A Network in a Laptop: Rapid Prototyping for Software-Defined Networks

Bob Lantz
Network Innovations Lab
DOCOMO USA Labs
Palo Alto, CA, USA
rlantz@cs.stanford.edu

Brandon Heller
Dept. of Computer Science,
Stanford University
Stanford, CA, USA
brandonh@stanford.edu

Nick McKeown
Dept. of Electrical Engineering
and Computer Science,
Stanford University
Stanford, CA, USA
nickm@stanford.edu

ABSTRACT

Mininet is a system for rapidly prototyping large networks on the constrained resources of a single laptop. The lightweight approach of using OS-level virtualization features, including processes and network namespaces, allows it to scale to hundreds of nodes. Experiences with our initial implementation suggest that the ability to run, poke, and debug in real time represents a qualitative change in workflow. We share supporting case studies culled from over 100 users, at 18 institutions, who have developed Software-Defined Networks (SDN). Ultimately, we think the greatest value of Mininet will be supporting collaborative network research, by enabling self-contained SDN prototypes which anyone with a PC can download, run, evaluate, explore, tweak, and build upon.

Categories and Subject Descriptors

C.2.1 [**Computer Systems Organization**]: Computer-Communication Networks—*Network communications*; B.4.4 [**Performance Analysis and Design Aids**]: Simulation

General Terms

Design, Experimentation, Verification

Keywords

Rapid prototyping, software defined networking, OpenFlow, emulation, virtualization

1. INTRODUCTION

Inspiration hits late one night and you arrive at a world-changing idea: a new network architecture, address scheme, mobility protocol, or a feature to add to a router. With a paper deadline approaching, you have a laptop and three months. What prototyping environment should you use to evaluate your idea? With this question in mind, we set out to create a *prototyping workflow* with the following attributes:

Flexible: new topologies and new functionality should be defined in software, using familiar languages and operating systems.

Deployable: deploying a functionally correct prototype on hardware-based networks and testbeds should require no changes to code or configuration.

Interactive: managing and running the network should occur in real time, as if interacting with a real network.

Scalable: the prototyping environment should scale to networks with hundreds or thousands of switches on only a laptop.

Realistic: prototype behavior should represent real behavior with a high degree of confidence; for example, applications and protocol stacks should be usable without modification.

Share-able: self-contained prototypes should be easily shared with collaborators, who can then run and modify our experiments.

The currently available prototyping environments have their pros and cons. Special-purpose testbeds are expensive and beyond the reach of most researchers. Simulators, such as ns-2 [14] or Opnet [19], are appealing because they can run on a laptop, but they lack realism: the code created in the simulator is not the same code that would be deployed in the real network, and they are not interactive. At first glance, a network of virtual machines (VMs) is appealing. With a VM per switch/router, and a VM per host, realistic topologies can easily be stitched together using virtual interfaces [13, 17, 15]. Our experience is that VMs are too heavyweight: the memory overhead for each VM limits the scale to just a handful of switches and hosts. We want something more scalable.

There are efforts underway to build programmable testbeds (e.g. Emulab [9], VINI [1], GENI [6], FIRE [5]) supporting realistic user traffic, at scale, and with interactive behavior. Our approach is complementary to these systems. We seek a *local* environment that allows us to quickly implement a functionally correct, well-understood prototype, then directly move it onto shared global infrastructure.

Mininet – the new prototyping environment described in this paper – supports this workflow by using lightweight virtualization. Users can implement a new network feature or entirely new architecture, test it on large topologies with application traffic, and then deploy the exact same code and test scripts into a real production network. Mininet runs surprisingly well on a single laptop by leveraging Linux features (processes and virtual Ethernet pairs in network namespaces) to launch networks with gigabits of bandwidth and hundreds of nodes (switches, hosts, and controllers). The entire network can be packaged as a VM, so that others can download, run, examine and modify it.

Mininet is far from perfect – performance fidelity and multi-machine support could be improved – but these are limitations of the implementation, not the approach. Other tools also use lightweight virtualization [9, 23] (see Section 7 for a comparison), but Mininet differs in its support for rapidly prototyping Software-Defined Networks, a use case we focus on throughout this paper.

