. 2022.

CHAPTER 4

SCALABLE DISTRIBUTED NETWORK EMULATION

We have previously argued that while emulation overcomes some limitations of both traditional testbeds and simulation methods, it does not offer a perfect, fault-free service. In particular, we have identified two problems that limit the performance of modern virtualisation- and containersation-based network emulators: scale and accuracy. In this chapter, we will examine how the former is not yet fully overcome in practice by distributed emulation. We will present the case of Distrinet, which is proved to be the most scalable Mininet-like emulator, and build a new lightweight distributed network emulator from our proposed solutions to Distrinet's limitations.

4.1 The Case Against Distrinet

4.1.1 Design Flaws

Total Compatibility with Mininet One of the key features of Distrinet (Chapter 2) born out of its design requirement to be perfectly compatible with Mininet, is its implementation with Mininet's API. The rationale behind such choice is to appeal to users who are already familiar with the original emulator's interface, and not to burden them with learning a new API. While this has been a noble ambition, the fact of the matter remains that Mininet's API is only suitable for a laptop setting and was never meant to work perfectly in distributed settings. In particular, Mininet provides its users with a shell-like interface through which they can run commands in any emulated end-host or networking node they wish, in addition to new Mininet-specific commands for network management. In single-machine settings, this is straightforward as commands can be simply redirected to their destination's network namespace[1]. In distributed settings, however, the commands need to be sent to the desti-

[1]In Linux, to run a command `cmd` in a namespace `ns`, one simply has to run `ip netns exec ns cmd` from the root namespace.

nation containers for execution. Distrinet achieves this by creating a *management network* (Figure 2.5) that connects all running containers to the client through the master machine, and by maintaining open SSH sessions with each container. This approach incurs significant cost in terms of:

- Number of open SSH connections: for each running container one SSH session needs to be maintained. This impacts scalability as Linux imposes a hard limit on the number of simultaneously open SSH connections;
- Number of open files: for each running container, multiple Linux files need to be kept open, both for the SSH session and for the shell-like interface. This also impacts scalability as Linux imposes limits on the number of open files.

Heavyweight Containers A second design choice made by the creators of Distrinet is the use of a natively-supported, open-source containerisation solution: LXC (Linux Containers). This Linux-supported technology offers more isolation at the cost of more resource usage. In particular, LXC provides *system containers*, which are different from regular *application containers* in that the latter achieve the illusion of isolation by running applications (and the libraries on which they rely) on virtual environments that use the host machine's Linux kernel. Application containers are additionally offered their own network stack, their own file subtree (rooted somewhere in the host's tree), and their own share of the machine's resources. On the other hand, system containers are closer to regular virtual machines, as each container runs its own operating system, while sharing only a minimal part of the host's kernel. This makes the LXC containers quite heavyweight: they consume more, and need more time to be deployed and stopped. On a related note, LXC is not the most popular containerisation tool and is therefore less often improved and does not benefit from supporting technologies (such as Kubernetes, Swarm, OpenShift for Docker).

Containerised Virtual Switches Another important feature of Distrinet is the isolation of virtual switches inside their own containers. Unlike its ancestor and its competitors, Distrinet was designed with the choice to run virtual switches in isolated environments similar to the emulated end-hosts. This is unfortunately a double-edge sword: while it lets emulated networking nodes be isolated and thus specific amounts of resources be allocated, it incurs containerisation overhead and elongates the path to and from the switches' ports.

4.2 Bignet: a Scalable Distributed Network Emulator

From the observed flaws of Distrinet and its predecessors we build Bignet: a new distributed network emulator capable of overcoming the former's limitations. The remainder of this chapter will lay out its design principles and introduce the blueprints of a lightweight first implementation. The section will end with a comparison against Distrinet and demonstration of our solution's performance.

4.2.1 Design and Implementation

Lightweight Containers Bignet's main technological difference compared to Distrinet is its use of lighter containers. Although far from being the micro-containers used by Mininet to emulate end-hosts, the containers used by Bignet still consume less resources (computing, memory, and storage) compared to other distributed emulators that rely on heavy containers and/or virtual machines. This important technological shift increases scalability both by optimising resource usage and by reducing the amount of time needed for starting and stopping the containers. These Bignet containers can be prepared beforehand as images, and can be tailored for the emulated scenario and for each individual host.

In particular, Bignet uses Docker [16] to create application containers that emulate end-hosts. This lets users prepare the end-host's configuration and their software into Docker images which will be deployed for the corresponding emulated hosts. As for virtual switches

and other networking nodes, they are by default run directly on the physical machines to increase performance and scalability by reducing overhead, with the option to be containerised as in Distrinet if the user wishes for more isolation.

API Another distinctive characteristic of Bignet compared to other emulators is the deviation from Mininet's requirement for an interactive interface. Instead, Bignet focuses on a scripted scenario API: the user writes in great details the flow of events in scripts that the emulator will run non-interactively. By breaking off from this restrictive assumption, Bignet can be implemented without the need for a shell environment and live connections to the containers –which would need to stay open during the emulation regardless of whether or not commands are sent for execution. We acknowledge that this limits any potential use of our emulator for educational purposes, but it consequently focuses its abilities on large-scale research-oriented emulation. Thus more attention is given to such application, by providing additional routines for non-interactive actions on the containers (asynchronous commands, file download from and upload to the containers, etc.)

In addition, connection to the containers is not achieved by a management network. Instead, all sent commands and all received results and files go through the hosting machines which act as gateways. Specifically, Bignet only maintains SSH sessions with the physical machines, which are in charge of forwarding instructions to the containers via `docker exec` commands (Figure 4.1). This is an important key point and is essentially what makes Bignet much more scalable than Distrinet.

Mapping Bignet does not innovate in the emulated network mapping aspect of distributed emulation. Instead, it implements basic algorithms (random and round-robin mapping) as well as the optimised heuristics developed by Distrinet's authors. It also offers an interface for implementing new mapping algorithms, potentially ones custom-made for the desired emulated scenario.

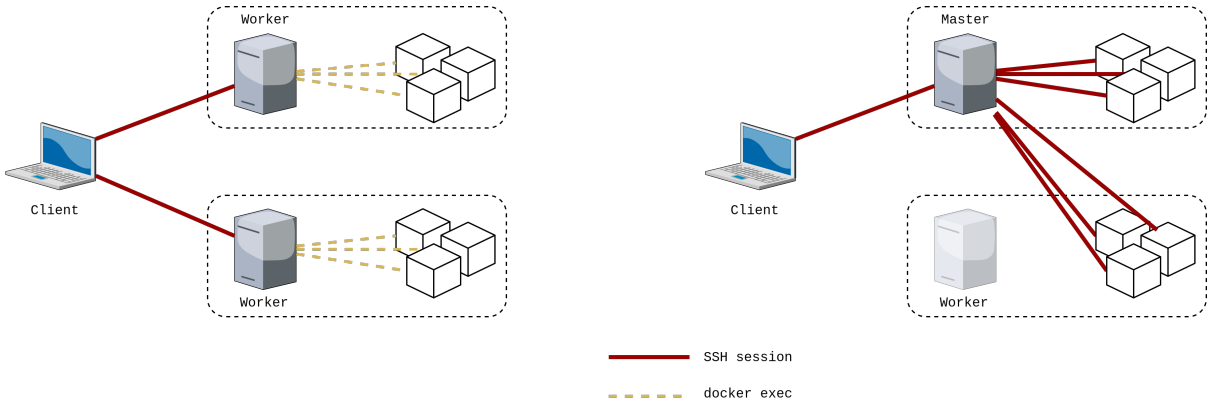


Figure 4.1: Command execution in containers in Bignet (left) and Distrinet (right). In Bignet, workers act as gateways to their hosted containers via the DOcker API (dashed yellow lines), which significantly reduces the number of open SSH sessions (red lines).

4.2.2 Performance Evaluation

In this section we present a series of performance evaluation tests to compare key performance indicators between Bignet and Distrinet. In particular, we show how the minimalist design and implementation of Bignet –particularly with regards to how virtual switches are not containerised and how the datapaths are somehow optimised– allows it to sustain larger emulated networks with more traffic and less delay.

Delay and Throughput To compare traffic emulation performance between Bignet and Distrinet, we conduct the following experiment on a single host[2]: a server sends a heavy TCP flow to a client through a variable-length network of cascading switches. All links are emulated with neither limits on bandwidth nor simulated delay, essentially to show the maximum speed and size of traffic that the emulated switches can forward. For each number of intermediate switches, we monitor the average throughput as well as the minimum delay over a 100-seconds long TCP flow.

Figure 4.2 shows the results. We can see how Bignet achieves (on average) more than Distrinet in terms of maximum throughput, and less in terms of delay. We also observe

[2]All experiments were conducted in the UVB cluster of Grid5000's Sophia site. More information about its hardware can be found at `https://www.grid5000.fr/w/Sophia:Hardware`.

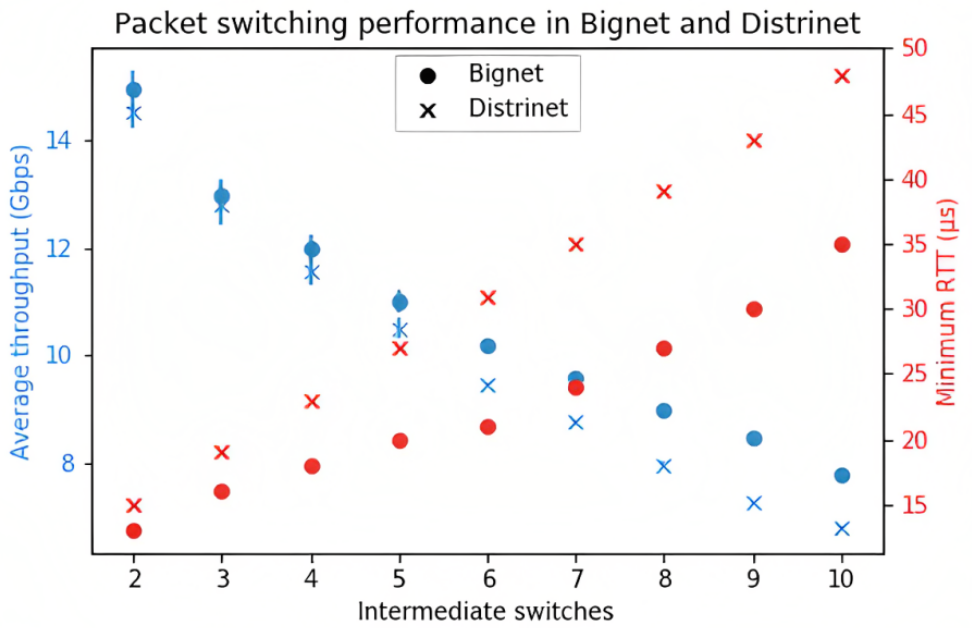


Figure 4.2: Maximum achievable throughput and minimum possible delay (y-axis) in a topology of many cascading switches (x-axis) emulated using Bignet and Distrinet.

that the gains in performance increase and the supremacy of Bignet gets more statistically significant as we add more switches, up to 1 Gbps more throughput and 15 microseconds less delay for a network of 10 intermediate switches. These gains in performance are mainly due to Bignet's shorter emulated data paths as the virtual switches run directly on the physical machines and do not need to be isolated inside their own containers.

Scalability The most important goal behind the development of this new lightweight distributed network emulator is to overcome the observed scale limits of Distrinet. To show how this has indeed been achieved by our shift from Distrinet's original design principles and implementation technologies, we perform another experiment: we emulate a variable-length linear topology[3] on a cluster of 10 machines using the same round-robin mapping algorithm[4].

[3]Mininet's famous linear topology is a cascade of n switches where one end-host is connected to each switch. A linear topology with parameter n therefore totals n switches, n end-hosts, and $2n-1$ links.

[4]The round-robin mapping algorithm distributes the emulated end-hosts and switches by cycling through the set of available hosting machines. In other terms, in an infrastructure of N machines, switch i and end-host i will be mapped to physical host $i+1 \mod N$.

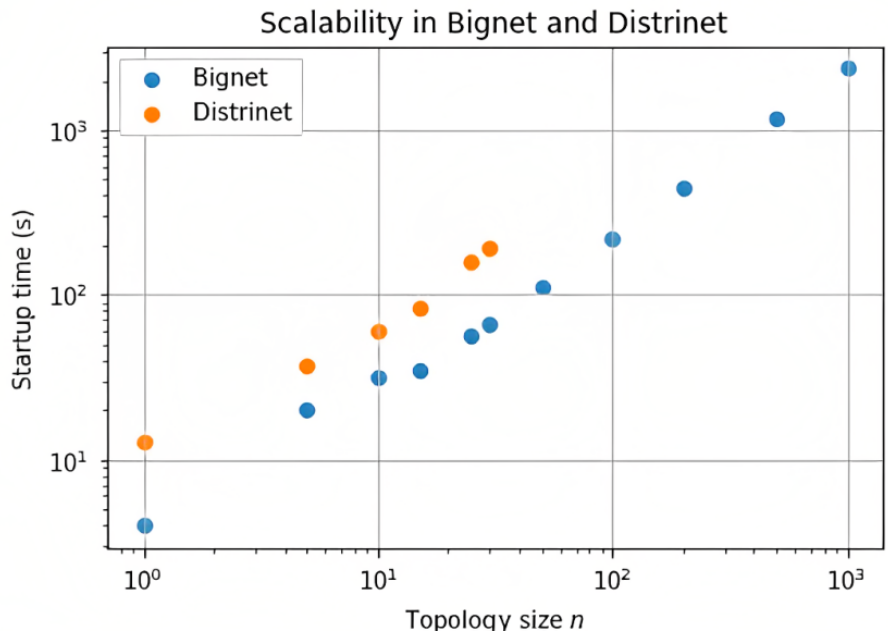


Figure 4.3: Startup time of variable-length linear topologies emulated using Bignet and Distrinet.

In this experiment, the key performance indicator is the start-up time, defined as the time required for all the nodes of the emulated network to start running, and which is considered infinite if it exceeds one hour or if the emulator fails to set-up the emulation and halts.

Figure 4.3 shows the results. Indeed, Bignet proves to be faster in setting-up the emulated networks for all sizes, and can set up networks of more than 2000 nodes (1000 end-hosts and 1000 switches), while Distrinet stops at less than 60 nodes ($n = 30$).

4.3 Conclusion

In this chapter we have presented Bignet: a lightweight distributed emulator inspired from Mininet (and particularly its distributed version Distrinet). Bignet is the result of a shift from Distrinet's design constraints and technological choices, and which trades off abstract values (compatibility with Mininet, use of Linux-native tools) for concrete increases in scalability and performance. We have shown that Bignet performs better than the most comprehensive

Mininet-like distributed network emulator in terms of traffic speed and deployment scale.

Bignet is not aimed at replacing Distrinet but only serves as a proof-of-concept to demonstrate how it could be improved to achieve its original goals: performance and scalability. In the next chapters, we will discuss the problem of emulation accuracy and will propose a framework for fidelity monitoring of network emulation. This will culminate in Hifinet (High-fidelity networks), an enhanced version of Bignet that implements the designed fidelity framework.

CHAPTER 5

FIDELITY MONITORING OF NETWORK EMULATION

In this chapter, we will present a theoretical framework for assessing the fidelity of network emulations, both single-machine and distributed. The end goal of such preliminary work is only to mitigate the inherent limits of realism by informing the user whether emulation failures during the experiment may have negatively impacted the results and, subsequently, whether the results should be discarded. We will first provide a conceptual and operational definition of emulation fidelity, then derive an axiomatic approach to fidelity monitoring through passive delay measurement, and finally conclude with examples of potential obstacles to emulation fidelity that our approach can detect.

5.1 Emulation Fidelity

Similar to network simulation, we have mentioned in the previous chapters that limited realism is one of the main issues with emulation. The understanding is that both of these paradigms only implement simplified models of the network environments they wish to replicate, and thus may leave out features and details which are assumed to be *irrelevant* but which can impact the experimented scenario and ultimately alter its results. In principle, the researcher is responsible for the results that the network emulator produces, and the burden of carefully and cautiously analysing the entire emulation process falls on them. However, as research becomes more and more specialised and siloed, the researcher may not possess the necessary technical knowledge about every component of the network and how it is emulated. Thus in practice, interpretation of emulation results is not properly conducted as separating emulation bias from experiment results is not easy. Hence the need for a framework to help evaluate the output.

5.1.1 Definition

Intuitively, fidelity (or realism) in the context of simulation and emulation is a measure of the quality of modeling and replication of networks and their components. However, it is a challenge to sketch a rigorous definition that is both *conceptual* –giving an abstract frame to reason about the concept of fidelity and its relationships to other concepts, and *operational* –allowing practical use in measurement and monitoring.

In the introduction, we have stated that empirical evaluation of any algorithm, protocol, architecture, or technology in the fields of computer networks and distributed systems requires testing it in the environments where it is intended to be implemented. But as we have argued, this is hardly ever possible, and researchers may instead need to resort to software solutions that provide facsimiles of *real* environments. For instance, in the case of network emulation, the evaluation of a certain solution A on a network N (whose results are denoted as $A(N)$) can be substituted by the evaluation of A on an emulated network N'. If we assume that there is a *comparison* function d that compares the results $A(N)$ and $A(N')$ (e.g., statistical metrics that compare values against ground truths such as precision and recall), then *emulation fidelity* can be viewed as a function ϕ that compares N and N' and which is positively correlated with d, i.e.,

$$\phi(N, N') \geq \phi(N, N'') \implies d\big(A(N), A(N')\big) \geq d\big(A(N), A(N'')\big),$$

for any two emulated networks N' and N'' from a real network N, and for any solution A implemented on N, N', and N''. Although such a definition of fidelity captures all intuition about the concept, as it would measure the similarity between a real network and its software replica, it is very difficult to implement in practice, and thus is not operational as we wish it to be. We will characterise any such definition as being *noumenal* [1] , because it evaluates the

[1] Our use of *noumena* and *phenomena* is analogous to kantian metaphysics [48], where a noumenon captures a thing-in-the-world *in-itself*, i.e. its essence independently of human perception, and where a

similarity between the objects (real and emulated networks) *in themselves*, independently of any higher level behaviour (in particular as perceived by implementing solutions A).

Conversely, a weaker definition of fidelity could instead only consider certain aspects of the real and emulated networks, which are perceived as phenomena by the user and which can be modeled, measured, and compared to assess similarity. Essentially, this technique projects the networks onto a plane of measurable phenomena that do not capture their full essences but that contain enough information to evaluate their resemblance to some degree. Thus any such definition of fidelity will be characterised as being *phenomenal*. As well as being conceptual definitions, these definitions also possess the additional advantage of being operational by design, as they can be measured and monitored exactly through the set of phenomena around which they are defined. The next paragraphs will present various phenomenal evaluations of fidelity.

5.1.2 Phenomenal Assessment of Emulation Fidelity

Phenomenal fidelity can be defined by choosing a subset of metrics from a large set of measurable phenomena. Some are specific to the solution A and the experimented scenario: if the experiment involves large flows, for instance, the user can monitor the network throughput and make sure that it fully uses and does not exceed links' bandwidths; if A uses TCP, they can monitor the evolution of the congestion control parameters (receiving window and congestion window); etc.

On the other hand, and knowing that network emulation is defined by the simulation of the hardware and of the communication media, it is sufficient to only consider these components for the evaluation of fidelity. We can thus choose network phenomena that directly manifest their behaviour. These network phenomena are especially convenient because they are *universal*, as they do not depend on the solution A and the experimented scenario, and

phenomenon is the projection of the thing-in-the-world onto the human mind and its a-priori features of perception (e.g. abstraction, causality, space, and time).

