Section 4: Week 8: Experiment with Mininet

Nate Bachmeier

TIM-7010: Computer Networking and Mobile Computing

September 18th, 2019

North Central University

# Section I: Experimentation with Mininet

Software-defined networking represents the next evolutionary step in device communication (Eissa, Bozed, & Younis, 2019). It enforces clear separation of duties between the control plane and the data plane.

The control plane is responsible for making decisions around flow control and is extensible through modules registering to network events (Azodolmolky, 2013). For instance, a simple learning switch can monitor for incoming packets and build a map of sender Media Access Control (MAC) to the physical switch port. Without this map, incoming traffic needs to be broadcasted to all physical switch ports, to ensure it arrives at the correct location (Dordal, 2005).

The data plane consists of simple packet forwarding switches and possesses the least amount of logic possible (Azodolmolky, 2013). The de-facto protocol used by these switches is called OpenFlow. OpenFlow leverages flow tables to perform match operations and then perform one or more actions on the incoming traffic. For instance, the flow table entry might say drop any traffic for network port 1234. Another entry could forward virtual endpoints, such as Named Data Network (NDN) resources, to a contextually dependent Internet Protocol (IP) address (Lei, Zong, Zhu, & Zhang, 2018).

## Application through Mininet

To explore these concepts, several researchers use a tool called mininet (Santos de Oliveira & Shinoda, 2014). The utility provides a scriptable interface for defining detailed network topologies and then materializing them as lightweight virtualized endpoints.

The materialization uses Linux’s Network Namespace feature to associate different routing tables with each launched application instances. For example, mininet could start two copies of the Secure Shell Daemon (sshd) and bind them to addresses 10.10.10.10 and 20.20.20.20 (Lantz, 2016). During binding, the kernel will create separate virtual network adapters and treat them as if they were different physical machines. This behavior enables technology practitioners to experiment on large topologies using a single server.

## Defining Topologies

Topologies can be significantly more complicated by combining multiple virtual switches to represent different network segments (Pal, Veena, Rustagi, & Murthy, 2014). Internally the mininet.topo.MultiGraph supports adding arbitrary nodes and edges. A node represents a virtual endpoint and uses a Python class to define its implementation. The edges expresse the routing table configuration.

For example, a Python class can derive from the mininet.node.Node base class and define an initialization script to become a Linux router (Lintz, 2016). Then one or more routers can be added into the topology, each with custom local configuration.

Links from virtual switches can connect to the router and other topology nodes. Each of these links can contain configuration parameters, such as latency or fault injection. These capabilities allow researchers to verify reliability scenarios that are otherwise difficult to reproduce.

## Customizing the Controller

The controller is responsible for making all flow decisions as the data plane is simple packet forwarding devices. Dordal demonstrates this by constructing a rectangular looped topology, then launching it without a controller. After issuing mininet’s pingall test command, an infinite loop occurs.

The standard mininet virtual machine image contains a POX controller. The topologies virtual switches will automatically detect and connect to once it has started. WireShark can watch the loopback adapter and report these traffic patterns (see Figure 2).

# Section II: Analyze Existing Research

## Using Mininet for Prototyping SDN (2014)

Santos de Oliveira and Shinoda describe the need for an economical and efficient mechanism to test large scale software-defined networks. Without these capabilities, budgetary and maintainability costs will limit innovation.

Consider the investment required to provision 256 Linux servers, each with custom routing tables. Then contrast that scenario to materializing topology with 256 process nodes. The first could easily take months of careful setup and troubleshooting. Meanwhile, mininet can emerge that same environment within a few minutes.

Their research shows that mininet scales linearly until roughly 128 nodes, then grows exponentially. A Hyper-v based virtual machine with eight cores and four gigabytes of memory demonstrated that the start and stop time is reasonably linear. Figure 1 shows the timed values for the command template: sudo mn --topo single,256 --test=build.

In the grand scheme, this is a minor flaw in their research results. Perhaps the differences come from five years of improvements to both mininet, and the Linux operating system. Assuming that is the cause, it only strengthens their point that this is a critical research area and is continuously improving.