2. SOFTWARE-DEFINED NETWORKS

In an SDN, the control plane (or "network OS") is separated from the forwarding plane. Typically, the network OS (e.g NOX [8], ONIX [10], or Beacon [2]) observes and controls the entire network state from a central vantage point, hosting features such as routing protocols, access control, network virtualization, energy management, and new prototype features. The network OS controls the forwarding plane via a narrow, vendor-agnostic interface, such as OpenFlow [18], which defines 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 main consequence of SDN is that the functionality of the network is defined after it has been deployed, under the control of the network owner and operator. New features can be added in software, without modifying the switches, allowing the behavior to evolve at software speeds, rather than at standards-

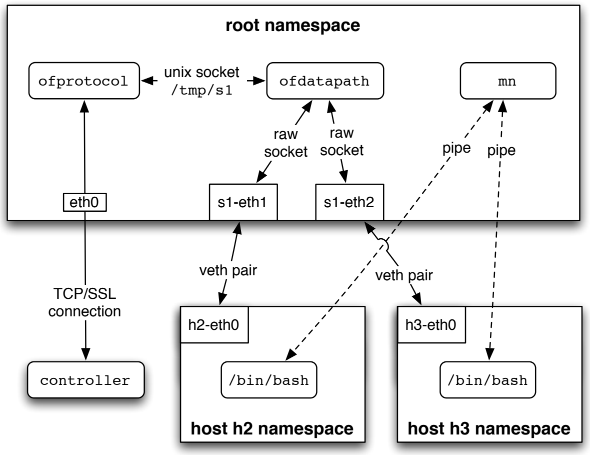


Figure 1: Mininet creates a virtual network by placing host processes in network namespaces and connecting them with virtual Ethernet (veth) pairs. In this example, they connect to a user-space OpenFlow switch.

body speed. SDN enables new approaches to state management (anywhere on the spectrum from centralized to distributed) and new uses of packet headers (fields with layer-specific processing become a layer-less sea of bits). Examples of software-defined networks include 4D [7], Ethane [4], PortLand [12], and FlowVisor [22]).

These examples hint at the potential of SDN, but we feel that a rapid prototyping workflow is a key to unlocking the full potential of software-defined networking. The variety of systems prototyped on Mininet supports this assertion, and we describe several such case studies in Section 6.

3. MININET WORKFLOW

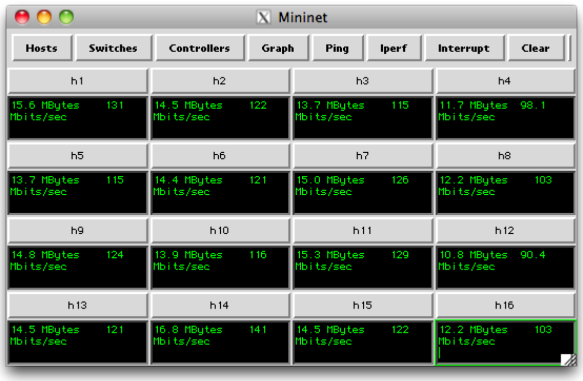
By combining lightweight virtualization with an extensible CLI and API, Mininet provides a rapid prototyping workflow to create, interact with, customize and share a software-defined network, as well as a smooth path to running on real hardware.

3.1 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
```

starts a network of OpenFlow switches. In this example, Open vSwitch [20] kernel switches are connected in a tree topology of depth 2 and fanout 8 (i.e. 9 switches and 64 hosts), under the control of NOX, followed by the `pingAll` test to check connectivity between every pair of nodes. To create this network, Mininet emulates links, hosts, switches, and controllers. Mininet uses the lightweight virtualization mechanisms built into the Linux OS: processes running in network namespaces, and virtual Ethernet pairs.



