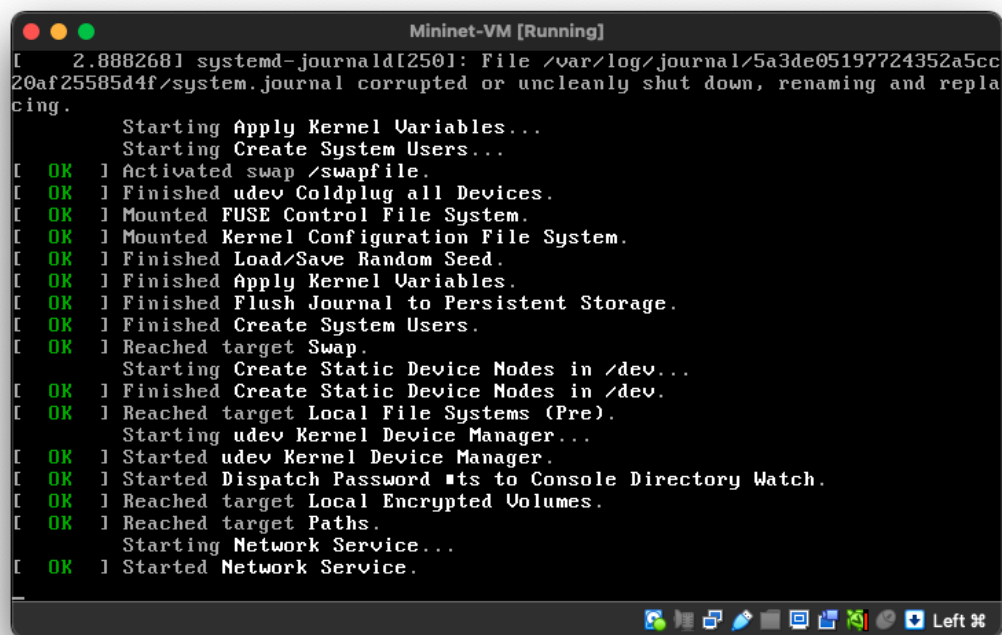


An Intelligent Transportation System (ITS) Approach to Teletraffic Engineering

NORMAN ANDERSON, University of Victoria, Canada



```
Mininet-VM [Running]
[ 2.888268] systemd-journald[250]: File /var/log/journal/5a3de05197724352a5cc20af25585d4f/system.journal corrupted or uncleanly shut down, renaming and replacing.
Starting Apply Kernel Variables...
Starting Create System Users...
[ OK ] Activated swap /swapfile.
[ OK ] Finished udev Coldplug all Devices.
[ OK ] Mounted FUSE Control File System.
[ OK ] Mounted Kernel Configuration File System.
[ OK ] Finished Load/Save Random Seed.
[ OK ] Finished Apply Kernel Variables.
[ OK ] Finished Flush Journal to Persistent Storage.
[ OK ] Finished Create System Users.
[ OK ] Reached target Swap.
Starting Create Static Device Nodes in /dev...
[ OK ] Finished Create Static Device Nodes in /dev.
[ OK ] Reached target Local File Systems (Pre).
Starting udev Kernel Device Manager...
[ OK ] Started udev Kernel Device Manager.
[ OK ] Started Dispatch Password #ts to Console Directory Watch.
[ OK ] Reached target Local Encrypted Volumes.
[ OK ] Reached target Paths.
Starting Network Service...
[ OK ] Started Network Service.
```

Fig. 1. Launching Mininet within VirtualBox.

A partial replication study in which I attempt to validate and extend previous work in Software Defined Networks (SDN) published in 2016 by Liu et al. entitled *Leveraging software-defined networking for security policy enforcement*, recreating the authors' proposed network topology in Mininet using Ryu to communicate via the OpenFlow 1.3 protocol. I evaluate the topology's performance through a basic experiment measured using the Linux ping utility. Also included is a tutorial of how to set up the study environment. Liu et al.'s research was selected from a group of several promising options, I weigh the pros and cons of each possible implementation. I also explain the initial ideas behind the creation of this project. Lastly, I explore future directions including extensions involving game theory and ITS approaches to teletraffic engineering.

CCS Concepts: • **Networks** → **Programmable networks**; *Topology analysis and generation*; • **General and reference** → *Validation*.

Additional Key Words and Phrases: Software defined networks, Mininet, Ryu, OpenFlow, network topology, replication study

CONTENTS

Abstract	1
----------	---

Author's Contact Information: Norman Anderson, normananderson@uvic.ca, University of Victoria, Victoria, Canada.