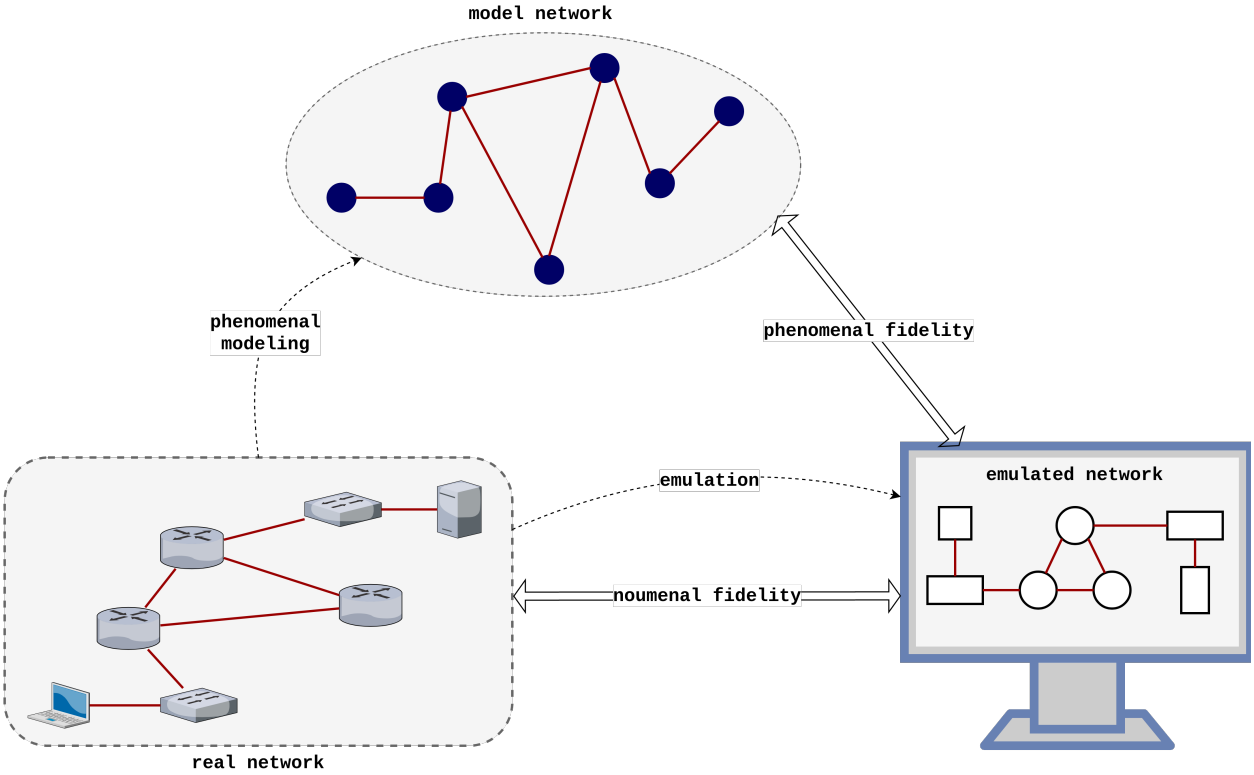


Figure 5.1: Noumenal fidelity evaluates the conformity of the emulated network to the real network, while phenomenal fidelity evaluates its conformity to a phenomenal model defined by aspects and metrics that are observable and measurable.

do not particularly raise challenges in their implementation. They are the *link bandwidth*, the *packet loss*, the *packet ordering*, and the *packet delay*.

To better define fidelity with regards to these phenomena, we first consider a phenomenal model of a network, defined as a set of unidirectional links (the communication media). Each link is defined by:

- Two ends (1) (transmission) and (2) (reception);
- A bandwidth B, a propagation delay π, and a loss rate λ;
- Three finite sets of packets $\mathcal{P}_0$, $\mathcal{P}_1$, and $\mathcal{P}_2$, containing the generated (from the upper layers), transmitted, and received packets respectively, and such that $\mathcal{P}_2 \subseteq \mathcal{P}_1 \subseteq \mathcal{P}_0$, additionally equipped with a size operator $|\cdot|$, and such that:

$$\frac{|\mathcal{P}_0| - |\mathcal{P}_1|}{|\mathcal{P}_0|} = \lambda; \tag{5.1}$$

- Two clock functions $t_1 : \mathcal{P}_1 \to \mathbb{R}_+$ and $t_2 : \mathcal{P}_1 \to \mathbb{R}_+ \cup \{\infty\}$, giving the timestamps of enqueuing of packets at the transmission and receiving at the reception, with $t_2(P) = \infty$ for all packets $P \notin \mathcal{P}_2$;
- A function $\delta : \mathcal{P}_1 \to \mathbb{R}_+$ associating to each packet P its delay $\delta(P)$, and which must satisfy the recursive formula for any two successively transmitted packets $P_i, P_{i+1} \in \mathcal{P}_1$:

$$\delta(P_{i+1}) = \pi + \frac{|P_{i+1}|}{B} + \max\left[0, \delta(P_i) - \pi - (t_1(P_{i+1}) - t_1(P_i))\right]. \tag{5.2}$$

The last equation is essentially a recursive formulation of the delay model presented in A.1. It states that the delay of a packet P_{i+1} is the sum of its propagation delay π along the link and its transmission delay by the hardware if it is not to wait in the queue; and otherwise is equal to the sum of its transmission delay, the total delay of the previous packet

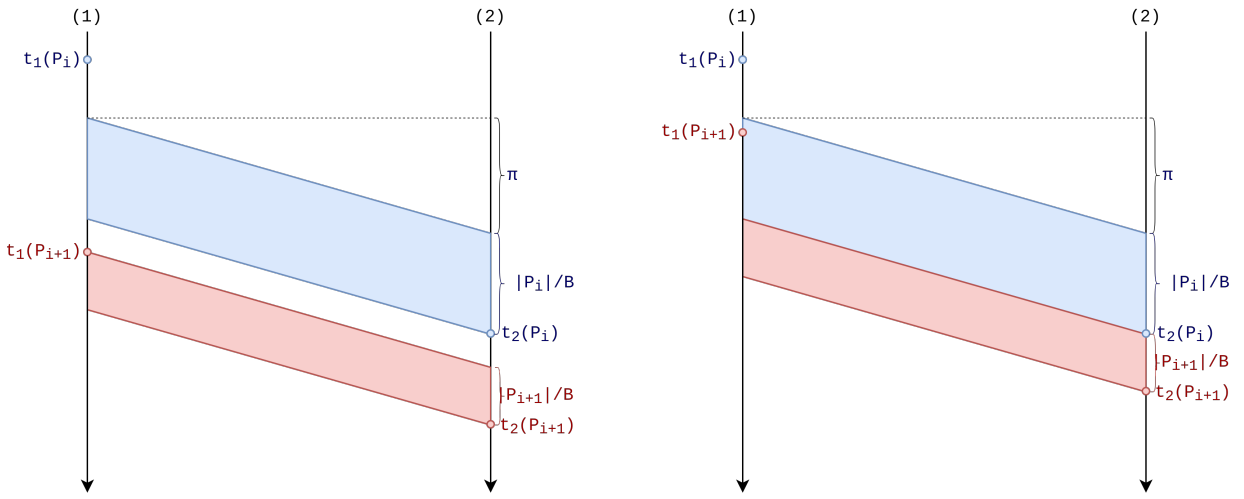


Figure 5.2: Queuing, transmission, and reception of two successively sent packets P_i and P_{i+1} in two cases: packet P_{i+1} is not queued (left); and P_{i+1} waits in the queue before transmission (right).

P_i minus the duration of time that the previous packet has spent in the queue before P_{i+1}'s arrival (see Figure 5.2).

Link bandwidth Link bandwidth is one of the most basic link-level network phenomena to monitor in order to assess the accuracy in emulating a wired communication medium, and by extension the fidelity of network emulation. Without knowledge about the rate at which packets are generated and prepared for transmission by the link, it is not possible to verify that the link capacity is fully utilised at all times. However, it is possible and easy to check that said capacity is not exceeded. In our phenomenal model of the network link, this can be formulated as follows:

Definition 1. *An emulated link is said to be* ***bandwidth-accurate*** *if, for any two successive packets* $P_i, P_{i+1} \in \mathcal{P}_1$,

$$t_2(P_{i+1}) - t_2(P_i) \geq \frac{|P_{i+1}|}{B},$$

which essentially ensures that no two packets are received faster than the bandwidth permits.

The examples in Figure 5.2 show the case of equality (right) and strict inequality (left).

Packet loss Packet loss is another basic link-level network phenomenon that can be monitored for link accuracy assessment. One may simply count the ratio of lost packets (generated but not received) and compare it to the loss rate of the emulated link. In our model, this can be formulated as:

Definition 2. *An emulated link is said to be* ***loss-accurate*** *if*

$$\frac{|\mathcal{P}_0| - |\mathcal{P}_2|}{|\mathcal{P}_0|} = \lambda.$$

Note that such formulation is ideal as in practice the ratio can, at best, only approach the configured loss rate as the size of $\mathcal{P}_1$ approaches ∞. Another important caveat is that only losses due to the unreliability of the communication medium should be accounted. In other words, any loss due to congestion should be recognised separately.

Packet reordering Our phenomenal model of the link also allows packet reordering to be a metric through which link emulation accuracy can be assessed:

Definition 3. *An emulated link is said to be* ***order-accurate*** *if, for any two packets* $P_i, P_j \in \mathcal{P}_1$,

$$t_1(P_i) < t_1(P_j) \implies t_2(P_i) < t_2(P_j),$$

which ensures that packets are always received in the order in which they are transmitted.

Packet delay The last metric our phenomenal model can define is delay accuracy. Its definition within our model is straightforward:

Definition 4. *An emulated link is said to be* ***delay-accurate*** *if, for any packet* $P \in \mathcal{P}_1$*:*

$$t_2(P) - t_1(P) = \delta(P).$$

5.2 Delay-based Fidelity Monitoring

The previous definitions of link emulation accuracy can be seen as different phenomenal aspects of emulation fidelity of communication media. The four metrics can be monitored separately or jointly to provide evidence of good phenomenal fidelity, but they are in fact loosely correlated. Indeed, we will prove in this section that the delay is the finest of the four phenomena, in the sense that it captures information contained in the others, and develop our fidelity monitoring framework around it.

Definition 5. *A phenomenon* X *is said to be* ***finer*** *than a phenomenon* Y *if* X*-accuracy necessarily ensures* Y*-accuracy.*

Properties 1. *1. Delay is finer than bandwidth.*

2. Delay is finer than order.

3. Delay is finer than loss.

Proof. 1. Suppose that an emulated link is delay-accurate. Let P_i and P_{i+1} be two successively transmitted packets. From 5.2, we have:

$$\delta(P_{i+1}) \geq \pi + \frac{|P_{i+1}|}{B} + \delta(P_i) - \pi - (t_1(P_{i+1}) - t_1(P_i)),$$

which simplifies to:

$$\delta(P_{i+1}) - \delta(P_i) + (t_1(P_{i+1}) - t_1(P_i)) \geq \frac{|P_{i+1}|}{B}.$$

Since the link is delay-accurate, we also have:

$$t_2(P_{i+1}) - t_2(P_i) = \delta(P_{i+1}) + t_1(P_{i+1}) - \delta(P_i) - t_1(P_i),$$

and therefore

$$t_2(P_{i+1}) - t_2(P_i) \geq \frac{|P_{i+1}|}{B}$$

which concludes the proof.

2. To prove that delay is finer than order, it is sufficient to prove that the order is not broken for pairs of successive packets. Suppose that a link is delay-accurate and let P_i and P_{i+1} be such a pair. From property (1), we know that the link is also bandwidth-accurate, and thus

$$t_2(P_{i+1}) - t_2(P_i) \geq \frac{|P_{i+1}|}{B} > 0,$$

which concludes the proof.

3. Suppose that a link is delay-accurate. Loss-accuracy is equivalent to proving that all losses occur between generation and transmission and that no additional packet is lost between the transmission and reception, i.e. $\mathcal{P}_1 = \mathcal{P}_2$. Let P be a packet in $\mathcal{P}_1$. Since the link is delay-accurate,

$$t_2(P) = t_1(P) + \delta(P) < \infty,$$

and thus $t_2(P)$ is well defined in $\mathbb{R}_+$ which, from our model, supposes that $P \in \mathcal{P}_2$ and concludes the proof.

□

These proofs show that all the considered network phenomena can be brought down to delay, which triumphs as the finest metric for link emulation accuracy –and thus network emulation fidelity– evaluation. Although this has only been formally proven within the frame of our approximate theoretical model, it demonstrates a broader truth: that all link-level network phenomena are ultimately features of time. Packet reordering can be understood as an inconsistency in the timestamps recorded by the two ends of the link; packet loss as infinite delay; and even bandwidth can expressed in terms of time. Knowing this, the delay imposes itself as the supreme criterion for phenomenal evaluation of fidelity.

The corollary of all the above reasoning is that delay is a good network metric on which we can build a definition of phenomenal fidelity. In principle, it is both conceptually and operationally valid. But in practice, however, it is difficult, and arguably impossible, to judge fidelity based on the ideal equation between timestamp difference and modeled delay. Neither emulators nor systems of measurement can achieve such perfection. Thus we first redefine delay-accuracy to allow some margins of inevitable error:

Definition 6. *Let $\gamma, \epsilon \in (0, 1)$. An emulated link is said to be (γ, ϵ)-**delay-accurate**, or **probably approximately delay-accurate with parameters** (γ, ϵ) if, for any packet $P \in \mathcal{P}_1$ and with probability at least $1 - \gamma$,*

$$|(t_2(P) - t_1(P)) - \delta(P)| \leq \epsilon.$$

In higher-level terms, the principle can be formulated as the following criterion:

Criterion 1. *For an emulation to have good fidelity, the deviation of the measured delays from the model delays of a sample set of packets throughout the duration of the experiment should not be too large.*

The criterion for fidelity raises important caveats regarding its implementation that we will discuss individually in the next chapter. The following section, last in this chapter, will be dedicated to citing examples of fidelity failure that manifest in the packet delays.

5.3 Typical Sources of Delay Emulation Error

In this section we show three typical causes of delay emulation error, either due to an inherent design of the emulators, or to the hardware in which they are intended to run. The first is the overload of computing resources that has been extensively studied about Mininet in the literature; the second is Mininet and its variants overlooking by default the precise emulation of the transmission delay; and the third is the additional delay caused by the physical network infrastructure in the case of distributed emulation.

5.3.1 CPU overload

Many previous studies have focused on the impact a lack of computing resources and their contention can have on Mininet's fidelity and effectiveness [65]. It is indeed reasonable to expect that multiplexing multiple, and often very numerous, virtual hosts and network nodes on top of the same, often limited, computing resources can have a negative impact on accuracy. Thus, any user who relies on emulation to run experiments and to measure performance indicators (network throughput, delay, etc.) must be careful while emulating computing- and traffic-intensive scenarios.

To demonstrate how much an overloaded CPU affects the emulated network delay, we emulate the following simple network in a distributed emulator (Figure 5.3): two virtual hosts H_1, H_2 are connected by a cascade of $N > 1$ virtual switches $S_1, ..., S_N$, where all links are configured with a capacity of 1 Gbps and a propagation delay of 1 ms. The virtual hosts are located in one physical machine, and the switches in a second. On this emulated testbed we run two scenarios: in a first scenario the virtual hosts simply exchange small

size (90 bytes) ICMP echo request/reply packets at a rate of 10 packets per second; while in the second scenario, H_1 also sends a heavy Iperf[2] TCP flow to H_2. We also run each of these scenarios in two different settings: in a first setting only the emulator threads and basic background kernel functionalities are running (low CPU load); while in the second setting we run CPU- and memory-intensive user processes to overload all cores of the machine hosting the virtual switches. The objective of such a design is to see how the lack of computing resources negatively impacts the emulation of networking components (switching nodes and network links) even when the applications that generate the packets (Ping and Iperf) run smoothly in a separate host. Our performance indicator of interest is the round-trip time (RTT) reported by Ping, which corresponds to the round-trip application-level delay between virtual hosts H_1 and H_2.

Figure 5.4 shows the results. Under low CPU load, the RTTs linearly increase with the number of intermediate switches as each link adds 1 ms of propagation delay in each direction, as well as a relatively low queuing delay in the presence of Iperf traffic. However, when the virtual switches and the emulated links experience low amounts of available CPU resources and memory bandwidth, the results are a very high increase in reported delay, especially when parallel network traffic is emulated. These results are not unexpected, but such a large impact of resource shortage, especially on delay emulation, has not been appropriately documented in the literature.

5.3.2 Non-emulation of Transmission Delay

Another source of emulation inaccuracy of Mininet and its derivatives is how they overlook the correct emulation of the transmission delay of packets. Indeed, while Mininet can use TC-Netem to delay packets by a fixed (or random) value, it does not do so based on their sizes, which is the distinguishing feature of the transmission delay. Instead, as it relies on the

[2]Iperf3: `https://iperf.fr/`

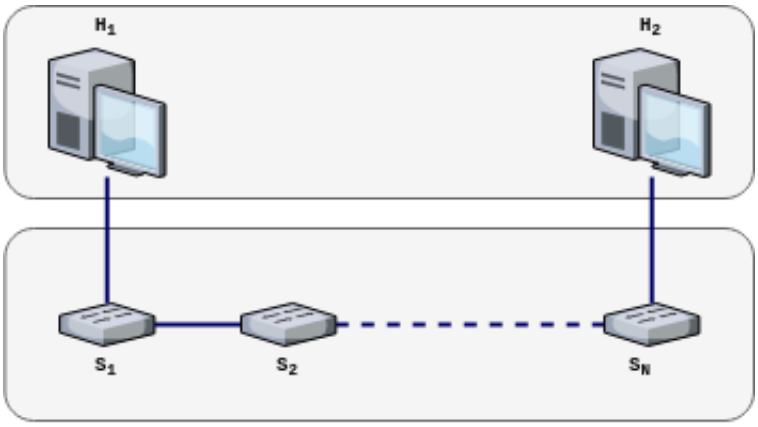


Figure 5.3: Emulated testbed. The virtual hosts H_1, H_2 and the virtual switches $S_1, \ldots, S_N$ run on two different physical machines.

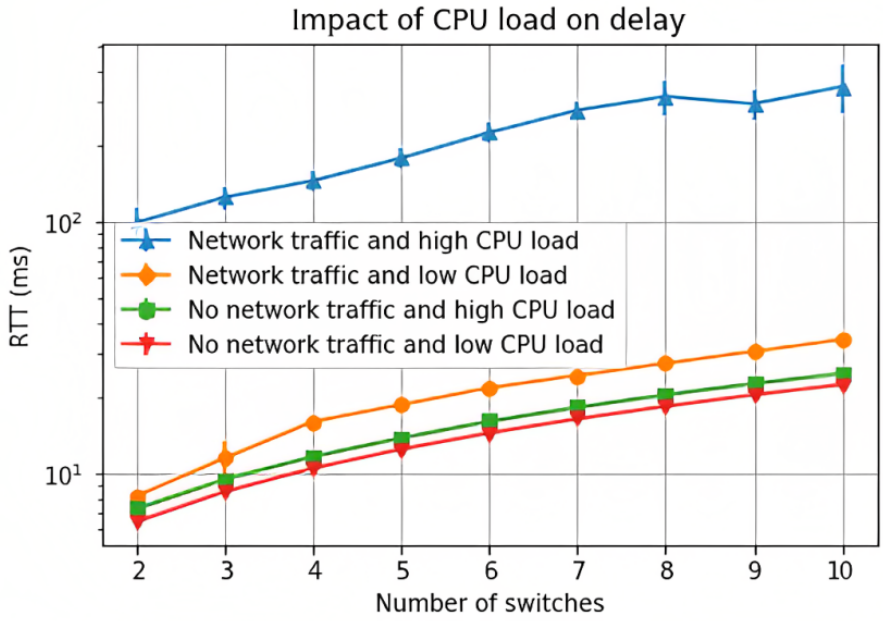


Figure 5.4: Reported Ping RTT vs number of virtual switches N.

Hierarchical Token Bucket (HTB) [24], Token Bucket Filter (TBF) [52], and Hierarchical Fair Service Curve (HFSC) [78] scheduling and queuing models, the *service time* is incurred on subsequent packets and not on the head-of-line packet itself. This error is harmless enough for the emulation of high-speed networks (less than a millisecond for regular-sized Ethernet packets in 100 Mbps and 1 Gbps links), but it can give biased results in low-bandwidth scenarios where the application-level QoS depends on the variation of the delay between packets. Thus the user must go beyond the available API to implement transmission delay if they need it in their emulation.

To see this error in practice, we emulate the same simple network as above: two hosts connected to two switches that are linked through one 10 Mbps link of 1 ms propagation delay. We then simply send ICMP echo request/reply packets of different sizes (1000 packets per size) at relatively large interarrival times (i.e., at a rate much lower than the link's capacity in order to avoid queuing) from one machine to the other, and log the measured packets' delays. These delays are measured at the application-level. Figure 5.5 shows the results. We can clearly visualize how the measured delay does not increase with the sizes of the packets: in fact their Pearson correlation coefficient is less that 5%. This non-consideration of the transmission delay can cause delay differences up to 2.2 ms for a pair of 1500 bytes packets on one single link.

However, it is possible to make use of mechanisms already available in TC-Netem to correctly emulate this missing delay. We propose a Mininet patch [3] to integrate this functionality. Figure 5.5 shows how using this patch can correct the error (the green curve). Note that the "expected delay" does not take into account system delays that are included in the "measured delay", hence the constant gap between the orange and green curves.

