REPRODUCIBLE NETWORK RESEARCH WITH HIGH-FIDELITY EMULATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Brandon Heller
June 2013

© 2013 by Brandon David Heller. All Rights Reserved.
Re-distributed by Stanford University under license with the author.

This dissertation is online at: http://purl.stanford.edu/zk853sv3422

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Nick McKeown, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christoforos Kozyrakis

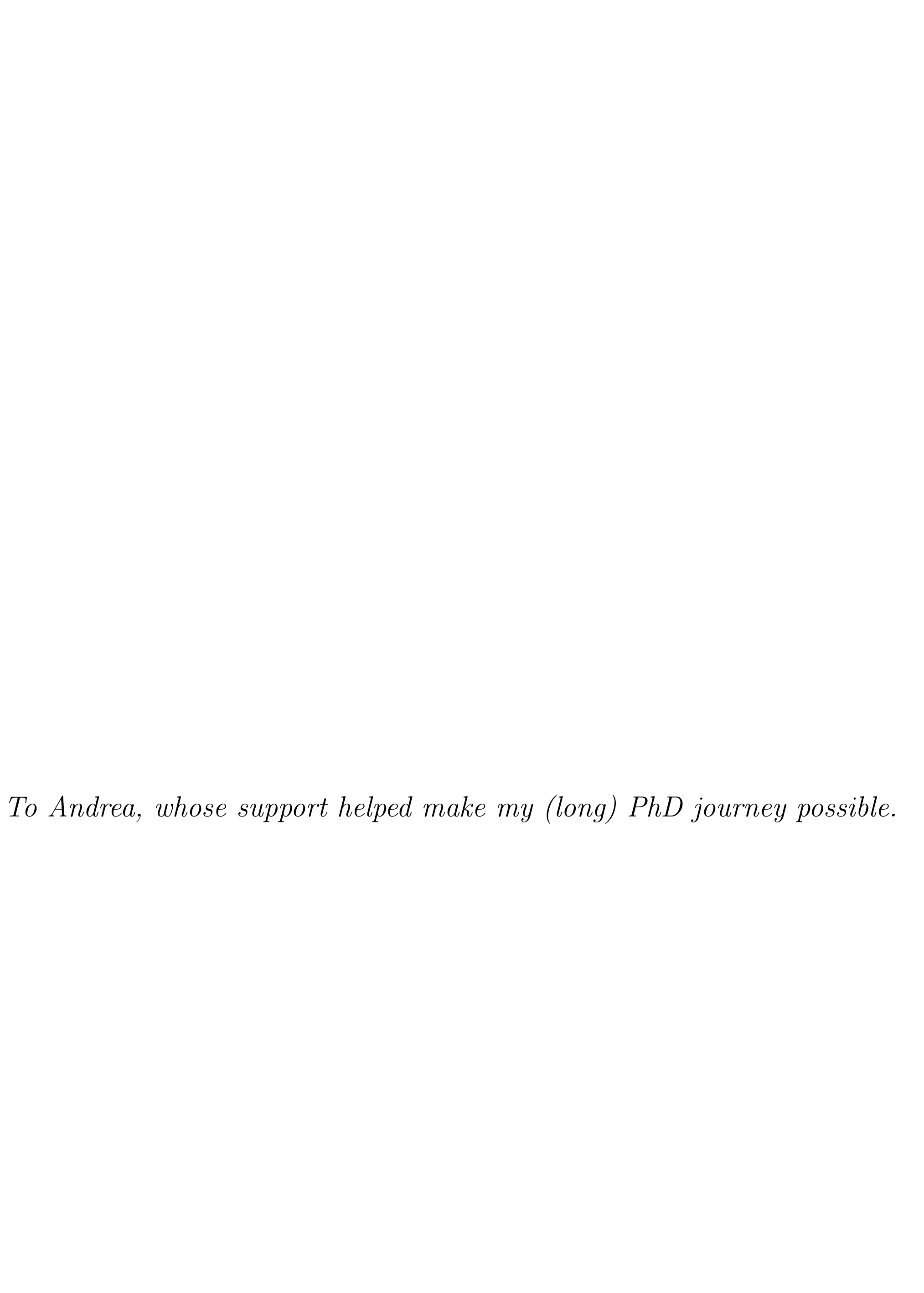
I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Gurudatta Parulkar

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.



To Andrea, whose support helped make my (long) PhD journey possible.

Abstract

In an ideal world, all research papers would be runnable: simply click to replicate the results, using the same setup as the authors. In many computational fields, like Machine Learning or Programming Languages, creating a runnable paper means packaging up the code and data in a virtual machine. However, for Network Systems, the path to a realistic, runnable paper is not so clear. This class of experiments requires many servers, network elements, and packets to run in parallel, and their results depend on accurate timing. Current platform options either provide realism but lack flexibility (e.g., shared testbeds like Emulab [30] cannot support arbitrary topologies) or provide flexibility but lack realism (e.g., discrete-event simulators like ns-2 [57] model end-host code).

This dissertation presents a new approach to enable realistic yet reproducible network experiments: *high-fidelity emulation*. High-fidelity emulation couples a resource-isolating emulator with a monitor to verify properties of the emulation run. Every (wired) network comprises the same basic components, like links, switches, and virtual hosts, and these components behave in highly predictable ways, since they are implemented in hardware. A correct emulation run will maintain the behavior of these components. For example, a wired link should have a constant delay, while a queue should have a fixed capacity. We call these properties "network invariants", and they are universal: they apply regardless of the experiment being run, the system upon which that experiment is run, and even the emulation code. By logging and processing network events, the monitor can quantify the error in an emulation run. Unlike a simulator, the code is fully real: it is the same code that would run on multiple systems, and it captures implementation quirks such as OS interactions, lock conflicts, and resource limits that simulation models inherently abstract away.

The second contribution of this dissertation is Mininet-HiFi, an open-source tool for creating *reproducible* network system experiments. Mininet-HiFi runs an environment of virtual hosts, switches, and links on a modern multi-core server, using real application and kernel code with software-emulated network elements. The approach builds upon recent advancements in lightweight OS-level virtualization to combine the convenience, flexibility, and low cost of discrete-event simulation with the realism of testbeds. To produce evidence that an experiment ran accurately, it logs system events and extracts indicators of fidelity from these logs. In addition to running experiments, the tool has proven useful for interactively developing, testing, sharing, and demonstrating network systems.

The third contribution of this dissertation is the collected outcomes of putting Mininet-HiFi to the test, by using it to reproduce key results from published network experiments such as DCTCP, Hedera, and router buffer sizing, as well as those done by students. In Stanford CS244 in Spring 2012, 37 students attempted to replicate 18 different published results of their own choosing, atop EC2 virtual machines in the cloud. Their experiences suggest that Mininet HiFi makes research results easier to understand, easier to reproduce, and most importantly, easier to build upon.

As a community we seek high-quality results, but our results are rarely reproduced. It is our hope that Mininet-HiFi will spur such a change, by providing networking researchers a way to transform their research papers from “read-only” to “read, write, and execute”.

Acknowledgements

My advisor, ~~James Bond~~ Nick McKeown, gets the first call-out [43]. The thing I find most amazing about Nick is his foresight. When I entered Stanford in fall 2007, the phrases OpenFlow and Software-Defined Network did not yet exist. In group talks, Nick and Guru Parulkar tried to persuade us that the simple, not-so-technically-interesting idea of separating the *control* of a forwarding device from its *hardware* was going to be *big* – no, **really big** – and that we could each play a role in disrupting a multi-billion-dollar networking industry, while simultaneously making networking researchers more relevant than ever to the practice of networking. It sounded too good to be true.

Nick shows us that the best way to predict the future is to create it. Six years later, the story is playing out just as Nick predicted, and I had the privilege to help make this change happen, by acting as the editor of the OpenFlow Specification for three years, doing countless demos to execs to show them the power of software to define network behavior, and spreading the word through tutorials. I blame Nick, in the best way possible, for providing me with these opportunities, as well as instilling the "benevolent criticality" needed to select meaningful problems, along with demonstrating the perseverance to solve them.

Second comes Team Mininet, which includes Bob Lantz, Nikhil Handigol, and Vimal Jeyakumar. Without their combined effort, a system with 1000+ users, or this dissertation, would never have happened, nor would it be as usable or appreciated by it users. This dissertation owes them all a great debt.

Next comes the McKeown group, with whom skills were forged in the sleep-deprivation fires of demo weeks and paper deadlines. Dave, Glen, Yiannis, TY,

Peyman, Jad, KK (and others I've missed), your feedback and support is much-appreciated.

Thanks to my past academic advisors, including Patrick Crowley and Jon Turner, for preparing me to enter the world of research. Thanks to my current reading committee, including Guru Parulkar and Christos Kozyrakis, for enabling me to exit the world of research :-) .

Thanks also to the students from CS244 Spring 2012, whose collected experiments made my thesis statement defensible: Rishita Anubhai, Carl Case, Vaibhav Chidrewar, Christophe Chong, Harshit Chopra, Elliot Conte, Justin Costa-Roberts, MacKenzie Cumings, Jack Dubie, Diego Giovanni Franco, Drew Haven, Benjamin Helsley, John Hiesey, Camille Lamy, Frank Li, Maxine Lim, Joseph Marrama, Omid Mashayekhi, Tom McLaughlin, Eric Muthuri Mibuari, Gary Miguel, Chanh Nguyen, Jitendra Nath Pandey, Anusha Ramesh, David Schneider, Ben Shapero, Angad Singh, Daniel Sommermann, Raman Subramanian, Emin Topalovic, Josh Valdez, Amogh Vasekar, RJ Walsh, Phumchanit Watanaprakornkul, James Whitbeck, Timothy Wong, and Kun Yi. Thanks also to the Hedera, Buffer Sizing, and DCTCP authors for sharing their experiment code, plus the Stanford University Clean Slate Program and a Hewlett-Packard Fellowship, which funded my graduate studies.

Lastly, thanks to Maria, whose unconditional support has been a great influence the entire time I have been in Palo Alto. And of course, many thanks to my family on the East Coast, for helping me to get there.

Contents

Abstract **vii**

Acknowledgements **ix**

1 Introduction **3**

1.1 Network Research Platforms 4
1.2 Platform Goals 7
1.3 Reproducibility 9
1.4 Our Approach: High-Fidelity Emulation 10
1.5 Challenges 12
1.6 Related Work Overview 13
1.7 Organization 14

2 Fidelity and Invariants **19**

2.1 Desired Workflow 19
2.1.1 Events to Log 21
2.1.2 Invariants 22
2.1.3 Invariant Tolerances 25
2.2 Evidence That High Fidelity Is Possible 25

3 Original Mininet: A Container-Based Emulator **33**

3.1 Motivation 33
3.2 Architecture Overview 36
3.3 Workflow 37
3.3.1 Creating a Network 37

3.3.2 Interacting with a Network . . . 40
3.3.3 Customizing a Network . . . 41
3.3.4 Sharing a Network . . . 41
3.3.5 Running on Hardware . . . 42
3.4 Scalability . . . 42
3.4.1 Topology Creation . . . 42
3.4.2 Individual Operations . . . 43
3.4.3 Available Bandwidth . . . 43
3.5 Use Cases . . . 43
3.6 Limitations . . . 46

4 Mininet-HiFi: A High-Fidelity Emulator 47
4.1 Motivation . . . 47
4.2 Design Overview . . . 48
4.2.1 Performance Isolation . . . 49
4.2.2 Resource Provisioning . . . 50
4.2.3 Fidelity Monitoring . . . 51
4.3 Validation Tests . . . 51
4.3.1 Testbed . . . 52
4.3.2 Tests . . . 54
4.3.3 UDP Synchronization Problem + Fix . . . 55
4.4 Microbenchmarks . . . 57
4.4.1 CPU Scheduling . . . 57
4.4.2 Link Scheduling . . . 59
4.4.3 Round-trip latency effects . . . 63
4.4.4 Summary . . . 64
4.5 Experimental Scope . . . 64

5 Experiences Reproducing Research 69
5.1 DCTCP . . . 70
5.2 Hedera . . . 73
5.3 Buffer Sizing . . . 76

5.4 CS244 Spring 2012 79
5.4.1 Project Assignment 80
5.4.2 Project Outcomes 82
5.4.3 Lessons Learned 83

6 Conclusion **87**
6.1 Status Report 88
6.2 Emulating Software-Defined Networks 89
6.3 Future Work 91
6.4 Closing Thoughts 92

List of Tables

1.1 Platform characteristics for reproducible network experiments. 8

3.1 Mininet topology benchmarks: setup time, stop time and memory usage for networks of H hosts and S Open vSwitch kernel switches, tested in a Debian 5/Linux 2.6.33.1 VM on VMware Fusion 3.0 on a MacBook Pro (2.4 GHz intel Core 2 Duo/6 GB). Even in the largest configurations, hosts and switches start up in less than one second each. 43

3.2 Time for basic Mininet operations. Mininet's startup and shutdown performance is dominated by management of virtual Ethernet interfaces in the Linux (2.6.33.1) kernel and `ip link` utility and Open vSwitch startup/shutdown time. 44

3.3 Mininet end-to-end bandwidth, measured with `iperf` through linear chains of user-space (OpenFlow reference) and kernel (Open vSwitch) switches. 44

4.1 **Validation Tests**.
Each graph shows measured interface counters for all flows in that test, comparing data from Mininet-HiFi running on a dual-core Core2 1.86GHz, 2GB machine against a hardware testbed with eight machines of identical specifications. In many cases the individual CDF lines for Mininet-HiFi results overlap.
Legend: **Mininet-HiFi: solid red lines.** **Testbed: dashed black lines.** 53

4.2 Comparison of schedulers for different numbers of *vhosts*. The target is 50% total utilization of a four core CPU. 58

5.1 Caption 80

6.1 Tutorials Using Mininet 89

List of Figures

1.1 Emulator realism suffers without adequate performance isolation. . . 6
1.2 The approach to high-fidelity emulation advocated by this dissertation. 11
1.3 Sources of emulator infidelity: event overlap, variable delays, and event timer precision. 12

2.1 A workflow for high-fidelity emulation. Questions raised are shown in blue. 20
2.2 CPU utilization logging is insufficient to check emulation fidelity. . . . 21
2.3 Typical wired network, showing measurable delays experienced by packets. 22
2.4 Simplified view of regular TCP vs Data Center TCP [9]. 26
2.5 Hardware results with DCTCP at 100 Mb/s. 27
2.6 Emulator results with DCTCP at varying speeds. 28
2.7 Packet Spacing Invariant with DCTCP. Going from 80 Mb/s to 160/s, the emulation falls behind and the 10% point on the CCDF increases by over 25x. 29

3.1 Traditional network on the left, with a Software-Defined Network (SDN) on the right. SDN physically separates the control plane from the forwarding devices in a network. Network features are implemented atop a centralized view of the network, which is provided by the Network OS. SDN was a primary motivator for developing and releasing Mininet. 34
3.2 Mininet creates a virtual network by placing host processes in network namespaces and connecting them with virtual Ethernet (veth) pairs. In this example, they connect to a user-space OpenFlow switch. . . . 38

3.3 The `console.py` application uses Mininet's API to interact with and monitor multiple hosts, switches and controllers. The text shows `iperf` running on each of 16 hosts. 39

3.4 MiniEdit is a simple graphical network editor that uses Mininet to turn a graph into a live network when the Run button is pressed; clicking a node opens up a terminal window for that node. 40

4.1 A Mininet-HiFi network emulates performance constraints of links and hosts. Dashed blue lines and text indicate added performance isolation and monitoring features that distinguish Mininet-HiFi from Original Mininet. 49

4.2 The testbed includes eight machines connected in a dumbbell topology using a single physical switch. 52

4.3 The Fork-In Test, where multiple UDP senders send to one receiver, originally led to one sender receiving all the bandwidth. 55

4.4 CPU timer fidelity as a function of N and t. Solid lines are for Test 1; dashed lines are for Test 2. 59

4.5 Rates for a single `htb` & `hfsc`-scheduled link, fed by a TCP stream. 60

4.6 Breakdown of CPU usage as we vary the number of links and bandwidth allotted to each link. 61

4.7 Provisioning graph for the 8-core platform. Y-axis is the *total* switching capacity observed in the pair test and x-axis is the number of parallel links. The numbers show the total CPU usage for a run at that configuration. 62

4.8 Round-trip latency in the presence of background load. 63

5.1 Topology for TCP and DCTCP experiments. 70

5.2 Reproduced results for DCTCP [9] with Mininet-HiFi and a identically configured hardware setup. Figure 5.2(b) shows that the queue occupancy with Mininet-HiFi stays within 2 packets of hardware. . . 71

5.3 Complementary CDF of inter-dequeue time deviations from ideal; high fidelity at 100 Mb/s, low fidelity at 1 Gb/s. 72

5.4 Effective throughput with ECMP routing on a $k = 4$ Fat Tree vs. an equivalent non-blocking switch. Links are set to 10 Mb/s in Mininet-HiFi and 1 Gb/s in the hardware testbed [6]. 74
5.5 Buffer sizing experiment topology. 76
5.6 Results: Buffers needed for 99% utilization, comparing results from the Internet2 testbed, Mininet-HiFi, and the theoretical upper bound. 77
5.7 Verifying Fidelity: A large number of flows increases the inter-dequeue time deviations only by 40% from that of an ideal 100% utilized link, and only for 1% of all packets in a 2s time window. 79

Chapter 1

Introduction

Computer networks make today's communication possible. They enable phone calls across an ocean, grease the wheels of commerce, fuel revolutions, and provide an endless stream of cat videos. As this list of uses grows, we continue to expect faster, cheaper, and more reliable networks.

Enter the network systems researcher, whose job is to uncover issues with today's networks, wherever these issues are, and to solve them, by changing whatever piece of the network they can, be it a host networking stack, hypervisor, switch, or topology. To provide two examples, network systems researchers design and evaluate new topologies and routing strategies to build scale-out data centers [6, 37, 55, 77], as well as design new transport protocols to make use of multiple paths, reduce delays, and reduce network hardware requirements [9, 90].

The challenge for a network systems researcher is that no matter how simple the solution, convincing others to publish it, try it, and eventually adopt it requires experimentation — not just to make sure the idea works in its intended context, but also to understand what happens to the results as parameters such as scale, speeds, and traffic assumptions, are changed. Turning a promising idea into a deployed system requires a higher level of validation: typically, a demonstration at sufficient scale to interest operators who might deploy the idea or vendors who might implement the idea in shipping products. However, our reliance on production networks prevents them from being a feasible or prudent place to test new ideas. Hence, many turn to network research platforms — often created *by* researchers, and *for* researchers —

to run interestingly-large-scale experiments, at a fraction of the cost of a full-scale hardware-based network.

1.1 Network Research Platforms

The most commonly used platform types in networking systems research are *simulators*, *testbeds*, and *emulators*. I now discuss each of these options with a particular focus on their reproducibility and realism properties. With today's platforms, these two goals are hard to achieve simultaneously, and the next two sections will explain why these two goals are so important and motivate a new kind of network research platform.

Discrete-Event Simulation. Network simulators advance virtual time as a result of simulated events [57, 58, 64]. Discrete is the key; packet and application events happen at an instant in virtual time. In a simulator, two events can occur at exactly the same time and execution will pause until all such events have been processed. These events are generated by a set of models that cover low-level hardware, e.g., queues and links, as well models that cover high-level application behavior, e.g., interacting applications and users. The underlying simulator and model code runs in user-space and produces the same result each time, regardless of the specifications of the machine on which the experiment runs;[1] hence, simulation results are considered easy to reproduce.

But, we don't trust simulation, because the results obtained from simulators may not be realistic. Simulators try to simplify the behavior of the world, but there's always a risk that by simplifying, one changes the resulting behavior and sees a different result from a hardware system configured equivalently to the simulator. Results are not believable unless they're validated, and each model must be validated. Plus, the code tends not to be the same code you would run on a real server; for example, the Linux TCP stack is necessarily different than the one used in NS-2, and it would be impractical for every simulator author to port every change made to Linux alone. To fit within a simulator framework, it is unlikely that simulator code

[1] Results are deterministic for a given hash seed but of course vary by run.

will use the same system calls, implement the same locking, or experience the same resource limits. These realism concerns motivate the use of testbeds.

Testbeds. The networking community has been great about making testbeds openly available. These can be at a single location with hosts and switches like Emulab (*vEmulab* in this paper) [30] or more spread out like GENI [34] and OFELIA[59]. They can be shared to amortize their construction cost across a community of users [34, 22, 25, 30] or they can be specific to one project [6, 9, 13]. Many testbeds provide time-shared access to dedicated hardware; physically isolating resources, such as links, switches and machines, prevents experiments from affecting each other. Furthermore, running with real hardware eliminates realism concerns.

However, the problem with testbed experiments is that their results can be hard to reproduce, or even just produce. First, an experiment may not be possible to run on a testbed because of topology restrictions; for example, GENI only supports tree topologies, and a new, more exotic topology like a random graph is unlikely to be supported [77]. Or, if we want to change the way a box forward packets, such as adding OpenFlow [63], we may not have access to change the firmware. There are also possible issues with availability of the testbed, as contention occurs before conference deadlines. Finally, since the testbed may not be available indefinitely, the result may not be reproducible in the future.[2]

Network Emulators. The third option, network emulation, describes software running on a PC that configures and runs everything that you would find in a real network: the switches, the links, the packets, and the servers. The core of the network is like a simulator, in that it processes events, but the difference is that these events happen in continuous time. The emulated servers run code, rather than a discrete-event model.

Like testbeds, emulators run real code (e.g., OS kernel, network applications) with real network traffic. Like simulators, they support arbitrary topologies and their virtual "hardware" costs very little. There are two main categories of network emulators. The first, *Full-System Emulation*, e.g. DieCast [39] or VMs coupled using

[2] One could address this somewhat by designing a testbed-abstraction layer, so that an experiment could be targeted to multiple testbeds, or implement a resource-constraint language to define the resource request, somewhat like the GENI RSpec [1].