Contents	1
1 Introduction	2
2 Project directions, planning, and goals	3
2.1 Initial goals and deliverables	3
2.2 Journey to a protocol selection	4
2.3 The remainder of this paper	4
3 Related Work	4
3.1 SDN	5
3.2 Google’s move to SDN	5
3.3 GameTE: A Game-Theoretic Distributed Traffic Engineering in Trustless Multi-Domain SDN	6
3.4 Leveraging software-defined networking for security policy enforcement	6
4 Environment	7
4.1 Mininet	7
4.2 Ryu	8
5 Experiment	8
5.1 Topologies	8
5.2 Performance capture	13
6 Results	13
7 Future Work	14
8 Conclusion	15
Acknowledgments	16
References	17
A Topology 1 in Mininet	17
B Topology 2 in Mininet	18

1 Introduction

Software-Defined Networking (SDN) is a networking paradigm aiming to separate a network’s control plane from its data plane and centralize control logic into a single place. Traditionally the network control plane (decisions made regarding which route packets should take to their destination) and the network data plane (actions taken to forward packets along this route) were indistinguishable: that is, the logic was decentralized, distributed among every router and switch within the network topology [13]. SDN, and in particular OpenFlow, the protocol which pioneered this approach, breaks these traditions. This shift was motivated by challenging “the limitations of current network infrastructures” [12]. The hope is that one day SDN will dramatically reduce the cost of things like deploying new routing algorithms.

Designing Intelligent Traffic Systems (ITS) is a mostly unrelated field of computer science, although it involves some networking. However, in this paper I hope to uncover some cases where it relates to the field of communication networks. The main potential link I will explore is through traffic engineering (TE), or “the process of controlling how traffic flows through one’s network” which aims to get the most out of a network’s resources to maximize its performance [23].

Teletraffic engineering is the discipline concerned with applying queueing theory to telecommunication network traffic engineering [2]. Despite fundamental differences, I believe teletraffic engineering practices can be applied to both communication networks and ITS. At a surface level, one can observe similarities between routing packets and navigating cars, but there are also deeper connections. For instance, research has previously applied game theory to both SDN [17]. However, I would argue that the primary intersection between ITS and traffic theory is in name only. Therefore my goal is to also invite the reader to consider where traffic theory itself applies to the situations throughout this paper.

2 Project directions, planning, and goals

When beginning to ideate on this project's direction, I had an ambitious plan. I knew that I wanted to combine several unrelated areas of computer science, to see if I could draw analogies between them. Two areas I have already introduced: SDN and teletraffic engineering. They ended up in the final version of this project. However, a third concept was scrapped. A Wireless Ad-Hoc Network (WANET) is a network which "consists of autonomous or mobile nodes which communicate with each other without a centralized control or assistance" [21]. Every node acts as both a router and a receiver [21]. WANETs are composed of autonomous nodes, and in contrast to software-defined networking, communicate with decentralized (rather than centralized) control logic [21]. They are closely linked to ITS, because vehicles need to communicate with each other in traffic. That is to say, WANETs fit in well with my project, but I decided to drop them from my project because it was too time consuming to learn about TE, SDN, and WANETs at the same time. However I wanted to mention that they fit very well with the other topics of this project: game theory has even also seen prior application within WANETs [20]!

The WANET component of this project was not the only part to change. I also scaled back the relevance ITS had to my project, and in this paper I only discuss superficial relationships to TE, because I did not have time to explore the connection in detail. Lastly, I originally wanted to implement a networks protocol, but I quickly decided that an algorithm would be a scope more appropriate for this course. Even still, conducting a small-scale replication was far more difficult than I anticipated.

2.1 Initial goals and deliverables

My project proposal asked the following: *How can teletraffic engineering approaches improve strategies for SDN or WANETs?* Armed with a direction, my next steps were outlined as follows: (1) select either a WANET or a SDN protocol which could benefit from either Intelligent Traffic Systems (ITS) and/or teletraffic engineering approaches, (2) conduct a small-scale replication, (3) propose improvements, and (4) evaluate the impact of the proposed improvements.

Like most parts of this project, this plan was only loosely adhered to. Eventually, I found myself committed to a plan to make two research contributions (which to be clear are not research questions! I do not want to be accused of HARKing [10]). They are as follows:

- (1) Answering how feasible is it to integrate ITS and/or teletraffic engineering approaches with SDN
- (2) A replication of the 2016 paper by Liu et al. with some additional contributions: an emulated version of the proposed topology.

Before I can talk about how successful I was in making these contributions, I want to explain how I arrived at my conclusion to make them.

Week	Scheduled Deliverables
W5	Proposal (Feb 7); narrow project ideas and scope
W6	Explore background literature in depth: [1], [4], [5]
W7	Biweekly update (Feb 21); present results of informal lit survey
W8	Dive deep into strategies of related work, select protocol to augment with my own ideas
W9	Midterm update (Mar 7); present progress on my improvements to the selected protocol
W10	Select simulation tools for analysis; attempt selected replication of paper results
W11	Biweekly update (Mar 21); present progress on simulation results of my improvements
W12	Complete queueing system analysis (or use another formal method)
W13	Final presentation (Apr 4); prepare rough materials for peers
W14	Final report (Apr 11); finishing touches

Table 1. Original tentative project schedule and deliverables

2.2 Journey to a protocol selection

I spent approximately the first two weeks of my project (covered to biweekly update 1) reading literature to determine possible paper candidates which had a concrete implementation. At the time I believed I would replicate some kind of networking protocol, and I also defined several considerations for selecting what I would replicate:

- (1) Feasibility of integrating ITS and/or teletraffic engineering approaches into the implementation.
- (2) Perceived difficulty of conducting a small-scale replication.
- (3) Relevance to WANETs or SDN.