[3] `https://github.com/distrinet-hifi/mininet`

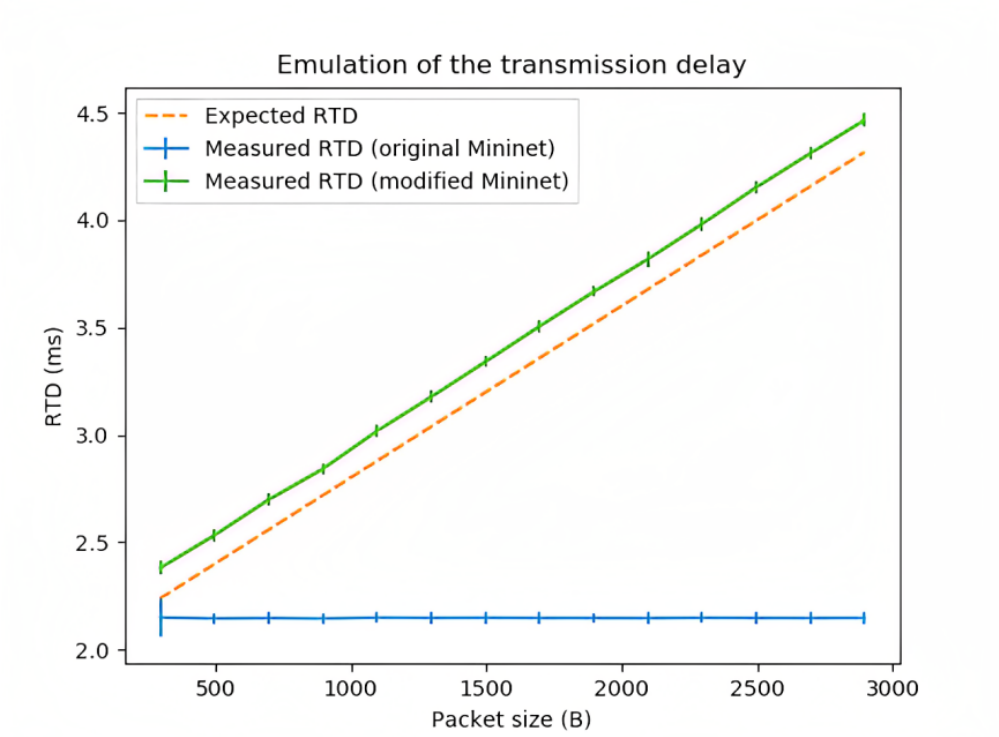


Figure 5.5: The blue line represents the average measured RTT and its confidence interval (y-axis) over all pairs of packets of same total size (x-axis) in the scenario emulated using the current official version of Mininet; the green line plots the average measured RTT and its confidence interval using the patched version of Mininet; the orange line plots the expected RTT using the model given in equation (A.1). The confidence intervals are invisible due to the small variance and the large number of the measurements.

5.3.3 Physical Network Delay

In the case of distributed emulation, computing resources are not much of an issue, as users can simply scale up their hosting infrastructure by adding more machines. However, in such settings the infrastructure network adds delay to the links that are emulated over it. This delay manifests itself in two ways which cause two different types of errors:

- The first type of error is due to the additional delay added to all the packets of a virtual link emulated over the infrastructure network. For example, on a virtual link connecting two switches emulated in two different machines separated by a physical network, all the packets will be delayed according to the propagation delay and the transmission speed of the virtual link, as well as according to those of the physical network. While the former is the correct delay to be expected, the latter is undesirable noise. The value of this error depends on the characteristics of the physical network, and can reach a few milliseconds in regular cluster networks. This again might seem negligible, but it can easily add up to tens or hundreds of milliseconds for large diameter emulated networks which is enough to cause abnormal behaviour and thus lead to biased results for scenarios involving delay-sensitive applications;

- The second type is due to the multiplexing of the virtual links by the infrastructure network that it hosts: as packets from different links share the same medium and the same queues to access that medium, they mutually add delay to each other. Consider for example a scenario where two virtual links with equal bandwidth of 500 Mbps share the same physical link of 1 Gbps capacity. Normally, packets from the same virtual link are first queued in the virtual interface which has a speed of 500 Mbps, and therefore should never be queued at the faster physical interface. However, because there is another virtual link emulated over the same physical link, queuing will inevitably happen, which will add considerable delay to packets and bias the emulation. The network embedding algorithms make sure not to emulate over physical links multiple

virtual links with total bandwidth exceeding the underlying physical capacity, but mutual added delay can reach high values even when this constraint is satisfied.

5.4 Conclusion

In this chapter we have presented and theorised the notion of *emulation fidelity*, which intuitively measures the similarity between real and emulated networks. We have argued that this notion can be defined either by considering the inherent characteristics and features of the networks in themselves, giving us the concept of *noumenal fidelity*; or by considering their higher-level operation and behaviour which can be observed and measured by a human user, defining the concept of *phenomenal fidelity*. We have argued that the latter provides both a conceptual and operational definition of fidelity, which we have attempted to define by contemplating a set of communication media-level network phenomena: link bandwidth, packet loss, packet order, and packet delay. Using a symbolic formalism, we have laid down the relationships between these phenomena and concluded that the packet delay is the finest metric for emulation fidelity. As a result, we have built a theoretical framework for delay-based fidelity monitoring, whose implementation will be discussed extensively in the next chapter.

We have also presented three typical sources of emulation *in*fidelity, which manifest as errors in the emulation of packet delays. These failures can be categorised into two classes: failures due to hardware infrastructure (contention and underlay communication media), and failures due to design flaws (improper emulation of network delay).

CHAPTER 6

IMPLEMENTING FIDELITY MONITORING OF NETWORK EMULATION

The last chapter served as a preliminary and theoretical presentation of our framework for emulation fidelity and its monitoring by looking at emulated packets' delays. We have broadly defined what fidelity is and argued that packet delay is its best measure among a set of other phenomenal metrics. This has resulted in Criterion 1 which postulates a necessary condition for emulation correctness.

6.1 Delay Measurement for Fidelity Monitoring

To implement passive delay measurement in distributed emulated networks which can generate and transport high speed, low latency traffic, is a complicated task due to requirements for low overhead and high precision. Indeed, to monitor the fidelity of emulation, comparing passively measured packet delays to their model values must be accomplished with surgical precision and without interfering with the emulation itself. This raises three important challenges: packet identification, clock synchronisation between hosting machines, and packet sampling. In this section we tackle these prolegomena before moving on to the implementation of our framework.

6.1.1 Packet identification

Identifying packets is necessary for passive delay measurement. In order to measure any type of delay, timestamps at both ends of the link have to be matched to compute the delay. Ideally, the packets can be identified by their order, i.e. the first packet sent from a source A to a destination B corresponds to the first packet received at the destination B from the source A. But as packets can be lost or arrive unordered, especially when the

link is mapped on top of a complex underlay infrastructure, more sophisticated mechanisms have to be implemented. Another straightforward solution is to tag all packets, either by a unique packet ID, or even directly by adding the packet timestamp to its header at the source. However, this requires unnecessary modifications to the operating system's network module, and can incur non-negligible network overhead at scale.

In our context of distributed network emulation, all packets are encapsulated in UDP datagrams as soon as they leave the emulated host (Distrinet and Bignet use VXLAN while Mininet Cluster Edition and Maxinet use GRE). We can therefore safely make the assumption that all packets are IP packets, and for each flow of packets sent from a certain source to a certain destination, use the native ID field of IP as identification tag. Unfortunately, this still has two major obstacles: the ID field in IP headers is shared between all fragments of a long packet and is encoded on 16 bits only which can lead to collisions. The first limitation can be managed by considering the pair *(ID, Fragment Offset)* as identification tag; the second limitation is trickier since packets with the same ID from the same source can arrive unordered. However this generally does not happen very often, but to make such assumption, we must ensure that packets take less time to get to their destination than it takes for their source to circle through the range of possible packet IDs (encoded in 16 bits). Formally, the assumption holds when $\Delta < 2^{16}\tau$, where Δ is an upper bound on the network delay, and τ is the average interarrival time of packets (equal to the average packet size over the bandwidth). It is generally the case because longer links (i.e. larger delay) correlate with lower bandwidth (i.e. larger interarrival time). For example, the delay in a 1 Gbps link can be as high as 300 ms without violating the assumption.

Figure 6.1 shows how the assumption works in practice. In this example, (1) has sent two packets P_i and P_j with the same ID x and recorded their timestamps t_i and t_j. P_j got lost and the other end (2) only received packet P_i at time $t_i + \delta$. As such, (1)'s records database is $\{(x, t_i), (x, t_j)\}$ while (2)'s only contains $\{(x, t_i + \delta)\}$. In the worst case scenario,

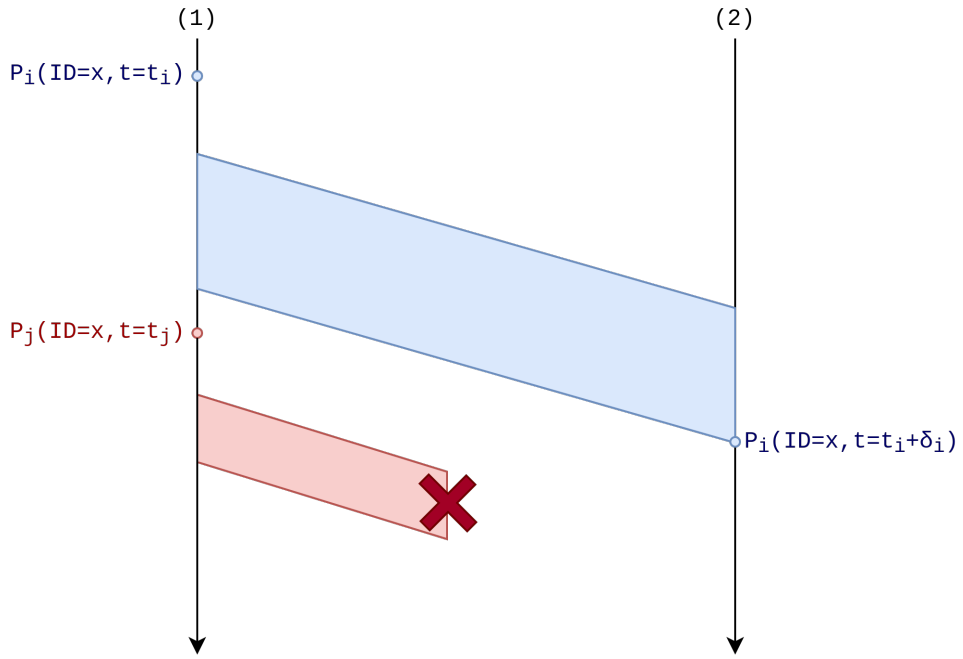


Figure 6.1: A link whose ends are (1) and (2) is used to transmit two packets P_i and P_j, the former arrives at its destination while the second got lost.

(1) has been sending packets of average size s at full speed B (resulting in an average inter-packet time $\tau = \frac{s}{B}$) and has thus circled through the space of IDs in an interval $2^{16}\tau$, i.e. $t_j = t_i + 2^{16}\tau$. The objective is to decide which packet in (1)'s database has been lost, and to do that we need to analyse the timestamps using the assumption:

$$t_i + \delta \leq t_i + \Delta < t_i + 2^{16}\tau = t_j.$$

Hence $t_i < t_i + \delta < t_j$, and therefore (2) can correctly conclude that $(x, t_i + \delta)$ corresponds to (x, t_i) while (x, t_j) has been lost.

6.1.2 Passive delay measurement and time synchronisation

Passive OWD measurement In this section we study the extent to which it is possible to passively measure the one-hop one-way delay (OWD), i.e., the delay of data packets exchanged between two ends of a link which may be hosted on two different underlay machines.

The approach is described in 1, where we simply log packets and their timestamps at both ends, then use our previous assumption to match the two databases.

Algorithm 1 Passive OWD measurement algorithm.

```
Require: packet dumps from (1) and (2): dump_1 and dump_2
  initialise arrays OWD_12 and OWD_21
  for (packet_ID, timestamp_1) in dump_1[outgoing] do
    lookup matching (packet_ID, timestamp_2) in dump_2[incoming] with the closest
    timestamp_2
    owd ← timestamp_2 − timestamp_1
    add (packet_ID, owd) to OWD_12
  end for
  for (packet_ID, timestamp_2) in dump_2[outgoing] do
    lookup matching (packet_ID, timestamp_1) in dump_1[incoming] with the closest
    timestamp_1
    owd ← timestamp_1 − timestamp_2
    add (packet_ID, owd) to OWD_21
  end for
  return OWD_12, OWD_21
```

However, as we have previously argued, great attention should be given to time synchronisation. Indeed, it is practically impossible to accurately measure the OWD in a link hosted in two machines with different clocks. Consider for example the plots in Figure 6.2. We show the OWDs of generated ICMP packets measured by our method with no clock synchronization for a large number of ICMP packets sent with a 1 ms interval (large enough to avoid queuing, thus in theory all packets should have equal delay). We can clearly see how the two machines' clocks drift over time, how this drift affects the measurement of the OWD, and how it is difficult to predict it as it also changes over time. In general, the clock skew depends on uncontrollable physical phenomena (e.g., hardware heating [18]) which cause clock offset between the machines that changes in a non-linear fashion. Note also how in this example the clocks largely drift over a relatively short period of time (17 milliseconds in a 100 seconds-long run), making the noise caused by the clock offset hide all the information from the actual network delay.

Nevertheless, running NTP with configuration parameters described in [54] on the testbed

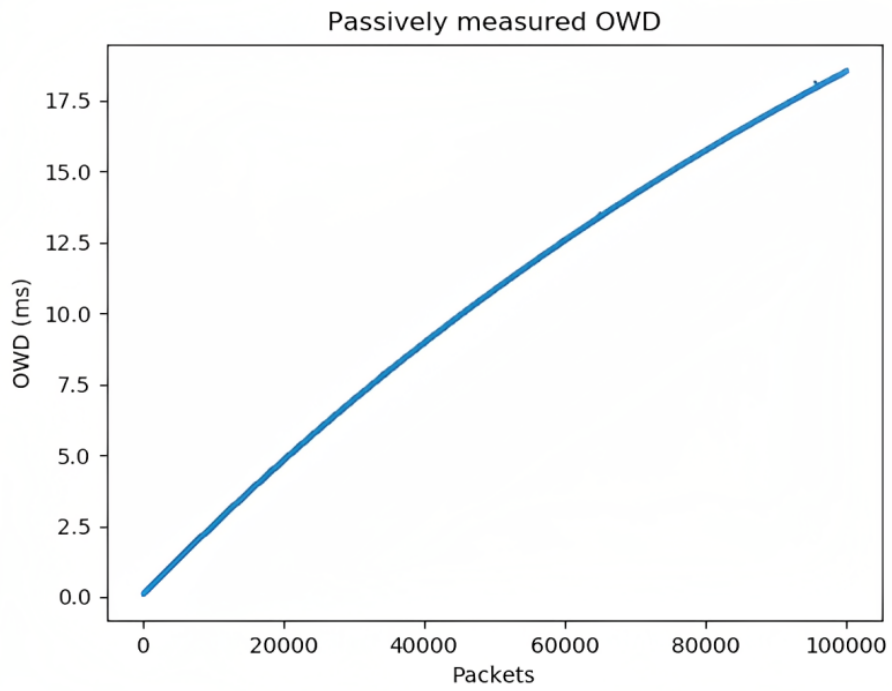


Figure 6.2: Measured OWD between two ends in two machines before clock synchronization.

almost perfectly solves the problem. At the convergence of the NTP process for clock synchronisation and frequency stabilisation, the clock offset and skew are almost neutralized and our method starts reporting *good* results. We can see this in Figure 6.3, where we report on the results of our method after NTP has stabilised: at convergence of NTP, the standard deviation of the measured OWD is less than $10\mu s$.

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

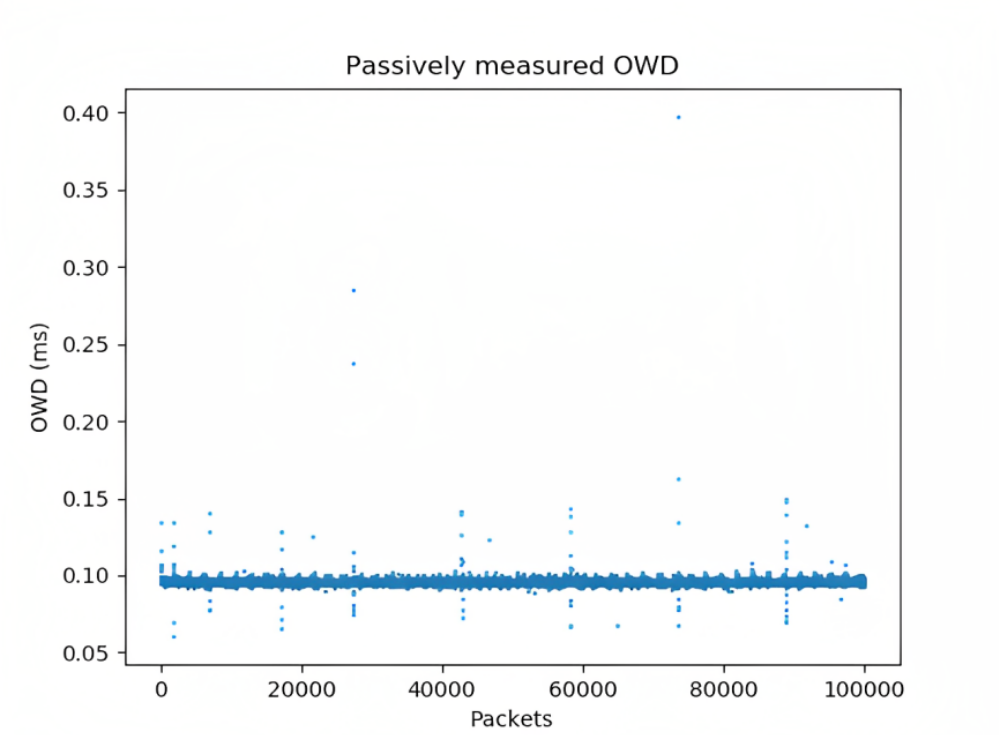


Figure 6.3: Measured OWD between two ends in two machines after clock synchronization.

does not require such strong assumptions.

The methods for passively measuring the round-trip and one-way delays follow a similar approach: we capture packets at both ends of the link and match packet IDs (eventually using our delay-bandwidth assumption to mitigate ID collisions) to compute round-trip delays (RTDs) from the recorded timestamps. In the case of the RTD, for each packet P sent from (1) at time $t_1(P)$ (in (1)'s clock) and received on (2) at time $t_2(P)$ (in (2)'s clock), and Q sent by (2) at time $t_2(Q)$ (in (2)'s clock) and received on (1) at time $t_1(Q)$ (in (1)'s clock), such that $t_1(Q) > t_1(P)$, the collector will report the RTD of packets P and Q as:

$$\widehat{RTD}(P,Q) = (t_1(Q) - t_1(P)) - (t_2(Q) - t_2(P)).$$

Similar to the previous passive OWD measurement method, this does not always give perfectly accurate estimations of the RTD. In fact, while it does eliminate any inaccuracy

Algorithm 2 Passive RTD measurement algorithm.

```
Require: packet dumps from (1) and (2): dump_1 and dump_2
  initialise array RTD
  compute arrays OWD_12 and OWD_21 using algorithm 1
  for (packetP_ID, owdP) in OWD_12 do
    lookup first (packetQ_ID, owdQ) in OWD_21
    if timestamps of packetP and packetQ are close enough then
      rtdPQ ← owdP + owdQ
      add (packetP_ID, packetQ_ID, rtdPQ) to RTD
    end if
  end for
  return RTD
```

due to constant clock drift between the two machines, (i.e., the clock drift at time $t = 0$) it is still vulnerable to its variation. In fact, the longer the time interval between the two packets P and Q, the more the clocks might have drifted during that interval, and the larger the error that will be induced. Thus, in practice, the collector should only stick to pairs of packets sent and received within a small enough time interval T so that the error caused by clock drifts on the estimation of RTD is no larger than a tolerance value ϵ. This ensures that whenever P and Q are such that $t_1(Q) - t_1(P) \leq T$, we have:

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

To evaluate this passive RTD measurement method, we conduct the same experiments as earlier, where we passively measure the delays of generated packets. However, to provide a baseline to compare our method against, we use the Ping tool to generate ICMP echo packets and measure their round-trip, application-level delay. Figure 6.4 shows how the RTDs measured by our method, in the absence of time synchronization by NTP, compare to the RTT reported by Ping.