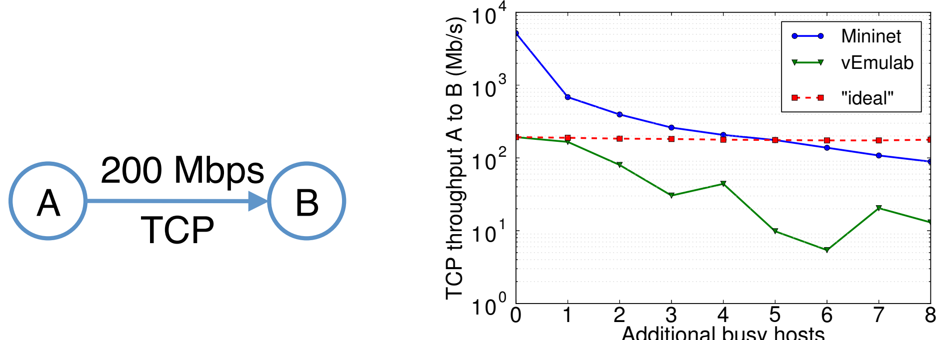


Figure 1.1: Emulator realism suffers without adequate performance isolation.

Open vSwitch [65], uses one full virtual machine per host. The second, *Container-Based Emulation* (CBE), e.g. virtual Emulab [42], NetKit [66], Trellis [14], CORE [5], Mininet [47] and many others, employs process-level virtualization – a lighter form of virtualization in which many aspects of the system are shared. By sharing system resources, such as page tables, kernel data structures, and the file system, lightweight OS-level containers achieve better scalability than VM-based systems, permitting a larger number of small virtual hosts on a single system [47, 78]. In *Container-Based-Emulators (CBEs)*, which use lightweight containers, each virtual host is a simply a group of user-space processes, and the cost of adding one is only the cost of spawning a new process.

An ideal emulator is indistinguishable from hardware, in that no code changes are required to port an experiment and no performance differences result. However, emulators, regardless of their type, may not provide adequate performance isolation for experiments. Figure 1.1 plots the TCP bandwidth for a simple benchmark where two virtual hosts communicate at full speed over a 200Mb/s link. In the background, we vary the CPU load on a number of other (non-communicating) virtual hosts. On an emulator with no isolation (Original Mininet), the TCP flow exceeds the desired performance at first, then degrades gradually as the background load increases. Though vEmulab correctly rate-limits the links, rate-limiting alone is not sufficient: the increasing background load affects the network performance of other virtual hosts, leading to unrealistic, load-dependent results. Ideally, the TCP flow would see a constant throughput of 200Mb/s, irrespective of the background load on the other virtual

hosts.

Unfortunately, even with properly implemented resource limits and isolation between virtual hosts, the emulator still provides no guarantee (or even a dependable indicator) that the experiment is running in a realistic way; high-level throughput numbers do not guarantee low-level event timing fidelity. Perhaps this issue has held back the use of software emulation results, as they are rarely found in networking papers.

1.2 Platform Goals

Having surveyed the available platforms, with each possessing advantages and disadvantages, we now describe the goals of a (hypothetical) ideal research platform a little more precisely. First, a convincing experiment requires *realism*, in at least three ways:

Functional realism. The system should have the same functionality as real hardware in a real deployment, and should execute exactly the same code.

Timing realism. The timing behavior of the system should be close to (or indistinguishable from) the behavior of deployed hardware. The system should detect when timing realism is violated.

Traffic realism. The system should be capable of generating and receiving real, interactive network traffic to and from the Internet, or from users or systems on a local network.

In addition to providing realism, the system should be *flexible*, to support arbitrary experiments:

Topology flexibility. It should be easy to create an experiment with any topology.

Scale. The system should support experiments with large numbers of hosts, switches, and links.

The third group relates to one's ability to duplicate results created with it, now and in the future.

Easy replication. It should be easy to duplicate an experimental setup and run

	Simulators	Testbeds		Emulators
		Shared	Custom	
Functional Realism		✓	✓	✓
Timing Realism	✓	✓	✓	???
Traffic Realism		✓	✓	✓
Topology Flexibility	✓	(limited)		✓
Scale	✓	✓	✓	(limited)
Easy Replication	✓	✓		✓
Low cost	✓			✓

Table 1.1: Platform characteristics for reproducible network experiments.

an experiment.

Low cost. It should be inexpensive to duplicate an experimental platform, e.g. for 1000 students in a massive online course.

Table 1.1 compares how well each platform described in §1.1 supports our goals for realism, flexibility, and reproducibility. Each platform falls short in at least one of these goals: simulation lacks functional realism, testbeds lack topology flexibility, and emulators lack demonstrated timing realism. Unfortunately, our takeaway here is that network research tends not to be both easily reproducible and — realistic. With today's platforms, you can have one, or the other, but not both.

Fortunately, there is no fundamental tradeoff here, just a design space that has not been fully explored. In particular, an emulator with provable timing fidelity would be attractive, because its results would be reproducible on any PC, and its realism could potentially match that of hardware. Such an emulator would enable *runnable papers* — a complete packaging of an experiment on a VM that runs all the real code.

Now we consider the broader issue of reproducibility in network systems, and discuss how a platform to enable runnable (realistic and reproducible) network systems research papers might fit in.

1.3 Reproducibility

The scientific method dictates that experiments must be reproduced before they are considered valid; in physics and medicine, reproduction is a part of the culture. Yet in computer networking, reproduced works are not the culture.[3] Papers are often released without the complete code, scripts, and data used to produce the results.

As the computational geophysics professor Donoho noted: [15],

> *An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.*

Reproducible, runnable papers do not appear to be standard practice in network systems research either, or indeed Computer Science at large, so calls from Knuth [45], Claerbout [23], Donoho [15] and Vandewalle [85] for reproducible experiments and results still resonate. If papers were runnable — that is, easily reproducible — a number of benefits would follow:

- It would be easier to understand and evaluate the work, because you could access details left out of the paper for space reasons, like parameter settings and algorithm specifics, and then you could go make a tweak and see what happens.

- It would be easier to transfer ideas, because they would be in a more usable form, one that does not require replicating infrastructure.

- The third, and possibly the most compelling reason, is that it becomes easier to build upon the work of others, both in academia and industry.

These benefits are in addition to adding confidence to results and weeding out questionable results.

[3]The NSDI community award is one rare exception, and is given to “the best paper whose code and/or data set is made publicly available by the final papers deadline.”

Why reproducibility is hard. If runnable papers would be a good thing, then why aren't all networking research papers like this? For many areas of CS, like machine learning or programming languages, creating a reproducible result is relatively straightforward. One packages the code, data, and scripts together on a VM, or releases instructions and code, and any PC will work as a platform to run the VM or code.

But for many network systems projects, running the experiment is not as simple. These experiments are inherently parallel: they all have servers, network elements, and packets running at the same time. This category includes projects like congestion control, routing, new topologies, or new mixes. It does not cover hardware design or measurement studies, but it does cover many of the ideas and papers in networking. For experiments with network systems, the platform must run a large set of processes (applications and protocols) concurrently, without losing accuracy, to be reproducible.

This dissertation considers the question of whether it is possible to enable reproducible network systems experiments that have believable, useful realism, on a single PC.

1.4 Our Approach: High-Fidelity Emulation

My solution to this problem of hard-to-reproduce networking results is to build an emulator whose results one can trust, as well as verify. This dissertation defends the following thesis statement:

High-fidelity emulation, the combination of resource isolation and fidelity monitoring, enables network systems experiments that are realistic, verifiable, and reproducible.

This dissertation advocates reproducible networking experiments that use *High-Fidelity Emulation*, where real code runs on an emulated network that has careful resource isolation and fidelity monitoring. This combination is depicted in Figure 1.2.

Resource isolation provides a network whose links, queues, and switches behave like hardware, and whose emulated hosts each get an equal amount of CPU.

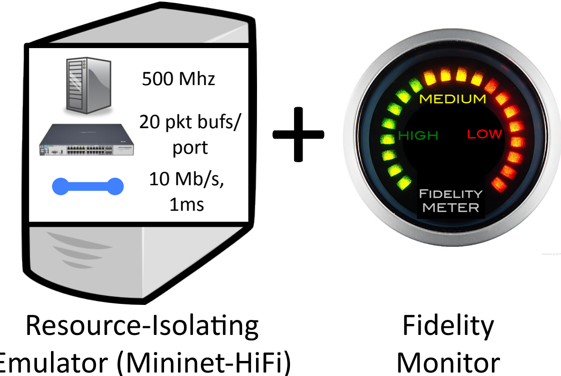


Figure 1.2: The approach to high-fidelity emulation advocated by this dissertation.

Fidelity Monitoring provides a gauge to measure the faithfulness of the network emulation, using the concept of *network invariants*, which are properties we can track to verify the timing error(s) in a particular emulation run. This technique is general, in that it does not depend on the particular emulator implementation, as all wired experiments share the same set of invariants. By quantifying the errors, and comparing them to known or measured hardware timing variability, we can reason about the fidelity of an emulation run. This technique is needed because it is hard to predict in advance whether an experiment has been provisioned with sufficient resources.

High-Fidelity Emulation provides the topology flexibility, low cost, and repeatability of simulation with the functional realism of testbeds. These two characteristics — realism and reproducibility — make it an appealing candidate for publishing research in an easily reproducible form, as a runnable paper.

My implementation of a High-Fidelity Emulator, *Mininet-HiFi*, is described later in §3 in full detail. In brief, Mininet-HiFi employs lightweight, OS-level virtualization techniques, plus packet and process schedulers. It builds on the kernel event system to store a log of relevant events and then post-processes them to report timing fidelity. As of March 2013, 45 networking experiments based on Mininet-HiFi have been released with blog entries on the Reproducing Network Research Blog [3], and the Mininet mailing list counts over 680 users.

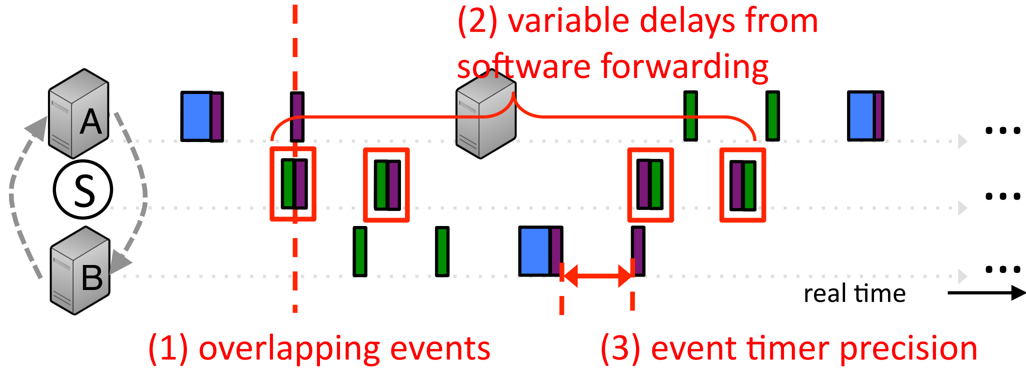


Figure 1.3: Sources of emulator infidelity: event overlap, variable delays, and event timer precision.

1.5 Challenges

Using real code provides a nice starting point for functional fidelity. However, achieving *timing* fidelity is more of a challenge, as an emulator potentially runs events and code in serial, rather than parallel, like a testbed. This serialization, which occurs whenever there are more active events than parallel cores, creates the potential for situations where an emulated network must incur timing errors and cannot exactly match the the realism of a testbed.

Figure 1.3 shows a simple experiment that helps to discuss sources of emulator timing infidelity. Hosts A and B are connected by a switch and are engaging in a continuous ping, where A sends two packets to B and then B sends two packets back to A. Time spent in the ping application is shown in blue, packet transmissions are shown in purple, and packet receptions are shown in green.

Overlapping Events. It just so happens that the first packet is getting forwarded by the switch at the same time host A is sending its second packet. If the system only has one processing core, then one of these events must get delayed.

Software Forwarding Delays. Another possible issue is variability in software forwarding delays. Hardware forwarding frequently employs parallel TCAM searches to handle MAC and IP table lookups in constant time. Software forwarding tends to use algorithmic techniques that can encounter variability from cache effects, as well as take longer, and hence increase the chance for event stack-up. For example, Open

vSwitch [65] sends the first packet of every new flow to user-space, where a lookup occurs, and then it pushes down a forwarding match for following packets in that flow. The speed of this lookup depends on the number of installed rules that must be traversed. Once cached, the lookup is much faster, as it takes place entirely in the kernel; compare this to an Ethernet switch that learns addresses and experiences no such first-packet-of-each-flow delays.

Event Timer Precision. A third source of errors is event timer precision. To prevent interrupts from saturating the processor and causing process starvation, OSes like Linux intentionally prevent the use of short timers in the kernel. In addition, the sources of time on which these timers are based may experience drifts and periodic corrections.

Even worse, these sources of variability can be encountered on every packet hop through a switch, and hence the error can increase proportionally in topologies with longer paths.

Each of these situations can affect the fidelity of the emulator, in the sense that its behavior can differ from hardware. To be trustworthy, an emulator must be able to track when these timing errors are happening, and ideally, should eliminate as many causes of these as possible.

1.6 Related Work Overview

Techniques and platforms for network *emulation* have a rich history and expanded greatly in the early 2000s. Testbeds such as PlanetLab [22] and Emulab [30] make available large numbers of machines and network links for researchers to programmatically instantiate experiments. These platforms use tools such as NIST Net [18], Dummynet [17], and `netem` [53], which each configure network link properties such as delays, drops and reordering. Emulators built on full-system virtualization [11], like DieCast [39] (which superseded ModelNet [84]) and several other projects [61], use virtual machines to realistically emulate end hosts. However, VM size and overhead may limit scalability, and variability introduced by hypervisor scheduling can reduce performance fidelity [89].

To address these issues, DieCast uses *time dilation* [38], a technique where a hypervisor slows down a VM's perceived passage of time to yield effectively faster link rates and better scalability. SliceTime [89] takes an alternate *synchronized virtual time* approach – a hybrid of emulation and simulation that trades off real-time operation to achieve scalability and timing accuracy. SliceTime runs hosts in VMs and synchronizes time between VMs and simulation, combining code fidelity with simulation control and visibility. FPGA-based simulators have demonstrated the ability to replicate data center results, including TCP Incast [86], using simpler processor cores [80].

The technique of *container-based virtualization* [78] has become increasingly popular due to its efficiency and scalability advantages over full-system virtualization. Mininet [47], Trellis [14], IMUNES [91], vEmulab [42], and Crossbow [82] exploit lightweight virtualization features built for their respective OSes. For example, Mininet uses Linux containers [49, 21], vEmulab uses FreeBSD jails, and IMUNES uses OpenSolaris zones.

Mininet-HiFi also exploits lightweight virtualization, but adds resource isolation and monitoring to verify that an experiment has run with high fidelity. In this dissertation, we demonstrate not only the feasibility of a fidelity-tracking Container-Based Emulator on a single system, but also show that these techniques can be used to replicate previously published results.

1.7 Organization

This dissertation makes the following contributions:

- The introduction of a range of *network invariants* as a means to track the realized fidelity of an emulation run (§2).

- Implementation of a High-Fidelity Container-Based Emulator, Mininet-HiFi,[4] which enables reproducible network experiments using resource isolation, provisioning, and monitoring mechanisms (§3).

[4] Available at `http://mininet.org`.

- Reproduced experiments from published networking research papers, including DCTCP, Hedera, and Sizing Router Buffers (§5.1-§5.2).
- Practical lessons learned from unleashing 37 budding researchers in Stanford's *CS244: Advanced Topics in Networking* course upon 13 other published papers (§5.4).

To demonstrate that network systems research can indeed be made repeatable, each complete experiment described in this dissertation can be repeated by running a script on an Amazon EC2 [29] instance or on a physical server, to meet the highest level of the reproducibility hierarchy defined by Vandewalle in [85]:

> *The results can be easily reproduced by an independent researcher with at most 15 min of user effort, requiring only standard, freely available tools.*

Following Claerbout's model [23] clicking on each figure in the PDF (when viewed electronically) links to instructions to replicate the experiment that generated the figure. We encourage network researchers to put the experiments described in this thesis to the test and replicate results for themselves.

Chapter 2

Fidelity and Invariants

This chapter walks through a workflow for realizing a high-fidelity experiment on a container-based emulator. Our goals are to motivate the use of *network invariants* and explain this approach from a user's perspective; we delay the technical details of how to build the emulator and monitor invariants to the next chapter.

2.1 Desired Workflow

Informally, by *high-fidelity emulation*, we mean emulation where the application behavior matches that of parallel systems running on a hardware-based network, to within some experimenter-defined error; this level of fidelity does not require the exact same timing of events. *Guaranteeing* high fidelity in advance with a real-time emulator is challenging due to the inherent sources of variability described in §1.5. Even if these sources of variability were eliminated, *predicting* fidelity would still be hard, because it would require prior knowledge of how many events will simultaneously occur, along with the maximum time that would be required to process them. However, *measuring* fidelity is possible.

This observation — that measuring fidelity may be enough to support our goal of high-fidelity emulation — motivates the workflow shown in Figure 2.1. The experimenter first provides the topology and all code required to run the experiment. Then she runs that experiment on emulator on a PC, with logging enabled. After the

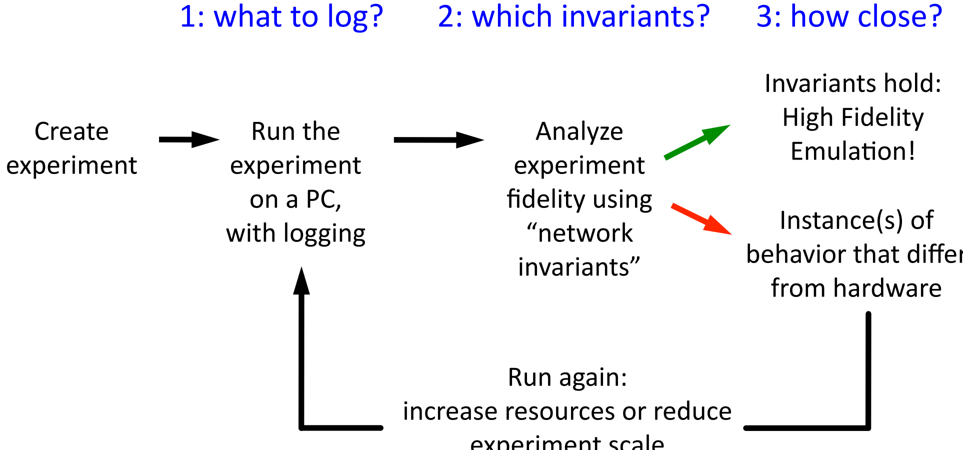


Figure 2.1: A workflow for high-fidelity emulation. Questions raised are shown in blue.

experiment is over, she uses an automated software tool that analyzes experiment fidelity by employing the concept of *network invariants*. A network invariant is simply a timing property that should always hold for any network that comprises wired links and output-queued switches. It should apply regardless of the topology, the traffic pattern, the behavior of the application-level code, and all the transport protocols in use.

If every one of those invariants hold, we consider the experiment to be high fidelity. But if there is even a single instance where an invariant was clearly violated, that is, where its behavior differs from hardware, then that experiment's results are in doubt. When this happens, the experimenter needs to either scale back the experiment in some way, like use fewer hosts or slower links, or add resources by procuring a machine with more or faster cores.

While it does not guarantee a high-fidelity result, this workflow is appealingly general; it applies to any emulator, any PC, and any experiment. It raises three questions, which the next three subsections address. First, what events should we log? Second, which network invariants should we check? Third, how close to perfect must the measured network invariants be to indicate a valid experiment?

2.1.1 Events to Log

Logging the utilization of the CPU running the emulator would be a natural starting point for measuring emulator fidelity. First, consider event-driven programs, which either fire a timer for the future, process a packet, or generate a new packet every time the program is scheduled in. The network protocols running below are also event driven, and similarly issue timers, consume packets, or generate packets. Each program or protocol event requires some amount of processor time; when the measured CPU utilization approaches 100%, one might assume that there is no slack in the system, and that it is approaching (or has already passed) the cliff beyond which the emulation run is invalid. However, at 99% CPU, the emulation run may be valid, with just enough CPU cycles to run the experiment, and evenly-spaced packets may have yielded no overlapping events.

Now consider a non-event-driven program, such as a traffic generator that spins in a loop. The CPU value tells us even less now; 100% CPU may indicate correct operation.

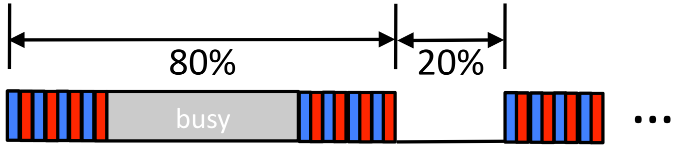


Figure 2.2: CPU utilization logging is insufficient to check emulation fidelity.

The problem is that for *any* CPU utilization value, an emulation run can still return wrong results. The reason could be as simple as the example shown in Figure 2.2, where a burst of busy time occurs, yet over the measurement interval, the CPU utilization is well below 100%. Busy time could be caused by any number of events. Perhaps a higher-priority OS process or hypervisor preempted network handling for a bit. A broadcast packet could take so long to replicate that it delays the time at which receivers can start processing the message. Or, a traffic sender could spin and block the execution of the receiver until it is preempted, if the system has fewer cores than the total of senders and receivers.