During my research I quickly found that I could not find previous work which applied principles of ITS to general communication networks. The lack of papers in the area led me to de-emphasize the relevance of the first point during the remainder of my protocol selection.

However, no matter where I looked, I could not find a protocol that seemed simple to implement. As I looked further into SDN in the past two weeks, implementing an algorithm rather than a protocol solidified as a clear choice. In fact, I could not justify implementing a protocol at all. They differed by orders of magnitude in complexity. Algorithms would be considerably easier to implement. As for usefulness: in SDN, protocols such as OpenFlow already exist to communicate routing decisions. I could have designed a protocol to make routing decisions, but if the controller is already centralized, what purpose would one serve? I resolved to select an algorithm.

2.3 The remainder of this paper

The remainder of this paper explores four deliverables for this project: (1) a selected protocol, (2) small-scale replication of existing work, (3) proposed improvements to existing work, (4) evaluation of impacts to existing work.

Section 2 weighs pros and cons of various potential papers concerning WANETs or SDN which I considered replicating, including the relevant literature explored which lead to each decision. I also explore the necessary details required to understand my replication. Section 3 discusses the technical details of implementing the selected system. Lastly in Section 4 I evaluate the improved system against the existing work, comparing systems through emulation.

3 Related Work

What follows is my process of selecting the work that I eventually decided to replicate. Since I had decided early on to work with SDN, the work left was to select a paper that I could replicate within the constraints of my topic. That led

me to believe that I might be able to deploy a small portion of a Tailscale network¹ over Mininet (which I discuss in detail in a future section). In fact, I believed it might be possible to run my own DERP (Designated Encrypted Relay for Packets) server² over Mininet.

Unfortunately, both protocols are quite complex and I did not find information on how to deploy a DERP server within a virtualized environment. I also could not determine if the resource constraints posed by a VM would make it possible to run DERP in the emulated environment. So in the end I concluded that the technical challenges were too great for me to overcome.

After trying and failing to find a feasible way to use these technologies in my project, I concluding that learning more information about SDN would help clarify what existing project would make most sense for me to replicate. So I started to learn what SDN was all about.

3.1 SDN

One of the primary motivating factors of SDN is to avoid misconfiguration [6]. Rohrer, Assistant Professor of Computer Science at the Naval Postgraduate School, cites that “[t]here have been a number of surveys of network operators have indicated that that misconfiguration is the number one cause of network outages across the Internet. So if we can decrease the likelihood that these misconfigurations will occur, we can go a long way to increasing the reliability of the Internet.” He goes on to note that SDN also allows for increased traffic management, such as differentiating between traffic with different latency and bandwidth requirements [6].

As discussed previously in the introduction, SDN was largely motivated by the OpenFlow [19] protocol, designed at Stanford and now maintained by the Open Networking Foundation (ONF).³ It separates the control plane from the data plane. As a refresher, the control plane determines how data packets are routed, and the data plane carries out this routing, based on the rules defined in the control plane.

The development of SDN was spurred on by being seen as having a balance between vision and pragmatism: it was something that could be easily adopted, but also had promising uses for the future. The field itself has seen development based on “technology pushes” and “use pulls”: certain technological advances have allowed more processing to be done within the network, but also serious time and effort in developing SDN itself has been invested by companies and individuals who will benefit from its uses [7]. In real terms, most early use cases came in the form of network virtualization.

So what are the use of SDN *controllers*? Their purpose is to transform generic SDN switches into other types of network devices. The reason that this control is centralized to the SDN is that if these switches encounter a new type of packet, they first need to communicate with the controller (using a protocol like OpenFlow). Furthermore, they eliminate issues arising from individual routers making contradictory decisions due to not sharing state. In the context of inter-domain routing, this means policy decisions can be informed by a consistent global state.

3.2 Google’s move to SDN

Vahdat et al. give a concrete example of what incentivizes a large company like Google to move internally to SDN [22]. SDN brings concepts from telecom network infrastructure; huge WANs like Google now carry approximately 10% as much traffic as the Internet, but their operation has a key difference. Every part of the network is owned and operated

¹<https://tailscale.com/kb/1136/tailnet>

²<https://tailscale.com/kb/1232/derp-servers>

³<https://www.opennetworking.org/sdn-resources/openflow/>

by the same company. When all hosts are trusted, these networks do not have the same “push” towards decentralization that was fundamental to the original Internet design.

In the future, the authors even predict SDN peering could overcome Border Gateway Protocol (BGP)’s distrustful view of the network by dynamically sharing (some) additional info about your network [22]. For example, info about downstream traffic patterns would improve performance for end users and let ISPs use their “lightly loaded paths” better, or even enable application-specific peering. However, they argue that currently SDN is most applicable in cases where the controller platform and controller application are tightly integrated and developed by the same people; that for SDN to apply more broadly we need reusable platforms to exist.