## Implementing Simplified Custom Topologies in Mininet (2014)

Pal et al. explored the capabilities of mininet when generating more elaborate network topologies. Real-world network environments use multiple hierarchical levels and enforce segmentations.

Consider a typical enterprise with branch offices in Seattle, Dallas, and New York. They would likely have connectivity between each hub, but not every device can see each other. For example, a person in the Seattle office should not be able to send documents to the New York printer. To prevent these erroneous decisions, network administrators define isolation rings.

If that same business wants to test the impact of changes to their operational design, then using tooling like mininet simplifies that testing process. To improve the reproducibility of those results, the organization can write Python scripts to create their simulated world.

Original Python code was authored to reproduce their “Custom Topology with Single IP Network Address” example (see figure 3). Creating the template and then verifying its behavior were very straight forward.

Next, a section of the “Custom Topology with Multiple IP Network Addresses” was recreated (see figure 4). To handle the multiple network addresses, I referenced Lintz’s Linux Router example. Afterward, the pingall validation test was able to confirm connectivity between all nodes.

The performance of the single-address and multi-address network was measured, and similar to Pal et al. showed trivial time differences. The multi-address was slightly slower most likely due to the extra hop across the router node. A more significant difference appeared after replacing the kernel-mode virtual switch with user-mode implementation. This change decreased the throughput from 45GB/s to around 850MB/s.