Our conclusion is that CPU utilization is insufficient for monitoring fidelity. We

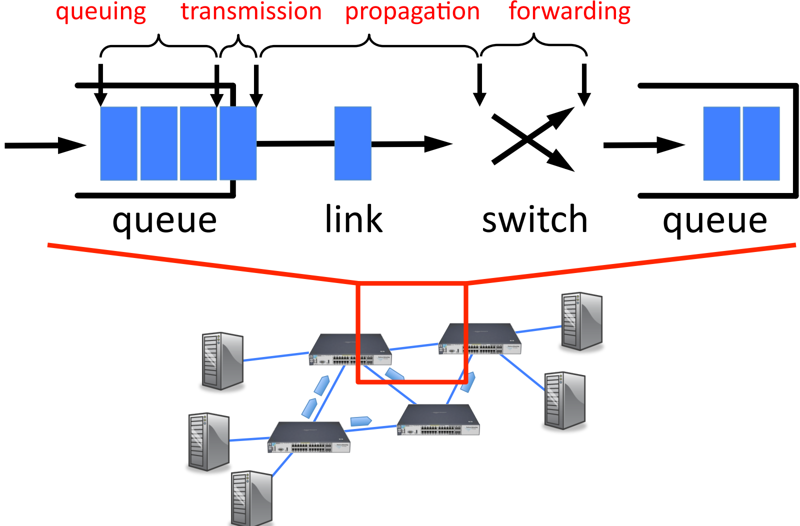


Figure 2.3: Typical wired network, showing measurable delays experienced by packets.

need something better that considers finer-grained event timings.

2.1.2 Invariants

We now consider the use of invariants that are based on expected timings of packets.

Network Invariants. Every (wired) network looks something like Figure 2.3: it has switches, hosts, and links, plus packets in flight. The top of this figure represents a zoomed-in view of the red box in the figure and traces a packet crossing from left to right. This packet will experience a number of delays. First, if there are any packets in the queue, it will need to wait, depending on how many bytes of packets are in front:

$$queue\ delay = t_{exit_queue} - t_{enter_queue} = Q_{size_in_bytes}/Rate \tag{2.1}$$

When the packet moves onto the wire, it will see a transmission delay based on the link rate and packet size:

$$transmission\ delay = t_{stop} - t_{start} = \frac{packet_size_in_bytes}{rate} \tag{2.2}$$

Passing through the link, the packet will experience a propagation delay of known length, which depends on the speed of light in the transmission medium:

$$propagation\ delay = wire_length/c_{in_medium} \tag{2.3}$$

Next, the packet experiences a forwarding delay as it traverses a switch, and this delay depends on the internal complexity of the switch. Hardware switches implemented with multiple chips or a crossbar scheduler impose variable delays. However, for many experiments, assuming a perfect output-queued switch is an acceptable choice. Transport protocols tend to have some timing slack that can absorb these variations in practice. Faithfully implementing the internal scheduler details tends to be impossible when such details are not disclosed. Lastly, if an experiment fails with a simple output-queued switch, it is unlikely to work with something more complex.

Assuming a perfect output-queued switch:

$$forwarding\ delay = t_{egress} - t_{ingress} = k \tag{2.4}$$

These equations are all simple and *testable*. In fact, to test them, one must only log timestamps for packet events.[1] A single-system emulator has a single clock domain, which enables easy, accurate timestamps. We can log all these timestamps and process the log after the experiment has completed.[2] These all happen to be examples of invariants defined on the timing of *single* packets, which requires logging enqueue and dequeue timestamps only.

Other network timing invariants deal with gaps between packets. The first is packet spacing. Whenever a queue is occupied, the gap between two packet transmissions, P1 and P2, should be equal to the time it takes to send the first packet:

$$transmission\ gap : t_{P2} - t_{P1} = \frac{bytes_{P1}}{rate} \tag{2.5}$$

This is a special case of the link capacity constraint, which says that sweeping

[1] Queue occupancy logs can help to check invariants, but these are optional, as they can be reconstructed from the enqueue and dequeue logs.

[2] Or, if the overhead is low enough, we can process these in real-time; we do not explore this option further.

across time and looking at a link, its configured rate should never be exceeded:

$$link\ capacity : rate_{measured} \leq rate_{configured} \tag{2.6}$$

Since packet transmissions in an emulator occur as an event, rather than a continuous hardware process, the measurement interval must be at least one packet size.

If all these conditions hold — that is, the measured invariant values are all a small difference away from the expected values — then the network behavior should match reality.

Host Invariants. However, there is still the potential fidelity issue of multiplexing the execution of hosts, especially when the number of virtual hosts actively processing events exceeds the number of available parallel processors. If the emulation is fully event-driven, and each event is handled infinitely fast, then the issue of multiplexing host events goes away. In reality, a host may spin in a loop or take a non-negligible time to process a packet and generate an event. These *serialization delays* occur when event execution is delayed due to the limited parallelism of an emulator, and we must track them to expose timing differences from truly parallel hardware.

The first host invariant, *preempt-to-schedule latency*, is the time between an event triggering (such as a packet reception) and the event getting processed (such as the corresponding virtual host receiving a wakeup event and starting to process the event).[3] Different hypervisor or process scheduler implementations may yield different preempt-to-schedule latencies, especially with multiple cores.

An additional host invariant check, *CPU capacity*, is similar to the link capacity invariant; it checks that hosts getting a fraction of a processor are, in fact, receiving no more processing time than their configured amount:

$$CPU\ Capacity : cpu_bandwidth_{measured} \leq cpu_bandwidth_{configured} \tag{2.7}$$

[3] A "trivially perfect" emulation operating on an instruction-by-instruction basis, similar to full-system simulators, would perfectly satisfy this invariant, but it would unduly slow down the experiment.

2.1.3 Invariant Tolerances

Having identified the timing invariants to check, the next question is how close the measured invariant values must be to ideal values to distinguish between high-fidelity and low-fidelity invariant runs. A conservative check is desirable here to avoid false positives, although in practice, many protocols, especially bulk transport ones like TCP, will tend to admit some slack in their timing before their behavior changes. Our observation is that matching the qualitative behavior of the original experiment — high fidelity — doesn't mean that timing errors must always reach zero, but instead, that they are reasonable and match what hardware would experience.

Here are some examples for timing delay in real hardware:

- **Clock drift** occurs when two clocks are slightly out of sync, which occurs due to manufacturing differences in crystals used for setting clock speeds. Clock drift causes one packet to alternate between being ahead of and behind the other, even with two seemingly-identical-rate links feeding into a single queue.

- **Bus contention** causes variability in getting a packet from a network interface to a CPU, up to tens of packet timings.

- **Scheduler non-determinism** can affect packet generation times, and tends to be on the order of many milliseconds. For example, the Linux minimum scheduler interval is 7.5 ms and will be seen if no kernel events transpire.

To be conservative, I choose a target error of one packet time, and in the next section, test whether this goal can be met. In practice, we expect a test to fail catastrophically. That is, when the system is keeping up with the events occurring in an emulation run, the network invariant should hold, and when it fails, it should fall off a steep cliff; most queueing systems show this behavior.

2.2 Evidence That High Fidelity Is Possible

To demonstrate that this approach of monitoring timing invariants can work, we now show an example where monitoring the packet gap invariant differentiates between

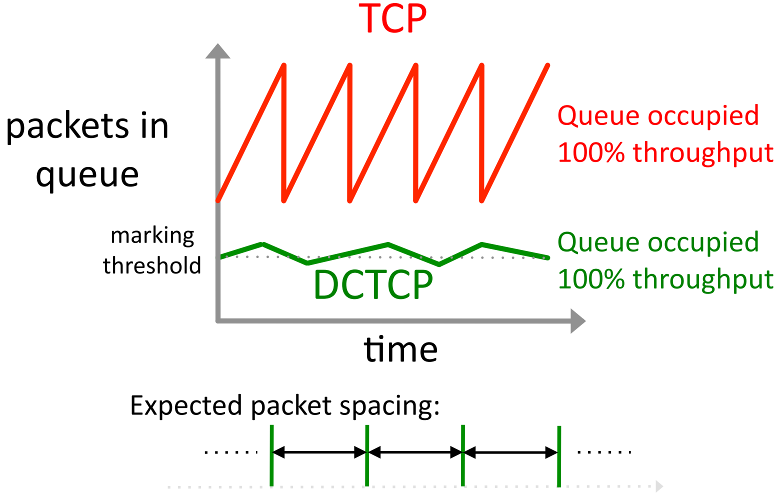


Figure 2.4: Simplified view of regular TCP vs Data Center TCP [9].

a high-fidelity and low-fidelity experiment. That is, as the experiment's demands increase, the fidelity indicator (packet gaps) changes accordingly. Completing the workflow shown in Figure 2.1, we will (1) log dequeue events, (2) parse the log to measure packet spacings, and (3) check if any packet is delayed by more than one packet time.

If this workflow is valid, then whenever step 3 says "pass", the results should match those from hardware. Our experiment uses Data Center TCP, or DCTCP [9], which is described fully later in § 5.1. This is an example of the kind of experiment one might actually run on a high-fidelity emulator, and since DCTCP depends on packet-level dynamics, it presents a good test of timing fidelity.

To understand DCTCP, it helps to start with regular TCP. As shown in Figure 2.4, a regular TCP flow on a link with enough buffering, i.e., $RTT * N$ for a single flow, will form a sawtooth in steady state. Since the queue never goes empty, the link achieves full link throughput.

DCTCP also tries to keep the queue occupied and achieve full throughput, but its goal is to do so with less buffering. DCTCP defines a marking threshold used by each switch; a packet is marked whenever it traverses a queue with an occupancy above this level. Each host then responds to the sequence of marked and unmarked packets

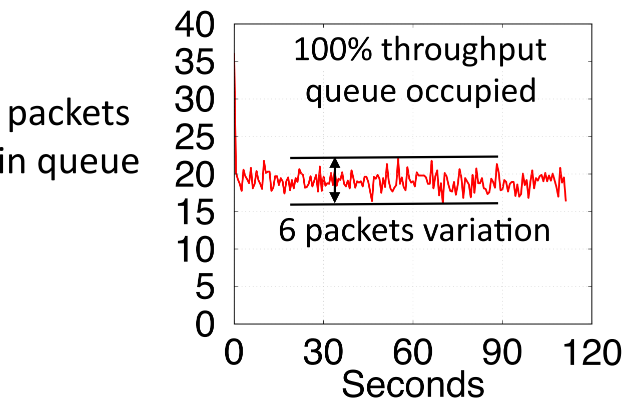


Figure 2.5: Hardware results with DCTCP at 100 Mb/s.

using a modified congestion control algorithm. Our packet spacing invariant should show nothing but constant gaps if the modified congestion control algorithm causes the queue to stay occupied.

Figure 2.5 shows the result from running DCTCP in hardware, using two 100 Mb/s Ethernet links to connect two servers with a switch in-between. The switch was configured with a marking threshold of 20 packets. The graph shows the number of packets in the switch queue over two minutes.

As expected, the queue stays occupied and the link reaches 100% throughput. The number of packets in the switch queue hovers around the 20-packet marking threshold and the experiment shows 6 packets of variation. The packet spacing invariant is held here, because the experiment is running on hardware. The next test is to run this same experiment on Mininet-HiFi.

Figure 2.6 shows the emulator results, where the server is intentionally restricted to a single 3 GHz core to better expose timing overlaps. Starting at 80 Mb/s, we see 100% throughput, with 6 packets of queue variation: the same result as hardware. The same variation occurs at 160 Mb/s. But moving to 320 Mb/s, the emulation clearly breaks down. The link speed varies, does not reach full throughput, and there is significantly more queue variation.

The question we ask is this: does checking invariants — in this case, packet spacing — identify this wrong result? This experiment has known-in-advance hardware results, but for many first-time emulator experiments, this will not be the case. To

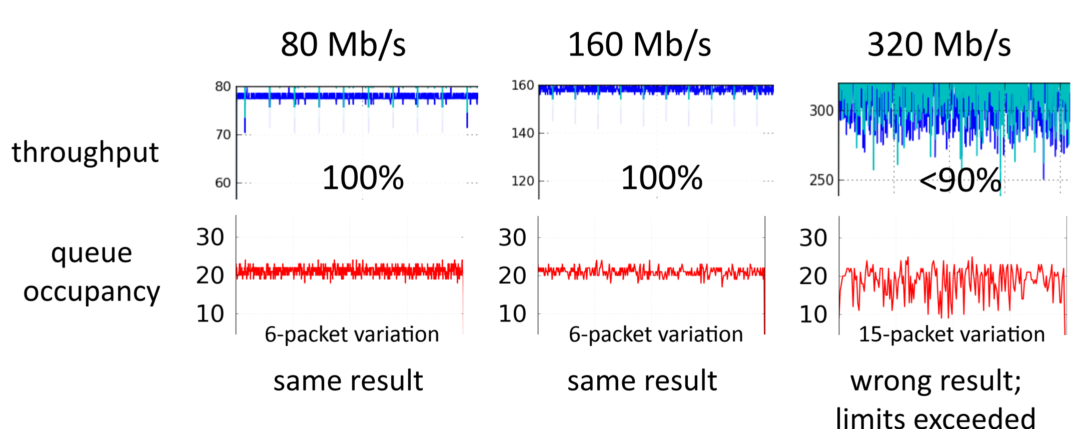


Figure 2.6: Emulator results with DCTCP at varying speeds.

enable *arbitrary* high-fidelity experiments, the invariant checks must reveal timing errors.

Checking Packet Gaps. Figure 2.7 shows the complementary cumulative distribution functions (CCDFs) of measured packet spacing values for emulator experiments run at a range of speeds. Both axes are logarithmic, and all numbers are in Megabits/s. The Y axis is percent of packets, and since this is a CCDF, it highlights the tail of the distribution of packet spacings. The X axis is the error. Below one packet are lines in green, for high fidelity; 1 - 25 is medium fidelity, where the result might be the same but we have evidence that the emulator was overloaded. The red lines correspond to low-fidelity emulation runs with more than 25 packets of timing error.[4]

We see full throughput for 10, 20, 40, and 80 Mb/s, with well under a packet of maximum error. The invariant check says “pass”. At 320 Megabits per second or higher, the emulator is way off, and the invariant check says “try again”, either with slower links or a bigger system.

160 Mb/s is the most interesting case here, shown in yellow. 90% of the time, the errors are below one packet, but some gaps are up to 10 packets in size. Our invariant check says “try again”, because there are measured instances where the

[4]The number 25 for packet times to define medium fidelity is arbitrary, and is meant to simplify the discussion.

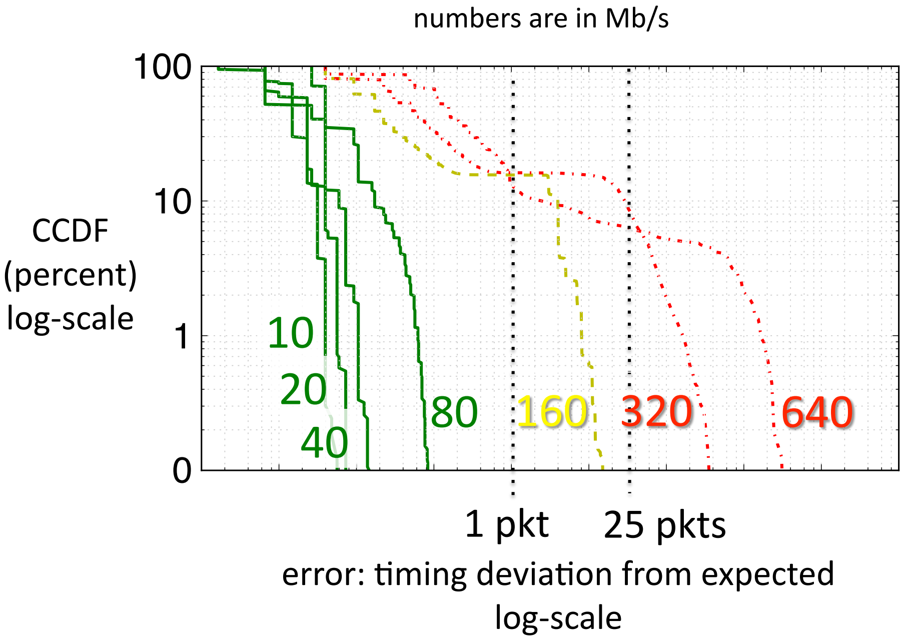


Figure 2.7: Packet Spacing Invariant with DCTCP. Going from 80 Mb/s to 160/s, the emulation falls behind and the 10% point on the CCDF increases by over 25x.

timing behavior is different.

So, what is the right thing to do in this case? The conservative approach seems prudent. One should probably not publish a paper on the result, because there are measured instances of lowered fidelity; even for valid emulation runs, one would want to publish the invariant checks along with the results, as evidence that the results are trustworthy.

This example shows the beauty of the invariants approach: it detects an emulation run that would otherwise appear correct if just checking that the output looked reasonable. DCTCP happens to be tolerant of a few packet delays, but there are other applications that would not tolerate these delays. Example include anything sensitive to latency, like a ping test, a key-value store, bandwidth testing that uses packet trains, or data center isolation mechanisms that operate at a fine time granularity [44].

Having demonstrated the potential of network invariant monitoring as a means to establish trust in results from an emulator, I now describe the architecture of the system used to produce these results, Mininet-HiFi.

Chapter 3

Original Mininet: A Container-Based Emulator

This chapter describes Original Mininet, an emulator that uses lightweight OS containers to run a complete network within a single OS instance. Original Mininet helps to verify the *functional* correctness of code used in network experiments [47]. Its components provide the basis for Mininet-HiFi, described in §4, which helps to verify *performance* properties for a much wider range of experiments [41].

In this section, I describe the specific problem that motivated the development of Original Mininet, walk through the system's operation, analyze its scalability, and then describe its use cases.

3.1 Motivation

In late 2009, OpenFlow was emerging as a tool for network researchers to change the behavior of a forwarding device, without needing access to a development toolkit or having to sign a non-disclosure agreement with a hardware vendor [63].[1] I now give a brief introduction to OpenFlow, along the broader paradigm into which it fits: Software-Defined Networking. Then I describe three motivating examples that encouraged Mininet development.

[1] Using OpenFlow is not required to use Original Mininet, but it did provide the original motivation.

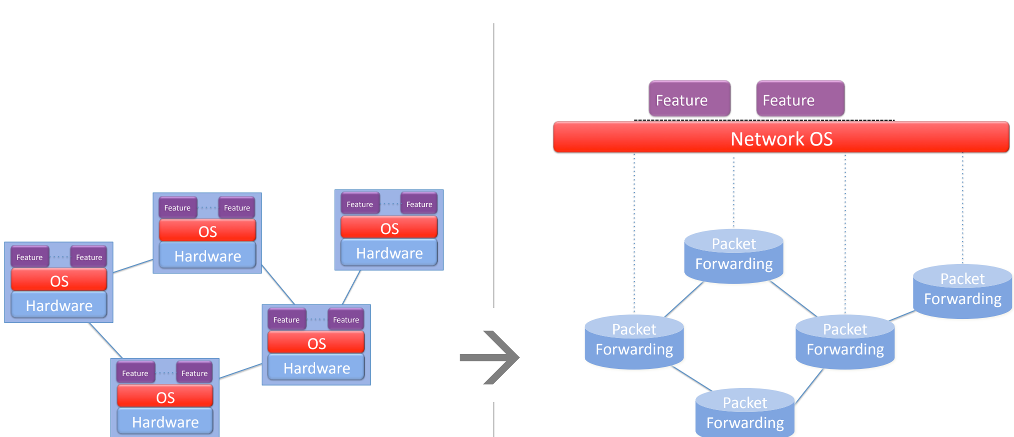


Figure 3.1: Traditional network on the left, with a Software-Defined Network (SDN) on the right. SDN physically separates the control plane from the forwarding devices in a network. Network features are implemented atop a centralized view of the network, which is provided by the Network OS. SDN was a primary motivator for developing and releasing Mininet.

Software-Defined Networking. In Software-Defined Networking (SDN), the control plane (or "network OS") is separated from the forwarding plane and the control plane interacts with switches through a narrow, vendor-agnostic interface, protocol such as OpenFlow [63]. Figure 3.1 shows the architecture. OpenFlow defines a simple abstraction for the low-level forwarding behavior of each forwarding element (switch, router, access point, or base station). For example, OpenFlow defines a rule for each flow; if a packet matches a rule, the corresponding actions are performed (e.g. drop, forward, modify, or enqueue).

The other side of an SDN, the network OS (e.g NOX [36], ONIX [46], or Beacon [12]), observes and controls OpenFlow switches from a central vantage point, hosting existing features such as routing protocols and access control. The network OS can also host more experimental features, such as network virtualization and energy management, which can leverage common functionality implemented inside the network OS, such as topology discovery and shortest-path routing algorithms.

Hassles with SDN development and testing. With SDN development, verifying controller functionality is typically a primary goal. Once a controller is known

to be functionally correct, it then has the potential to be deployed onto a testbed supporting OpenFlow, such as GENI [34] or FIRE [32]. However, in late 2009, both GENI and FIRE were in their infancy. At the time, one could build a custom testbed with an OpenFlow software switch [65], NetFPGA [48], or a hardware switch, and then connect the physical switch to physical servers, but this approach would not provide the ability to easily change the topology beyond that of a single switch. In addition, this small-number-of-switches testbed would require time-slicing to support multiple users.

An attractive alternative at the time was to use the network-of-virtual-machines approach. With a VM per switch/router, and a VM per host, realistic topologies can be stitched together using virtual interfaces. Both the NOXrepo VM scripts [56] and its Python counterpart, OpenFlow VMs [61], configure multiple virtual machines in a given topology, on a single server, by coupling QEMU instances on Linux with Virtual Distributed Ethernet (VDE) links [87]. However, even using reduced, 32MB-footprint Debian disk images, these approaches would take minutes to boot any network with a significant number of hosts or switches. In my experience, VMs were more convenient than hardware but still impractically slow for many uses. In particular, the memory overhead for each VM limited the scale to just a handful of switches and hosts.