One last note is that David Clark, another author of the paper [22], shares more recent thoughts on how fate-sharing is affected by the SDN paradigm. I mention this because they differ from the thoughts he shared in his original DARPA paper which coined the term [5], and they are worth the read.

3.3 GameTE: A Game-Theoretic Distributed Traffic Engineering in Trustless Multi-Domain SDN

Armed with some knowledge about SDN, I began by looking at an initial candidate for replication: GameTE.

When performing inter-domain routing, Liu et al. note it is not straightforward to maintain trust between multiple parties [17]. One way to solve this problem is to use game theory, which can model behaviour of two rational selfish actors (who are looking to maximize the benefit they receive as an individual).

Game theory is, very briefly, studying the math behind strategic thinking. GameTE proposes a game-theoretic traffic engineering algorithm, used between multiple domains which do not trust each other. It has both a theoretical argument (algorithm’s reaction to “when selfish domains act as transit domains” and “when selfish domains act as source domains”) as well as results from an experimental setup.

Unfortunately, while the authors do provide their algorithm and provide details regarding their experimental setup (Floodlight SDN controllers within Mininet), the raw data from the experiment is not released nor is the testbed setup the authors used. Additionally, it is hard to assess the impact of the GameTE paper since it was published in 2024 [17]. The algorithm is also intended for use among autonomous systems (ASes), which lead me to believe that developing it might be harder than the average SDN algorithm. Lastly, the paper also used topologies constructed from the Internet Topology Zoo [11], which are very cool, but I worried that they would be difficult to construct myself. These facts led me to eventually conclude that the complexity of replicating this work would likely be beyond the scope of the timeline for this class project.

3.4 Leveraging software-defined networking for security policy enforcement

Since I did not select GameTE, I searched for related works and found a paper by Hadi et al. which used an SDN controller for an institutional security policy enforcement system [8]. I didn’t select this paper due to quality concerns (did not recognize journal, authors’ emails were not academic domains, and overall paper structure seemed poor). Instead I selected one of the competing algorithms by Liu et al. that the paper referenced [16]. I do not believe it is the perfect choice, but I believe the Liu et al. paper is of high enough quality to warrant an implementation.

Why do I think the paper warrants a replication and/or extension? It introduces a relatively simple algorithm that I believed I could replicate in Mininet using Ryu. The paper is important because incorrectly updating flow tables exposes security holes and “basic” solutions to this problem are expensive or impossible to carry out.

The paper has two primary contributions: (1) a proposed two-layer OpenFlow switch topology for policy enforcement and (2) a safe update strategy for the resulting configuration. To clarify some terminology: *flow tables* are used by

OpenFlow-enabled switches to determine where to route packets. They are composed of “flow entries” or “rules” which are read and used each time a packet must be forwarded. *Policy enforcement* is the act of enforcing a particular set of rules for routing packets on your network based on their type (source/dest IP address, originating MAC address, etc.).

A very brief summary of the technical details:

- (1) The paper defines S-switches and F-switches.
- (2) An S-switch classifies packets (eg. ssh or https traffic).
- (3) An F-switch implements security policies (eg. dropping packets of certain traffic classes).
- (4) The main contribution is an algorithm which securely updates the security policies in the F-switches without incorrectly dropping approved traffic and/or letting denied traffic through at any point during execution.

The difference between this policy enforcement strategy and the basic approaches are that these other approaches “group” security policies and feed packets through a “chain” of filtering switches, or put all flow rules required for policy enforcement into the available edge switches (OVS or ToR).

4 Environment

This section describes the environmental setup for my project. I decided to start from scratch without leaning on an existing implementation because Mininet and Ryu are ideal for prototyping this project, but existing research lacks open-source prototypes using those technologies.

4.1 Mininet

Why Mininet? It is a step up from the original paper’s approach, I wanted to aim higher than the original paper’s simpler approach of a simulation. Simulation often hides complexity; things work nicely when they are running on a single thread in a Python program, but when you are using real hardware (or emulated hardware) new problems crop up.

I decided to install Mininet within VirtualBox. This was a challenge the installation instructions were designed for an outdated version of macOS. Here are the instructions to reproduce the test environment on an Intel-based Mac running macOS Sequoia 15.3.1, based on the original instructions for installing Mininet.⁴

- (1) Download and install VirtualBox (v7.1.8) for macOS Intel hosts.⁵
- (2) If an older version of VirtualBox is pre-installed, run the `VirtualBox_Uninstall.tool` that is part of the package.
- (3) Download and install XQuartz (v2.8.5) used for GUI access to Wireshark.⁶
- (4) Running `echo $DISPLAY` in a new terminal window should no longer result in blank output.
- (5) Download and import the Mininet VM Image (v2.3.0) into VirtualBox.⁷
- (6) In order to have SSH access to the VM from your host machine, create a network adapter (see fig. 2): In the VirtualBox menu, click *Tools* and select the *Host-only Networks* tab on the right-hand side. Click the *Create* button in the menu. A new network with the name *HostNetwork* should be displayed.
- (7) Add this network adapter in the VM’s network settings (see fig. 3): In the VirtualBox menu, select the Mininet VM. Click the *Settings* button. In the settings, select *Network*. You need to change *Adapter 1* such that *Adapter*

⁴Installing Mininet, official Mininet documentation: <https://mininet.org/download/>

⁵<https://www.virtualbox.org/wiki/Downloads>

⁶<https://www.xquartz.org/>

⁷<https://github.com/mininet/mininet/releases/>

Type: *Intel PRO/1000 MT Desktop (82540EM)*. Next select *Adapter 2*. Click *Enable Network Adapter* and make the following changes: *Attached to: Host-only Network, Name: HostNetwork, Adapter Type: Intel PRO/1000 MT Desktop (82540EM)*. All other settings remain default. Click *OK* to apply the settings.