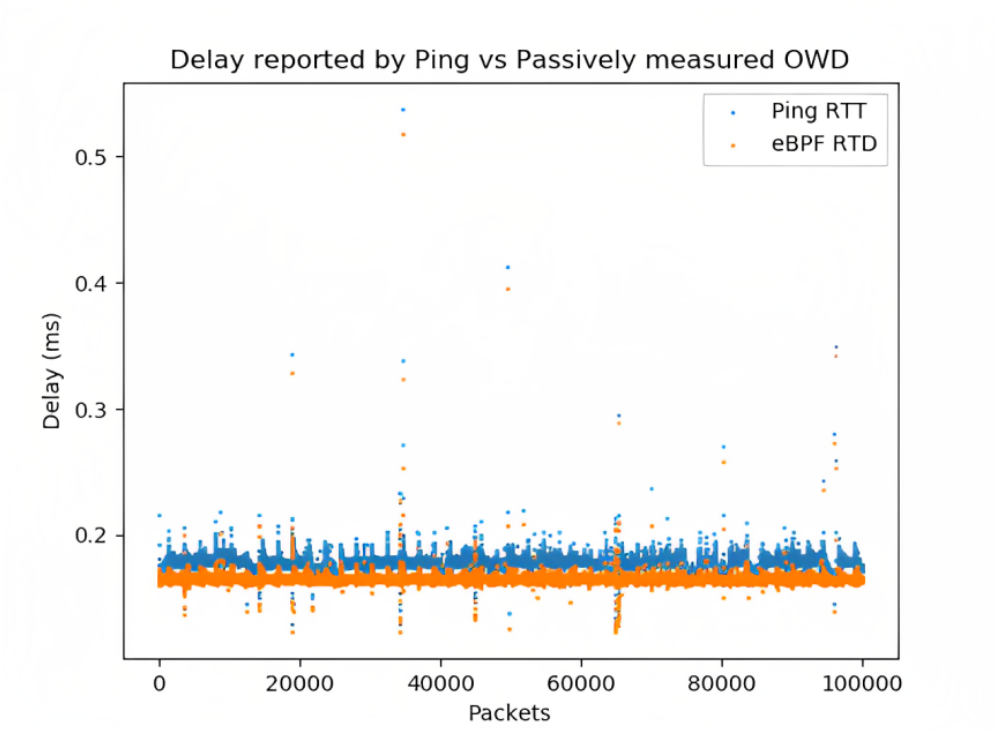


Figure 6.4: Ping RTT (blue) vs passively measured RTD (orange).

6.1.3 Optimisations

In its preliminary form, our approach to delay measurement, using regular packet sniffers and the above two algorithms, cannot be implemented for emulation scenarios. The algorithms have too high asymptotic time and space measures of complexity and the standard packet interception and logging tools can incur a large overhead which may instead reduce the link emulation accuracy and network emulation fidelity. In this section we will discuss ways to optimise these aspects to pave the way for a good design and implementation of our fidelity monitoring framework.

Algorithmic complexity In a bidirectional link (1)-(2) where N packets were sent from (1) to (2) and M from (2) to (1) during a run, the running time complexity of algorithm 1 is $O(N^2 + M^2)$, as the timestamp of reception of each packet has to be looked up by going through all records at reception. This is a naive implementation but it can be optimised by

choosing appropriate data structures and adding a pre-processing step. In particular, packet records can be organised in *(ID, list of timestamps)*-maps where we correspond to each ID the list of timestamps of all packets that it identifies. These lists can be first sorted in the pre-processing phase of the algorithm, which can make the lookup much faster using binary search. With this update, the worst-case[1] of both pre-processing time and running time complexity of the algorithm drop down to $O(N \log N + M \log M)$.

In the fine-grained analysis of network traffic, space complexity is a much bigger issue than time complexity. In high-speed large networks, the number of packets can be orders of magnitude high, and storing information about each individual packet may be impossible. This is particularly the case in emulated networks experimenting SDN and/or data centre scenarios where traffic is high by design. Therefore logging timestamps of all packets in all links is not a reasonable approach. One solution is to analyse traffic on-the-go, and measure packet delays online as the emulated network is running, but this can incur computing overhead which might lower emulation quality. A second solution is to implement packet sampling: instead of intercepting and timestamping every packet, the packet logging tool may only select a small subset based on a certain sampling strategy, either selecting packets of interest (those having large size and/or large expected queuing delay which tend to be the most problematic) or doing blind random sampling. However, as each packet needs to be logged twice (in both ends of the link), a random sampling strategy with a rate s must adapt to such constraint, otherwise (if random sampling processes are independent in the two ends) we may end up with $s \cdot (N + M)$ records in the database but only $s^2 \cdot (N + M)$ usable. A random sampling therefore needs to be hash-based [87] and thus deterministic on the packet ID. In conclusion, by reducing the number of packets from $N+M$ with a sampling rate s, we reduce the number of entries to $s \cdot (N + M)$ and subsequently the running-time complexity to $O(s \cdot (N \log N + M \log M))$.

[1]The worst-case scenario is when all packets share the same ID.

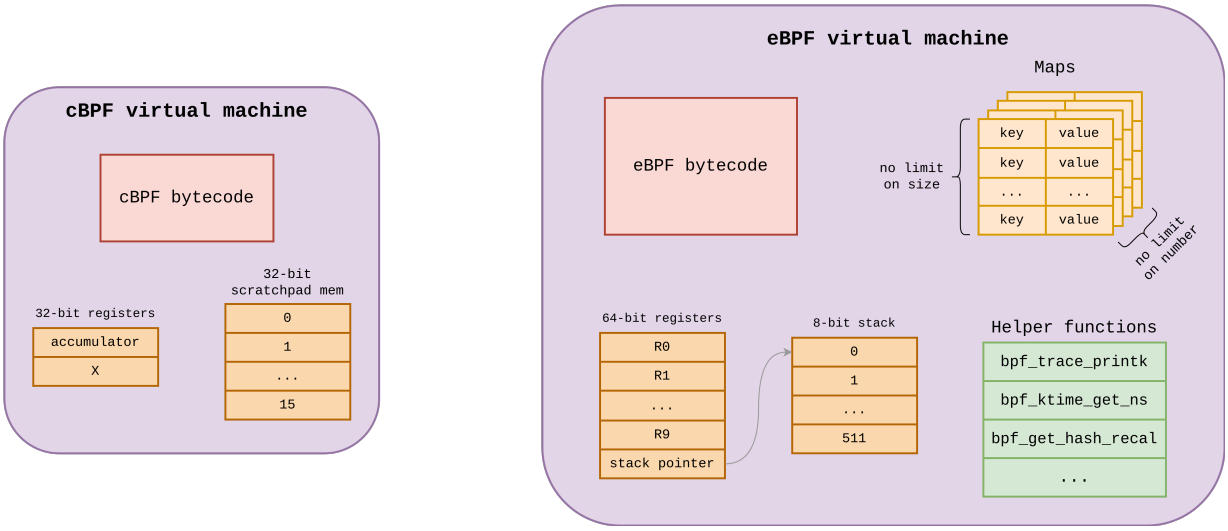


Figure 6.5: Classical Berkley Packet Filter (left) vs Extended Berkley Packet Filter (right).

Packet interception A second room for optimisation is in the packet interception tools. *libpcap* is the standard library for packet interception and analysis in Linux. It is used by the most popular sniffers (*Wireshark*, *tcpdump*) and is based on the Berkley Packet Filter (BPF) kernel tracing framework, which provides a *virtual machine*[2] comprised of (see Figure 6.5):

- A classical BPF code written in cBPF bytecode language, which does not implement `goto` instructions;
- Two 32-bit registers for standard arithmetic operations;
- A finite 16-units long, 32-bits wide scratchpad memory space.

As such, the classical BPF virtual machine can be understood as a minimalist runtime environment that can run very simple instructions in kernel space with very limited features to protect the kernel and avoid crashing the system.

A more sophisticated packet interception and processing framework is the extended

[2] *Virtual machine* must be understood here similarly to the *Java Virtual Machine*, i.e. a virtual code running environment with its own instruction set, bytecode language, registers, etc., and not in its modern sense, i.e. an isolated similacrum of a computer running its own applications, services, and operating system.

Berkley Packet Filter (eBPF). It inherits classical BPF's design and goals but offers much more advanced features (see Figure 6.5). In particular, it provides:

- A larger instruction set, and the possibility for *bounded* loops;
- Wider (64 bits) and more (10) registers;
- A 512-units long, 8-bit wide stack;
- An unlimited memory space through key-value maps;
- A set of *helper functions*, which are pre-compiled routines that can achieve some standard operations similar to actual kernel routines from the Linux source code and the kernel API. For instance, `bpf_trace_printk` functions similarly to `printk` which logs messages in persistent storage; `bpf_ktime_getns` functions similarly to `ktime_getns` which provides high-accuracy timestamping; and other packet processing-specific routines like `bpf_get_hash_recal` which recomputes and returns the hash of a packet.

In practice, eBPF works by writing restricted[3] C code functions and plugging them into *hooks*, which are a set of Linux API and kernel code functions that are eBPF-compatible. The code is then executed every time the hook function is called.

Another important perk of eBPF is its native support by the Linux Traffic Control (TC) subsystem, which we have shown is used by the most popular network emulators to simulate some link characteristics. In practice, most routines in the kernel code for TC can be used as hooks, and TC can also offer an additional entry point for eBPF code through classifiers and filters: while defining queues and queuing disciplines, the user can instruct the execution of pre-compiled eBPF code whenever a packet satisfies some conditions going into or coming out of a queue.

[3] C11 language with some restrictions to guarantee security. In principle, the code should be cleared as *halting* (no infinite loops and recursions) and segmentation fault-free at compile-time.

6.2 Hifinet

Now that we have addressed the major issues underlying any attempt to passively and efficiently measure the network delay, we can present how our fidelity monitoring framework works in practice. We will first introduce its design and argue for the choice of certain principles, then move on to describe its actual implementation within the previously described Bignet network emulator, without any loss of generality as to its implementation within other distributed network emulation software. The last part will evaluate experimentally how all choices and assumptions work in a concrete scenario.

6.2.1 Design principles

Recall the delay-based phenomenal fidelity criterion described in the previous chapter:

> **Criterion 2.** *For an emulation to have good fidelity, the deviation of the measured delays from the model delays of a sample set of packets throughout the duration of the experiment should not be too large.*

Model delay An important caveat about the fidelity criterion is the *model delay* that is used for baseline against which we compare the measured delays. One way to estimate this theoretical/expected delay is to use the standard delay model previously established:

$$\delta(P) = \frac{l}{v} + \frac{|P|}{B} + \frac{|Q(P)|}{B},$$

where the first term is the propagation delay (independent of the packet) configured by the user, where the second term is the transmission delay which depends on the size of the packet and the configured bandwidth (or transmission speed), and where the last term is the queueing delay which depends on the size of the queue when the packet enters and the configured bandwidth. All of these parameters can be known to the fidelity monitoring tool,

and logged alongside the timestamps of arrival, departure, and reception of the packet. This equation perfectly models the behaviour of wired links given certain known variables, but needs to be carefully adapted to radio channels and other wireless media.

Passive delay measurement Because no assumption about time synchronization is reasonable, we will stick to measuring the RTD of pairs of packets P_1, P_2 which is to be compared against the sum of their own delays $\delta(P_1) + \delta(P_2)$ expected from a correct emulation. The major downside of such a choice is that information about the individual OWDs is lost in the RTD. As such, an assessment made from a certain measurement is made on both the packets of the pair.

Statistical metrics Another important caveat is that the deviation between measured and model packet delays can be evaluated using different statistical metrics. A straightforward approach is to consider the mean absolute error (MAE) as a measure of deviation between measured values ($\widehat{RTD}(P_1, P_2)$) and expected values ($RTD(P_1, P_2) = \delta(P_1) + \delta(P_2)$). The emulation can then be considered incorrect if the MAE (over all considered pairs of packets) exceeds a certain threshold established beforehand and which expresses how much fidelity is expected from the emulation. If the user is not able to decide on such a threshold, then they can instead consider the mean *percentage* absolute error (MPAE) by measuring the deviation relative to the model values. The user can then work with a *universal* threshold value (such as 1% or 5% for strict fidelity standards, or up to 50% for looser ones).

However, these two metrics share the common drawback of being measures of *averages* and do not consider values individually, which leads to higher errors being compensated by lower ones. Thus, in the presented version of our framework, we consider quantiles: the emulation will be presumed correct if a certain number of measured values (e.g., 95% of all measures) do not deviate from the estimations by more than the threshold error value.

Here again it is possible to reason in terms of relative error[4], but while this is certainly an upgrade compared to averages, it still treats all packets with equal importance and makes an assessment based on the overall measure of quantile. This is efficient and more precise but not yet ideal as major errors happening in a short period of time and on a limited number of packets can have propagating macro-level effects on the emulation while being considered negligible by the system at the scale of the entire set of packets. In this case the user can consider looking at the *sliding time window quantiles*, i.e., by assessing the experiment to be accurate if, given a number K (resp. time window T), the condition holds for all sequences of K successive pairs of sampled packets (resp. for all pairs of sampled packets in any time window T).

6.2.2 Implementation

Our solution for the measurement and estimation of delay is built up from three main components.

Packet loggers This component is a collection of eBPF functions that are plugged into a number of specific kernel routines and which therefore run in kernel space. Its goal is to capture and log information about sampled packets in persistent storage (Figures 6.6 and 6.7). After a message is made into a packet and then into a Linux data structure, it is enqueued by the TC subsystem –provided the queue is not full– and waits for a period of time before being dequeued and sent to the virtual NIC for transmission, or randomly dropped with a certain probability to simulate loss if it is enabled. And if the packet is to be successfully transmitted, the packet logger logs in raw files information about its enqueue event (event timestamp, packet size, and the length of the queue at the packet's arrival) and dequeue event (event timestamp) if it is sampled for monitoring.

[4]The quantile percentage absolute error (QPAE) is thus a network-level transposition of the *probably approximately delay-accurate* concept we have previously defined for single emulated links: the quantile being the probability parameter γ and the threshold being the approximation parameter ϵ.

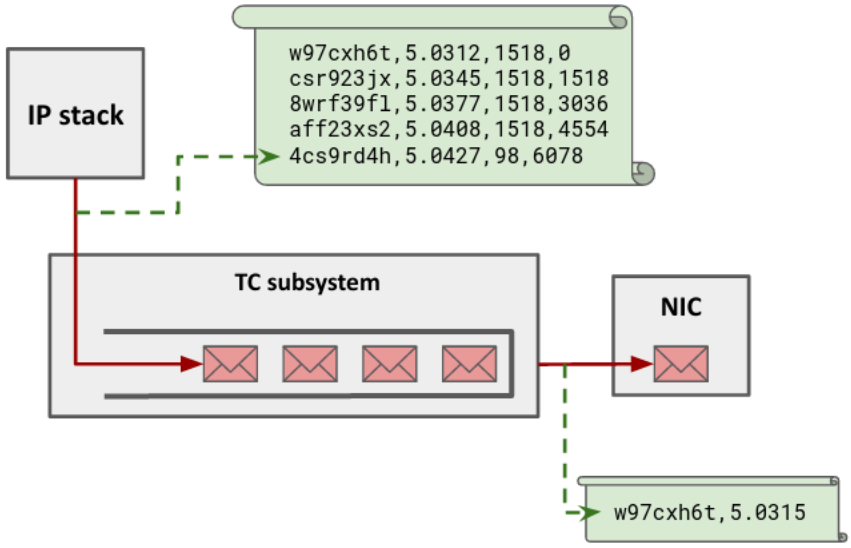


Figure 6.6: TC datapath interception by packet loggers.

Specifically, using eBPF we embed low-level instructions into the TC datapath, at both ends of each virtual link, which run whenever a packet is received by the TC subsystem. This ensures that our passive packet monitoring methodology incurs no significant computing overhead on the kernel (particularly networking) and on application processes.

Local monitoring agents These agents are user space programs that run on the hosts and whose goal is to parse the logs from the packet loggers and compile them into tables that can later be used for analysis (Figure 6.8). This is executed after the emulated experiment has finished running, and therefore does not interfere with the emulation.

A collector/analyser This component is the brain of the system. Its job is to collect and analyse packet information compiled by the monitoring agents (Figure 6.9). It is logically unique and achieves its goal in two steps:

- First, the data is collected from the monitoring agents as tables, which are then cross-examined to match information about packets distributed over multiple tables. For instance, a table from one monitoring agent (and therefore from one machine of the

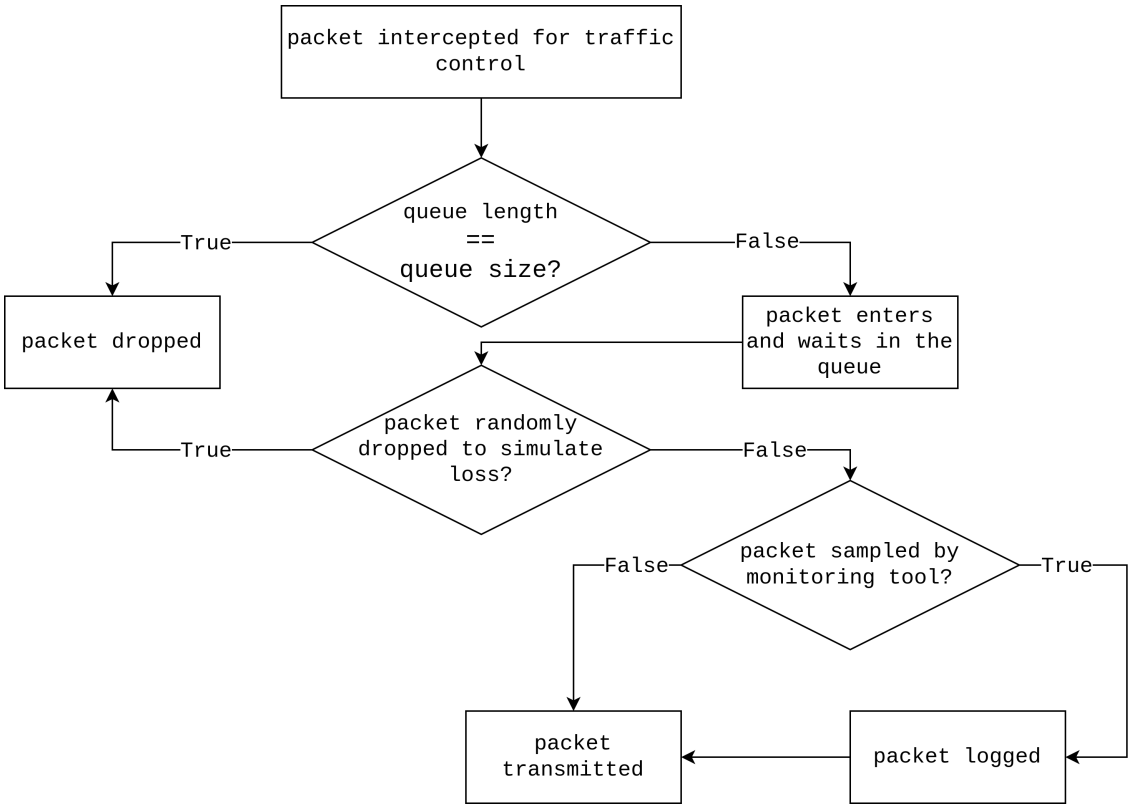


Figure 6.7: Interception and logging of packets.

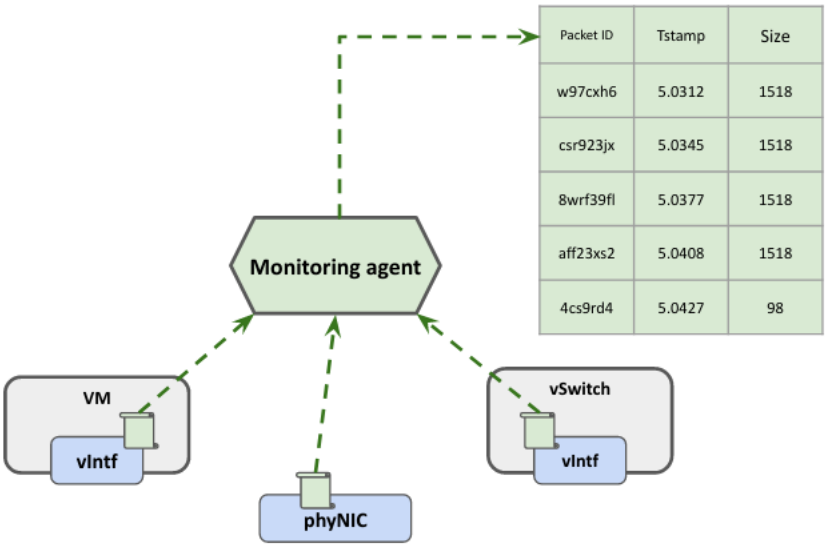


Figure 6.8: Architecture and operation of the monitoring agents.