The Ripcord project. The Ripcord data center network controller [19] was an effort to generalize the control of scale-out OpenFlow-based data centers, by decoupling the control logic from the topology. At the core of Ripcord is a structured topology object that describes a range of scale-out topologies and enumerates path choices to be used by routing algorithms. Previous papers had tightly coupled the topology to the control; for example, the paper on PortLand was evaluated only on a Fat Tree [55], while a paper on another scale-out data center, VL2, had its evaluation tied to a particular Clos topology [35]. In Ripcord, a PortLand-style routing module could run atop a VL2 topology, with no changes, and any applications, such as energy management or traffic engineering, would work on both.

Ripcord presented a worst case for the development environment:

Multiple concurrent developers. Project developers were physically distributed between two schools.

Frequent topology changes. Each application and control algorithm required testing on each topology, at multiple scales.

Large, highly connected topologies. As an example, the smallest three-layer Fat Tree topology requires 20 switches and 16 hosts [7].

A project like this would have no chance of completing on time if the development environment could not support concurrent development with large topologies and frequent changes.

OpenFlow Tutorials. A third motivator was OpenFlow tutorials. The first two were held at Hot Interconnects in August 2008 and SIGMETRICS in June 2009, and both used the NOXrepo scripts [56]. In these tutorials, attendees would learn basic OpenFlow operation, in a hands-on way, by turning a basic hub into a flow-accelerated learning switch. Each attendee was provided a virtual machine image. However, in practice, the VM scripts were slow to load, because of double-virtualization; not only would the primary Linux VM need to be virtualized, generally through software, but QEMU would run a second level of virtualization to run the minimized Linux instances that each ran a single host or OpenFlow switch. This double-virtualization issue plagued the tutorials, because any issue that required re-starting the network topology would take too long to resolve.

3.2 Architecture Overview

An initial Mininet script, written in Python in late 2009 by Bob Lantz, demonstrated the use of lightweight Linux containers for OpenFlow network emulation. This script grew into a more flexible, documented, and open-source system called Mininet, driven by the motivating examples of general SDN development pain, distributed development of data centers with Ripcord, and running OpenFlow tutorials. This section describes the architecture of Mininet, which largely derives from the initial prototype.

Original Mininet [47] use lightweight, OS-level virtualization to emulate hosts, switches, and network links. Mininet employs the Linux *container* mechanism, which enables groups of processes to have independent views of system resources such as

process IDs, user names, file systems and network interfaces, while still running on the same kernel. Containers trade the ability to run multiple OS kernels for lower overhead and better scalability than full-system virtualization.[2]

Figure 3.2 illustrates the components and connections in a two-host network created with Mininet. For each virtual host, Mininet creates a container attached to a *network namespace*. Each network namespace holds a virtual network interface, along with its associated data, including ARP caches and routing tables. Virtual interfaces connect to software switches (e.g Open vSwitch [65]) via virtual Ethernet (`veth`) links. The design resembles a server hosting full virtualized systems (e.g., Xen [11] or VMware [88]), where each VM has been replaced by processes in a container attached to a network namespace.

Since each network namespace has a distinct set of kernel structures for networking, two processes in different network namespaces can do things that would be impossible without network namespace isolation, such as hosting web servers on port 80 or using the same assigned IP address. Having described the architectural pieces, we now walk through a typical Mininet workflow.

3.3 Workflow

By combining lightweight virtualization with an extensible CLI and API, Mininet provides a rapid prototyping workflow to create, interact with, customize and share a Software-Defined Network, as well as a smooth path to running on real hardware.

3.3.1 Creating a Network

The first step is to launch a network using the `mn` command-line tool. For example, the command

```
mn --switch ovsk --controller nox --topo \
   tree,depth=2,fanout=8 --test pingAll
```

[2]This lightweight virtualization approach is similar to other network emulators, such as Imunes [91] and vEmulab [42], and the Linux *containers* Mininet employs are effectively equivalent to *jails* in BSD and *zones* in Solaris.

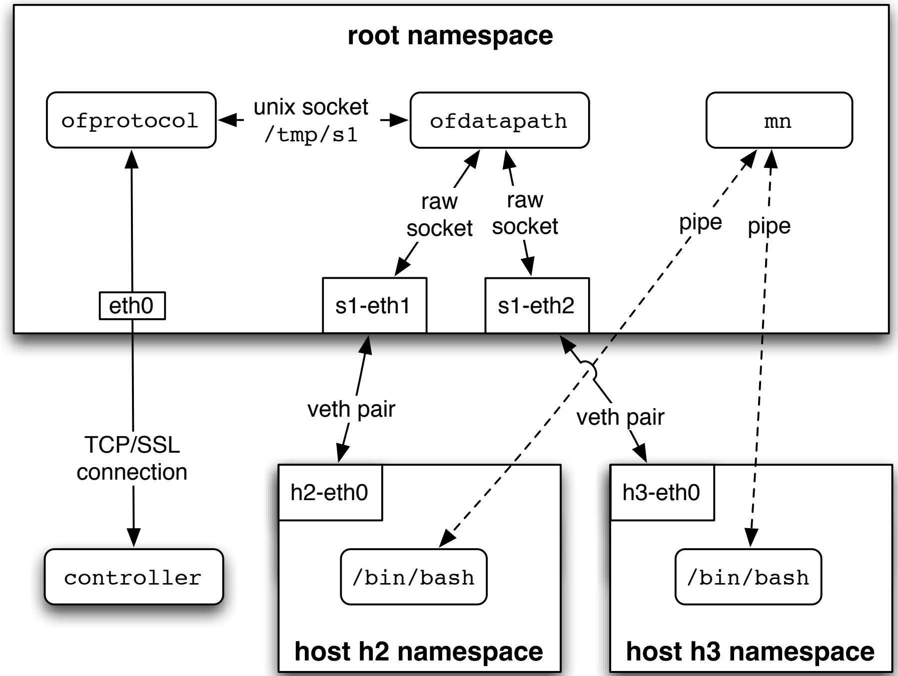


Figure 3.2: Mininet creates a virtual network by placing host processes in network namespaces and connecting them with virtual Ethernet (veth) pairs. In this example, they connect to a user-space OpenFlow switch.

starts a network of OpenFlow switches. In this example, Open vSwitch [65] kernel switches will be connected in a tree topology of depth 2 and fanout 8 (i.e. 9 switches and 64 hosts), under the control of NOX, an OpenFlow controller [36]. After creating the network, Mininet will run a `pingAll` test to check connectivity between every pair of nodes. To create this network, Mininet must emulate links, hosts, switches, and controllers:

Links: A virtual Ethernet pair, or veth pair, acts like a wire connecting two virtual interfaces; packets sent through one interface are delivered to the other, and each interface appears as a fully functional Ethernet port to all system and application software. Veth pairs may be attached to virtual switches such as the Linux bridge or a software OpenFlow switch.

Hosts: Network namespaces [49] are containers for network state. They provide processes (and groups of processes) with exclusive ownership of interfaces, ports, and

Figure 3.3: The `console.py` application uses Mininet's API to interact with and monitor multiple hosts, switches and controllers. The text shows `iperf` running on each of 16 hosts.

routing tables (such as ARP and IP). For example, two web servers in two network namespaces can coexist on one system, both listening to private `eth0` interfaces on port 80.

A host in Mininet is simply a shell process (e.g. `bash`) moved into its own network namespace with the `unshare(CLONE_NEWNET)` system call. Each host has its own virtual Ethernet interface(s) (created and installed with `ip link add/set`) and a pipe to a parent Mininet process, `mn`, which sends commands and monitors output.

Switches: Software switches, such as Open vSwitch [65], provide the same packet delivery semantics as a hardware switch. Both user-space and kernel-space switches are available, acting as basic Ethernet bridges, remotely-controlled OpenFlow switches, or even managed switches running distributed protocols like Spanning Tree Protocol.

Controllers: OpenFlow controllers can be anywhere on the real or simulated network, as long as the machine on which the switches are running has IP-level connectivity to the controller. For Mininet running in a VM, the controller could run inside the VM, natively on the host machine, or in the cloud.

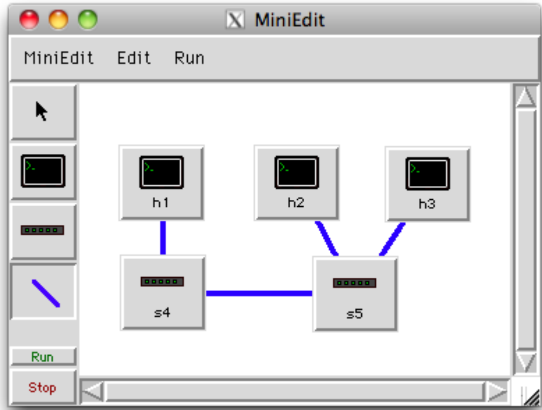


Figure 3.4: MiniEdit is a simple graphical network editor that uses Mininet to turn a graph into a live network when the Run button is pressed; clicking a node opens up a terminal window for that node.

3.3.2 Interacting with a Network

After launching the network, we want to interact with it: to run commands on hosts, verify switch operation, and maybe induce failures or adjust link connectivity. Mininet includes a network-aware command line interface (CLI) to allow developers to control and manage an entire network from a single console. Since the CLI is aware of node names and network configuration, it can automatically substitute host IP addresses for host names. For example, the CLI command

```
mininet> h2 ping h3
```

tells host `h2` to ping host `h3`'s IP address. This command is piped to the bash process emulating host 2, causing an ICMP echo request to leave `h2`'s private `eth0` network interface and enter the kernel through a veth pair. The request is processed by a switch in the root namespace, then exits back out a different veth pair to the other host. If the packet needed to traverse multiple switches, it would stay in the kernel without additional copies; in the case of a user-space switch, the packet would incur user-space transitions on each hop. In addition to acting as a terminal multiplexer for hosts, the CLI provides a variety of built-in commands and can also evaluate Python expressions.

3.3.3 Customizing a Network

Mininet exports a Python API to create custom experiments, topologies, and node types: switch, controller, host, or other. A few lines of Python are sufficient to define a custom regression test that creates a network, executes commands on multiple nodes, and displays the results. An example script:

```
from mininet.net import Mininet
from mininet.topolib import TreeTopo
tree4 = TreeTopo(depth=2,fanout=2)
net = Mininet(topo=tree4)
net.start()
h1, h4  = net.hosts[0], net.hosts[3]
print h1.cmd('ping -c1 %s' % h4.IP())
net.stop()
```

creates a small network (4 hosts, 3 switches) and pings one host from another, in about 4 seconds.

The current Mininet distribution includes several example applications, including text-based scripts and graphical applications, two of which are shown in Figures 3.3 and 3.4. The hope is that the Mininet API will prove useful for system-level testing and experimentation, test network management, instructional materials, and applications that will surprise its authors.

3.3.4 Sharing a Network

Mininet is distributed as a VM with all dependencies pre-installed, runnable on common virtual machine monitors such as VMware, Xen and VirtualBox. The virtual machine provides a convenient container for distribution; once a prototype has been developed, the VM image may be distributed to others to run, examine and modify. A complete, compressed Mininet VM is about 800 MB. Mininet can also be installed natively on Linux distributions that ship with `CONFIG_NET_NS` enabled, such as Ubuntu 10.04, without replacing the kernel.

3.3.5 Running on Hardware

The final step in a typical Mininet workflow is porting the experiment to a custom or shared testbed, running physical switches, links, and servers. If Mininet has been correctly implemented, this step should require no additional work on the part of the experimenter. The OpenFlow controller should see a network whose only differences are minor, like different numbers of links or MAC addresses, and hence, should require no code changes. To justify this assertion, a week before a SIGCOMM deadline, the Ripcord controller code developed on Mininet was "ported" to a custom hardware testbed, and this controller was able to provide full network connectivity, with no code changes.

3.4 Scalability

Lightweight virtualization is the key to scaling to hundreds of nodes while preserving interactive performance. This section measures overall topology creation times, times to run individual operations, and available bandwidth.

3.4.1 Topology Creation

Table 3.1 shows the time required to create a variety of topologies with Mininet. Larger topologies which cannot fit in memory with full-system virtualization can start up on Mininet. In practice, waiting 10 seconds for a full fat tree to start is quite reasonable (and faster than the boot time for hardware switches).

Mininet scales to the large topologies shown (over 1000 hosts) because it virtualizes less and shares more. The file system, user ID space, process ID space, kernel, device drivers, shared libraries and other common code are shared between processes and managed by the operating system. The roughly 1 MB overhead for a host is the memory cost of a shell process and plus the network namespace state; this total is almost two orders of magnitude less than the 70 MB required per host for the memory image and translation state of a lean VM. In fact, of the topologies shown in Table 3.1, only the smallest one would fit in the memory of a typical laptop if full-system

Topology	H	S	Setup(s)	Stop(s)	Mem(MB)
Minimal	2	1	1.0	0.5	6
Linear(100)	100	100	70.7	70.0	112
VL2(4, 4)	80	10	31.7	14.9	73
FatTree(4)	16	20	17.2	22.3	66
FatTree(6)	54	45	54.3	56.3	102
Mesh(10, 10)	40	100	82.3	92.9	152
Tree(4ˆ4)	256	85	168.4	83.9	233
Tree(16ˆ2)	256	17	139.8	39.3	212
Tree(32ˆ2)	1024	33	817.8	163.6	492

Table 3.1: Mininet topology benchmarks: setup time, stop time and memory usage for networks of H hosts and S Open vSwitch kernel switches, tested in a Debian 5/Linux 2.6.33.1 VM on VMware Fusion 3.0 on a MacBook Pro (2.4 GHz intel Core 2 Duo/6 GB). Even in the largest configurations, hosts and switches start up in less than one second each.

virtualization were used.

3.4.2 Individual Operations

Table 3.2 shows the time consumed by individual operations when building a topology. Surprisingly, link addition and deletion are expensive operations, taking roughly 250 ms and 400 ms, respectively. These operations likely have room for optimizations.

3.4.3 Available Bandwidth

Mininet also provides a usable amount of bandwidth, as shown in Table 3.3: 2-3 Gbps through one switch, or more than 10 Gbps aggregate internal bandwidth through a chain of 100 switches. For this chain test, the aggregate bandwidth starts to drop off around 40 switches.

3.5 Use Cases

As of July 2010, Mininet had been used by over 100 researchers in more than 18 institutions, including Princeton, Berkeley, Purdue, ICSI, UMass, University of Alabama

Operation	Time (ms)
Create a node (host/switch/controller)	10
Run command on a host ('echo hello')	0.3
Add link between two nodes	260
Delete link between two nodes	416
Start user space switch (OpenFlow reference)	29
Stop user space switch (OpenFlow reference)	290
Start kernel switch (Open vSwitch)	332
Stop kernel switch (Open vSwitch)	540

Table 3.2: Time for basic Mininet operations. Mininet's startup and shutdown performance is dominated by management of virtual Ethernet interfaces in the Linux (2.6.33.1) kernel and `ip link` utility and Open vSwitch startup/shutdown time.

S (Switches)	User(Mbps)	Kernel(Mbps)
1	445	2120
10	49.9	940
20	25.7	573
40	12.6	315
60	6.2	267
80	4.15	217
100	2.96	167

Table 3.3: Mininet end-to-end bandwidth, measured with `iperf` through linear chains of user-space (OpenFlow reference) and kernel (Open vSwitch) switches.

Huntsville, NEC, NASA, Deutsche Telekom Labs, Stanford, and a startup company, as well as seven universities in Brazil. The set of use cases has certainly expanded since then, but this initial set covers most of the common use cases.

The first set of use cases roughly divided into prototyping, optimization, demos, tutorials, and regression suites; §3.1 already described the tutorial and prototyping cases. For each additional use, I describe a project, a challenge it faced, and how Mininet helped.

Degugging: The OpenFlow controller NOX builds a topology database by sending periodic LLDP packet broadcasts out each switch port [36]. A production network was brought down by an excessive amount of these topology discovery messages, experiencing 100% switch CPU utilization. Reproducing the bug proved hard in the production network because of topology and traffic changes. Mininet provides a way

to try many topologies to reproduce the error, experiment with new topology discovery algorithms, and validate a fix.

Demos: Several users have created live interactive demonstrations of their research to show at overseas conferences. While connecting to real hardware is preferred, high latency, flaky network access, or in-flux demo hardware can derail a live demo. Maintaining a version of the demo inside Mininet provides insurance against such issues, in the form of a local, no-network-needed backup.

Regression Suites: Mininet can help to create push-button regression suites to test new network architectures. One example is SCAFFOLD [76], a service-centric network architecture that binds communication to logical object names (vs. addresses), provides anycast between object group instances, and combines routing and resolution in the network layer. Without Mininet, its creators would have to either employ a simulator, and build a model for user requests and their systems's responses, or build a physical network.

Distributed Controllers: The in-packet Bloom Filter architecture uses a distributed network of rack controllers to provide routes, directory services, topology info, and split identity location [74]. This project was unable to build sufficiently large topologies with full-system virtualization on a limited number of machines, so they migrated to Mininet to support topologies with more nodes.

Network Monitoring: Monitoring the live performance of an network requires getting metrics from real applications, from as many vantage points as possible. Unfortunately, Linux does not support multiple Ethernet interfaces on the same subnet, so to get multiple vantage points, one typically uses full-system virtualization with multiple Ethernet jacks or uses distinct machines. After learning that Mininet could spawn multiple virtual hosts on a single physical server, our local network administrator leveraged the lightweight host virtualization in the Mininet API to get 8 vantage points from a cheap, low-power, low-memory embedded PC – one with insufficient memory to run fully virtualized hosts. This example is not a direct use of Mininet for network emulation, but it does show another way that network namespace virtualization can prove useful in managing networks.

3.6 Limitations

While Original Mininet can handle a number of distinct use cases, as described in the last section, the system does have limitations that prevent it from applying to a number of other use cases. In particular, the most significant limitation of Original Mininet is a lack of performance fidelity, especially at high loads. CPU resources are multiplexed in time by the default Linux scheduler, which provides no guarantee that a virtual host that is ready to send a packet will be scheduled promptly, or that all virtual switches will forward at the same rate. Original Mininet has no way to provision a link to run at a given rate, like 1 Gb/s. In addition, software forwarding may not match the observed latency of hardware. As mentioned in §1.5, $O(n)$ linear lookup for software tables cannot approach the $O(1)$ lookup of a hardware-accelerated TCAM in a vendor switch, causing the packet forwarding rate to drop for large wildcard table sizes.

In addition, Mininet's partial virtualization approach also limits what it can do. It cannot handle different OS kernels simultaneously. All hosts share the same filesystem, although this can be changed by using `chroot`. Hosts cannot be migrated live like VMs. These losses are reasonable tradeoffs for the ability to try ideas at greater scale.

In the next section, I describe additions to Original Mininet to address the first and most pressing issue, the lack of performance fidelity.

Chapter 4

Mininet-HiFi: A High-Fidelity Emulator

This chapter describes Mininet-HiFi, a Container-Based Emulator with sufficient accuracy to support experiments that demand performance fidelity, including transport protocol, queueing, and topology experiments, among others. These types of experiments, described briefly in §4.1, would have been impossible with Original Mininet. After describing the architectural pieces of Mininet-HiFi to isolate resource and monitor performance in §4.2, I present a range of validation tests in §4.3 to ensure that the system operates as intended. Then I present micro-level benchmarks to help with choices between process and packet scheduler options in §4.4. Section 4.5 concludes this chapter by describing the scope of experiments that fit well on Mininet-HiFi.

4.1 Motivation

Original Mininet lacks performance fidelity; fixing this limitation enables experiments where the goal is to *measure performance properties*, not just ensure functional correctness. The following experiment types are a poor fit for Original Mininet, as each requires additional emulation features to trust the results. For all of these, the ability to real Linux code with trustworthy performance numbers, on a single machine, would be useful.

Transport Protocols. Transport protocols like TCP exist to maximize transmission speeds, fairness, or delay, and their experiments are unlikely to provide valid results when links speeds and delays cannot be emulated.

Queueing. Queueing methods might try to provide per-flow or per-host fairness on a link. These experiments would be more likely to provide valid results with accurately emulated queues, as well as limits to ensure that an aggressively-sending virtual host cannot delay or prevent other virtual hosts from generating traffic.

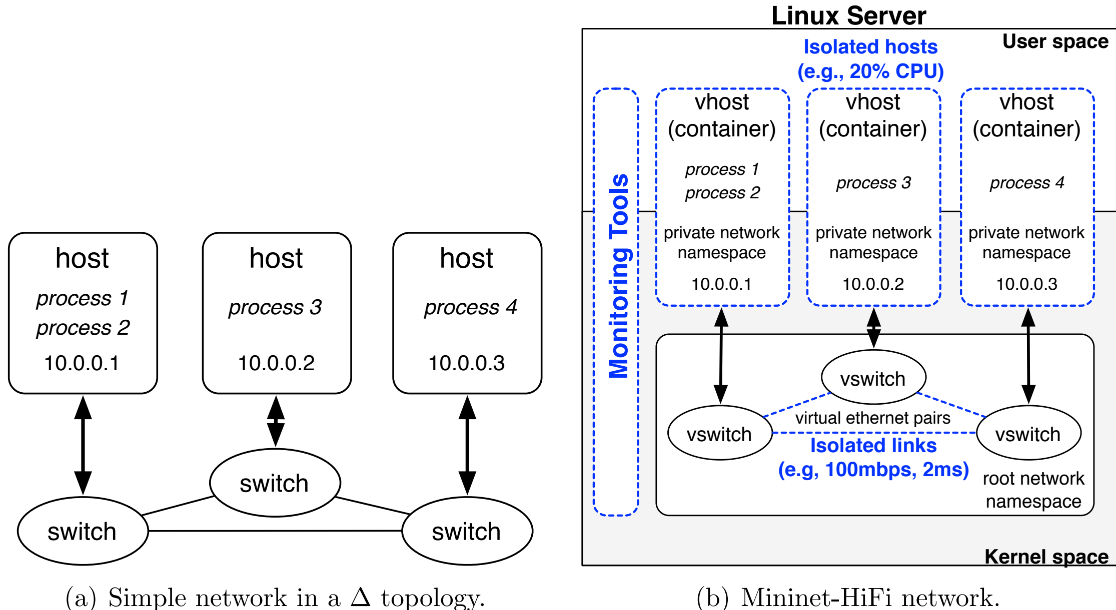
Topology Experiments. Data-center topologies like Fat Trees [7] and random topologies [77] benefit from new routing strategies [6, 70]. These experiments require large numbers of hosts, queues, and switches, which all must be accurately emulated. In addition, since the forwarding delays through virtual switches cannot be bounded, they should at least be tracked.