- (8) Boot up Mininet, and log in with the username (mininet) and password (mininet).
- (9) Verify network access works within the VM (`sudo dhclient eth0`). You can use `ifconfig -a` to see which IP your host network interface was assigned. Mine defaulted to the `eth0` interface with a `192.168.X.X` IP, and this persisted across reboots.
- (10) Add the following configuration to `/etc/network/interfaces` (using the information gained from the previous step) to automatically enable network access on boot: `auto eth0 && iface eth0 inet dhcp`
- (11) You should now have SSH access to the machine. In a terminal outside of the VM, try `ssh -Y mininet@192.168.X.X` where the IP address also matches what you found in the previous step. `-Y` will enable trusted X11 forwarding.
- (12) Log in via SSH.
- (13) Test that Wireshark can run by running `wireshark` within your SSH session. If it cannot you will need to follow additional troubleshooting steps.⁸ You may also try running `sudo dpkg-reconfigure wireshark-common && sudo usermod -a -G wireshark mininet` from within Mininet to reconfigure Wireshark to be runnable by regular users.

See the Mininet FAQ for answers to many questions.⁹

4.2 Ryu

Ryu is an SDN framework which allows you to dictate how a controller sends and responds to OpenFlow messages. I decided to use it (as opposed to another SDN framework) because it is ideal for experimentation. It is only recently without a maintainer, so it is mostly up to date with the latest OpenFlow improvements. It is also Python-based, which I find familiar because Python was the language used by the majority of my undergraduate programming classes. It is also lightweight. Lastly, Mininet recommends its use in their tutorial, and after making my way through their tutorial, I was feeling more confident with the framework than with the other choices.

5 Experiment

To take advantage of the opportunity that each Mininet host allows you to run any Linux commands you would like, or even open a shell, I decided to implement two of the topologies (seen in fig. 4 and fig. 5) from my selected paper and attempt to analyze how they would perform while routing generated traffic in both unloaded and congested situations.

5.1 Topologies

The first topology I implemented was the main topology proposed by the Liu et al. [16] paper. One point that the Liu et al. paper was not clear on is whether the topology is supposed to support connecting multiple F-switches to a single rack, or whether only one F-switch is supposed to connect to any rack. I assumed the multiple F-switches scenario, because the paper comments that ToR switches cannot implement policy enforcement efficiently since their resource constraints typically mean all of the necessary flow rules do not fit into their memory.¹⁰ This makes sense, because

⁸<https://github.com/mininet/mininet/wiki/FAQ#wireshark-xcb-error>

⁹<https://github.com/mininet/mininet/wiki/FAQ>

¹⁰ToR switches use TCAM to store flow rules, which means they are fast but have tiny flow tables compared to a virtual switch running on a x86 node. See <https://learningnetwork.cisco.com/s/article/tcam-demystified>

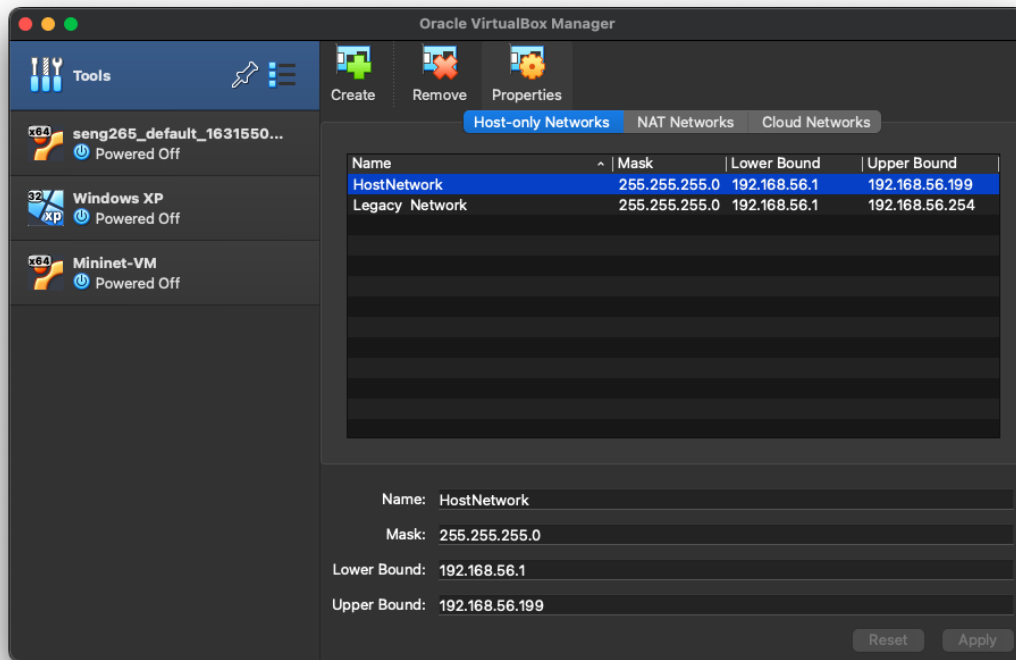


Fig. 2. After creating a new host-only network in VirtualBox.