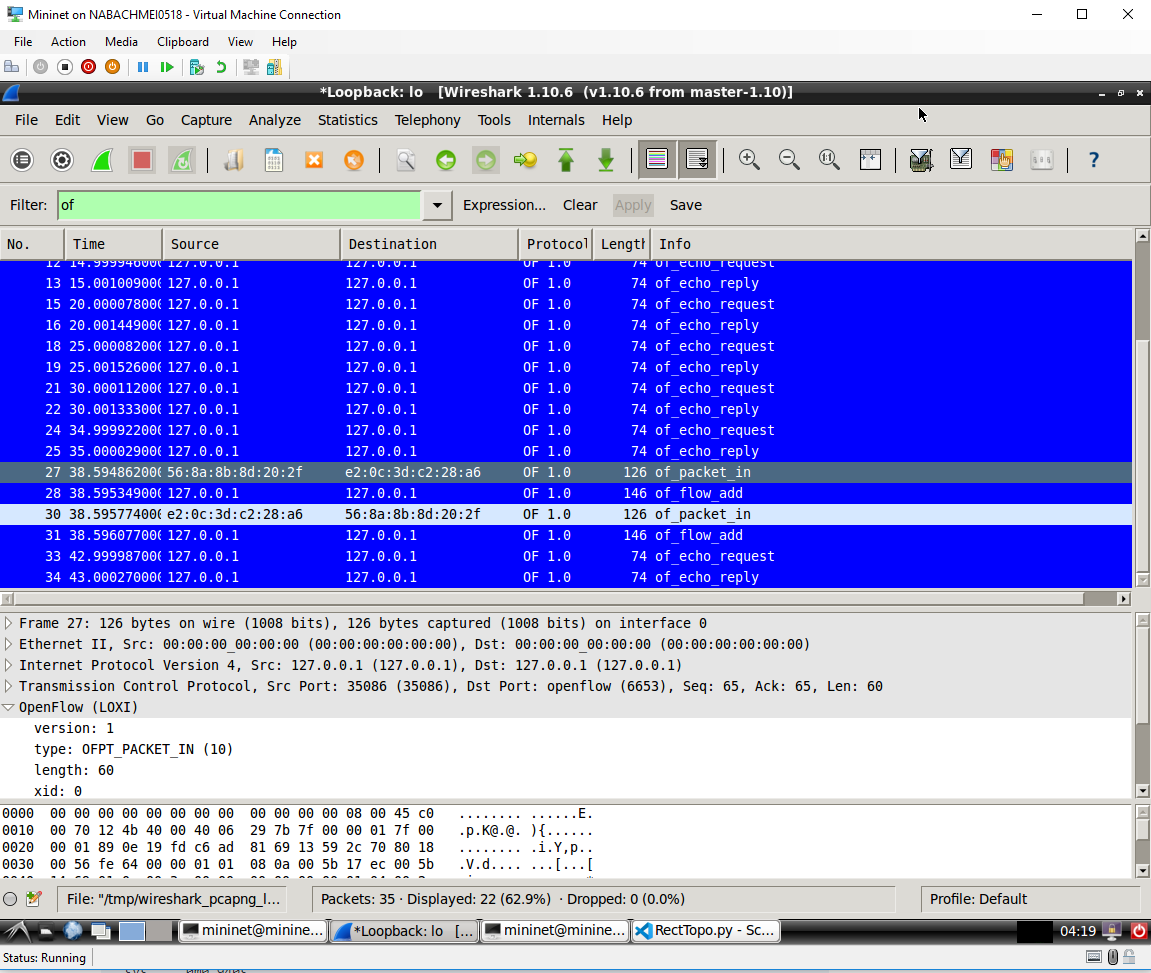
Given

# Conclusions

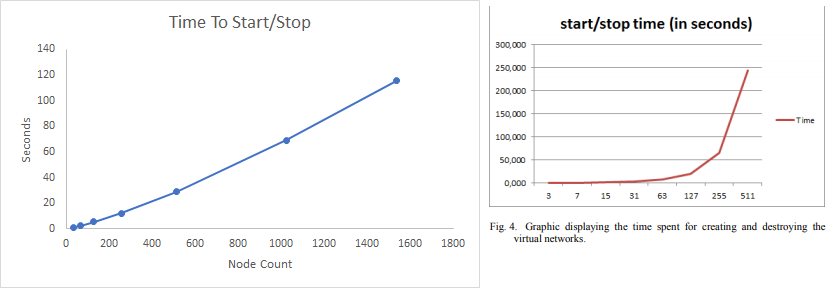
Software-defined networking

# Appendix and Figures

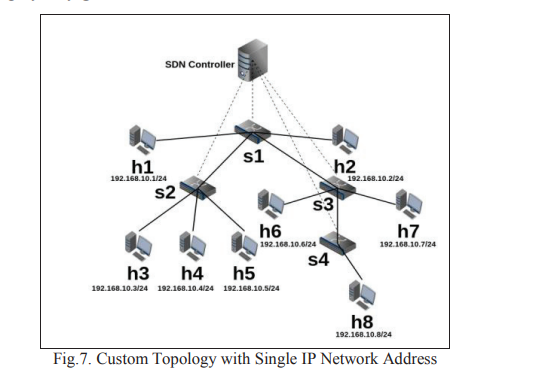
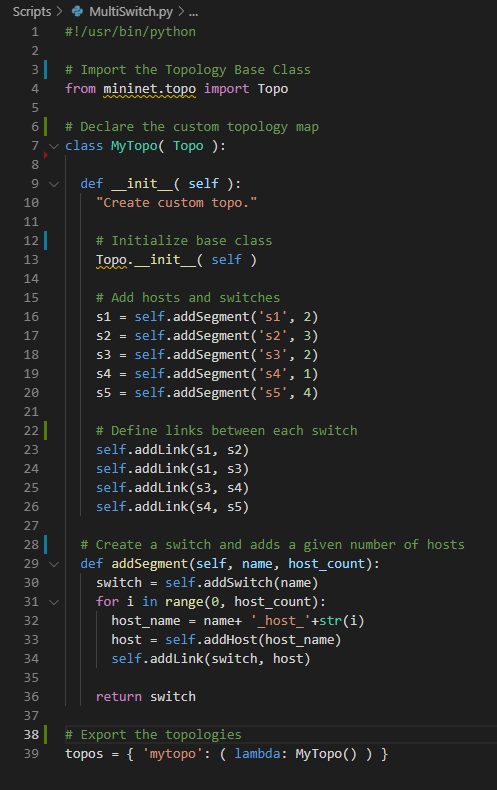
## Figure 1: Wire Share Traffic



## Figure 2: Start/Stop Time



## Figure 3: Reproduced Topology



## Figure 4: Reproducing Multi-Address Topology