Ideally, even combinations of these methods would be supported, such as a new transport protocol used within a new datacenter fabric with switches that implement a new queueing strategy.

4.2 Design Overview

Mininet-HiFi extends the original Mininet architecture with mechanisms for performance isolation, resource provisioning, and performance fidelity monitoring.

The individual components of Mininet-HiFi tend not to be new, or even designed with network in emulation in mind, but the combination of these pieces leads to a new emulation workflow. This approach of using off-the-shelf components is a practical choice to maximize installability, usability, and maintainability; in particular, this choice requires no changes with new kernel revisions.



(a) Simple network in a Δ topology. (b) Mininet-HiFi network.

Figure 4.1: A Mininet-HiFi network emulates performance constraints of links and hosts. Dashed blue lines and text indicate added performance isolation and monitoring features that distinguish Mininet-HiFi from Original Mininet.

4.2.1 Performance Isolation

Original Mininet provides no assurance of performance fidelity because it does not isolate the resources used by virtual hosts and switches. One option for running network experiments, vEmulab, provides a way to limit link bandwidth, but not CPU bandwidth. As we saw earlier in Figure 1.1, link bandwidth limiting alone is insufficient: a realistic emulator requires both CPU and network bandwidth limiting at minimum. Mininet-HiFi implements these limits using the following OS-level features in Linux:

Control Groups or *cgroups* allow a group of processes (belonging to a container/virtual host) to be treated as a single entity for (hierarchical) scheduling and resource management [21].[1] To use cgroups to control process execution, rather than just monitor it, a mechanism like the one described next is required.

[1]Resources optionally include CPU, memory, and I/O. CPU caches and Translation Lookaside Buffers (TLBs) cannot currently be managed.

CPU Bandwidth Limits enforce a maximum time quota for a `cgroup` within a given period of time [83]. The period is configurable and typically is between 10 and 100 ms. CPU time is fairly shared among all `cgroup`s which have not used up their quota (slice) for the given period. This feature was originally implemented by Google to reduce completion-time variability, by preventing a single process from monopolizing a CPU, whether due to a bug, normal variation, or an unusually expensive query. It has similar goals to the Linux Real-Time (RT) scheduler, which also limits process-time execution, but differs in that when no limits are set, it acts in a work-conserving mode identical to the default Linux Completely Fair Scheduler (CFS).

Linux Queueing Disciplines enforce link properties such as bandwidth, delay, and packet loss. Packet schedulers such as Hierarchical Token Bucket (HTB) [26] and Hierarchical Fair Service Curve (HFSC) [79] can be configured using `tc` to apply these properties to each link.

Figure 4.1(a) shows the components of a simple hardware network, and Figure 4.1(b) shows its corresponding realization in Mininet-HiFi using the above features. Adopting these mechanisms in Mininet-HiFi solves the performance isolation problem shown in Figure 1.1: the “ideal” line is in fact the measured behavior of Mininet-HiFi.

4.2.2 Resource Provisioning

Each isolation mechanism must be configured appropriately for the system and the experiment. The challenge with advance provisioning is that the exact CPU usage for packet forwarding will vary with path lengths, lookup complexities, and link loads. It is impractical to predict in advance whether a particular configuration will provide enough cycles for forwarding. Moreover, it may be desirable to overbook the CPU to support a larger experiment if some links are partially loaded. Mininet-HiFi lets the experimenter allocate link speeds, topologies, and CPU fractions based on their estimated demand. More importantly, as described next, Mininet-HiFi can also monitor performance fidelity to help verify that an experiment is operating realistically.

4.2.3 Fidelity Monitoring

Any Container-Based Emulator must contend with infidelity that arises when multiple processes execute serially on the available cores, rather than in parallel on physical hosts and switches. Unlike a simulator running in virtual time, Mininet-HiFi runs in real time and does not pause a host's clock to wait for events. As a result, events such as transmitting a packet or finishing a computation can be delayed due to serialization, contention, and background system load, as described in depth in §2.

Mininet-HiFi uses hooks in the Linux Tracing Toolkit [2] to log process and packet scheduler events to memory. Each event is written to a log as the experiment runs and processed afterwards with a Python script to build and check the timing distributions.

4.3 Validation Tests

It is not immediately obvious that the combination of mechanisms described in §4.2 will in fact yield a network whose behavior matches hardware switches, even on the simplest of topologies. This section presents results from validation tests that check for basic performance correctness, on topologies small enough to set up an equivalent hardware testbed. We run tests on simple networks with multiple TCP flows, then compare the bandwidths achieved to those measured on a testbed with an equivalent topology. For each test, we know roughly what to expect – multiple flows sharing a bottleneck should get an equal share – but we do not know the expected variation between flows, or whether TCP will yield equivalent short-term dynamics.

The validation tests confirm that the combination of elements in Mininet-HiFi works properly. Mininet-HiFi yields similar TCP results, but with greater repeatability and consistency between hosts when compared to hardware. However, for one simple test topology, a dumbbell with UDP traffic, it produces the wrong result, and we describe one solution to this problem in §4.3.3.

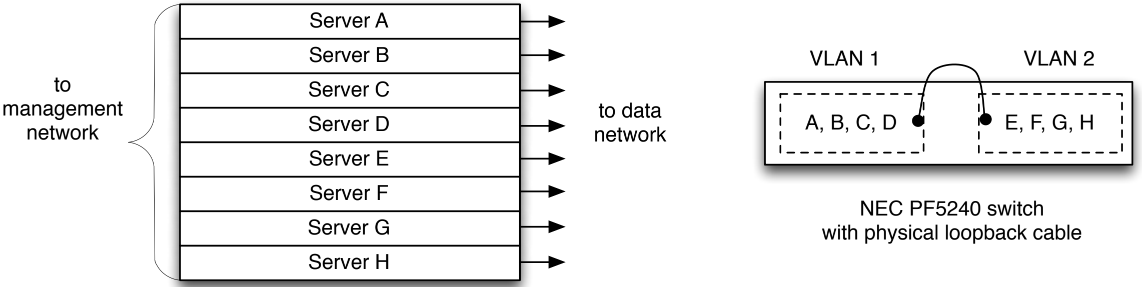
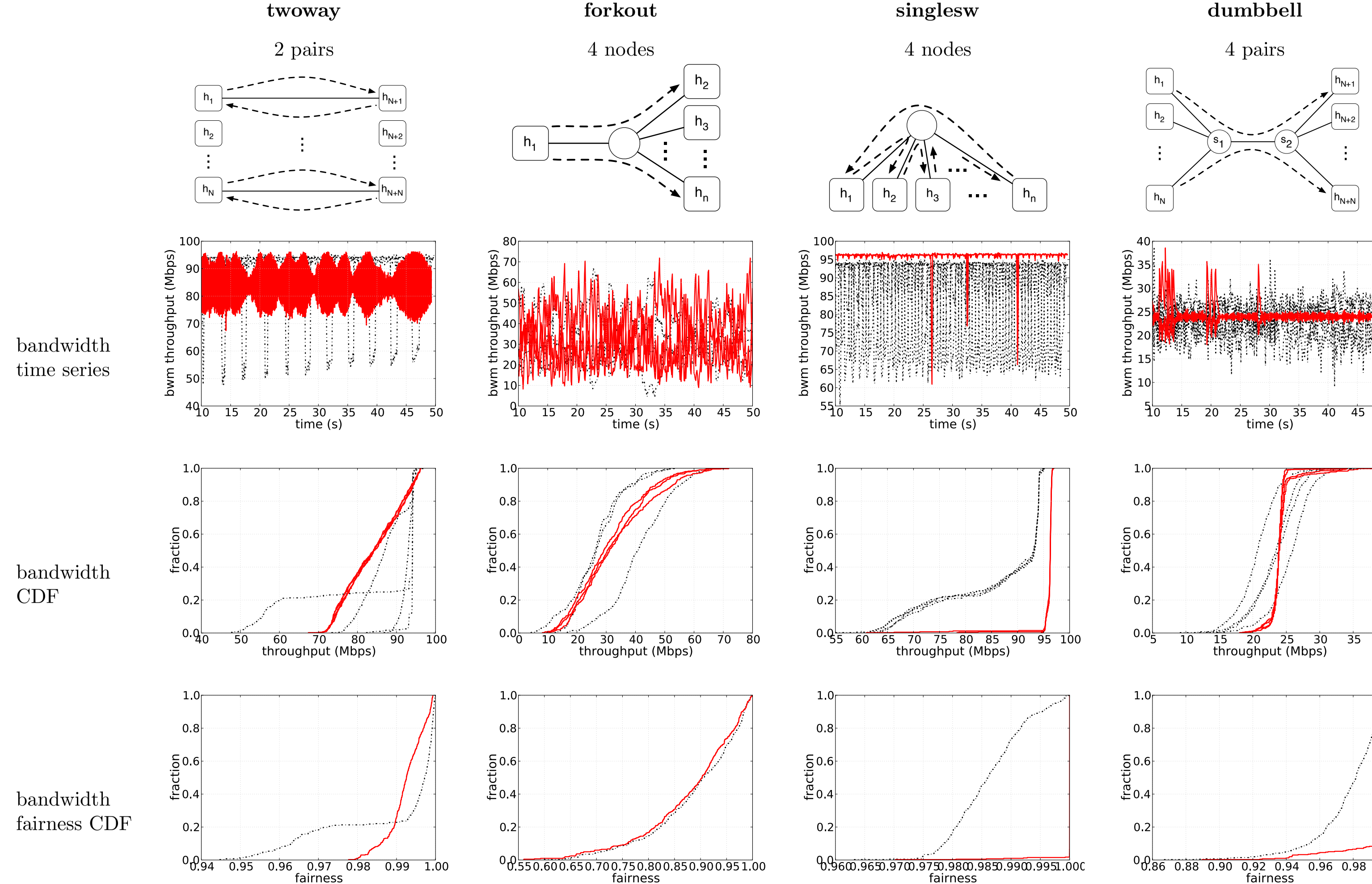


Figure 4.2: The testbed includes eight machines connected in a dumbbell topology using a single physical switch.

4.3.1 Testbed

Our testbed includes eight identical dual-core Core2 1.86 GHz, 2 GB RAM Linux servers. These eight machines are connected as a dumbbell with four hosts on each end through an NEC PF5240 switch, as shown in Figure 4.2. The switch is split into two VLANs and connected with a physical loopback cable. Each server has two motherboard ports; one provides remote management access and the other provides access to the isolated test network (the dumbbell). When showing Mininet-HiFi results in this section, we use a single dual-core PC from within the testbed. For both hardware and software, all links are set to 100 Mbps full-duplex.

Table 4.1: **Validation Tests**.
Each graph shows measured interface counters for all flows in that test, comparing data from Mininet-HiFi running on a dual-core Core2 1.86GHz, 2GB machine against a hardware testbed with eight machines of identical specifications. In many cases the individual CDF lines for Mininet-HiFi results overlap.
Legend: Mininet-HiFi: solid red lines. **Testbed: dashed black lines.**



4.3.2 Tests

Table 4.1 presents our validation results. Each column corresponds to an experiment with a unique topology and configuration of flows. Each row corresponds to a different way to present the same bandwidth data, as a time series, CDF, or CDF of fairness. To measure fairness between flows we use Jain's Fairness Index, where 1/n is the worst possible and 1 represents identical bandwidths. Mininet-HiFi results are highlighted in solid red lines, while testbed results are dashed black lines. In the time series and bandwidth plots, each TCP Reno flow has its own line.

Two-Way Test. The goal of this test is to verify whether the links are "truly duplex" and whether the rate-limiters for traffic flowing in either direction are isolated. The network consists of two two-host pairs, each sending to the other. One hardware port appears to experience significant periodic dips. Mininet-HiFi does not show this dip, but it does have more variation over time and the bandwidths for each host line up more closely.

Fork-Out Test. The goal of this test is to check if multiple flows originating from the same virtual host get a fair share of the outgoing link bandwidth, or an uneven share, if they are unequally affected by the CPU scheduler. The test topology consists of four hosts connected by a single switch, with one sender and three receivers. The bandwidths received by the three Mininet-HiFi hosts are squarely within the envelope of the hardware results, and the fairness CDF is nearly identical.

Single-Switch Test. The goal of this test is to see what happens with potentially more-complex ACK interactions. The single-switch test topology consists of four hosts, each sending to their neighbor, with the same total bandwidth requirement as the Two-Way test. As with the Fork-Out test, the Mininet-HiFi results here show less variation than with the hardware testbed.

Dumbbell Test. The goal of this test is to verify if multiple flows going over a bottleneck link get their fair share of the bandwidth. The dumbbell test topology consists of two switches connected by a link, with four hosts at each end. Four hosts each send to the opposite side. We see throughput variation in the TCP flows, due

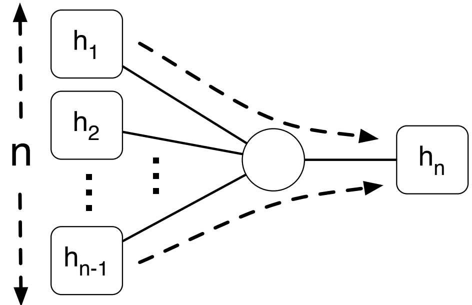


Figure 4.3: The Fork-In Test, where multiple UDP senders send to one receiver, originally led to one sender receiving all the bandwidth.

to ACKs of a flow in one direction contending for the queue with the reverse flow, increasing ACK delivery delay variability. The median throughput is identical between Mininet-HiFi and hardware, but again the Mininet results show less variability over time and between hosts.

Discussion. Note that our hardware is unlikely to exactly match that from other testbeds, and in fact, we may not be able to measure our own testbed with enough fidelity to accurately re-create it in Mininet-HiFi. For example, the amount of buffers in the switch and its internal organization are unknown. Also, without a hardware NIC across a bus, we expect much less variability from the NIC and a noticeably lower RTT between hosts on the same switch. Hence, one would not expect the results to match exactly.

In general, Mininet-HiFi shows less variation between hosts with TCP flows than the testbed. For example, the repeating pattern seen in the Two-Way test occasionally appeared on hardware, but never on Mininet-HiFi. However, as we see next, putting the resource-isolation pieces together in Mininet-HiFi does not ensure behavior that approximates hardware for UDP flows.

4.3.3 UDP Synchronization Problem + Fix

The Fork-In test sends UDP traffic from multiple hosts to one receiver, as shown in Figure 4.3. This test revealed that bandwidth was not getting equally shared among the flows. One flow was receiving all the bandwidth, while the others starved.

Repeating the test with two pairs of hosts produced the same result; one flow received all the bandwidth.

The reason for this behavior is that all the emulated nodes in the system operate under a single clock domain. This perfect synchronization leads the UDP streams to become perfectly interleaved at the bottleneck interface. When there was room for a packet in the bottleneck queue, a packet from the first flow would always take the slot, and the next packet from each other flow would be dropped. While not technically a bug in the queue implementation, this behavior does prevent us from getting results that we would expect to obtain on real networks. In real networks, hosts, switches, and switch interfaces all operate on different clock domains. This leads to an inherent asynchrony, preventing pathological scenarios like the one observed. The continuing drift of one interface relative to the other, due to inaccuracies in crystal manufacturing and temperature differences yields fairness over time.

This single clock domain synchronization problem is not unique to our emulator; it also appears in simulators, where the typical solution is to randomly jitter the arrival of each packet event so that events arriving at the same virtual time get processed in a random order. Since our emulator only knows real time, we use the following algorithm to emulate the asynchrony of real systems and jittered simulators in Mininet-HiFi:

- On each packet arrival, if the queue is full, temporarily store it in an overflow buffer (instead of dropping it, as a droptail queue would).

- On each dequeue event, fill the empty queue slot with a packet randomly picked from the overflow buffer (instead of the first packet), and drop the rest.

After implementing the jitter-emulation solution, we observed the flows achieving fairness close to that measured on real hardware, similar to the TCP case. We saw similar synchronization with a Fork-In test with two hosts sending to one host, except instead of complete starvation, the bandwidth was arbitrarily split between the flows, with a split ratio that remains consistent for the duration of the experiment. The modified link scheduler code fixed this bug, too.

This tweak highlights a way in which a CBE is like a simulator and may yield behavior different than hardware, even if it perfectly hits timing goals for network invariants.

4.4 Microbenchmarks

The High Fidelity Emulation approach relies on the accuracy and correctness of the CPU scheduler and the link scheduler. This section focuses on microbenchmarks to verify the behavior of these components (and options for them), as well as explore their limits, when pushed. Each microbenchmark was run on an Intel i7 CPU with 4 cores running at 3.20GHz with 12GB of RAM.

4.4.1 CPU Scheduling

CPU schedulers differ in the precision by which they provide each virtual host with a configured fraction of the CPU. For each virtual host, we measure how precisely it is scheduled over time (i.e., the CPU utilization during each second for each process). We also measure delays caused by serialization in scheduling (i.e., the time from when a process should be scheduled, until the time it is, as well as the time from when a packet should depart, until the time it does).

Time-Allocation Accuracy. Table 4.2 compares the CPU allocation we *want* a vhost to receive against the *actual* CPU time given to it, measured once per second. The vhosts are simple time-waster processes, and the total CPU utilization is 50%.

Consider the first row: the *bwc* (CFS Bandwidth Limiting) scheduler allocates each of the 50 vhosts 4% of a CPU core, and since there are four cores, the total CPU utilization is 50%. Each second we measure the amount of time each process was *actually* scheduled and compare it with the target. The maximum deviation was 0.41% of a CPU, i.e., in the worst-case second, one process received 3.59% instead of 4%. The RMS error was 2% of the target, i.e., 0.008% of a CPU core within a second. The results are similar for 100 vhosts and for the *rt* scheduler. In other words, the *bwc* and *rt* schedulers both give the desired mean allocation, during each second, within a

Sched	vhosts	goal cpu%	mean cpu%	maxerr cpu%	rmserr %
bwc	50	4.00	4.00	0.41	2.1
	100	2.00	2.00	0.40	2.9
rt	50	4.00	4.00	0.39	1.9
	100	2.00	2.00	0.40	3.2
default	50	4.00	7.98	4.44	100.0
	100	2.00	3.99	2.39	99.6

Table 4.2: Comparison of schedulers for different numbers of *vhosts*. The target is 50% total utilization of a four core CPU.

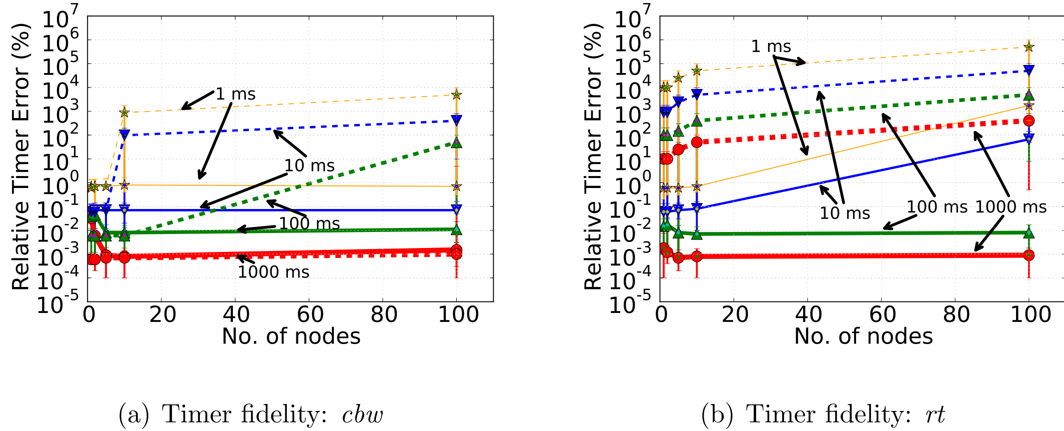
small margin of error. As expected, in both cases, the *default* scheduler gives a very different allocation, because it makes no attempt to limit the per-process CPU usage.

The results in Table 4.2 are limited by our measurement infrastructure, which has a resolution of only 10ms – which for 200+ vhosts is not much longer than each process is scheduled. This limitation prevented us from making reliable per-second measurements for larger numbers of vhosts.

Delay Accuracy. The second set of CPU microbenchmarks checks whether processes are scheduled at precisely the right time. This consideration is important for applications that need to perform tasks periodically, e.g., a video streaming application sending packets at a specific rate.

We conduct two tests. In each case, vhost-1 sets an alarm to fire after time t and we compare the actual versus desired waking time. $N-1$ vhost processes run time-waster processes. In Test 1 vhost-1 is otherwise idle. In Test 2 vhost-1 also runs a time-waster process to create "foreground load".