```
h1
15.6 MBytes    131
Mbits/sec
h2
14.5 MBytes    122
Mbits/sec
h3
13.7 MBytes    115
Mbits/sec
h4
11.7 MBytes  98.1
Mbits/sec
h5
13.7 MBytes    115
Mbits/sec
h6
14.4 MBytes    121
Mbits/sec
h7
15.0 MBytes    126
Mbits/sec
h8
12.2 MBytes    103
Mbits/sec
h9
14.8 MBytes    124
Mbits/sec
h10
13.9 MBytes    116
Mbits/sec
h11
15.3 MBytes    129
Mbits/sec
h12
10.8 MBytes  90.4
Mbits/sec
h13
14.5 MBytes    121
Mbits/sec
h14
16.8 MBytes    141
Mbits/sec
h15
14.5 MBytes    122
Mbits/sec
h16
12.2 MBytes    103
Mbits/sec
```

Figure 2: 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.

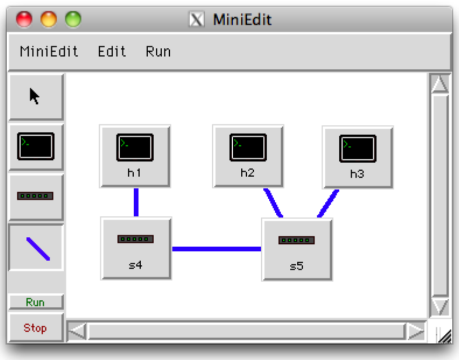


Figure 3: MiniEdit is a simple graphical network editor that uses Mininet to turn a graph into a live network when the Run button is pressed; clicking a node opens up a terminal window for that node.

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 [11] are containers for network state. They provide processes (and groups of processes) with exclusive ownership of interfaces, ports, and 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 OpenFlow switches provide the same packet delivery semantics that would be provided by a hardware switch. Both user-space and kernel-space switches are available.

Controllers: 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 1 illustrates the components and connections in a two-host network created with Mininet.

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 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

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

graphical applications, two of which are shown in figures 2 and 3. 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 the authors.

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.

3.5 Running on Hardware

To successfully port to hardware on the first try, every Mininet-emulated component must act in the same way as its corresponding physical one. The virtual topology should match the physical one; virtual Ethernet pairs must be replaced by link-level Ethernet connectivity. Hosts emulated as processes should be replaced by hosts with their own OS image. In addition, each emulated OpenFlow switch should be replaced by a physical one configured to point to the controller. However, the controller does not need to change. When Mininet is running, the controller "sees" a physical network of switches, made possible by an interface with well-defined state semantics. With proxy objects representing OpenFlow datapaths on physical switches and SSH servers on physical hosts, the CLI enables interaction with the network in the same way as before, with unmodified test scripts.

4. SCALABILITY

Lightweight virtualization is the key to scaling to hundreds of nodes while preserving interactive performance. In this section, we measure overall topology creation times, available bandwidth, and microbenchmarks for individual operations.

Table 2 shows the time required to create a variety of topologies with Mininet. Larger topologies which cannot fit in memory with 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).

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(4^4)	256	85	168.4	83.9	233
Tree(16^2)	256	17	139.8	39.3	212
Tree(32^2)	1024	33	817.8	163.6	492

Table 2: 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.

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: 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.

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 small 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 2, only the smallest one would fit in the memory of a typical laptop if system virtualization were used. Mininet also provides a usable amount of bandwidth, as shown in Table 1: 2-3 Gbps through one switch, or more than 10 Gbps aggregate internal bandwidth through a chain of 100 switches.

Table 3 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. As we gain a better understanding of Mininet's resource usage and interaction with the Linux kernel, we hope to further improve its performance and contribute optimizations back to the kernel as well as Open vSwitch.

5. LIMITATIONS

The most significant limitation of Mininet today 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 host that is ready to send a packet will be scheduled promptly, or that all switches will forward at the same rate. In addition, software forwarding may not match hardware. $O(n)$ 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.

To enforce bandwidth limits and quality of service on a link, the linux traffic control program (`tc`) may be used. Linux CPU containers and scheduler priorities offer additional options for improving fairness. Mininet currently runs on a single machine and emulates only wired links; as with performance fidelity, these limitations do not seem fundamental, and we expect to address them later.

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. We feel that these losses are a reasonable tradeoff for the ability to try ideas at greater scale.

6. CASE STUDIES

Mininet has been used by over 100 researchers in more than 18 institutions, including Princeton, Berkeley, Purdue, ICSI, UMass, University of Alabama Huntsville, NEC, NASA, Deutsche Telekom Labs, Stanford, and a startup company, as well as seven universities in Brazil. The use cases roughly divide into prototyping, optimization, demos, tutorials, and regression suites. For each use, we describe a project, a challenge it faced, and how Mininet helped.

Prototyping: Ripcord is a modular and extensible platform for creating scale-out data center networks [3]. The main challenge was developing a common codebase — without hardware — across multiple geographic locations. A second challenge was regression testing: every change needed to be tested against many topologies. Mininet enabled concurrent development, plus easy regression testing for new topologies. Better still, the code was directly portable to hardware: when a hardware testbed became available a week before a deadline, the code and test scripts transferred without modification, allowing the paper to include hardware results.

Optimization: The OpenFlow controller NOX builds a topology database by sending periodic LLDP packet broadcasts out each switch port [8]. 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. With Mininet, we could try many topologies to reproduce the error, experiment with new topology discovery algorithms, and validate a fix.

Tutorials: In OpenFlow hands-on tutorials, attendees turn a simple hub controller into a flow-accelerated Ethernet switch, giving them experience with OpenFlow debugging tools and writing controller code. Initially, the tutorial used optimized QEMU VM instances for switches and hosts, with VDE connecting them, distributed as a VM. It was too slow to be usable. After re-implementing the tutorial on Mininet [16], it started up minutes faster. An unexpected bonus was that attendees could run the tutorial on small netbook computers with little memory.