different VMs can have different IPs even when running on the same hypervisor, in which case a ToR switch may need to implement rules for more traffic classes than there are physical machines connected to its ports. However, in practice this topology did not function correctly. It introduced ambiguity as to which F-switch a ToR switch should forward its outbound traffic through; multiple paths (through any F-switch) were available, and no S-switch was present to perform traffic classification. If these F-switches enforced different policies, then making an incorrect forwarding choice would introduce a policy security hole.

Lastly, another problem with the setup was that if rules do need to be applied within the same rack, by default the ToR switch did not pass the packet towards the S-switch because it knew that the other hosts in the rack were on the same subnet accessible via one of the ToR switches other links.

For the topology to function correctly, I also needed to implement the F-switch and S-switch logic within a Ryu-based controller. My implementation inherited features from the sample Spanning Tree Protocol (STP) switch implementation `simple_switch_stp_13.py` that ships with Ryu. Listing 1 contains a brief example of logic for coordinating one F-switch to drop packets belonging to SSH or HTTPS for a certain traffic class:

```

1 elif dpid > 0x1A: # F-switch: filter traffic
2     if dpid == 0x21: # F-switch #1: drop SSH and HTTPS traffic
3         for tcp_port in (22, 443):

```

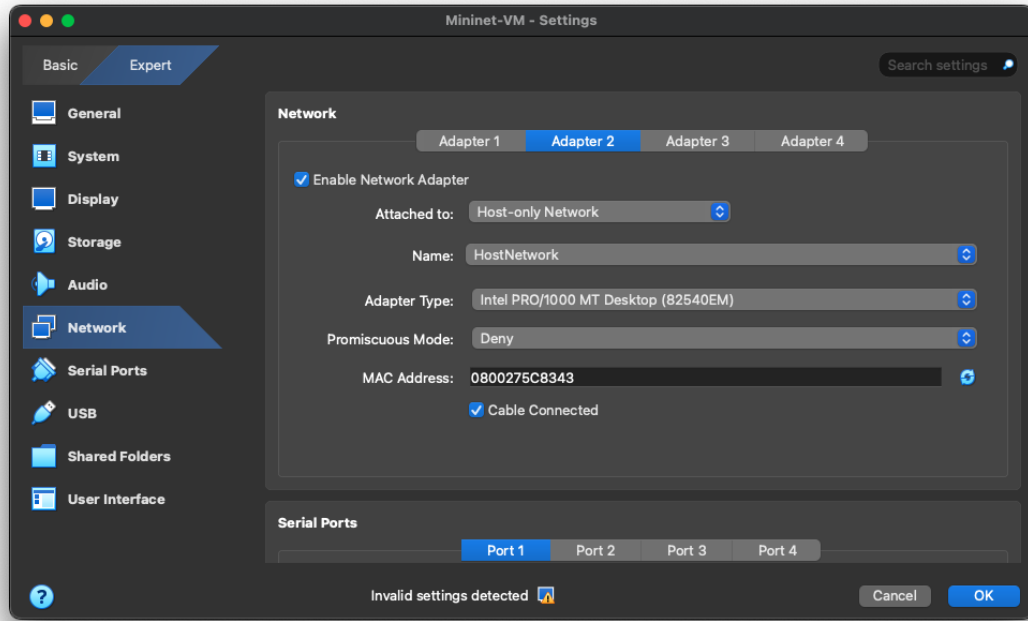


Fig. 3. After configuring the VM's new network adapter.

```

4      match = parser.OFPMatch(eth_type=0x0800, # must specify IPv4
5                                ip_proto=6, # only TCP
6                                tcp_dst=tcp_port)
7
8      inst = [parser.OFPInstructionActions(ofp.OFPIT_CLEAR_ACTIONS, [])]
9      mod = parser.OFPFlowMod(datapath=dp, priority=2,
10                               match=match, instructions=inst)
11
12      dp.send_msg(mod)

```

Listing 1. F-switch logic based on Liu et al.

I also tried to implement the basic topology described by Liu et al. as one of the inferior “straightforward solutions” in their paper. This topology is briefly described as a chain of filtering switches connected to each other that perform policy enforcement on every packet passing through the network. Fig. 5 contains a reference diagram for the topology. Like the previous topology, I also implemented the controller for this topology by inheriting from `simple_switch_stp_13.py`. Note that I interpreted the links between the F-switches ($F1 \rightarrow F2 \rightarrow F3 \rightarrow F4$) and ToR switches to be unidirectional. Therefore I attempted to implement control logic which dropped all incoming packets to F4 that did not originate from F3, and all outgoing packets from F1 which were not destined for F2. When I attempted to implement this topology (and corresponding controller which could properly filter packets), I found that network performance was severely degraded. I cannot determine whether this is a limitation of the topology itself, or if there are bugs in my implementation.