Figure 4.4 shows the relative timer error in the two tests for various values of N, t, and for the two schedulers - *cbw* and *rt*. Both introduce small errors in the timer for Test 1, increasing as t approaches the scheduling period (10ms), where it may fall either side of the boundary. But the main takeaway is that *rt* fails to schedule accurately in Test 2. We discounted *rt* from further consideration.



(a) Timer fidelity: *cbw*

(b) Timer fidelity: *rt*

Figure 4.4: CPU timer fidelity as a function of N and t. Solid lines are for Test 1; dashed lines are for Test 2.

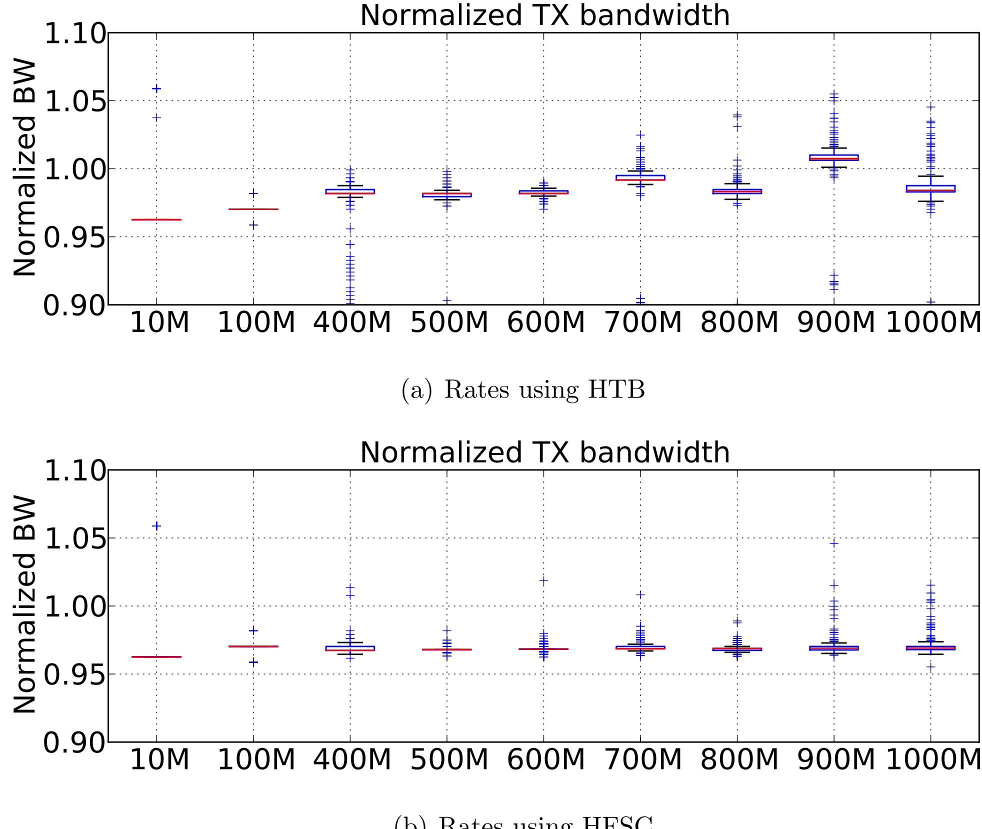
4.4.2 Link Scheduling

We created microbenchmarks to test the accuracy to which the OS can emulate the behavior of a link, for both TCP and UDP flows.

Throughput: We measure the throughput of vhost A transmitting to vhost B over a `veth` pair using TCP. Figures 4.5(a) and 4.5(b) show the distribution of attained rates (normalized) over a 10ms time interval, for the `htb` and `hfsc` schedulers, respectively. Both link schedulers constrain the link rate as expected, although the variation grows as we approach 1Gb/s. The table provides a reminder that Container-Based Emulation runs in real time, and will show measurable errors if run near system CPU limits.[2]

Our next test runs many `veth` pairs in parallel, to find the point at which the CPU can no longer keep up, where most CPU cycles go to switching data through the kernel. Figure 4.6(a) shows the total time spent by the kernel on behalf of end hosts, including system calls, TCP/IP stack, and any processing by the kernel in the context of the end host. The system time is proportional to the number of links, for a given link rate (10Mb/s - 1Gb/s). The graph shows where the system time is

[2]The benchmark revealed a bug in the link scheduler, which remains to be resolved. At rates close to 1Gb/s, it appears we must read from a buffer about 20Mb/s faster than we write to it. This appears to be a bug, rather than a fundamental limitation.



(a) Rates using HTB

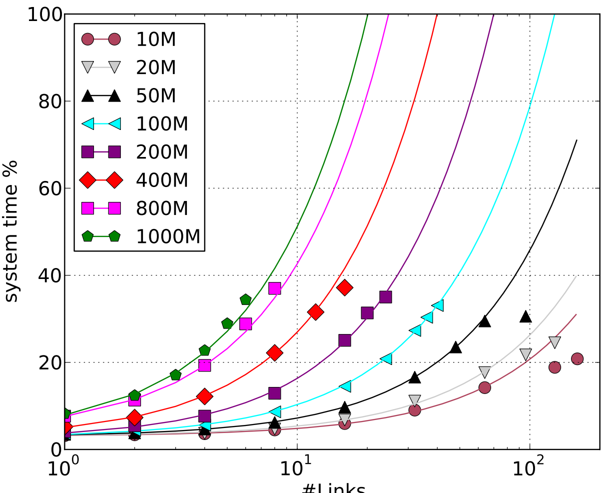
(b) Rates using HFSC

Figure 4.5: Rates for a single `htb` & `hfsc`-scheduled link, fed by a TCP stream.

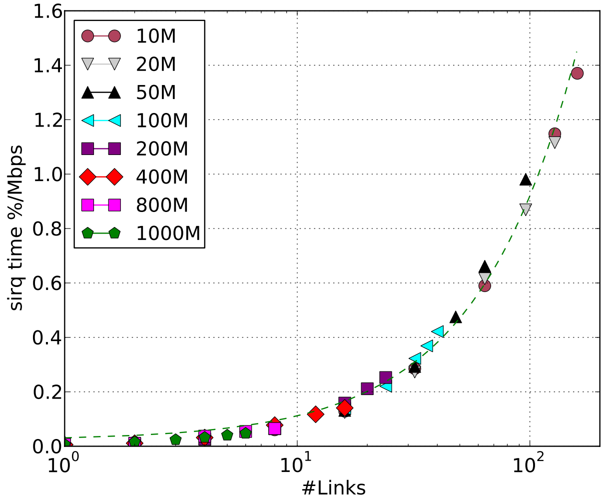
insufficient to keep up (where the points lie below the proportionality line) and the performance fidelity breaks down.

Figure 4.6(a) does not include processing incurred by the network switches, which are accounted as `softirq` time; these are shown in Figure 4.6(b). As expected, the system time is proportional to the aggregate switching activity. Should a measured point deviate from the line, it means the system is overloaded and falling behind.

We can now estimate the total number of `veth` links that we can confidently emulate on our server. If we connect L links at rate R between a pair of vhosts, we expect a total switching activity of $2LR$. Figure 4.7(a) shows the measured aggregate switching capacity for $L = 1 - 128$ `veth` links carrying TCP, at rates $R = 10Mb/s - 1Gb/s$. The solid lines are the ideal aggregate switching activity ($2LR$) when there are sufficient CPU resources. Measured data points below the line represent "breaking

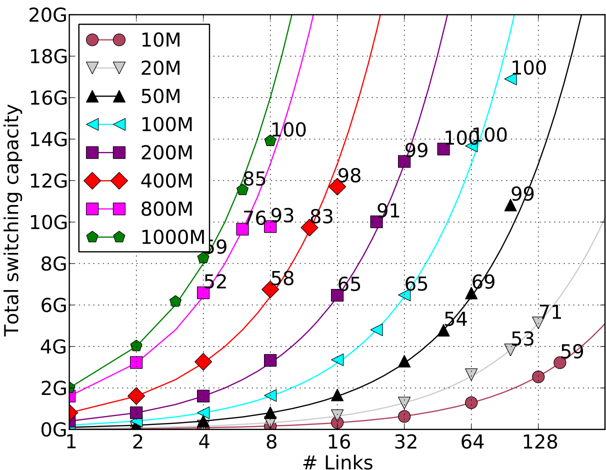


(a) Kernel CPU usage for various rates and number of links

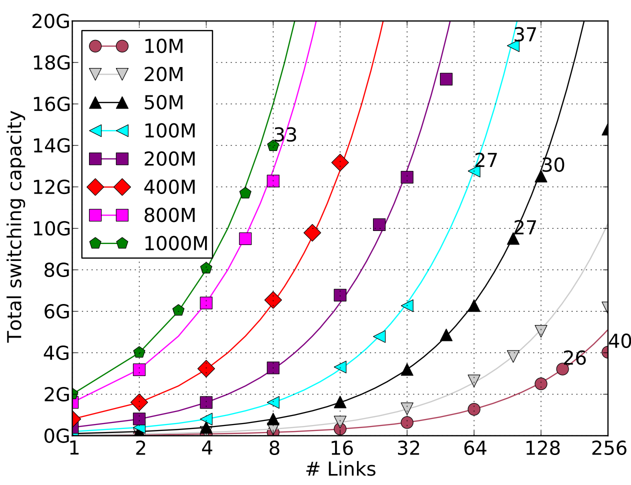


(b) SoftIRQ CPU per Mbps for various rates and number of links

Figure 4.6: Breakdown of CPU usage as we vary the number of links and bandwidth allotted to each link.



(a) Multiple links in parallel



(b) Multiple links in series

Figure 4.7: Provisioning graph for the 8-core platform. Y-axis is the *total* switching capacity observed in the pair test and x-axis is the number of parallel links. The numbers show the total CPU usage for a run at that configuration.

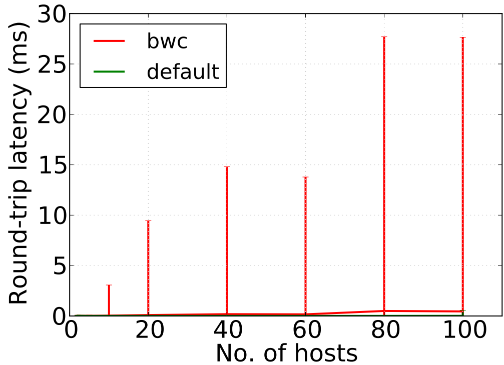


Figure 4.8: Round-trip latency in the presence of background load.

points" when the system can no longer yield high fidelity results; as expected, we see that fidelity is lost as we push the CPU utilization to 100%.

4.4.3 Round-trip latency effects

To examine the effects of background load on round-trip latency, we ran a series of experiments where a client and server communicate in an RPC-like scenario: the client sends a packet to the server which then sends a reply, while other hosts within the experiment run CPU-intensive programs. Since I/O-bound processes will have a lower overall runtime than CPU-bound processes, Linux's default CFS scheduler automatically prioritizes them over CPU-bound processes, waking them up as soon as they become runnable. The goal of this test is to verify that CFS bandwidth limiting preserves this behavior when hosts are limited to 50% of overall system bandwidth.

Figure 4.8 shows round-trip latency in the presence of background load. Although the average request-response time does not increase dramatically from 2 to 80 hosts, the variation (shown by error bars of +/- twice the standard deviation, corresponding to a 95% confidence interval) is larger at higher levels of multiplexing and background load. This graph indicates that it is more likely that a client or server which is ready to run may have to wait until another process completes its time slice.

4.4.4 Summary

The main takeaways from these tests include:

- Process schedulers in Linux sufficiently accurate, as-is. The RMS error for both the *bwc* and *rt* schedulers was on the order of 2%.

- Packet schedulers in Linux are sufficiently accurate, as-is. The 95th percentile of enforced rates of packet schedulers like `htb` and `htb` is within 5% of the actual rate for a wide range of rates from 10Mbps to 1Gbps.

- The queue fidelity of `hfsc` is better than `htb` for higher link speeds.

- The available network bandwidth for an experiment depends depends not only on the total number of links (i.e. total CPU), but also the diameter of the network.

Process and packet schedulers provide resource isolation, but one should still verify the correctness of the experiment through the use of network invariants, as it may not be obvious that resource limits are being reached.

4.5 Experimental Scope

Before presenting successful examples of reproduced results generated using Mininet-HiFi, I want to clearly set expectations for those experiments for which the system may not be the best choice.

The main limitation with Mininet-HiFi is its inability to support experiments with large numbers of hosts at high bandwidths. In its current form, Mininet-HiFi targets experiments that (1) are *network-limited* and (2) have aggregate resource requirements that fit within a single modern multi-core server.

Network-limited refers to experiments that are limited by network properties such as bandwidth, latency, and queueing, rather than other system properties such as disk bandwidth or memory latency; in other words, experiments whose results would not change on a larger, faster server. For example, testing how a new version of

TCP performs in a specific topology on 100 Mb/s links would be an excellent use of Mininet-HiFi, since the performance is likely to be dependent on link bandwidth and latency. In contrast, testing a modified Hadoop would be a poor fit, since MapReduce frameworks tend to stress memory and disk bandwidth along with the network.

Generally, Mininet-HiFi experiments use less than 100 hosts and links. Experiment size will usually be determined by available CPU cycles, virtual network bandwidth, and memory. For example, on a server with 3 GHz of CPU and 3 GB RAM that can provide 3 Gb/s of internal packet bandwidth, one can create a network of 30 hosts with 100 MHz CPU and 100 MB memory each, connected by 100 Mb/s links. Unsurprisingly, this configuration works poorly for experiments that depend on several 1 Gbps links.

Overall, Mininet-HiFi is a good fit for experiments that benefit from flexible routing and topology configuration and have modest resource requirements, where a scaled-down version can still demonstrate the main idea, even with imperfect fidelity. Compared with hardware-only testbeds [34, 22, 25, 30], Mininet-HiFi makes it easier to reconfigure the network to have specific characteristics, and doesn't suffer from limited availability before a conference deadline. Also, if the goal is to scale out and run hundreds of experiments at once, for example when conducting a massive online course or tutorial, using Mininet-HiFi on a public cloud such as Amazon EC2 or on the laptops of individual participants solves the problem of limited hardware resources.

If an experiment requires extremely precise network switching behavior, reconfigurable hardware (e.g. NetFPGAs) may be a better fit; if it requires "big iron" at large scale, then a simulator or testbed is a better choice. However, the Container-Based Emulation approach is not fundamentally limited to medium-scale, network-limited experiments. The current limitations could be addressed by (1) expanding to multiple machines, (2) slowing down time [38], and (3) isolating more resources using Linux Containers [49], as described later in §6.3.

The next section (§5) shows network-limited network experiments that Mininet-HiFi appears to support well. All are sensitive to bandwidth, queues, or latency.

Chapter 5

Experiences Reproducing Research

Past chapters have shown that Mininet-HiFi can yield similar results to hardware on simple topologies with simple traffic patterns. This chapter ups the ante to test Mininet-HiFi on more complex experiments that were previously published in conference papers, covering queueing, transport protocols, and topology evaluations. If one can successfully reproduce results that originally came from measurements on hardware testbeds — using Mininet-HiFi, on a single system — then it strongly suggests that Mininet-HiFi is a platform capable of reproducing networking network research, and potentially one on which to generate original results.

In particular, we want to understand these aspects of Mininet-HiFi relating to reproducing network research:

Range of Experiments. The first goal is to see whether results measured on Mininet-HiFi can qualitatively match the results generated on hardware, for a range of network-limited network experiments.

Fidelity Monitoring. The second goal is to see whether the monitoring mechanisms in Mininet-HiFi can successfully indicate the fidelity of an experiment, by using network invariants to distinguish "emulation runs to trust" from "emulation runs to ignore".

Practicality: The third goal is to evaluate the practicality of replicating network systems research, and to identify common pitfalls that may arise in the process.

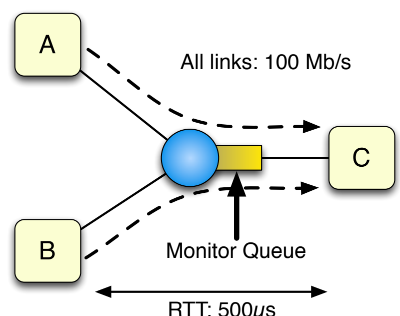


Figure 5.1: Topology for TCP and DCTCP experiments.

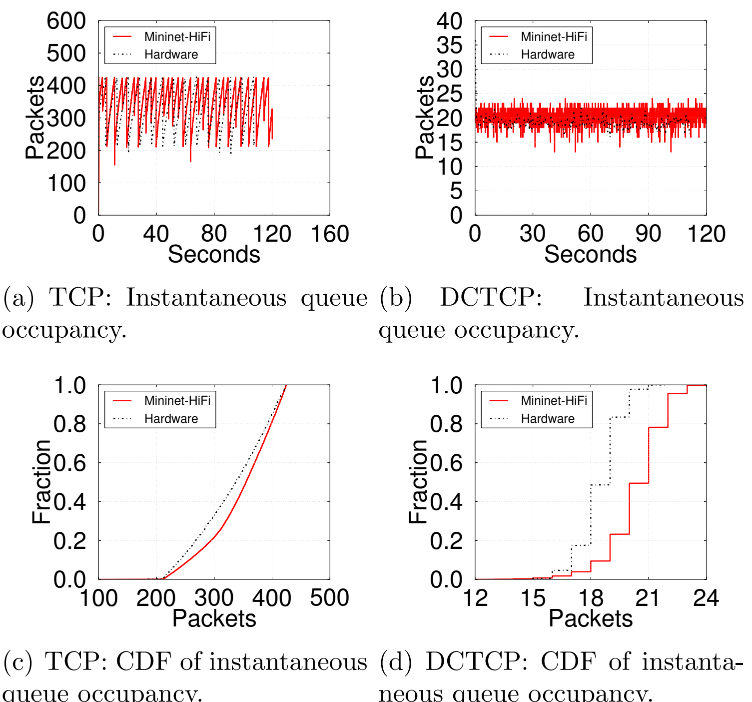
Each published result in the first three sections originally used a custom testbed, because there was no shared testbed available with the desired characteristics; either the experiment required custom packet marking and queue monitoring (§5.1), a custom topology with custom forwarding rules (§5.2), or long latencies and control over queue sizes (§5.3). For the Mininet-HiFi runs, I use an Intel Core i7 server with four 3.2 GHz cores and 12 GB of RAM.

The fourth section covers experiences from the Spring 2012 editions of Stanford CS244, Advanced Topics in Networking, where the class project was to attempt to reproduce a previously published result. Each student project, as well as the first three in-depth experiments, links to a "runnable" version; clicking on each figure in the PDF links to instructions to replicate the experiment.

5.1 DCTCP

Data-Center TCP was proposed in SIGCOMM 2010 as a modification to TCP's congestion control algorithm [9] with the goal of simultaneously achieving high throughput and low latency. DCTCP leverages the Explicit Congestion Notification [72] feature in commodity switches to detect and react not only to the presence of network congestion but also to its *extent*, measured through the sequence of ECN marks stamped by the switch.

To test the ability of Mininet-HiFi to precisely emulate queues, we attempt to replicate an experiment in the DCTCP paper showing how DCTCP can maintain



(a) TCP: Instantaneous queue occupancy.

(b) DCTCP: Instantaneous queue occupancy.

(c) TCP: CDF of instantaneous queue occupancy.

(d) DCTCP: CDF of instantaneous queue occupancy.

Figure 5.2: Reproduced results for DCTCP [9] with Mininet-HiFi and a identically configured hardware setup. Figure 5.2(b) shows that the queue occupancy with Mininet-HiFi stays within 2 packets of hardware.

high throughput with very small buffers. This experiment uses the Linux DCTCP patch as the paper authors [24]. In both Mininet-HiFi and on real hardware in our lab,[1] we created a simple topology of three hosts A, B and C connected to a single 100 Mb/s switch, as shown in Figure 5.1. In Mininet-HiFi, we configured ECN through Linux Traffic Control's RED queuing discipline and set a marking threshold of 20 packets.[2] Hosts A and B each start one long-lived TCP flow to host C. We monitor the instantaneous output queue occupancy of the switch interface connected to host C. Figure 5.2 shows the queue behavior in Mininet-HiFi running DCTCP and from an equivalently configured hardware setup.

Importantly, the *behavior* of the experiment stays the same. On Mininet-HiFi,

[1] We used the same Broadcom switch as the authors.

[2] As per [9], 20 packets exceeds the theoretical minimum buffer size to maintain 100% throughput at 100 Mb/s.

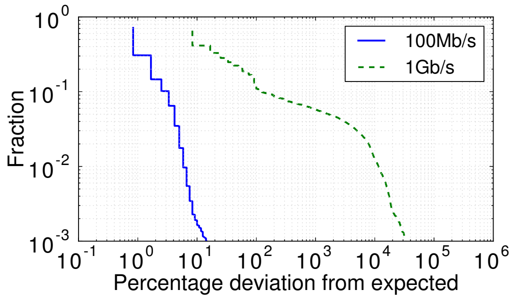


Figure 5.3: Complementary CDF of inter-dequeue time deviations from ideal; high fidelity at 100 Mb/s, low fidelity at 1 Gb/s.

running on a single core, TCP does the familiar sawtooth, with the same extents, and DCTCP shows queue lengths that in 20% of cases exceed the maximum seen on hardware, of 21 packets, but the minimum of 15 packets is the same. Both of these differences in the queue distributions can be explained by a different effective round-trip time (RTT). For the Mininet experiments, the RTT was not extended to match the necessary RTT and RTT variability induced by NIC receive and transmit hardware. Since Mininet never uses the NIC driver routines, its NIC forwarding delay is effectively zero.