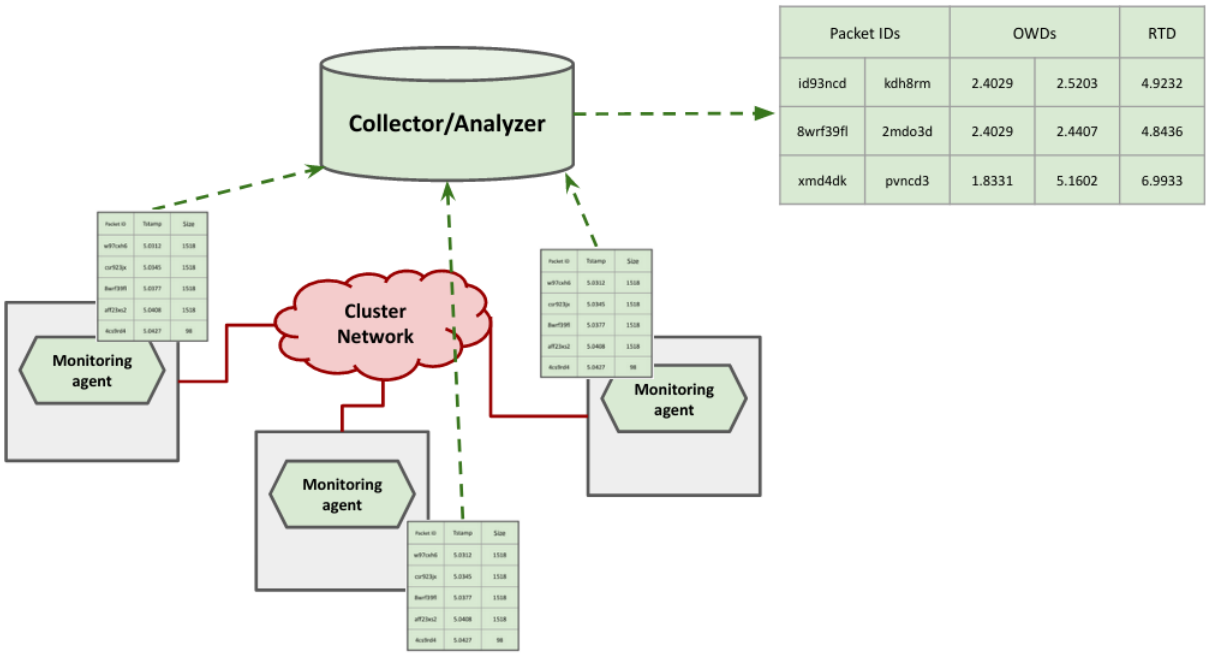


Figure 6.9: The collector/analyser component.

cluster in the case of distributed emulation) can contain the sending timestamp of a packet, and another table from a different monitoring agent can contain the reception timestamp of the same packet. The output of this step is a unique large table where each entry corresponds to a packet and is identified by its ID, and which contains all information about it;

- Finally, packets from the table are paired together according to a pairing rule and their joined RTD is measured (from the logged timestamps) and estimated (from other information such as packet sizes, queue lengths, etc.). These two values are then compared for all considered pairs of packets and an overall judgement about the emulation can be made.

6.2.3 Evaluation

Overhead In the previous subsection, we have demonstrated the implementation of our methodology, which is designed in a distributed and hierarchical fashion: a central and logically unique component analyses the monitoring data sent from multiple local agents, which in turn gather their data from lightweight packet loggers. In practice, the central collector/analyser and the local monitoring agents perform their tasks –of analysis and data collection– *offline*, i.e., after the emulation has completed, and therefore do not disturb its execution. However, as packet loggers intercept the transmission of emulated packets on which they perform certain processing instructions, these can cause overhead by reducing the switching capacity of the emulated network (defined as the amount of packets that can be transmitted per unit of time) and/or inflict additional delay in the emulated links. We have argued that a sampling strategy can trade off a narrow decrease in statistical accuracy of the results to mitigate these overheads, which we show experimentally in this subsection.

To do this, let us consider the following emulated network: a switch connects two emulated nodes H_1 and H_2 with links of unlimited capacity (i.e., no traffic control to limit the bandwidth is used) and a small 1 ms propagation delay. In a first run, we send a heavy long Iperf TCP flow from H_1 to H_2 and record the average achieved goodput over a window of 100 seconds. In a second run, we send 10 000 Ping echoes initiated by H_1 and record the minimum RTT. These experiments are emulated on a single physical host and use our fidelity monitoring tool with random packet sampling, and repeated for different sampling rates: 100%, 10%, 1%, and 0.1%. The former metrics (average Iperf goodput and minimum Ping RTT) are compared between the different sampling rate, and against a setting where the monitoring tool is turned off (corresponding in the following figure to a sampling rate of 0%).

The results of this evaluation are shown in Figure 6.10. As expected, the throughput performance of the emulation decreases when the tool is used (from 16.30 Gbps down to

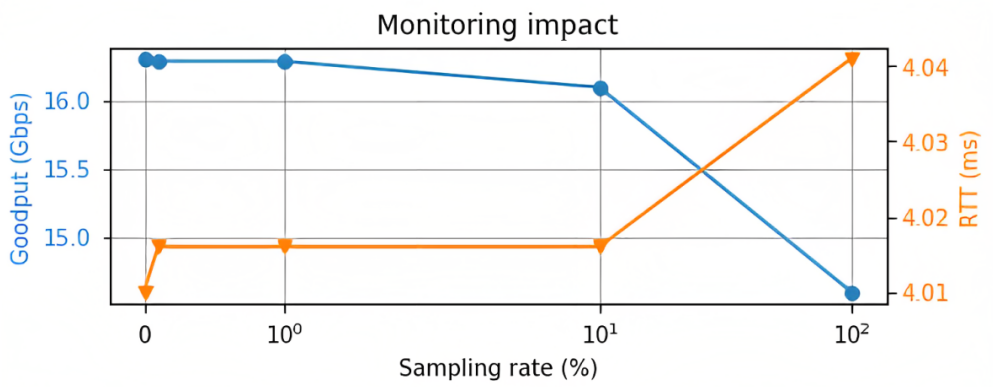


Figure 6.10: Impact of monitoring on the emulation performance. The orange plot shows the average achieved goodput and the blue plot shows the minimum RTT.

14.60 Gbps for a sampling rate of 100%), while the delay per packet slightly increases (few microseconds of overhead). These changes depend on the packet sampling rate configured for the packet loggers, and seem to follow the performance-sampling law found in [33]. Overall, the observed relatively small drops in performance prove that our fidelity monitoring implementation does not paralyse the emulation even at large sampling rates. In all following experiments, we will configure a sampling rate of 10% which does not impact networking capacities by more than 1%.

Example So far we have presented the design of our delay-based monitoring system and explored the possible sources behind delay emulation errors. The underlying principle is that incorrect emulation of delay leads to higher-level errors that can bias the overall results of the emulation. In this section we show through an example how this assumption performs in practice. More specifically, we present a simple network emulation scenario and correlate the delay monitoring metrics with application-level metrics.

Testbed In this scenario, 40 clients are synchronously downloading a 100 MB file from a random server (out of 5) located on the same Ethernet segment. The client hosts are separated into three groups: clients from Group I are connected to the switch by 10 Mbps-

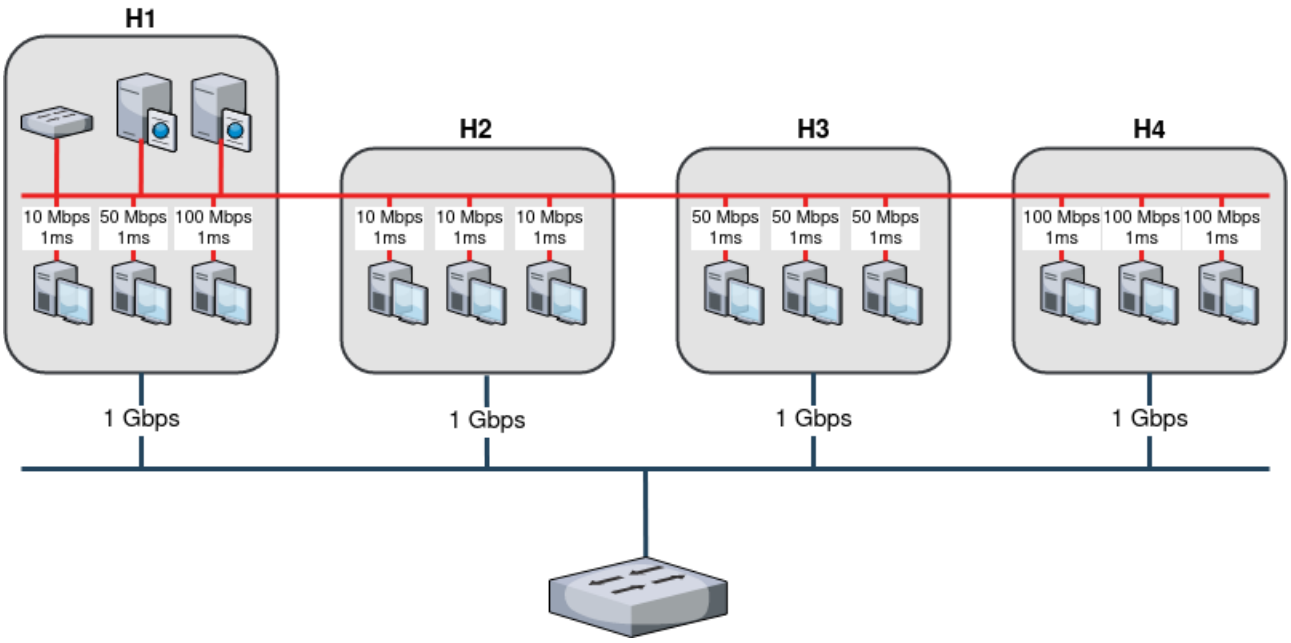
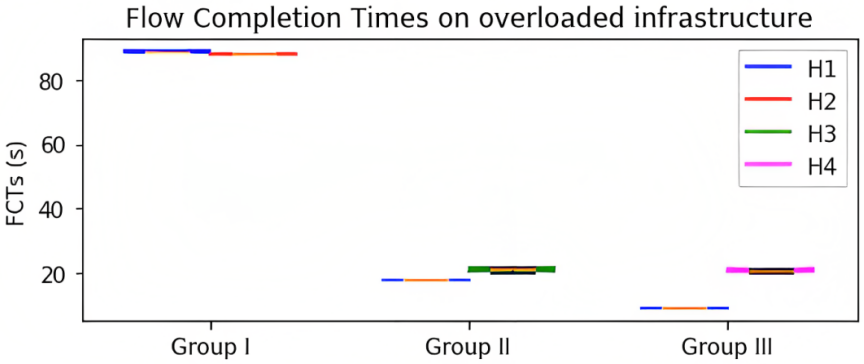


Figure 6.11: Emulated network (red) and underlying cluster network. Clients from Group I are emulated in $H1$ and $H2$; from Group II in $H1$ and $H3$; and from Group III in $H1$ and $H4$.

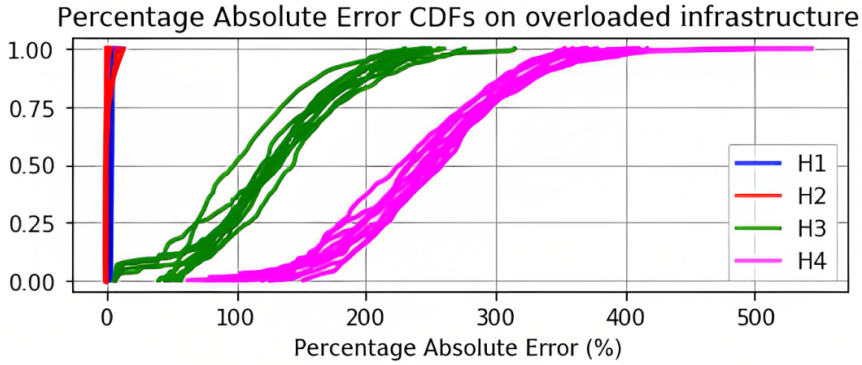
bandwidth and 1 ms-delay links; clients from Group II by 50 Mbps-bandwidth and 1 ms-delay links; and clients from Group III by 100 Mbps-bandwidth and 1 ms-delay links. The servers are connected by links with no traffic control. The experiment is run using the latest version of Distrinet to date (v1.2) on four nodes of the R2Lab cluster[5], which are connected in a star topology to one single switch (Figure 6.11). Furthermore, our embedding algorithm is configured in a way that all emulated file servers and the virtual switch are hosted in the same machine (host $H1$); the emulated clients from Group I are hosted in $H1$ (4 clients) and $H2$ (10 clients), from Group II in $H1$ (3 clients) and $H3$ (10 clients), and from Group III in $H1$ (3 clients) and $H4$ (10 clients) (see Figure 6.11).

The idea is to compare the flow completion times (FCTs) and the measured delay errors between and within the groups. If this scenario were emulated with perfect fidelity (i.e., behaving exactly as it would in real networks), (a) there should be no difference in the FCTs within each group as all clients from the same group are equivalent in the emulated topology,

[5]Reproducible Research Lab: `https://r2lab.inria.fr/index.md`.



(a) Flow Completion Times for the clients of each group. From left to right: Group I, Group II, and Group III. The left-side box plot shows the FCTs of clients hosted in host $H1$ (blue), and the right-side box plot for clients in hosts $H2$ (red), $H3$ (green), and $H4$ (magenta).



(b) CDFs of the percentage absolute errors (PAE). The blue plots correspond to the CDFs of locally emulated links in host $H1$; the red, green, and magenta to the CDFs of links overlay emulated between hosts $H2$ and $H1$, hosts $H3$ and $H1$, and hosts $H4$ and $H1$ respectively.

Figure 6.12: High-level (a) and low-level (b) indicators of emulation fidelity.

regardless whether or not they are hosted *locally* with the server, and (b) clients in Group I (10 Mbps bandwidth links) should experience the largest FCTs, followed by clients in Group II (50 Mbps bandwidth links) and finally the clients in Group III (100 Mbps bandwidth links) should experience the lowest FCTs.

Results Figures 6.12 shows the results. The first thing to note is how the clients in Groups II and III experience different FCTs depending on whether they are hosted in $H1$ or $H3$ and $H4$ (Figure 6.12a). In particular, these get an average of 17.81 seconds vs 21.24 seconds for Group II, and 9.08 seconds vs 20.92 seconds for Group III. This is also evident

from the CDFs of percentage absolute error: overlay links for clients in hosts $H3$ and $H4$ experience higher relative delay emulation error (Figure 6.12b).

Discussion This example essentially demonstrates how higher-level incorrect behaviour, which occurs silently and which can lead to false analyses, is in fact correlated with *objectively* incorrect lower-level behaviour easily perceivable from a delay perspective. In this example, the differences in delay emulation errors between local and overlay links are mainly due to the additional delay (cf. Section 5.3) that emulated packets experience as they cross the infrastructure network. To troubleshoot the causes behind this perceived inaccuracy, it is important to see that the bandwidths of all overlay links sum to a total of 1.6 Gbps, while all these emulated links have to cross the physical link connecting the cluster switch to host $H1$, whose capacity is limited to 1 Gbps. The congestion control algorithm used by clients to download the file distributes the available bandwidth in a way that the throughput of greedy clients (Group II and III hosted in $H1$) is lowered: clients from Group I get a throughput of around 10 Mbps (equal to their bandwidth), clients from Group II get a throughput of around 45 Mbps (5 Mbps less than their bandwidth), and clients from Group III get a throughput of around 45 Mbps (55 Mbps less than their bandwidth). This results in clients from Groups II and III hosted in $H3$ and $H4$ getting longer FCTs than their counterparts hosted in $H1$. And while clients from Groups II and III hosted in $H3$ and $H4$ get the same throughput (and thus experience the same FCTs), the links connecting them to the Ethernet switch show different PAEs: the median for links emulated between $H3$ and $H1$ is around 119%; while the median for links between $H4$ and $H1$ is around 238%. This is due to the former links having a higher bandwidth –and thus their packets a lower RTD on average– while both experiencing approximately the same added delay, which leads to different errors in relative values.

Emulating this scenario in such an infrastructure might seem artificial, and while it does demonstrate the failures of distributed network emulators in certain settings and how those

failures can be captured by our fidelity monitoring methodology, such settings might appear unrealistic at first glance: to avoid these problems the user need only analyse the capacities of the infrastructure and distribute the emulated nodes accordingly. However, this is not always possible as they may be using a shared infrastructure –cloud or grid— over which they have a very limited amount of control and/or knowledge. In such cases, the maximum bandwidth of each physical link may be disclosed, but the fraction available to the user at all times generally is not.

Nevertheless, the delay emulation error strongly correlates with higher-level inaccuracies independently of the infrastructure usage. In this scenario, the load[6] ρ on the physical link connecting host H_1 to the switch S can reach 160% ($N = 40$):

$$\rho = \frac{(10 + 50 + 100) \cdot \frac{N}{4}}{1000} = 160\%.$$

By varying the number of emulated clients N, we can vary this maximum load, and observe different degrees of high-level and low-level emulation infidelity for values below or above 100%. Figure 6.13 shows how these two indicators correlate for different loads (their Pearson correlation coefficient is approximately equal to 0.91). The *deviation* is a chosen application-level metric that measures the relative difference in FCTs between clients from group III hosted in the same machine (H_1) as the server, and clients from group III hosted in H_4.

6.3 Conclusion

Fidelity monitoring is essential in emulation-based experimentation to ensure certain guarantees on accuracy. A good measure of fidelity is how the finest, most elementary network phenomenon is emulated: the packet delay. We have presented in this paper an approach to fidelity monitoring of network emulation by passively measuring delays of packets in emulated

[6]The load or the usage of a link is defined here as the volume of traffic it transports relative to its bandwidth.

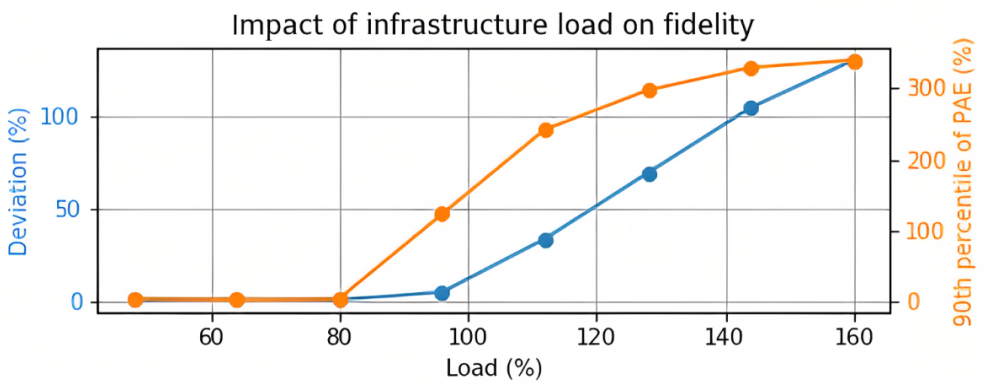


Figure 6.13: High-level (blue) and low-level (orange) indicators of emulation fidelity (y-axis) vs. link load (x-axis).

links and comparing them to values estimated based on a simple network delay model. We have used the extended Berkeley Packet Filter's (eBPF) native packet monitoring capabilities to implement our methodology in an accurate and efficient manner. This implementation, together with a good sampling strategy, can highly limit the impact of monitoring on the emulation itself. We have demonstrated how our methodology can help predict emulation anomalies that are otherwise indistinguishable to the user from normal network behaviour.

The passively collected delay measurements can help troubleshoot the emulation failures and accurately diagnose their causes. The use of network delay tomography can help translate the measurements collected at the emulation-level into information about the underlying infrastructure. The next chapter will attempt to explore this approach, with the ultimate objective of helping the user re-conduct their experiment under better conditions and with better guarantees on fidelity.

CHAPTER 7

TROUBLESHOOTING DISTRIBUTED NETWORK EMULATION

The previous chapters have presented the design and implementation of our emulation fidelity monitoring framework. Its main objective is to detect and inform when emulation fidelity is compromised by relying on a network-level fine-grained metric: the packet delay. This is achieved by passively collecting emulation-level delay measurements. The immediate next question is to wonder whether these measurements can be used to infer the potential underlying root causes of emulation failure, ultimately in order to redo the emulated experiment under better conditions. This chapter will attempt an exploration of the different aspects of this question.

7.1 Problem Modeling

The first and most important step in troubleshooting the lack of emulation fidelity is inferring physical infrastructure load to determine with controlled degree of confidence the location of bottlenecks potentially responsible for emulation failures. This is to be achieved using passive network measurements collected at the virtual level[1]. The generalised delay is a particularly useful metric for such task. We know from queuing theory that an increased load in a link results in increased queuing time of the packets and thus in an increase in their network delay. This additional underlay delay, measurable through our fidelity monitoring framework, depends on the depth of the physical queues: large queues induce high delays when full; and small queues cause packets to be dropped much earlier which is experienced

[1]Distinguishing between the emulated and infrastructure networks is of critical importance in this chapter. We will refer to the emulated network, which is constituted from the emulated nodes and links and which has a user-defined structure and characteristics, as the *overlay* network; conversely, the network defined by the physical machines over which the overlay network is emulated, and the switches, links, and routers that connect them, will be referred to as the *underlay* network.

as infinite delay in the emulated network. In both cases, our delay measurement strategy captures such incidents and performs statistical analysis to indicate failure. In this section, we describe the problem in more details by presenting our working hypotheses, its mathematical modeling, and by discussing raised challenges.