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 is being used to create push-button regression suites to test prototype network architectures. One example is SCAFFOLD [21], 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. Another is a higher-level API and runtime environment for OpenFlow-enabled networks that allows programmers to describe network behavior in a declarative manner on top of Python.

7. RELATED WORK

Mininet builds upon recent work which uses OS-level virtualization for network emulation. For brevity, we leave out related work on OpenFlow, hardware testbeds, and simulators, and instead focus on lightweight virtualization techniques.

IMUNES [23] added virtual Ethernet interfaces and a feature similar to network namespaces into the BSD kernel. The IMUNES work asked the right question: "How much virtualization do you really need?" Lightweight virtualization enables rapid prototyping, but in itself does not provide a path to hardware deployment or a means for distribution and sharing.

EMULAB [9] took another OS-level virtualization technology, FreeBSD jails, and modified it to allow multiple virtual interfaces per process group, similar to network namespaces. Jails provide coarser-grained control than Linux containers over which aspects of virtualization to use.

Unlike Mininet, EMULAB's virtual nodes attempt to carefully reproduce the full environment of EMULAB hardware nodes, allowing for identical system images to be used both in both. EMULAB virtual nodes represent a different design point, emulating 10 or more nodes on a single PC with close fidelity; Mininet gives up fidelity to emulate 100 or more nodes on a laptop. Although EMULAB doesn't currently support OpenFlow, the ProtoGENI evolution of EMULAB will include hardware OpenFlow switches.

8. DISCUSSION

As the case studies in Section 6 show, Mininet can yield a more efficient use of time and resources compared to other workflows. It provides a local environment for network innovation that complements shared global infrastructure [6], with interactive prototyping, scalability, a seamless path to hardware deployment and straightforward sharing and collaboration. Combined with software-defined networking, we think it yields an easier and faster path to real systems, in three phases:

Prototyping: Anyone (student, researcher, network administrator, etc.) with a laptop may use Mininet to rapidly prototype an SDN idea. Quick startup time and low overhead facilitates exploring a design space and building a system of interesting scale that can be run in emulation on modest hardware. Multiple researchers can share scripts, configurations and topologies, and work concurrently without interference.

Deployment: Once an idea works on Mininet, it can be deployed on research or production networks for validation, measurement, and general use. Mininet facilitates this transfer by leveraging software-defined networking (notably OpenFlow) and preserving switch, application, and script semantics between emulation and hardware. Hardware deployment can be on a locally available cluster of PCs and switches, or a shared research infrastructure such as GENI.

Sharing: A design that runs on Mininet can easily be shrink-wrapped in a VM image and redistributed. Mininet leverages lightweight process virtualization internally, but using system virtualization, an entire Mininet-based system can be packaged and distributed. We ship a VM image that includes Mininet pre-installed, along with all the pieces required to create and run a new SDN design without additional configuration or installation.

Wrapping a Mininet-based design in a VM creates a "network appliance" that can be distributed over the internet. Instead of relying solely on a conference paper, a written specification, or even a recorded video, one can download and run a living, breathing example of a new networked system. We look forward to creating, and working with others[1] to create, a library of Mininet-based software-defined networks that anyone can download, examine, run, modify and build upon in exciting and unexpected ways.

9. REFERENCES

[1] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI veritas: realistic and controlled network experimentation. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, page 14. ACM, 2006.

[2] Beacon: a Java-based OpenFlow Control Platform. `http://www.beaconcontroller.net/`.

[3] 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 UCB/EECS-2010-93, EECS Department, University of California, Berkeley, Jun 2010.

[4] M. Casado, M. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, page 12. ACM, 2007.

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

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

[7] A. Greenberg, G. Hjalmtysson, D. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *ACM SIGCOMM Computer Communication Review*, 35(5):54, 2005.

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

[9] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX 2008 Annual Technical Conference*, pages 113–128. USENIX, 2008.

[10] T. Koponen, M. Casado, N. Gude, J. Stribling, P. 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 *Operating Systems Design and Implementation*. USENIX, 2010.

[11] lxc linux containers. `http://lxc.sf.net`.

[12] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009*, pages 39–50. ACM, 2009.

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

[14] The network simulator - ns-2. `http://www.isi.edu/nsnam/ns/`.

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

[16] Openflow tutorial. `http://www.openflowswitch.org/wk/index.php/OpenFlowTutorial`.

[17] Openflow virtual machine simulation. `http://www.openflowswitch.org/wk/index.php/OpenFlowVMS`.

[18] The openflow switch. `http://www.openflowswitch.org`.

[19] Opnet modeler. `http://www.opnet.com/solutions/network_rd/modeler.html`.

[20] Open vSwitch: An Open Virtual Switch. `http://openvswitch.org/`.

[21] Service-centric architecture for flexible object localization and distribution. `http://sns.cs.princeton.edu/projects/scaffold/`.

[22] R. Sherwood et al. Carving research slices out of your production networks with OpenFlow. *ACM SIGCOMM Computer Communication Review*, 40(1):129–130, 2010.

[23] M. Zec and M. Mikuc. Operating system support for integrated network emulation in imunes. In *Proc. of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[1]Mininet is available at `openflowswitch.org/mininet`