Verifying fidelity: DCTCP's dynamics depend on queue occupancy at the switch, so the experiment relies on accurate link emulation. Figure 2.7, plus the complete discussion in §2, show measured distributions of packet spacing errors for a range of link speeds. To summarize the DCTCP results, a single core shows errors entirely less than the delay of a single packet time up to 80 Mb/s. At 160 Mb/s, 90% of packets experience sub-one-packet delays, but a remaining fraction of packets exceed this, and even though the experiment yields the same full-throughput results, the failed invariant check suggest the run should be ignored. At 320 Mb/s, a greater fraction of packets are overly delayed, and the link does not even achieve full throughput.

Lessons: The main lesson learned from this experiment was that transport protocols, which depend on queue fidelity, can be emulated using Mininet-HiFi. These

algorithms can be tested within Mininet-HiFi, as long as their kernel implementations support network namespaces. For a different transport protocol, Multipath TCP (MPTCP), adding network namespace support required the addition of a small kernel patch [70].

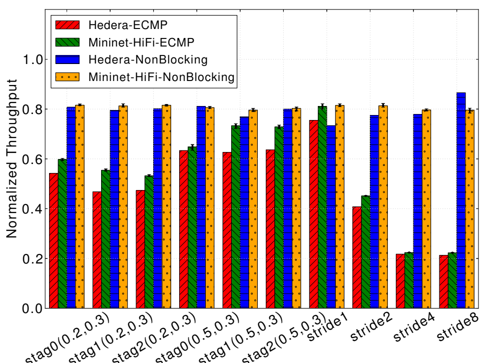
5.2 Hedera

Our second example uses Mininet-HiFi to reproduce results that were originally measured on a real hardware testbed in *Hedera* [6], a dynamic flow scheduler for data center networks, as presented at NSDI 2010. With Equal-Cost Multi-Path (ECMP) routing, a strategy implemented on hardware switches today, and commonly used to spread packets in data centers and across WAN links, flows take a randomly picked path through the network based on a hash of the packet header. The ECMP hash function takes as input the standard IP five-tuple, which includes the IP source, IP destination, IP protocol, L4 source port, and L4 destination port. Any hash based on the five-tuple prevents packet reordering by ensuring that all packets belonging to a flow take the same path [81]. Hedera shows that this simple approach leads to random hash collisions between "elephant flows" – flows that are a large fraction of the link rate – causing the aggregate throughput to plummet. This behavior is especially pronounced in Fat Trees, where the speed of the edge links matches that of the core links. With this throughput loss result as its motivation, Hedera proposes to intelligently re-route flows to avoid collisions, and thus, exploit all the available bandwidth.

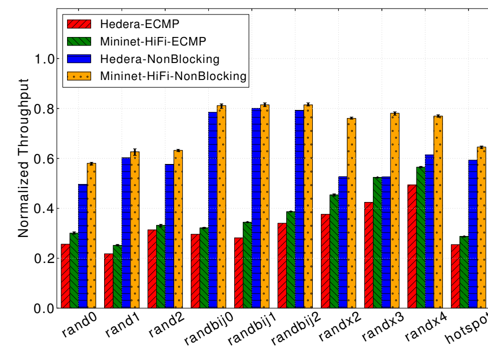
More specifically, as part of the evaluation, the authors compare the throughput achieved by ECMP with that of an ideal "non-blocking" network (the maximum achievable) for 20 different traffic patterns (Figure 9 in the original paper [6]). The authors performed their evaluation on a hardware testbed with a $k = 4$ Fat Tree topology with 1 Gb/s links. The main metric of interest is the aggregate throughput relative to the full bisection bandwidth of the network.

To test the ability of Mininet-HiFi to emulate a complex topology with many links, switches, and hosts, we replicate the ECMP experiment from the paper. We use the

same $k = 4$ Fat Tree topology and the same traffic generation program provided by the Hedera authors to generate the same traffic patterns. To route flows, we use RipL-POX [4], a Python-based OpenFlow controller. We set the link bandwidths to 10 Mb/s and allocate 25% of a CPU core on our eight core machine to each of 16 hosts (i.e. a total of 50% load on the CPU). We set the buffer size of each switch to 50 packets per port, our best estimate for the switches used in the hardware testbed.



(a) Benchmark tests from Hedera paper (Part 1).



(b) Benchmark tests from Hedera paper (Part 2).

Figure 5.4: Effective throughput with ECMP routing on a $k = 4$ Fat Tree vs. an equivalent non-blocking switch. Links are set to 10 Mb/s in Mininet-HiFi and 1 Gb/s in the hardware testbed [6].

Figure 5.4 shows the normalized throughput achieved by the two routing strategies – ECMP and non-blocking – with Mininet-HiFi, alongside results from the Hedera paper for different traffic patterns. The Mininet-HiFi results are averaged over three runs. The traffic patterns in Figure 5.4(a) are all bijective; they should all achieve maximum throughput for a full bisection bandwidth network. This is indeed the case for the results with the "non-blocking" switch. The throughput is lower for ECMP because hash collisions decrease the overall throughput. We can expect more collisions if a flow traverses more links. All experiments show the same behavior, as seen in the `stride` traffic patterns. With increasing stride values (1, 2, 4 and 8), flows traverse more layers, decreasing throughput.

The ECMP results obtained on the Hedera testbed and Mininet-HiFi differed significantly for the `stride-1, 2, 4 and 8` traffic patterns. Figure 5.4(a). At higher

stride levels that force all traffic through core switches, the aggregate throughput with ECMP should reduce. However, the extent of reduction reported for the Hedera testbed was exactly consistent with that of the spanning-tree routing results from Mininet-HiFi. We postulate that a misconfigured or low-entropy hash function may have unintentionally caused this style of routing. After several discussions with the authors, we were unable to explain a drop in the ECMP performance reported in [6]. Therefore, we use spanning tree routing for Mininet-HiFi Hedera results for *all* traffic patterns. For consistency with the original Hedera graphs, we continue to use the "ECMP" label.

After this change, the Mininet-HiFi results closely matched those from the hardware testbed; in 16 of the 20 traffic patterns they were nearly identical. In the remaining four traffic patterns (`randx2,3,4` and `stride8`) the results in the paper have lower throughput because – as the authors explain – the commercial switch in their testbed is built from two switching chips, so the total buffering depends on the traffic pattern. To validate these results, we would need to know the mapping of hosts to switch ports, which is unavailable.

One lesson learned from this experiment is that Mininet-HiFi can reproduce performance results for data-center networking experiments with a complex topology. This experiment demonstrates that with a CBE, it is possible to collect meaningful results in advance of (or possibly without) setting up a hardware testbed. If a testbed is built, the code and test scripts used in Mininet-HiFi can be reused.

A second lesson was a reminder of the value of reproduction and the difficulty in doing an exact reproduction. This experiment revealed ambiguous methodology details that would never be disclosed in the limited text space of a paper. Beyond this, the original authors may not even be aware of hardware parameters that can affect the results, as these may be undisclosed and hard to test.

Verifying fidelity: Unlike DCTCP, the Hedera experiment depends on coarse-grained metrics such as aggregate throughput over a period of time. To ensure that no virtual host starved and that the system had enough capacity to sustain the network demand, we measured idle time during the experiment (as described in §4.3). In all runs, the system had at least 35% idle CPU time, measured each second.

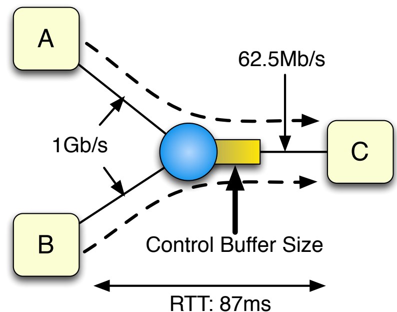


Figure 5.5: Buffer sizing experiment topology.

Scaling the experiment: In the Hedera testbed, machines were equipped with 1 Gb/s network interfaces. We were unable to use Mininet-HiFi to replicate Hedera's results even with 100 Mb/s network links, as the virtual hosts did not have enough CPU capacity to saturate their network links. While Hedera's results do not qualitatively change when links are scaled down, it is a challenge to reproduce results that depend on the absolute value of link/CPU bandwidth.

5.3 Buffer Sizing

Our third experiment example reproduces results that were measured on a custom hardware testbed to determine the number of packet buffers needed by a router. All Internet routers contain buffers to hold packets during times of congestion. These buffers must be large enough to ensure that the buffer never goes empty, which will ensure full link utilization of 100%.

At SIGCOMM 2004, a paper by Appenzeller et al. questioned the commonly-held wisdom for sizing buffers in Internet routers [10]. Prior to the paper, the common assumption was that each link needs a buffer of size $B = RTT \times C$, where RTT is the average round-trip time of a flow passing across the link and C is the data-rate of the bottleneck link. The authors showed that a link with n flows requires no more than $B = \frac{RTT \times C}{\sqrt{n}}$; when the number of flows is large, the needed buffering reduces to a small fraction of that required to support a single large flow at full line rate. The original paper included results from simulation and measurements from a real router,

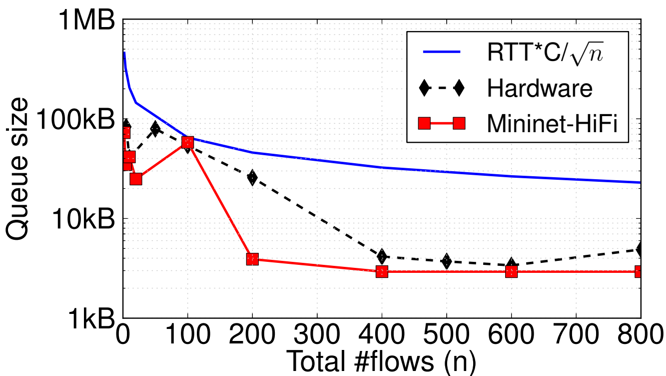


Figure 5.6: Results: Buffers needed for 99% utilization, comparing results from the Internet2 testbed, Mininet-HiFi, and the theoretical upper bound.

but not for a real network. Later, at SIGCOMM 2008, this result was demonstrated on a hardware testbed running on the Internet2 backbone.[3]

To test the ability of Mininet-HiFi to emulate hundreds of simultaneous, interacting flows, we replicated this hardware experiment. We contacted the researchers and obtained results measured on their hardware testbed, then compared them with results from Mininet-HiFi; the Mininet-HiFi topology is shown in Figure 5.5. In the hardware experiments, a number of TCP flows go from a server at Stanford University (California) to at a server at Rice University (Houston, Texas) via a NetFPGA [48] IPv4 router in the Internet2 POP in Los Angeles. The link from LA to Houston is constrained to 62.5 Mb/s to create a bottleneck and the end-to-end RTT is set to 87 ms to match the measured value. Once the flows are established, a script runs a binary search to find the buffer size needed for 99% utilization on the bottleneck link. Figure 5.6 shows results from theory, hardware, and Mininet-HiFi. Both Mininet-HiFi and hardware results are averaged over three runs. On Mininet-HiFi, the average CPU utilization did not exceed 5%.

Both results are bounded by the theoretical limit and confirm the new rule of thumb for sizing router buffers. Mininet-HiFi results show similar trends to the hardware results, with some points being nearly identical. If Mininet-HiFi had been

[3]Video of demonstration at http://www.youtube.com/watch?v=ykga6N_x27w.

available for this experiment, the researchers could have gained additional confidence that the testbed results would match the theory.

There are some notable differences in the results, particularly at 200 and 300 flows, which are likely explained by different measurement scripts. In the process of doing a clean-room reimplementation of the binary search script, we came across a surprising number of factors that would influence the test results, on both hardware and Mininet-HiFi. One key parameter was the "settling time" parameter. This parameter describes the time after any queue-size change during which the test script must wait before measuring the queue occupancy. The script must be careful to give the TCP connections sufficient time to stabilize. If set too low, the measurement variability will increase. In the binary search direction of an increasing queue size, this parameter doesn't matter as much, especially if the measured queue occupancy already exceeds the target, because the change will cause no additional packet drops. However, in the opposite binary search direction, where the queue size is reduced, losses are more likely to be instantaneously triggered, such that a flow might experience a TCP timeout or back off more than it usually would in steady-state operation. Because of this issue, the testing script must wait for multiple RTTs, at minimum.

Another example of the practical challenge with a seemingly simple binary search is specific to Mininet-HiFi. Links in Mininet-HiFi are not perfect, and they will vary slightly in speed. When the queue utilization target (the signal) is close to 99%, or even higher, link-speed variation (the noise) must stay below 1% for the full duration of the experiment to succeed. Any single error in the binary search will effectively chop the search space and prevent the search from approaching the correct value. Hence, we averaged multiple results, and reduced the target from an original value of 99.5% to a smaller value of 99%.

Verifying fidelity: Like the DCTCP experiment, the buffer sizing experiment relies on accurate link emulation at the bottleneck. However, the large number of TCP flows increases the total CPU load. We visualize the effect of system load on the distribution of deviation of inter-dequeue times from that of an ideal link. Figure 5.7 plots the CDF of deviations (in percentage) for varying numbers of flows. Even for 800 flows, more than 90% of all packets in a 2 s time interval were dequeued

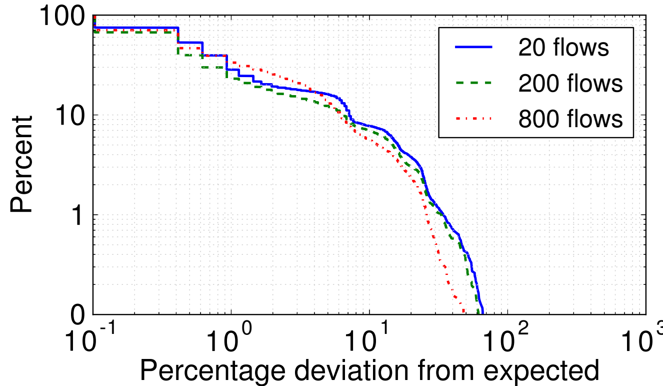


Figure 5.7: Verifying Fidelity: A large number of flows increases the inter-dequeue time deviations only by 40% from that of an ideal 100% utilized link, and only for 1% of all packets in a 2s time window.

within 10% of the ideal dequeue time (of 193.8 μs for full-sized 1514 byte packets). Even though inter-dequeue times were off by 40% for 1% of all packets, results on Mininet-HiFi qualitatively matched that of hardware.

Scaling the experiment: The experiment described in the original paper used multiple hosts to generate a large number of TCP flows. To our surprise, we found that a single machine was capable of generating the same number (400–800) of flows and emulating the network with high fidelity. While results on Mininet-HiFi qualitatively matched hardware, we found that the exact values depended on the version of the kernel (and TCP stack).

5.4 CS244 Spring 2012

After successfully replicating several experiments with Mininet-HiFi, the next step was to attempt to reproduce as broad a range of networking research results as possible, to learn the limits of the approach as well as how best to reproduce others' results. For this task we enlisted students in CS244, a masters-level course on Advanced Topics in Networking in Spring quarter 2012 at Stanford, and made reproducing research the theme of the final project. In this section, we describe the individual project outcomes along with lessons we learned from their assignments.

5.4.1 Project Assignment

The class included masters students, undergraduate seniors, and remote professionals, with systems programming experience ranging from a few class projects all the way to years of Linux kernel development. We divided the class of 37 students into 18 teams (17 pairs and one triple).

For their final project, students were given a simple, open-ended request: choose a published networking research paper and try to replicate its primary result using Mininet-HiFi on an Amazon EC2 instance. Teams had four weeks: one week to choose a paper, and three weeks to replicate it. Amazon kindly donated each student $100 of credit for use on EC2. Each team created a blog post describing the project, focusing on a single question with a single result, with enough figures, data, and explanation to convince a reader that the team had actually reproduced the result – or discovered a limitation of the chosen paper, EC2, or Mininet-HiFi. As an added wrinkle, each team was assigned the task of running another team's project to reproduce their results, given only the blog post.

Table 5.1: Student projects for CS244 Spring 2012, in reverse chronological order. Each project was reproduced on Mininet-HiFi on an EC2 instance. The image for each project links to a full description, as well as instructions to replicate the full results, on the class blog: http://reproducingnetworkresearch.wordpress.com.

Project	Image	Result to Replicate	Outcome
CoDel [54]		The Controlled Delay algorithm (CoDel) improves on RED and tail-drop queueing by keeping delays low in routers, adapting to bandwidth changes, and yielding full link throughput with configuration tweaking.	replicated + extra results
HULL [8]		By sacrificing a small amount of bandwidth, HULL can reduce average and tail latencies in data center networks.	replicated
MPTCP [71]		Multipath TCP increases performance over multiple wireless interfaces versus TCP and can perform seamless wireless handoffs.	replicated

Continued on next page

Table 5.1 – continued from previous page

Project	Image	Result to Replicate	Outcome
		Over 3G and WiFi, optimized Multipath TCP with at least 600 KB of receive buffer can fully utilize both links.	replicated, w/differences
Outcast [67]		When TCP flows arrive at two input ports on a tail-drop switch and compete for the same output port, the port with fewer flows will see vastly degraded throughput.	replicated
		The problem described for Outcast [67] occurs in a topology made to show the problem, as well as in a Fat Tree topology, and routing style does not necessarily alleviate the issue.	replicated + extra results
Jellyfish [77]		Jellyfish, a randomly constructed network topology, can achieve good fairness using k-shortest paths routing, comparable to a Fat Tree using ECMP routing, by using MPTCP.	replicated + extra results
		Jellyfish, a randomly constructed network topology, can achieve similar and often superior average throughput to a Fat Tree.	replicated
DCTCP [9]		Data Center TCP obtains full throughput with lower queue size variability than TCP-RED, as long as the ECN marking threshold K is set above a reasonable threshold.	replicated
Hedera [6]		The Hedera data center network flow scheduler improves on ECMP throughput in a $k = 4$ Fat Tree, and as the number of flows per host increase, the performance gain of Hedera decreases.	replicated
Init CWND [28]		Increasing TCP's initial congestion window can significantly improve the completion times of typical TCP flows on the Web.	replicated
		Increasing TCP's initial congestion window tends to improve performance, but under lossy network conditions an overly large initial congestion window can hurt performance.	replicated
Incast [86]		Barrier-synchronized TCP workloads in datacenter Ethernets can cause significant reductions in application throughput.	unable to reproduce
DCell [37]		DCell, a recursively-defined data center network topology, provides higher bandwidth under heavy load than a tree topology.	replicated + extra results

Continued on next page

Table 5.1 – continued from previous page

Project	Image	Result to Replicate	Outcome
		The routing algorithm used for DCell can adapt to failures.	replicated
FCT [27]		The relationship between Flow Completion Time (FCT) and flow size is not ideal for TCP; small flows take disproportionately long.	replicated
TCP Daytona [75]		A misbehaving TCP receiver can cause the TCP sender to deliver data to it at a much higher rate than its peers.	replicated
RED [33]		Random Early Detection (RED) gateways keep average queue size low while allowing occasional bursts of packets.	unable to reproduce

5.4.2 Project Outcomes

Table 5.1 lists the teams' project choices, the key results they tried to replicate, and the project outcomes. If you are viewing this paper electronically, clicking on each experiment image in the table will take you to the blog entry with instructions to reproduce the results. Students chose a wide range of projects, covering transport protocols, data center topologies, and queueing: MPTCP [70, 71], DCTCP [9], Incast [86], Outcast [67], RED [33], Flow Completion Time [27], Hedera [6], HULL [8], Jellyfish [77], DCell [37], CoDel [54], TCP Initial Congestion Window [28], and Misbehaving TCP Receivers [75]. In eight of the eighteen projects, the results were so new that they were only published after the class started in April 2012 (MPTCP Wireless, Jellyfish, HULL, TCP Outcast, and CoDel).

After three weeks, 16 of the 18 teams successfully reproduced at least one result from their chosen paper; only two teams could not reproduce the original result.[4] Four teams added new results, such as understanding the sensitivity of the result

[4]The inability to replicate RED could be due to bugs in network emulation, a parameter misconfiguration, or changes to TCP in the last 20 years; for Incast, we suspect configuration errors, code errors, or student inexperience.

to a parameter not in the original paper. By "reproduced a result", we mean that the experiment may run at a slower link speed, but otherwise produces qualitatively equivalent results when compared to the results in the papers from hardware, simulation, or another emulated system. For some papers, the exact parameters were not described in sufficient detail to exactly replicate, so teams tried to match them as closely as possible.

All the project reports with the source code and instructions to replicate the results are available at `reproducingnetworkresearch.wordpress.com`, and we encourage the reader to view them online.

5.4.3 Lessons Learned

The most important thing we learned from the class is that paper replication with Mininet-HiFi on EC2 is reasonably *easy*; students with limited experience and limited time were able to complete a project in four weeks. Every team successfully validated another team's project, which we credit to the ability to quickly start a virtual machine in EC2 and to share a disk image publicly. All experiments could be repeated by another team in less than a day, and most could be repeated in less than an hour.

The second takeaway was the breadth of replicated projects; Table 5.1 shows that the scope of research questions for which Mininet HiFi is useful extends from high-level topology designs all the way down to low-level queueing behavior. With all projects publicly available and reproducible on EC2, the hurdle for extending, or even understanding these papers, is lower than before.

When was it easy? Projects went smoothly if they primarily required configuration. One example is TCP Outcast. In an earlier assignment to learn basic TCP behavior, students created a parking-lot topology with a row of switches, each with an attached host, and with all but one host sending traffic to a single receiver. Students could measure the TCP sawtooth and test TCP's ability to share a link fairly. With many senders, the closest sender to the receiver saw lower throughput. In this case, simply configuring a few `iperf` senders and monitoring bandwidths was enough to demonstrate the TCP outcast problem, and every student did this inadvertently.

Projects in data center networking such as Jellyfish, Fat Tree, and DCell also went

smoothly, as they could be built atop open-source routing software [63, 4]. Teams found it useful to debug experiments interactively by logging into their virtual hosts and generating traffic. A side benefit of writing control scripts for emulated (rather than simulated) hosts is that when the experiment moves to the real world, with physical switches and servers, the students' scripts can run without change [47].