7.1.1 Hypotheses

Consider for example the simple scenario in Figure 7.1: a virtual overlay network (consisting of a virtual client and a virtual server connected to a virtual switch) is emulated over an underlay 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: e.g., limiting link bandwidth, adding propagation delay, and introducing packet loss. Given these traffic shaping parameters, each packet P that is not dropped by the virtual link should experience a certain *normal* delay $d(P)$ depending on its size, its position in the virtual links' queues, and other intrinsic characteristics of the emulated link. As this packet moves over the virtual network, the links L_1, L_2, and L_3 of the underlay 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 undesirable delay exceeds some tolerance value, its negative impact on the emulation may become non-negligible. Unfortunately, the user does not have full control over the physical infrastructure to monitor the underlay delay in all network nodes and links and thus cannot predict such an event nor determine its exact cause.

In the example, as the packet P crosses the virtual link v_1 it 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 undesirable 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

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

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, delay tomography theory suggests resorting 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 underlay paths). Given some prior information on the mapping of the virtual network to the infrastructure, statistics on the underlay link delays can then 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 underlay link i corresponds a sequence of variables $(x_i(T))_{T\in\mathcal{T}}$, and to each overlay link[3] 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

[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.

then be related by linear equations of the form:

$$\sum_i a_{i,j}(T) \cdot x_i(T) = \overline{\epsilon}_j(T), \tag{7.1}$$

where $a_{i,j}(T)$ is a binary value equal to 1 if virtual link j crosses physical link i and 0 otherwise[4].

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

$$\mathbf{A}(T) \cdot \mathbf{X}(T) = \mathbf{b}(T), \tag{7.2}$$

where $\mathbf{A}(T)$ is defined as the *mapping matrix*[5] whose coefficients are $(a_{i,j}(T))$, $\mathbf{X}(T)$ is a vector of variables $(x_i(T))$, and $\mathbf{b}(T)$ is a vector of collected delay errors $(\overline{\epsilon}_j(T))$.

Our problem then translates into solving the set of equations in (7.2) under the following three main hypotheses:

- The underlying topology and the mapping 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 in order 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 available capacities and characteristics and

[4]Clearly, the existence of an equation during interval T is conditioned by having packets flowing over the corresponding virtual link.

[5]The mapping matrix is also called *routing matrix* in delay tomography literature.

cannot access them for direct monitoring. This hypothesis is the default scheme in shared infrastructure such as grids and clouds, where static information (topology and hardware characteristics) can be provided but the user cannot directly access networking nodes and/or measure dynamic information[6] (load, delay, and 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 the emulation scenario and can implement a monitoring tool to passively measure the delays of emulated packets. One such monitoring tool is the fidelity monitoring framework we have previously presented, which provides exhaustive information about emulated packets and their delays. Such tools essentially intercept a subset of the emulated packets (based on a preconfigured sampling rate) and use information available to the emulator (e.g., queue lengths and virtual link speed) 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 (GRE and VXLAN) to create virtual Ethernet links on top of an infrastructure network. Thus, neither differentiated treatment of virtual links nor QoS mechanisms are used.

7.1.2 Challenges

Time synchronization Being an explicit measure of time, network delay measurement inevitably requires some degree of time synchronization. We have previously 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

[6]Certain cloud providers can offer a measurement service but it is generally limited to high-level application metrics and it is far from being the norm in public cloud settings.

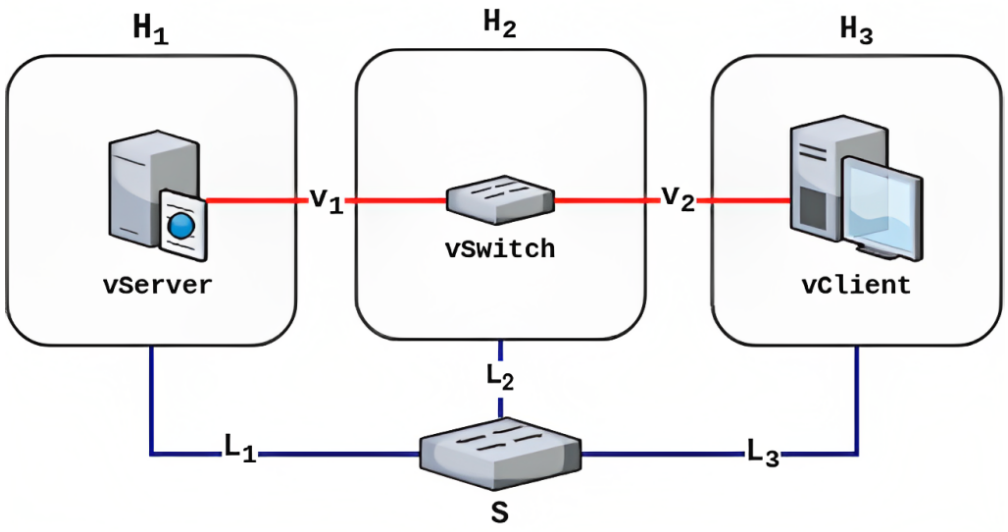


Figure 7.1: Emulated and infrastructure topologies.

time synchronization protocols (see Chapter 6). In cases where this cannot be achieved, we have suggested measuring 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 $\bar{\epsilon}_j$ are measures of mean round-trip delays on virtual links, then x_i will also be measures of round-trip delays.

Time decomposition In addition to the complexity of measuring one-way and round-trip delays, time introduces a new challenge for delay tomography: to assign overlay delay measurements to time bins also requires precise and synchronised timestamping by the machines that send and receive the packets. This problem, however, is unavoidable and can only be mitigated by considering coarse enough time intervals. This helps reduce the impact of timestamping imprecision but can challenge the assumption that the underlay link delay distributions are stationary.

Mapping identifiability The set of equations (7.2) have unique solutions $x_i(T)$ only if there are enough overlay links that cross the diverse set of underlay infrastructure links, i.e., when the mapping 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 mapping matrix $\mathbf{A}$:

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

However, the inevitable lack of precision of any tool used to passively measure the delay, as well as the aforementioned time asynchronisation problem, can potentially add noise to the measurements. This noise can be large enough that the equations have negative solutions, 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 mapping matrix. In the previous example (Figure 7.1), the system of equations in (7.1) transforms into 2 equations (corresponding to 2 virtual links) and 3 variables (corresponding to 3 physical links), which unfortunately does not have a unique solution. This problem is the main challenge in delay tomography, and is referred to as the *identifiability problem*.

In this context and following our notation, the mapping is said to be *identifiable* for a certain time interval T if the overlay measurements can determine a unique solution of underlay link delays. In other words, it is identifiable if the rank of the matrix $\mathbf{A}(T)$ is equal to the number of variables.

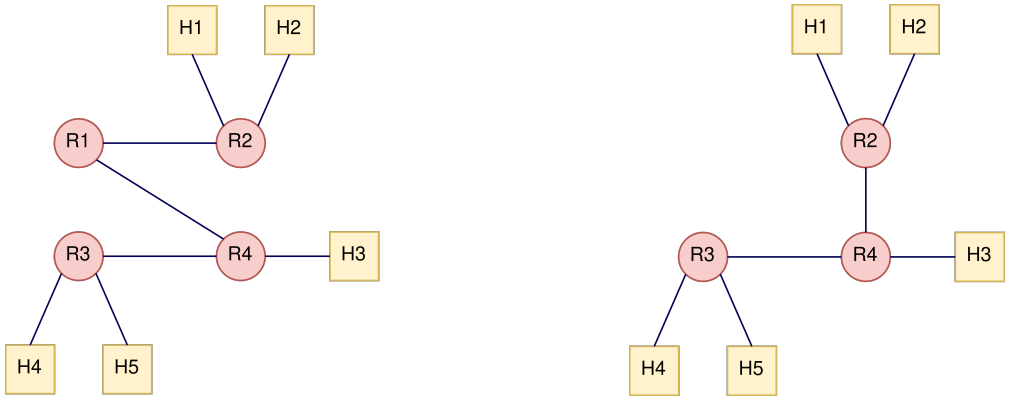
If an infrastructure underlay network is represented as an undirected graph $G = (V, E)$ whose vertices $v \in V$ are the nodes (switches, routers, and hosting machines) and $e \in E \subseteq$

$[V]^2$ are the underlay links, then it is said to be *identifiable* (intrinsically and independent of the overlay emulated network and its mapping) if there exists a mapping which is identifiable on G. In general delay tomography theory, intrinsic identifiability ensures that delay measurements between end-hosts (vertices of degree 1, corresponding to hosting machines in distributed emulation contexts) can fully determine the delays on the individual links of the underlay infrastructure. The representation of the network structure as mathematical objects simplifies reasoning about the concept and figuring out sufficient and necessary conditions. The language and toolset of graph theory is particularly valuable here, and provides us with useful properties such as:

Property 1. *A connected, acyclic, undirected graph $T = (V, E)$ is identifiable if and only if there is no node of degree 2,*

which ensures that on a tree underlay topology, corresponding to an infrastructure network without load-balancing and with clearly defined routing, it is possible to embed the overlay virtual network in a way that the passive delay measurements allow the inference of each individual underlay link delay, as long as there are no internal nodes of degree 2 (see Figures 7.2a and 7.2b, where R1 is one such node of degree 2). But while such structures are only a minority in the set of all possible undirected trees, in practice they are much more common, as a node of degree 2 would correspond to a switch or a router that simply forwards traffic between two interfaces and acts as a repeater.

In conclusion, while the problem of delay tomography for emulation troubleshooting can be easily modelled, some inherent limitations challenge its resolution. The following section thus aims at working around these constraints where we offer heuristics to solve the problem suboptimally with the minimum possible error in all cases.



(a) Unidentifiable network. (b) Identifiable network.

Figure 7.2: Examples of unidentifiable (a) and identifiable (b) graphs. The second graph is constructed by removing R1 and merging its two links into one.

7.2 Algorithms

Considering all discussed challenges, a resolution methodology necessarily requires controlling measurement imprecision and circumventing the identifiability problem. 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 j. The system then has the form

$$\mathbf{A}(T) \cdot \mathbf{X}(T) = \mathbf{b}(T) + \varepsilon(T) \; . \tag{7.3}$$

While this mitigates measurement errors, it adds more unknown variables to an already underdimensioned problem. In practice, the presented measurement tool designed for fidelity monitoring is implemented with high precision as an important specification to thoroughly reduce these errors. This observation can help us control those measurement errors $\varepsilon_j(T)$ by assigning them the smallest possible values that allow a solvable set of equations.

That being said, our resolution methodology will operate in two steps. First, starting from an underdimensioned linear system 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:

$$\begin{aligned} &\text{minimize}_{\mathbf{X},\varepsilon} && \|\varepsilon(T)\|^2 \\ &\text{subject to} && \mathbf{A}(T)\cdot\mathbf{X}(T) = \mathbf{b}(T) + \varepsilon(T) \\ & && \mathbf{X}(T) \geq 0\ . \end{aligned} \tag{7.4}$$

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., underlay 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 7.1 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:

$$\begin{aligned} &\text{minimize}_{\mathbf{X}} && f(\mathbf{X}(T)) \\ &\text{subject to} && \mathbf{A}(T)\cdot\mathbf{X}(T) = \mathbf{b}(T) + \varepsilon^*(T) \\ & && \mathbf{X}(T) \geq 0\ . \end{aligned} \tag{7.5}$$

Next, we present three heuristics that follow this model with incremental complexity and comment on each of their significations.

Heuristic 0: Lower and upper bounds of delay This first heuristic aims at finding very loose lower and upper bounds of 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 heuristics 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? by solving the formulated abstract problem 7.5 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 the simple algorithm:

Algorithm 3 Troubleshooting algorithm: lower and upper bounds

```
solve convex programme 7.4 and get values for ε*
for i = 1, ..., n do
  solve linear programme 7.5 with f(x_1, ..., x_n) = x_i and get x_i^m
  solve linear programme 7.5 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
```

Heuristic 1: Occam's razor The event where a large number of links in a network are overloaded is not very common. Instead, failures due to congestion are more likely to be caused by a small number of causes. This heuristic draws from this observation and selects, among all solutions to the linear system, those that describe a situation where the least number of overloaded underlay links are the cause of delay emulation errors in the overlay

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[7]. 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,

[7]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.

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 to the previous objective function but is continuous and does not involve integer variables:

$$\begin{aligned} &\text{minimize}_{\mathbf{X}} && \sum_i x_i(T) \\ &\text{subject to} && \mathbf{A}(T) \cdot \mathbf{X}(T) = \mathbf{b}(T) + \varepsilon^*(T) \\ & && \mathbf{X}(T) \geq 0\ . \end{aligned} \tag{7.6}$$

Heuristic 2: Dynamic adaptive Occam's razor The above heuristic reduces the set of solutions by choosing those with 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. 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 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 i and j, we have $\alpha_i > \alpha_j$ if and only if link j is more likely to cause delay emulation error than link 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 (log-)probabilities of overload of links 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 version of the Occam's razor heuristic.

Algorithm 4 Troubleshooting algorithm: Occam's razor

```
α_i ← 1
for Tin𝒯 do
    solve convex programme 7.4 and get values for ε*
    solve linear programme 7.5 with f(x_1, ..., x_n) = Σ_i α_i x_i
    update coefficients α_i
end for
```

7.3 Evaluation

We will explore in this section how the designed algorithms perform in practice. Though we use delay tomography, the end-goal is not to estimate delay values, which are only a tool to indirectly determine whether underlay links are overloaded. Thus, throughout this section we will not use the precision of delay estimation as the metric for our evaluation; instead, we will judge the algorithms based on how accurately they predict link overload, in terms of *true and false positives and true and false negatives*. The evaluation will start with a series of numerical simulations on a real testbed to explore its full potential before presenting how the implementation performs on a sample subset of cases.

7.3.1 Testbed

Underlay network (physical infrastructure) We run the simulations and the emulated experiment on a testbed which consists of a subset (10) of the machines in the Rennes site of the Grid5000 shared infrastructure. Figure 7.3 shows its 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. This also makes delay tomography easier by enforcing intrinsic identifiability on the network, even though we use unindentifiable overlay-to-underlay mappings. The testbed thus involves 13 links (10 access and 3 inter-switch links), which amounts to $\tilde{8}000$ configurations of overloading (any of the 13 link 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.

Overlay network (emulated scenario) On the physical testbed we run an emulated experiment about a country-wide telco network where multiples ISPs provide connectivity to clients and servers located in multiple regions (10) of a modelled metropolitan France. Figure 7.4 shows the telco network. 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 all directions.

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

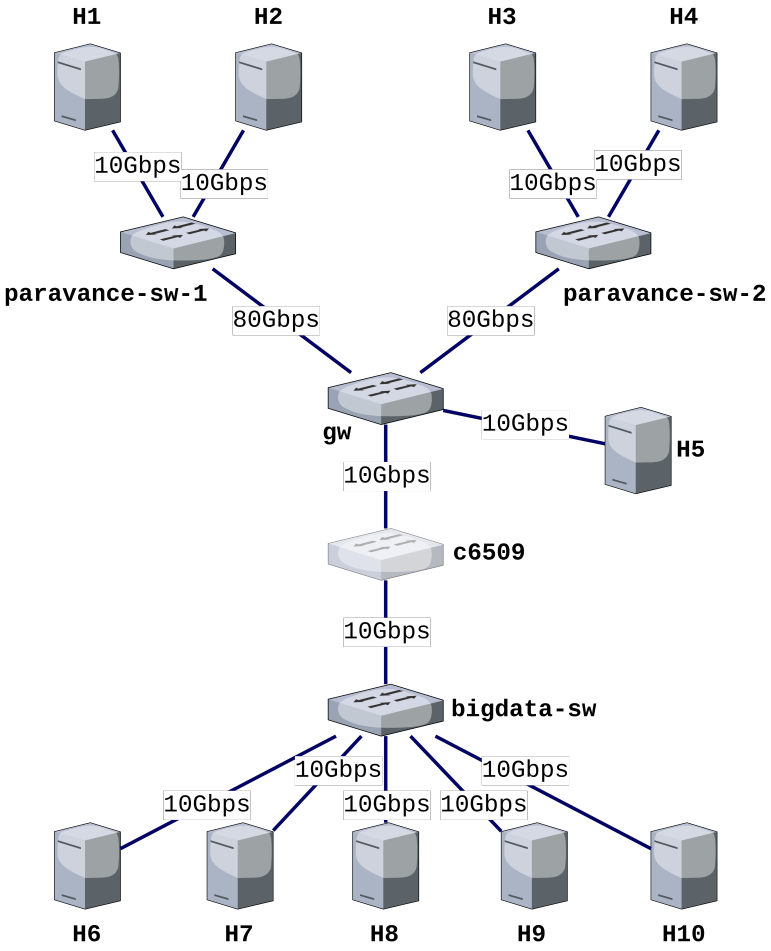


Figure 7.3: Underlay infrastructure network.

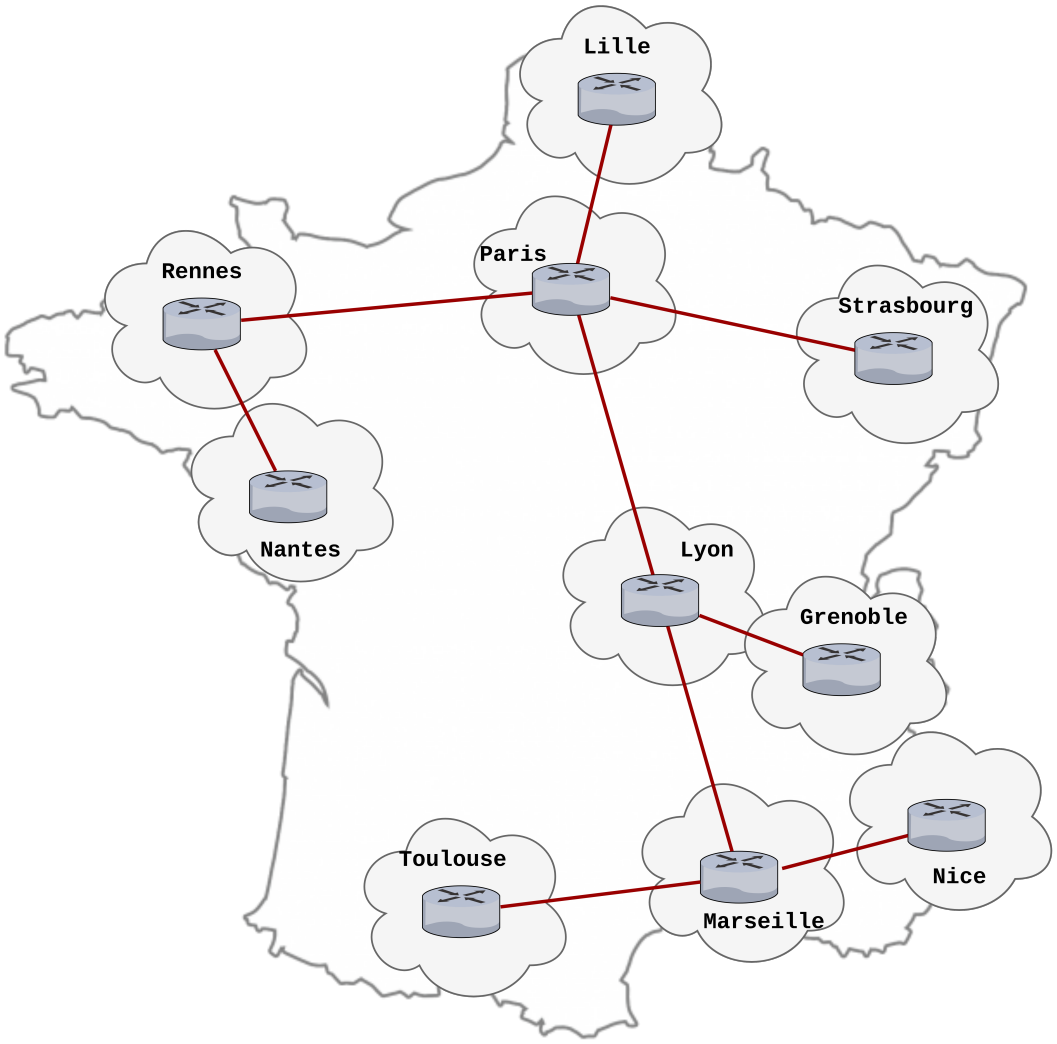


Figure 7.4: Overlay emulated network.

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

of the infrastructure runs the same number of virtual nodes. This is achieved by assigning an entire site from the overlay emulated network to each own unique physical host from the underlay infrastructure. The following table summarises the mapping.