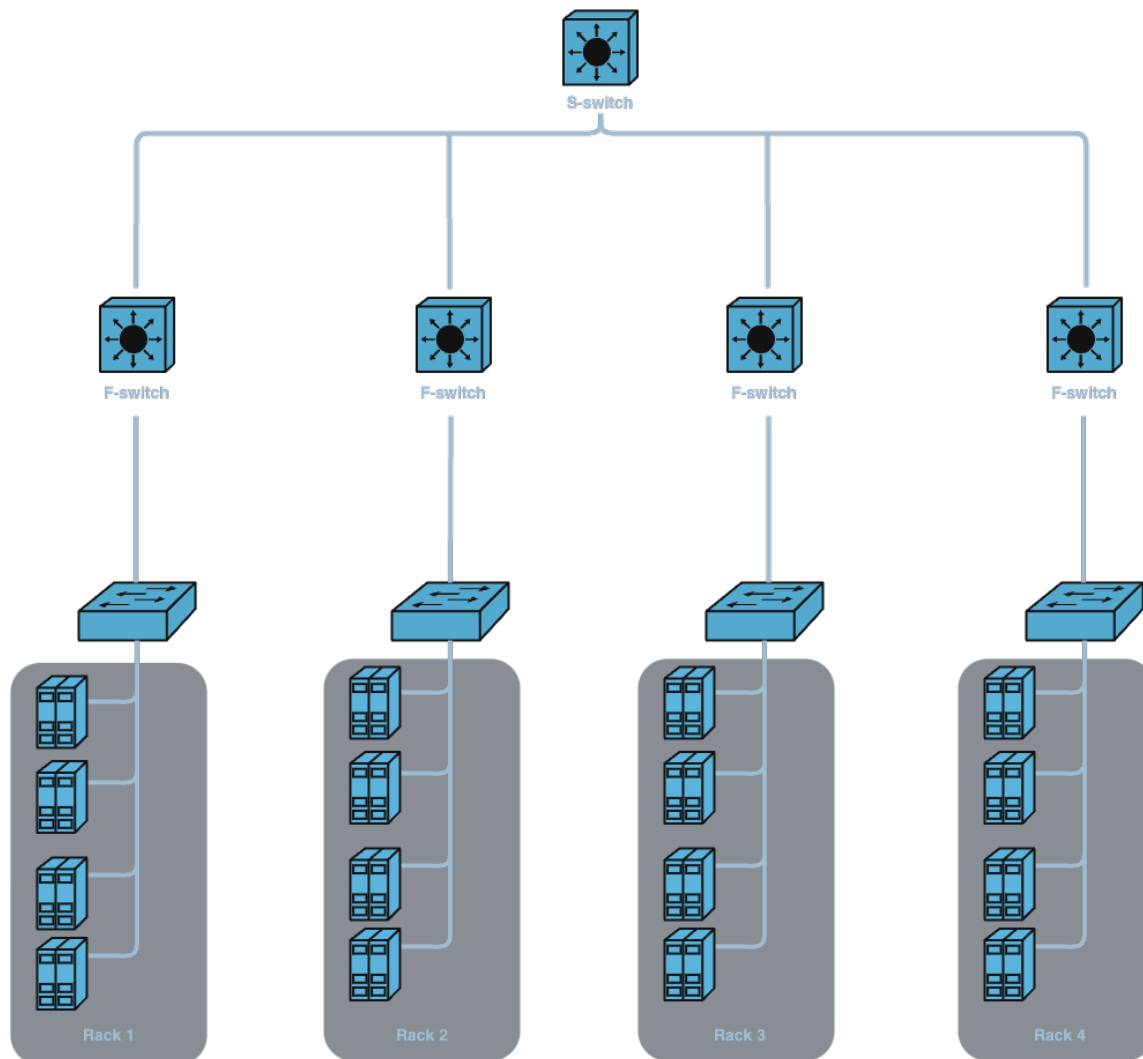


Fig. 4. Proposed experimental topology from Liu et al.

My hypothesis is that the performance degradation is related to the loop within the topology, possibly that all OpenFlow messages are being routed through the F-switch bottleneck and destroying performance. This leads me to believe that I have implemented the topology incorrectly since Liu et al. did not indicate that it would have unusable performance. However I have yet to discover a better way. It is possible that the intended topology is a chain of filter switches between every single rack and the main switch connecting all of them, but this seems unlikely because it would cause the number of filter switches to grow proportionally to the number of racks in the topology.