When was it hard? When kernel patches were unavailable or unstable, teams hit brick walls. XCP and TCP Fast Open kernel patches were not available, requiring teams to choose different papers; another team wrestled with an unstable patch for setting microsecond-level TCP RTO values. In contrast, projects with functioning up-to-date patches (e.g. DCTCP, MPTCP, and CoDel) worked quickly. Kernel code was not strictly necessary – the Misbehaving TCP Receivers team modified a user-space TCP stack – but kernel code leads to higher-speed experiments.

Some teams reported difficulties scaling down link speeds to fit on Mininet-HiFi if the result depended on parameters whose dependence on link speed was not clear. For example, the Incast paper reports results for one hardware queue size and link rate, but it was not clear when to expect the same effect with slower links, or how to set the queue size [86]. In contrast, the DCTCP papers provided guidelines to set the key parameter K (the switch marking threshold) as a function of the link speed.

Chapter 6

Conclusion

The thesis statement of this dissertation was the following:

High-fidelity emulation, the combination of resource isolation and fidelity monitoring, enables network systems experiments that are realistic, verifiable, and reproducible.

The past three chapters have covered each piece of this thesis statement, in depth. Chapter 2 defined high-fidelity emulation as a combination of resource isolation and fidelity monitoring. Chapter 3 described the architecture and implementation of Mininet-HiFi, a realization of high-fidelity emulation. Chapter 5 showed examples of successfully reproduced network system experiments; in each case, the results match those originally generated using hardware testbeds.

Having demonstrated the feasibility of High-Fidelity Emulation to enable reproducible research, this conclusion zooms out to the bigger picture:

Status Report. What progress has been made toward enabling a culture of reproducible networking research with HFE? Section 6.1 details the community forming around Mininet and Mininet-HiFi, as well as interest in using it for reproducible network experiments.

Emulating Software-Defined Networks. A recent trend in networking is Software-Defined Networking (SDN). Section 6.2 explains how High-Fidelity Emulation

can be a great fit for SDN development and explains why most Mininet adoption has occurred in the SDN community.

Future Work. The most acute limitations of Mininet-HiFi are a direct result of running on a single server, in real time. Section 6.3 describes possibilities to relax this core assumption, as well as improve the overall usability of the system.

The conclusion ends with a few closing thoughts, in Section 6.4.

6.1 Status Report

"Runnable" is not yet the default for networks system papers, but high-fidelity emulation is becoming more accessible and existence proofs of runnable papers are becoming more common and visible. This section describes specific progress indicators for the adoption of High-Fidelity Emulation.

Software Availability. Original Mininet has been publicly available since 2010. Resource isolation was the main feature addition for the Mininet 2.0 release, released at the end of 2012. Pre-built VMs for Mininet, along with documentation and install instructions, are available at `www.mininet.org`. The second half of the HiFi additions, the fidelity monitoring code, are generally available [52], as is the additional code required to run experiments described in this thesis [51] As of early 2013, Mininet is actively developed and supported through Open Networking Labs [62].

Software Community The Mininet mailing list [50] averages multiple new posts each day, with 720 members as of April 25, 2013. Questions that relate to experiment reproduction are starting to appear, and the software is being used for class assignments [31].

Reproducible Research Examples. The Reproducing Network Research Blog [3] has 49 posts, as of April 25, 2013, covering the experiments described in this

Name	Location	Date	Attendance
SBRC	Brazil	5/25/10	40
GEC8	San Diego	7/22/10	40
HOTI	Mountain View	8/20/10	20
GEC9	Washington, DC	11/1/10	35
EU OpenFlow	Berlin	2/1/11	40
GEC10	Puerto Rico	3/1/11	30
HOTI	San Jose	8/1/11	50
Ofelia	Berlin	11/1/11	40
ONS	Stanford	10/15/11	70
ONS	San Jose	4/16/12	180
ONS	San Jose	4/15/13	150

Table 6.1: Tutorials Using Mininet

thesis, class projects by students, and a few others. The first external contribution on this blog would be a great sign of interest in reproducible networking research, but has not happened yet.

Tutorials. Table 6.1 shows details of past SDN tutorials that have employed Mininet. Over 600 people have gained hands-on experience with Container-Based Emulation through these tutorials, 7 of which I led. A similar number of people were likely introduced through the online OpenFlow tutorial, which I wrote [60].

6.2 Emulating Software-Defined Networks

There is no strict dependency between High-Fidelity Emulation and Software-Defined Networking; one is not required to use the other, and vice versa. However, since SDN is gaining momentum within both academia and industry, and SDN prototyping is the most visible use case of Mininet, the synergy here is worth discussing.

In short, SDN describes a network where control plane functionality is implemented as functions to change states of a logically centralized network view, rather than the more traditional way: as eventually-consistent, fully distributed software, spread across many forwarding boxes. §3.1 provides more detail, including an architectural picture. By moving functionality into software that (1) does not require the

design or standardization of distributed protocols, (2) can build on primitives to perform common “protocol” tasks, like topology discovery and routing, and (3) can be more easily changed, SDN opens up architectural design choices for network designers and operators. When network functionality is increasingly designed in software, and frequently in ways that require no hardware changes, tools to test network software become relatively more valuable.

The community adoption of Mininet came from riding this wave of early SDN programmers looking for development tools. In some cases, an inability to procure hardware OpenFlow switches with the necessary scale or features made it the only available choice. In other cases, the ability to script regression tests to verify a controller atop a range of topologies made it the preferred choice. Having a tool that “runs the same code” makes Mininet attractive when compared to a simulator, as real code is required to use a network controller in a real network, and Mininet removes the need to build a separate model for this network controller. In addition, as network controllers increasingly build on open-source code, the ability to run the same code becomes even more attractive.

I personally expect the HiFi extensions to be a big part of Mininet usage going forward. Step one is “Does my code work?”; this step is well-supported now and may even become subject to automatic testing in the near future that builds on the defined network behavior of OpenFlow [16]. Step two is “How fast does my network perform?”, which is where the HiFi extensions come into play. Example applications that need to build a performance understanding before deployment include network-integrated load balancing [40], routing on randomly-constructed data center topologies [77], and even basic network controllers that use fine-grained flow management techniques [20]. The challenge is that this step can only be addressed when we can trust the performance numbers that result, and this is the one question upon which my disseration has focused.

6.3 Future Work

The most natural follow-on work would expand the scale of the experiments that high-fidelity emulation can support, in both the space and time dimensions. Other follow-on could improve scheduling to reduce the likelihood of invariant violations.

Space. In the space dimension, multiple servers could support network experiments with higher total bandwidths, as well as larger numbers of hosts and switches. ModelNet [84] and DieCast [39] demonstrated scaling to multiple servers, but both projects use a completely different set of experiments that are more focused on large-scale distributed systems behavior than repeatable, low-level network behavior. In addition, the approach of monitoring network invariants requires an extension to multiple servers.

In a dedicated cluster, multiple-server support appears possible through a combination of (1) fine-grained time synchronization support and (2) careful resource provisioning. IEEE 1588 (Precision Time Protocol, or PTP) [68] uses link-level packet timestamps to theoretically achieve sub-8-nanosecond time-synchronization precision, given hardware timestamp support and a software daemon [69]. Such support would extend the single clock domain of Mininet to multiple clocks domains with bounded error. The second piece, a topology-aware resource scheduler to "place" virtual switches and hosts on specific machines, similar to the `assign` program used to solve the Testbed Mapping Problem for the main Emulab testbed [73], could ensure that link capacity and CPU capacity constraints are satisfied. Such static, centralized networking provisioning would be required to minimize queueing delays within the network and prevent unbounded timing errors.

However, using *multiple* servers in the cloud appears to represent a greater research challenge. In this deployment environment, machines can have time-varying and placement-dependent network performance. Using Mininet-HiFi with only one machine in the cloud works well, as long as the one machine gets at least one dedicated CPU core, as with XL or larger instances on Amazon EC2; perhaps the dedicated network provided to cluster compute instances on EC2 would provide guaranteed bandwidth, but this assumption must be tested.

Time. In the time dimension, Time Dilation [38] and SliceTime [89] demonstrate methods to slow down the time perceived by applications to run an experiment with effectively larger resources. Time Dilation implements a global time dilation factor by wrapping the experiments within virtual machines and altering their time-based system calls, while SliceTime changes the VMM scheduler implementation to bound the time by which one VM can exceed the others. The network invariants approach, as well as the container-based emulation approach, should still work with each of these methods, provided the invariant timings process virtual timestamps. One natural extension is to automatically dilate time to run an experiment at the minimum slowdown that yields the required level of fidelity.

Scheduler. Mininet-HiFi employs the CFS scheduler with bandwidth limits [83], but this scheduler has no "network invariant awareness", in that it is completely unaware of the concept of network invariants. Despite this limitation, it works well in practice, because transport protocols implemented in the kernel reduce the need to schedule processes in and out to handle basic network activity and event-driven protocols tend to create a yield point whenever they send a packet. Reducing scheduler parameters seems like a worthwhile next set of tests, as it would reduce the worst-case delay for any single process, at the expense of increased scheduler overhead.

However, deeper changes to the scheduler could further reduce invariant violations, yet still run in real time. For example, the scheduler might be modified to prioritize the execution of the process that was scheduled out the farthest in the past, in an attempt to minimize event-to-run delays. This idea is inspired by the real-time scheduler, but more like the CFS scheduler targeting scheduler latency fairness rather than CPU bandwidth fairness. Another option might be to prioritize packet schedulers in the kernel over process schedulers, to minimize the chance of a timing violation; in a sense, this would be a lightweight version of SliceTime [89].

6.4 Closing Thoughts

Let us say I'm reading through a paper and I come across an interesting result that I'd like to reproduce. I click on the figure, and a tab in my browser opens up with

a description of the experiment. I scroll to bottom to get explicit instructions for how to reproduce the result. I can run this exact same experiment on the cloud, so I launch my own little testbed, an EC2 instance. I run a command in that instance, and get my result, 8 minutes later, after spending 8 cents. Now I have a working version of the paper in front of me, running all the same code on the same network.

If every published network-research work is easily reproducible, like this – *runnable* – then it becomes easier to stand on the shoulders, rather than the toes, of past network systems researchers, to build and demonstrate better networks. The sheer number and scope of the experiments covered in this paper – 19 successfully reproduced – suggest that this future direction for the network systems research community is possible with Container-Based Emulation. CBE, as demonstrated by Mininet-HiFi, meets the goals defined in Section 1.2 for a reproducible research platform, including functional realism, timing realism, topology flexibility, and easy replication at low cost.

As a community we seek high-quality results, but our results are rarely reproduced. It is my hope that this paper will spur such a change, by convincing authors to make their next paper a runnable one, built on a CBE, with public results, code, and instructions posted online. If enough authors meet this challenge, the default permissions for network systems papers will change from "read only" to "read, write and execute" – enabling "runnable conference proceedings" where every paper can be independently validated and easily built upon.

Bibliography

[1] GENI RSpec. http://groups.geni.net/geni/wiki/GeniRspec.

[2] Linux Trace Toolkit - next generation. http://lttng.org/.

[3] Reproducing network research blog. http://reproducingnetworkresearch.wordpress.com/.

[4] Ripcord-Lite for POX: A simple network controller for OpenFlow-based data centers. https://github.com/brandonheller/riplpox.

[5] J. Ahrenholz, C. Danilov, T.R. Henderson, and J.H. Kim. CORE: A real-time network emulator. In *Military Communications Conference, MILCOM '08*, pages 1 –7. IEEE, 2008.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI '10*. USENIX, 2010.

[7] M. Ala-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM '08*, pages 63–74. ACM, 2008.

[8] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *NSDI '12*. USENIX, 2012.

[9] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *SIGCOMM '10*, pages 63–74. ACM, 2010.

[10] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. In *SIGCOMM '04*, pages 281–292. ACM, 2004.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03*, pages 164–177. ACM, 2003.

[12] Beacon: a Java-based OpenFlow control platform. `http://www.beaconcontroller.net/`.

[13] N. Beheshti, Y. Ganjali, R. Rajaduray, D. Blumenthal, and N. McKeown. Buffer sizing in all-optical packet switches. In *Optical Fiber Communication Conference*. Optical Society of America, 2006.

[14] S. Bhatia, M. Motiwala, W. Muhlbauer, Y. Mundada, V. Valancius, A. Bavier, N. Feamster, L. Peterson, and J. Rexford. Trellis: a platform for building flexible, fast virtual networks on commodity hardware. In *CoNEXT '08*, pages 72:1–72:6. ACM, 2008.

[15] Jonathan B Buckheit and David L Donoho. Wavelab and reproducible research. *Time*, 474:55–81, 1995.

[16] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, and Jennifer Rexford. A nice way to test openflow applications. *NSDI, Apr*, 2012.

[17] M. Carbone and L. Rizzo. Dummynet revisited. *SIGCOMM Computer Communication Review*, 40(2):12–20, 2010.

[18] M. Carson and D. Santay. NIST Net – a Linux-based network emulation tool. *SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.

[19] M. Casado, D. Erickson, I.A. Ganichev, R. Griffith, B. Heller, N. Mckeown, D. Moon, T. Koponen, S. Shenker, and K. Zarifis. Ripcord: A modular platform for data center networking. Technical report, Technical Report UCB/EECS-2010-93, EECS Department, University of California, Berkeley, 2010.

[20] M. Casado, M.J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM '07*, page 12. ACM, 2007.

[21] cgroups. http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[22] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[23] J. Claerbout. Electronic documents give reproducible research a new meaning. *Proc. 62nd Ann. Int. Meeting of the Soc. of Exploration Geophysics*, pages 601–604, 1992.

[24] DCTCP patches. http://www.stanford.edu/~alizade/Site/DCTCP.html.

[25] J. DeHart, F. Kuhns, J. Parwatikar, J. Turner, C. Wiseman, and K. Wong. The open network laboratory. In *SIGCSE '06 Technical Symposium on Computer Science Education*, pages 107–111. ACM, 2006.

[26] M. Devera. Hierarchical token bucket. *accessible at http://luxik. cdi. cz/˜ devik/qos/htb.*

[27] N. Dukkipati and N. McKeown. Why flow-completion time is the right metric for congestion control. *SIGCOMM Computer Communication Review*, 36(1):59–62, 2006.

[28] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An argument for increasing TCP's initial congestion window. *SIGCOMM Computer Communication Review*, 40(3):27–33, 2010.

[29] Amazon Elastic Compute Cloud. http://aws.amazon.com.

[30] Emulab - network emulation testbed. http://emulab.net/.

[31] Nick Feamster and Jennifer Rexford. Getting students hands dirty with clean-slate networking. *ACM SIGCOMM Education Workshop*, August 2011.

[32] Future Internet Research and Experimentation. http://www.ict-fireworks.eu/.

[33] S. Floyd and V. Jacobson. Random Early Detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[34] Global Environment for Network Innovations. http://www.geni.net/.

[35] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM '09*. ACM, 2009.

[36] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, and N. McKeown. NOX: Towards an operating system for networks. In *SIGCOMM CCR: Editorial note*, July 2008.

[37] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM '08*. ACM, 2008.

[38] D. Gupta, K. Yocum, M. McNett, A.C. Snoeren, A. Vahdat, and G.M. Voelker. To infinity and beyond: time warped network emulation. In *SOSP '05*, pages 1–2. ACM, 2005.

[39] Diwaker Gupta, Kashi V. Vishwanath, and Amin Vahdat. DieCast: Testing distributed systems with an accurate scale model. In *NSDI '08*, pages 407–421. USENIX, 2008.

[40] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-Serve: Load-balancing web traffic using OpenFlow. *SIGCOMM '09 Demo*, 2009.

[41] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264. ACM, 2012.

[42] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale virtualization in the Emulab network testbed. In *USENIX '08 Annual Technical Conference*, pages 113–128, Berkeley, CA, USA, 2008. USENIX.

[43] Vint cerf special @ons 2013. http://www.youtube.com/watch?v=Q0OL-GdfkZU.

[44] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazieres, Balaji Prabhakar, Changhoon Kim, and Windows Azure. Eyeq: practical network performance isolation for the multi-tenant cloud. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*, pages 8–8. USENIX Association, 2012.

[45] D. E. Knuth. Literate Programming. *The Computer Journal [Online]*, 27:97–111, 1984.

[46] T. Koponen, M. Casado, N. Gude, J. Stribling, Poutievski. L., M. Zhu, R. Ramanathan, Y Iwata, H. Inouye, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *OSDI '10*. USENIX, 2010.

[47] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *HotNets '10*, pages 19:1–19:6. ACM, 2010.

[48] J.W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and Jianying Luo. NetFPGA – an open platform for gigabit-rate network switching and routing. In *Microelectronic Systems Education, MSE '07*, pages 160–161. IEEE, 2007.

[49] lxc linux containers. http://lxc.sf.net.

[50] Mininet-discuss mailing list. https://mailman.stanford.edu/mailman/listinfo/mininet-discuss.

[51] Mininet tests code repository. https://github.com/mininet/mininet-tests.

[52] Mininet tracing code repository. https://github.com/mininet/mininet-tracing.

[53] Linux network emulation module. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.

[54] K. Nichols and V. Jacobson. Controlling queue delay. *Communications of the ACM*, 55(7):42–50, 2012.

[55] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM '09*, pages 39–50. ACM, 2009.

[56] noxrepo.org virtual testing environment. http://noxrepo.org/manual/vm_environment.html.

[57] The network simulator - ns-2. http://nsnam.isi.edu/nsnam/.

[58] The ns-3 network simulator. http://www.nsnam.org/.

[59] OFELIA: OpenFlow in Europe. http://www.fp7-ofelia.eu/.

[60] OpenFlow tutorial. http://www.openflow.org/wk/index.php/OpenFlowTutorial.

[61] OpenFlow Virtual Machine Simulation. http://www.openflow.org/wk/index.php/OpenFlowVMS.

[62] Open network labs. http://onlab.us/.

[63] The OpenFlow switch. http://www.openflow.org.

[64] OPNET modeler. http://www.opnet.com/solutions/network_rd/modeler.html.

[65] Open vSwitch: An open virtual switch. http://openvswitch.org/.

[66] M. Pizzonia and M. Rimondini. Netkit: easy emulation of complex networks on inexpensive hardware. In *International Conference on Testbeds and research infrastructures for the development of networks & communities*, TridentCom '08, pages 7:1–7:10, Brussels, Belgium, 2008. ICST.

[67] P. Prakash, A. Dixit, Y.C. Hu, and R. Kompella. The TCP outcast problem: Exposing unfairness in data center networks. In *NSDI '12*. USENIX, 2012.

[68] Ieee 1588: Standard for a precision clock synchronization protocol for networked measurement and control systems. http://www.nist.gov/el/isd/ieee/ieee1588.cfm.

[69] Ptpd: A ptp daemon. http://ptpd.sourceforge.net/.

[70] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *SIGCOMM Computer Communication Review*. ACM, 2011.

[71] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath TCP. In *NSDI '12*, pages 29–29. USENIX, 2012.

[72] K. Ramakrishnan and S. Floyd. A proposal to add explicit congestion notification (ECN) to IP. Technical report, RFC 2481, January 1999.

[73] Robert Ricci, Chris Alfeld, and Jay Lepreau. A solver for the network testbed mapping problem. *ACM SIGCOMM Computer Communication Review*, 33(2):65–81, 2003.

[74] Christian Esteve Rothenberg, Carlos A. B. Macapuna, and Alexander Wiesmaier. In-packet bloom filters: Design and networking applications. *CoRR*, abs/0908.3574, 2009.

[75] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *SIGCOMM Computer Communication Review*, 29(5):71–78, 1999.

[76] Service-Centric Architecture For Flexible Object Localization and Distribution. http://groups.geni.net/geni/wiki/SCAFFOLD/.

[77] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *NSDI '12*. USENIX, 2012.

[78] S. Soltesz, H. Pötzl, M.E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *ACM SIGOPS Operating Systems Review*, 41(3):275–287, 2007.

[79] Ion Stoica, Hui Zhang, and T. S. Eugene Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *SIGCOMM '97*, pages 249–262, New York, NY, USA, 1997. ACM.

[80] Z. Tan, K. Asanovic, and D. Patterson. An FPGA-based simulator for datacenter networks. In *Exascale Evaluation and Research Techniques Workshop (EXERT '10), at ASPLOS '10*. ACM, 2010.

[81] D. Thaler and C. Hopps. Multipath issues in unicast and multicast next-hop selection. Technical report, RFC 2991, November 2000.

[82] S. Tripathi, N. Droux, K. Belgaied, and S. Khare. Crossbow virtual wire: network in a box. In *LISA '09*, pages 4–4. USENIX, 2009.

[83] Paul Turner, Bharata B Rao, and Nikhil Rao. CPU bandwidth control for CFS. In *Linux Symposium '10*, pages 245–254, 2010.

[84] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI '02*, pages 271–284. USENIX, 2002.

[85] Patrick Vandewalle, Jelena Kovacevic, and Martin Vetterli. Reproducible research. *Computing in Science Engineering*, pages 5–7, 2008.

[86] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D.G. Andersen, G.R. Ganger, G.A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. *SIGCOMM Computer Communication Review*, 39(4):303–314, 2009.

[87] Virtual distirbuted ethernet. http://vde.sourceforge.net/.

[88] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[89] E. Weingärtner, F. Schmidt, H. Vom Lehn, T. Heer, and K. Wehrle. Slicetime: A platform for scalable and accurate network emulation. In *NSDI '11*, volume 3. ACM, 2011.

[90] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 8–8. USENIX Association, 2011.

[91] M. Zec and M. Mikuc. Operating system support for integrated network emulation in IMUNES. In *Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.