7.3.2 Numerical simulations

We first conduct simple 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 cases, which for a lack of time and resources cannot all be run using emulation.

The simulation flow is as follows:

- first a binary vector $\mathbf{v}$ of size 13 is generated, where each element v_i indicates whether underlay link i is overloaded ($v_i = 1$) or not ($v_i = 0$);
- from the binary vector we generate random underlay link delays $\mathbf{X}$, where $x_i > 1ms$ if and only if link 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 $\hat{\mathbf{v}}$ are determined from the delay estimations, and compared to the ground truth $\mathbf{v}$.

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

- *precision*: a very conservative metric which is either 1 if the estimation perfectly mirrors the truth ($\mathbf{v} = \hat{\mathbf{v}}$), 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. It is equal to 1 if and only if the estimation is perfect ($\mathbf{v} = \hat{\mathbf{v}}$).

Figure 7.5 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.

7.3.3 Sample runs

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

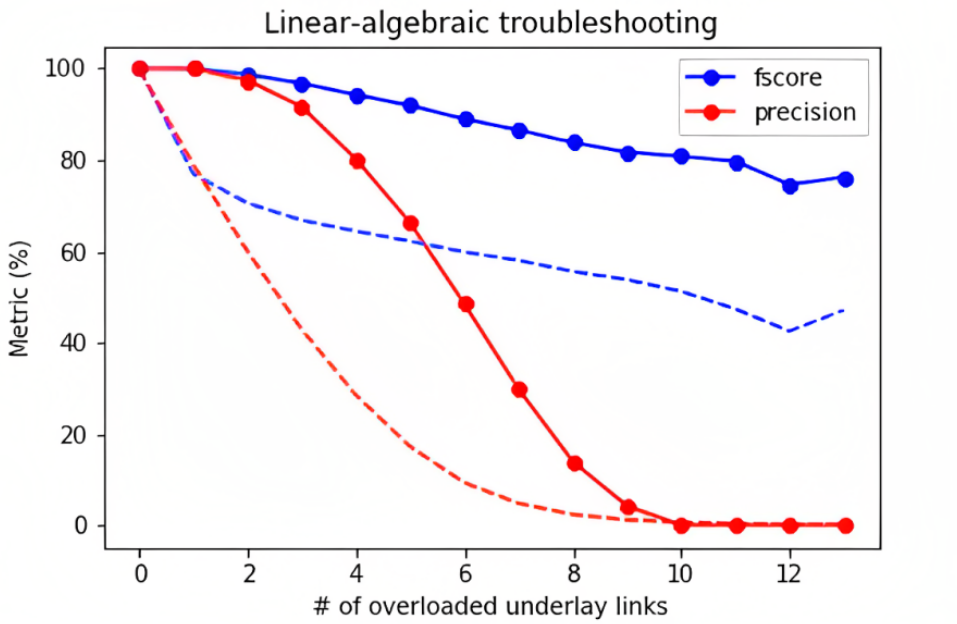


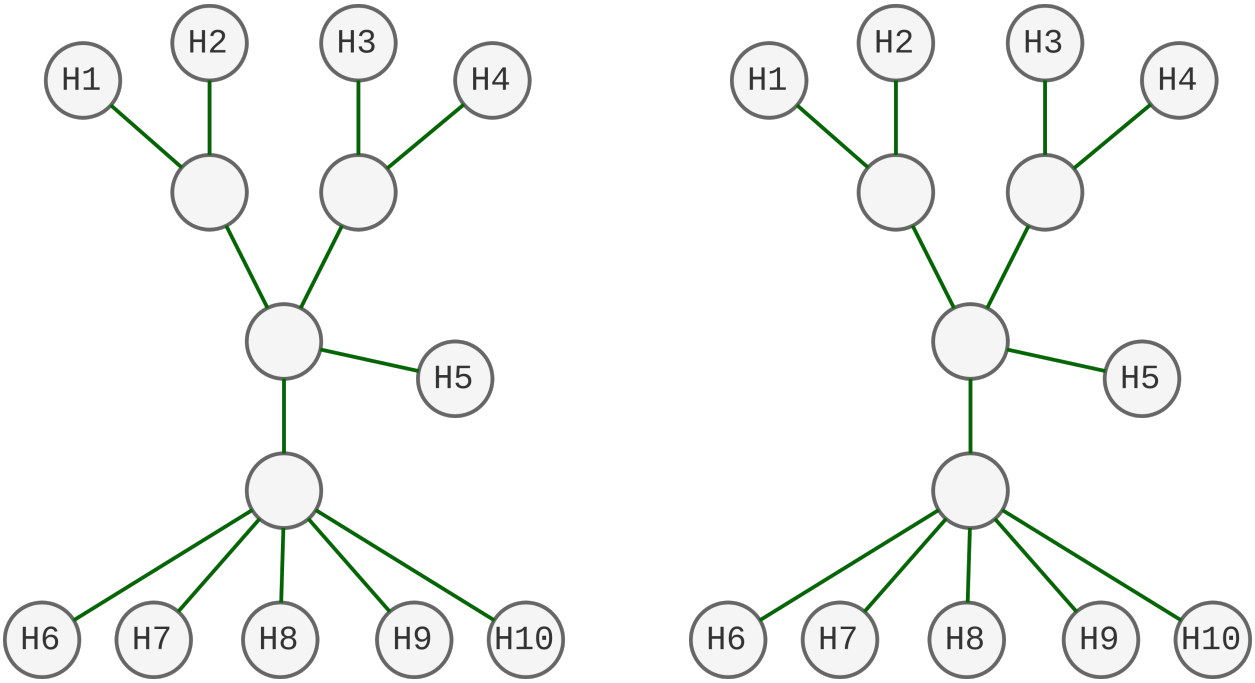
Figure 7.5: Simulation results on all 8192 overloading cases. The continuous lines show the performance of linear-algebraic troubleshooting with Occam's razor heuristic (Heuristic 1); and the dotted lines by relying only on lower and upper bounds (Heuristic 0).

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 on a cluster of machines running a 18.04 Ubuntu distribution with 4.15.0 Linux kernel. Full description of the hardware can be found at `https://www.grid5000.fr/w/Rennes:Hardware`.

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 further runs.

Using the emulation measurements, the fidelity monitoring tool does not observe irregular

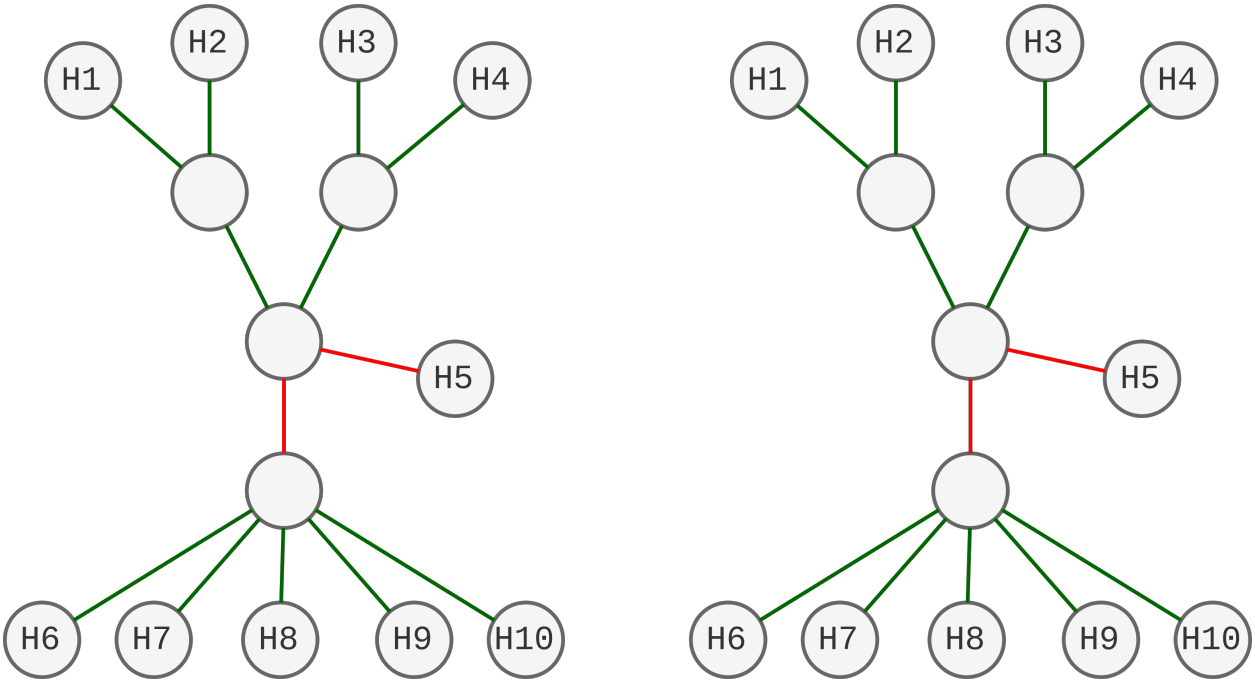


(a) Ground truth. (b) Algorithm output.

Figure 7.6: Run 0. Perfect prediction: 100% precision and F_1-score.

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 7.6) that the infrastructure is not saturated and therefore that no link is overloaded. The results, an average flow completion time of 7.80 seconds, are thus to be trusted.

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 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 7.7). From the perspective of the user, this has translated to inaccurate results: an average flow completion time of 10.20 seconds, mostly between clients and servers hosted in



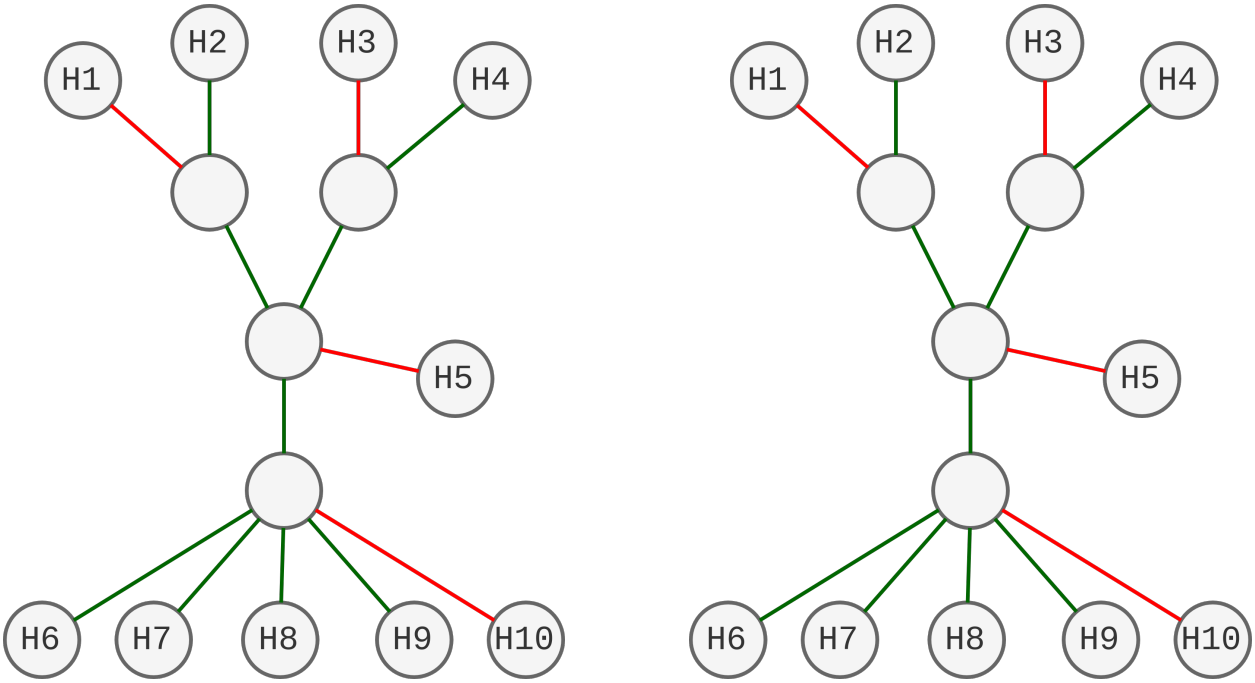
(a) Ground truth. (b) Algorithm output.

Figure 7.7: Run 1. Perfect prediction: 100% precision and F_1-score.

different sides of the country (North-to-South and South-to-North traffic).

Run 2: uncorrelated bottlenecks In this run, we artificially overload certain random links in the infrastructure network with multiple uncorrelated traffic flows and using external machines (see Figure 7.8). As the number of overloaded links is relatively low, the proposed heuristics can still correctly troubleshoot the failures with perfect precision.

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 incurs high delays on the emulated packets. The fidelity monitoring tool captures this delay increases and raises the alarm for emulation failure. Subsequently, the troubleshooting algorithm analyses the overlay delays to estimate 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, it decides (wrongly) that congestion



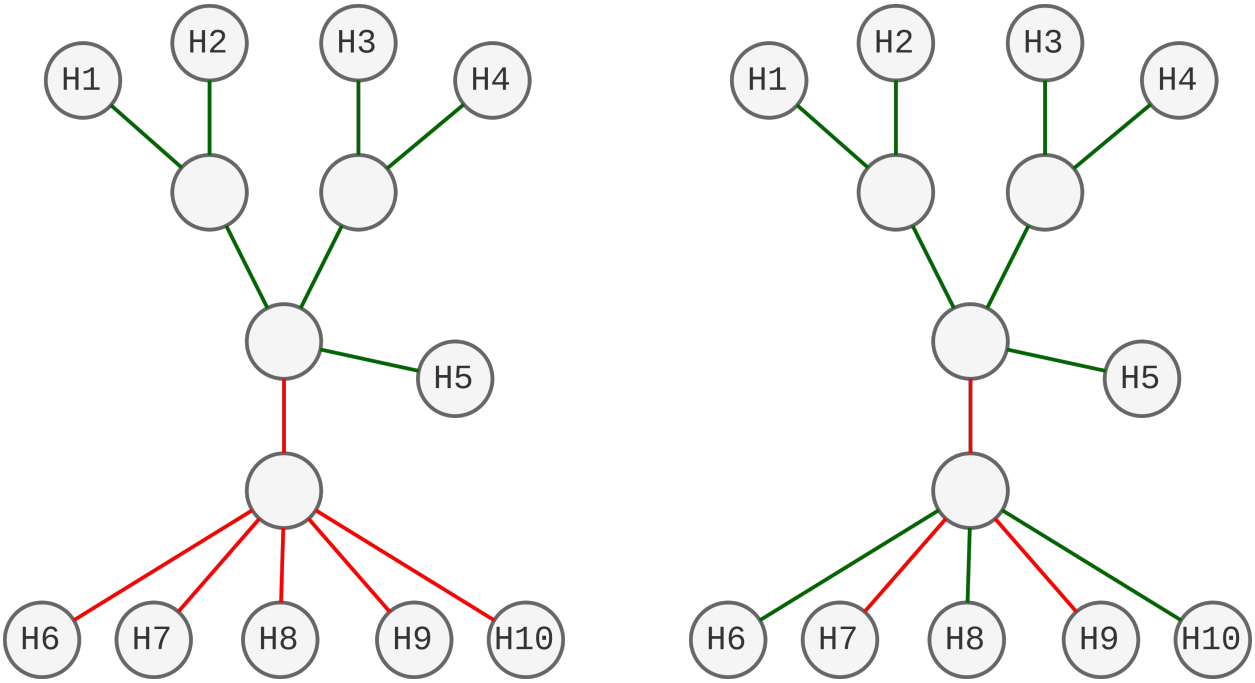
(a) Ground truth. (b) Algorithm output.

Figure 7.8: Run 2. Perfect prediction: 100% precision and F_1-score.

in links `H7-gw`, `H9-gw`, and `gw-(c6509-)bigdata-sw` is behind the measured high delays because these constitute a sufficient explanation. However, other links are also overloaded and their congestion contributes to the measured delays.

7.4 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. The idea behind is 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:



(a) Ground truth. (b) Algorithm output.

Figure 7.9: Run 3. Erroneous prediction: 0% precision and $66.\overline{6}\%$ F_1-score.

- 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:

- 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 loads have

Algorithm 5 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
```

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 who 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.

7.5 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 chapter, 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 mapping 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 and how to dynamically update its coefficients, for instance, can be improved to better quantify the likelihood of each component being faulty. Additionally, we have sketched a remapping algorithm that uses the troubleshooting data to offer a better overlay-to-underlay mapping. This algorithm reestimates the capacities of each component of the underlay infrastructure (link bandwidths and compute resources) to avoid problematic underlay links and instead localise the emulated links on single physical hosts.

CHAPTER 8

CONCLUSION

8.1 Summary

This thesis has tackled the complex problem of scientific experimentation in the context of distributed systems and computer networks, specifically by addressing network emulation as one of its paradigms. The initial hypothesis was that emulation is the best approach, especially using lightweight container-based network emulators such as Mininet, as it performs well in terms of reproducibility, efficiency, accessibility, flexibility, and openness. Indeed, such software makes the setup and operation of complex networks and intricate scenarios easy to accomplish and easy to share. The only issues were those of scalability and realism: as all the components of the emulated network share the same pool of resources running a large scenario is a delicate task; and as the emulator relies on software tools –based on fallible models– to simulate some pieces of the network the user cannot expect perfect fidelity of behaviour. The general objective of the thesis was to address these limitations individually and together, ultimately in order to produce blueprints of a fidelity-aware large-scale distributed network emulator.

We can safely say that the first objective has been met, or rather we have proven it easy to meet it. We have built a proof-of-concept lightweight distributed network emulator using widely popular tools: Docker containers, OvS switches, and TC traffic shaping. We have worked our way around all the flaws of state-of-the-art emulators we could identify, by carefully changing design approaches and obsolete technologies. The takeaway of our effort is that scalability issues in distributed network emulation are not ontological, and it is possible –both theoretically and in practice– to make an emulator that can sustain any network however its size and complexity, provided enough resources are available. The challenge, however, is in designing software that achieves this with minimal amounts of resources. In

this route we have put the first bricks by identifying ways that any such emulator can be optimised.

The second objective has been undeniably more challenging. Fidelity –more than once used interchangeably with realism and accuracy– has been a difficult concept to define and manipulate and more difficult to write in the language of mathematics. We have attempted to approach it phenomenally by considering the packet delay –as the network equivalent for timing– as a measure that contains evidence about an emulation's fidelity. We have formalised the idea into a theoretical framework and followed it up with an example implementation as a way to address all the sub-problems that it raises in practice: time synchronisation and accurate delay measurement, packet identification and tracing, and non-intrusive packet monitoring. Having implemented it, we were also able to show how it performs in practice through an example scenario where our measure of fidelity was correlated to objectively inaccurate network behaviour.

In the last contribution we have gone one step further and explored whether collected information about fidelity in the network delay could be used to shed light on the root causes behind emulation *in*fidelity. We have speculated that, indeed, the passively measured overlay delays contain information about the underlay network and can help troubleshoot potential failures. We have drawn from the classical network tomography theory and added particularities of our problem to build heuristics for solving it. The result is an algorithm which uses the passively measured emulated delays to infer the delays caused by the underlay network, and eventually associate high underlay delays to congestion failures. This troubleshooting algorithm has been complemented with a remapping strategy to avoid the identified causes of emulation failure.

8.2 Perspectives on Future Research

Though many of our tools and methodologies were powered by the wider literature on network measurement and modeling, the presented work attempts to deal with a novel concept (fidelity) in a very niche area (network emulation). We hope this will inspire further research into the topic, which we believe can tremendously help network researchers produce better quality scholarships. In particular, we believe the following two axes to be interesting initial questions for such endeavour (the last is another interesting but unrelated side question).

Formal Modeling of Emulated Networks: One Step Closer to Noumenal Fidelity We have argued in Chapter 5 that noumenal fidelity, while much more difficult to use as a measure of emulation accuracy, is a more powerful concept compared to (delay-based) phenomenal fidelity. We have briefly explored the possibility to use tools and theories from formal software and hardware specification and verification: propositional calculus, automata theory, and program semantics. These tools are traditionally used to prove the correctness of designed algorithms and developed systems, and there is no doubt that they can provide *finer* modeling of network operation against which emulations could be evaluated. Such models would be somewhat close to our concept of noumenal fidelity, and could inform with more certainty whether emulation results are more accurate. The only drawback would be scalability, where all the challenge in researching this subject lies.

Delay Tomography with less Assumptions We have built our delay tomography algorithms on the assumption that some knowledge of the underlay network is established: namely, its topology. This is not unrealistic in the contexts of interest to us (infrastructure owned by the user or shared with other users in a grid-like fashion), but can be unworkable in other contexts such as public clouds. The problem of delay tomography can become much harder but its answers paradoxically more valuable as it can help better run the emulated experiment or even renegociate service-level agreements.

eBPF-based Efficient Network Monitoring Our implementation of passive delay measurement used eBPF as a framework to do it in a non-intrusive and efficient manner. This use in fact only brushes the surface of what eBPF can truly achieve in the scope of network monitoring. We believe this tool does not get the attention it deserves as it can replace high-overhead monitoring software (sFlow, NetFlow, etc.) and most often could replace hardware monitoring devices. It is also particularly useful for efficiently monitoring overlay virtual networks where the use of dedicated devices is not possible.

APPENDIX A

PASSIVE DEALY MEASUREMENT: OTHER USE-CASES

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

A.1 Testbed

For the following experiments, we emulate a simple network consisting of two hosts connected by three cascading switches (Figure A.1). This scenario is emulated using Distrinet in a single node of the R2Lab cluster, which is equipped with a CPU Intel Core i7-2600 processor and 8 gigabytes of RAM, and runs a Ubuntu 18.04 Linux distribution (kernel v4.15.0) with basic functionalities and no particular application running in the background. Each host then generates a flow of random-sized packets to the other. No other traffic runs between the two emulated hosts.

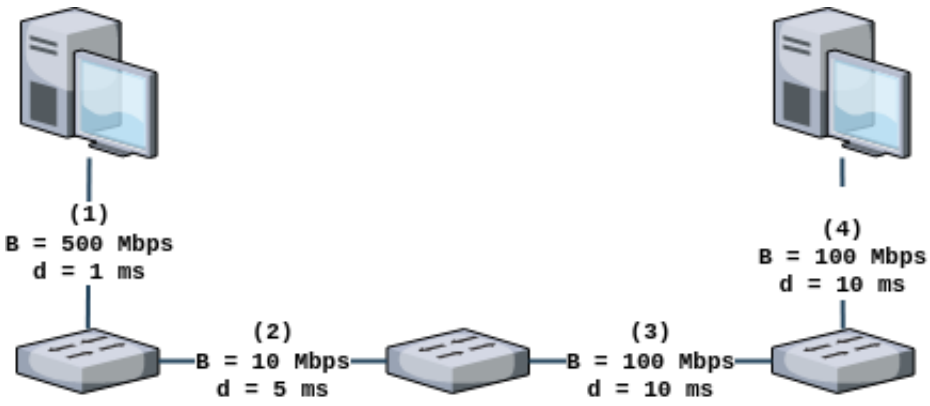


Figure A.1: Emulated testbed for bandwidth estimations. Each link is a full duplex wired link of bandwidth B and propagation delay d.

A.2 One-hop Link Bandwidth

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

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

When enough variables are known, this formula can be used for the estimation of the bandwidth B. In fact, according to the method famously described by the authors in [47, 28], by generating probe packets of varying sizes and then measuring their delays, it is possible to infer the bandwidths of each link along the path. However, thanks to our passive measurement methodology, it is possible to achieve this without injecting packets into the network but rather only from the passively measured delays of data packets.

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

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

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

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

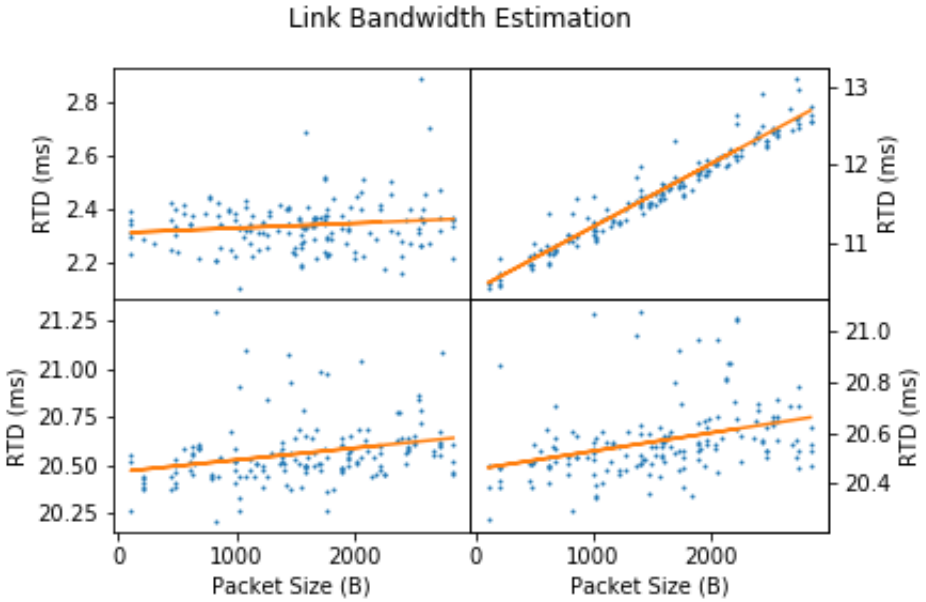


Figure A.2: Transmission speed estimation from passive measurement of RTD. Each data point corresponds to the RTD measurement (y-axis) of a pair of packets of a certain total size (x-axis); the orange lines plot the above formula using the estimated transmission speed. Clockwise from top-left: Link (1),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

(3); and 111.329 Mbps for Link (4)[1]. The small inaccuracy of these estimations is due to the imperfection of the emulator (which adds small processing delay to emulated packets) and the measurement tools. These imperfections can be seen in Figure A.2 where they manifest as small stationary noise added to all packets, which causes a constant drift to the measured delay (captured as an intercept by the linear regression algorithm) and as deviations around the regression line. The estimation accuracy can be made arbitrarily better, provided enough measurements are collected. In general, higher bandwidths cause lower transmission delays, which require more measurements to be distinguished from added noise and captured by the linear regression algorithm.

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

A.3 End-to-end Bottleneck Capacity

Another known method to estimate network bandwidth is *packet pair* [44]. It consists in sending pairs of packets back-to-back while timestamping them both at the source and at the destination, which are generally end-user machines and/or servers, and measuring their spacing difference. Intuitively, two packets sent back-to-back will get spaced along the path each time they cross a link of lower bandwidth. As such, the difference in their timestamps at the destination will be a function of their sizes and of the transmission speed of the slowest link, i.e. the bottleneck capacity of the path. This method has been extensively studied in the scientific literature. In this appendix, however, we only implement it in a *passive measurement* framework using only the tools we have proposed.

Consider two packets P and Q sent from one host A at instants t_P^A and t_Q^A respectively, and received by a host B at instants t_P^B and t_Q^B respectively, through a path with n links of bandwidths $B_1, B_2, ..., B_n$. If all the links of the path are fast enough, the packets will not be further spaced by transmission delay, i.e. $t_Q^B - t_P^B \approx t_Q^A - t_P^A$. However, each slow link i will try to impose its transmission delay on the packet spacing, and we would have $t_Q^B - t_P^B \geq \frac{S_Q}{B_i}$ where S_Q is the size of packet Q. In fact, the authors in [44] argue that if the packets are sent with small enough interpacket time, then their spacing at the destination will be equal to the transmission delay of the second packet on the slowest link, according to the formula:

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

where l is the bottleneck link of the path.

In the same scenario as above, we leverage the timestamps collected at the end hosts to apply this method. For each pair of packets sent *and* received successively, we compute their spacing $t_Q^A - t_P^A$ at the source and $t_Q^B - t_P^B$ at the destination, and use it to estimate the bottleneck bandwidth according to the previous formula. Then from the estimations gotten

from each pair of packet we select the one with the maximum likelihood. Figure A.3 shows the results, from which we obtain an estimated bottleneck bandwidth between 9.997 and 10.013 Mbps.

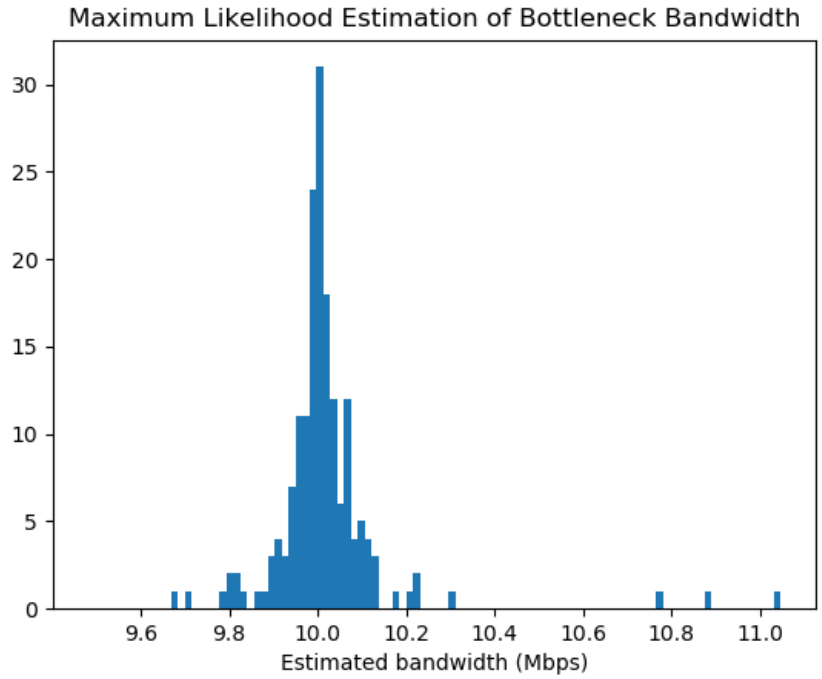


Figure A.3: Estimated bandwidth from different pairs of packets.

REFERENCES

[1] Cisco packet tracer: network simulation tool.

[2] Imunes: Ip networks emulator/simulator, 2004.

[3] Jist/swans, 2004.

[4] Mininet: an instant virtual network on your laptop (or other pc), 2012.

[5] Mininet cluster edition, 2016.

[6] Planetlab bibliography, 2019.

[7] Md F Ahmed, Brendan M Quine, Stoyan Sargoytchev, and AD Stauffer. A review of one-way and two-way experiments to test the isotropy of the speed of light. *Indian Journal of Physics*, 86(9):835–848, 2012.

[8] G Almes, S Kalidindi, and M Zekauskas. Rfc2679: A one-way delay metric for ippm, 1999.

[9] G Almes, S Kalidindi, and M Zekauskas. Rfc2681: A round-trip delay metric for ippm, 1999.

[10] Werner Almesberger et al. Linux network traffic control—implementation overview, 1999.

[11] David G Andersen, Nick Feamster, Steve Bauer, and Hari Balakrishnan. Topology inference from bgp routing dynamics. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 243–248, 2002.

[12] Brice Augustin, Xavier Cuvellier, Benjamin Orgogozo, Fabien Viger, Timur Friedman, Matthieu Latapy, Clémence Magnien, and Renata Teixeira. Avoiding traceroute anoma-

lies with paris traceroute. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 153–158, 2006.

[13] Brice Augustin, Xavier Cuvellier, Benjamin Orgogozo, Fabien Viger, Timur Friedman, Matthieu Latapy, Clémence Magnien, and Renata Teixeira. Avoiding traceroute anomalies with paris traceroute. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 153–158, 2006.

[14] Monya Baker. Reproducibility crisis. *Nature*, 533(26):353–66, 2016.

[15] Kamal Benzekki, Abdeslam El Fergougui, and Abdelbaki Elbelrhiti Elalaoui. Software-defined networking (sdn): a survey. *Security and communication networks*, 9(18):5803–5833, 2016.

[16] Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[17] Andre Broido et al. Internet topology: Connectivity of ip graphs. In *Scalability and traffic control in IP networks*, volume 4526, pages 172–187. SPIE, 2001.

[18] D Brunelli, D Balsamo, G Paci, and L Benini. Temperature compensated time synchronisation in wireless sensor networks. *Electronics letters*, 48(16):1026–1028, 2012.

[19] Mosharaf Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. Vineyard: Virtual network embedding algorithms with coordinated node and link mapping. *IEEE/ACM Transactions on networking*, 20(1):206–219, 2011.

[20] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[21] Dave Clark, Bill Lehr, Steve Bauer, Peyman Faratin, Rahul Sami, and John Wroclawski. Overlay networks and the future of the internet. *Communications and Strategies*, 63:109, 2006.

[22] 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.

[23] James Conley, Ed Andros, Priti Chinai, and Elise Lipkowitz. Use of a game over: Emulation and the video game industry, a white paper. *Nw. J. Tech. & Intell. Prop.*, 2:261, 2003.

[24] Martin Devara. Htb: Hierarchical token bucket, 2003.

[25] Giuseppe Di Lena, Andrea Tomassilli, Frédéric Giroire, Damien Saucez, Thierry Turletti, and Chidung Lac. A right placement makes a happy emulator: a placement module for distributed sdn/nfv emulation. In *ICC 2021-IEEE International Conference on Communications*, pages 1–6. IEEE, 2021.

[26] 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.

[27] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *2010 24th IEEE international conference on advanced information networking and applications*, pages 27–33. Ieee, 2010.

[28] Allen B Downey. Using pathchar to estimate internet link characteristics. *ACM SIGCOMM Computer Communication Review*, 29(4):241–250, 1999.

[29] Nick G Duffield, Joseph Horowitz, F Lo Presti, and D Towsley. Network delay tomography from end-to-end unicast measurements. In *Thyrrhenian Internatinal Workshop on Digital Communications*, pages 576–595. Springer, 2001.

[30] John C Eidson, Mike Fischer, and Joe White. Ieee-1588™ standard for a precision clock synchronization protocol for networked measurement and control systems. In *Proceedings of the 34th Annual Precise Time and Time Interval Systems and Applications Meeting*, pages 243–254, 2002.

[31] Albert Einstein et al. On the electrodynamics of moving bodies. *Annalen der physik*, 17(10):891–921, 1905.

[32] Martin Eklöf, Jenny Ulriksson, and Farshad Moradi. Netsim–a network based environment for modelling and simulation. 01 2004.

[33] Houssam ElBouanani, Chadi Barakat, Guillaume Urvoy-Keller, and Dino Lopez-Pacheco. Collaborative traffic measurement in virtualized data center networks. In *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*, pages 1–3. IEEE, 2019.

[34] Dino Farinacci, Tony Li, Stan Hanks, David Meyer, and Paul Traina. Generic routing encapsulation (gre). Technical report, 2000.

[35] Nick Feamster. Coursera: Software defined networking.

[36] Andreas Fischer, Juan Felipe Botero, Michael Till Beck, Hermann De Meer, and Xavier Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys & Tutorials*, 15(4):1888–1906, 2013.

[37] Ramon R Fontes, Samira Afzal, Samuel HB Brito, Mateus AS Santos, and Christian Esteve Rothenberg. Mininet-wifi: Emulating software-defined wireless networks. In *2015*

11th International Conference on Network and Service Management (CNSM), pages 384–389. IEEE, 2015.

[38] Michel Foucault. *Archaeology of knowledge*. routledge, 2013.

[39] Ramesh Govindan and Hongsuda Tangmunarunkit. Heuristics for internet map discovery. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, volume 3, pages 1371–1380. IEEE, 2000.

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

[41] Brandon Heller. *Reproducible network research with high-fidelity emulation*. Stanford University, 2013.

[42] Stephen Hemminger et al. Network emulation with netem. In *Linux conf au*, volume 5, page 2005. Citeseer, 2005.

[43] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 14(14):527, 2008.

[44] Ningning Hu and Peter Steenkiste. Estimating available bandwidth using packet pair probing. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2002.

[45] Teerawat Issariyakul and Ekram Hossain. Introduction to network simulator 2 (ns2). In *Introduction to network simulator NS2*, pages 1–18. Springer, 2009.

[46] Konstantin Ivanov. *Containerization with LXC*. Packt Publishing Ltd, 2017.

[47] Van Jacobson. Pathchar: A tool to infer characteristics of internet paths, 1997.

[48] Immanuel Kant. *Critique of Pure Reason*. The Cambridge Edition of the Works of Immanuel Kant. Cambridge University Press, New York, NY, 1998. Translated by Paul Guyer and Allen W. Wood.

[49] Ethan Katz-Bassett, Harsha V Madhyastha, Vijay Kumar Adhikari, Colin Scott, Justine Sherry, Peter Van Wesep, Thomas E Anderson, and Arvind Krishnamurthy. Reverse traceroute. In *NSDI*, volume 10, pages 219–234, 2010.

[50] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

[51] Srinivasan Keshav. *REAL: A network simulator*. University of California Berkeley, Calif, USA, 1988.

[52] Alexey N. Kuznetsov. tc-tbf linux man page.

[53] Earl Lawrence, George Michailidis, and Vijayan N Nair. Network delay tomography using flexicast experiments. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(5):785–813, 2006.

[54] Antonio Libri, Andrea Bartolini, Michele Magno, and Luca Benini. Evaluation of synchronization protocols for fine-grain hpc sensor data time-stamping and collection. In *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pages 818–825. IEEE, 2016.

[55] Jean-Francois Lyotard. The postmodern condition. *Manchester: Manchester*, 1994.

[56] Liang Ma, Ting He, Kin K Leung, Ananthram Swami, and Don Towsley. Monitor placement for maximal identifiability in network tomography. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 1447–1455. IEEE, 2014.

[57] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T Sridhar, Mike Bursell, and Chris Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. Technical report, 2014.

[58] Amir Malekzadeh and Mike H MacGregor. Network topology inference from end-to-end unicast measurements. In *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pages 1101–1106. IEEE, 2013.

[59] G Malkin. Rfc1393: Traceroute using an ip option, 1993.

[60] Nivedita Manohar. A survey of virtualization techniques in cloud computing. In *Proceedings of international conference on vlsi, communication, advanced devices, signals & systems and networking (vcasan-2013)*, pages 461–470. Springer, 2013.

[61] Johann Marquez-Barja, Bart Lannoo, Dries Naudts, Bart Braem, Carlos Donato, Vasilis Maglogiannis, Siegfried Mercelis, Rafael Berkvens, Peter Hellinckx, Maarten Weyn, et al. Smart highway: Its-g5 and c2vx based testbed for vehicular communications in real environments enhanced by edge/cloud technologies. In *EuCNC2019, the European Conference on Networks and Communications*. IEEE, 2019.

[62] Steven McCanne, Sally Floyd, and Kevin Fall. ns version 1-lbnl network simulator.

[63] Donald N McCloskey. The rhetoric of economics. *Journal of economic literature*, 21(2):481–517, 1983.

[64] David L Mills. Network time protocol (ntp). Technical report, 1985.

[65] 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.

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

[67] 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.

[68] Gideon Parchomovsky. Publish or perish. *Michigan Law Review*, 98(4):926–952, 2000.

[69] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open {vSwitch}. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, pages 117–130, 2015.

[70] Jon Postel. Internet control message protocol darpa internet program protocol specification. *RFC 792*, 1981.

[71] F Lo Presti, Nick G Duffield, Joseph Horowitz, and Don Towsley. Multicast-based inference of network-internal delay distributions. *IEEE/ACM Transactions On Networking*, 10(6):761–775, 2002.

[72] 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.

[73] Riccardo Ravaioli, Guillaume Urvoy-Keller, and Chadi Barakat. Characterizing icmp rate limitation on routers. In *2015 IEEE International Conference on Communications (ICC)*, pages 6043–6049. IEEE, 2015.

[74] Hartmut Rosa. *Social acceleration: A new theory of modernity*. Columbia University Press, 2013.

[75] Robert Rose. Survey of system virtualization techniques. 2004.

[76] Meng-Fu Shih and Alfred O Hero. Unicast-based inference of network link delay distributions with finite mixture models. *IEEE Transactions on Signal Processing*, 51(8):2219–2228, 2003.

[77] Saba Siraj, A Gupta, and Rinku Badgujar. Network simulation tools survey. *International Journal of Advanced Research in Computer and Communication Engineering*, 1(4):199–206, 2012.

[78] Ion Stoica, Hui Zhang, and TS Eugene Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. *ACM SIGCOMM Computer Communication Review*, 27(4):249–262, 1997.

[79] Marco Antonio To, Marcos Cano, and Preng Biba. Dockemu–a network emulation tool. In *2015 IEEE 29th international conference on advanced information networking and applications workshops*, pages 593–598. IEEE, 2015.

[80] Martino Trevisan, Danilo Giordano, Idilio Drago, Marco Mellia, and Maurizio Munafo. Five years at the edge: Watching internet from the isp network. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pages 1–12, 2018.

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

[82] 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.

[83] Caroline White. Suspected research fraud: difficulties of getting at the truth. *Bmj*, 331(7511):281–288, 2005.

[84] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.

[85] Lisa Yan and Nick McKeown. Learning networking by reproducing research results. *ACM SIGCOMM Computer Communication Review*, 47(2):19–26, 2017.

[86] Li Zhang, Zhen Liu, and C Honghui Xia. Clock synchronization algorithms for network measurements. In *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 160–169. IEEE, 2002.

[87] Tanja Zseby, Maurizio Molina, Nick Duffield, Saverio Niccolini, and Fredric Raspall. Sampling and filtering techniques for ip packet selection. Technical report, 2009.