There were several limitations to my topology and controller implementations. I did not implement automatic port discovery within the controllers, I instead manually assigned certain switches to only connect to others using certain ports. This may not be ideal for a deployment scenario (perhaps it could be better if switches would communicate

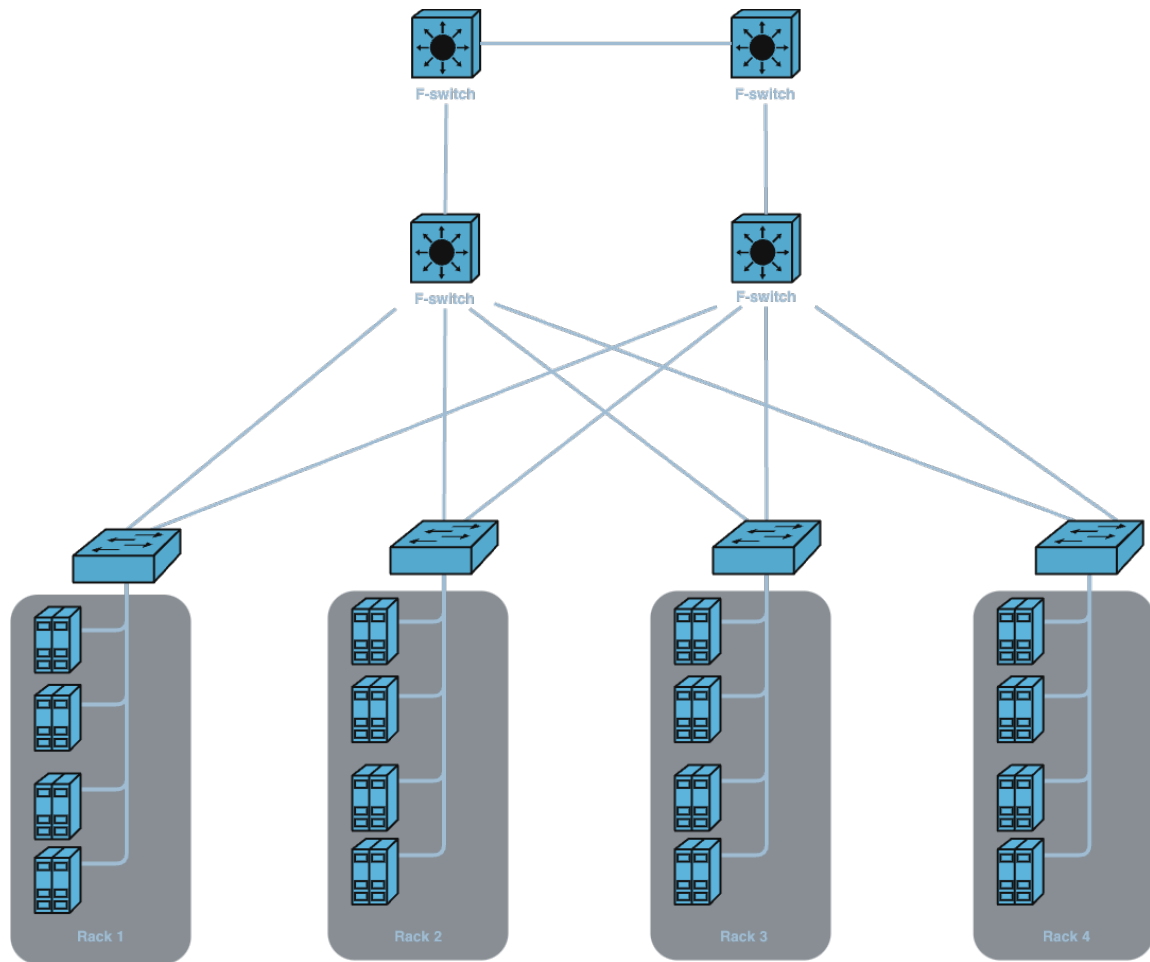


Fig. 5. Naive topology discussed in Liu et al.

to understand where they are connected to each other in the topology). However I believe it is semi-realistic as the network administrator who is implementing this topology would have control over which ports correspond to which links.

I referenced a variety of online resources to complete my code:

- (1) Inside-Openflow: Custom Mininet topologies and introducing Atom.¹¹
- (2) Understanding the Ryu API: Reimagining Simple Switch.¹²
- (3) The Ryu book.¹³
- (4) The Ryu tutorial.¹⁴

¹¹<https://web.archive.org/web/20170321151800/https://inside-openflow.com/2016/06/29/custom-mininet-topologies-and-introducing-atom/>

¹²<https://web.archive.org/web/20210623055222/https://inside-openflow.com/2016/08/05/ryu-api-reimagining-simple-switch/>

¹³<https://book.ryu-sdn.org/en/html/index.html>

¹⁴<https://ryu.readthedocs.io/en/latest>

- (5) Implementing different network topologies using Mininet.¹⁵

5.2 Performance capture

I wanted to perform a thorough analysis by running a Linux traffic-creation utility (for example trafgen), but this project concluded before I could get that far. Instead, I designed a simple performance comparison based around the tools that I used while debugging (ping and wireshark). I was severely limited by my implementation not functioning correctly, but I nevertheless performed the following comparison collection procedure:

- (1) Power on Mininet VM
- (2) Create two SSH connections
- (3) For each topology:
 - (a) Execute `sudo mn -c` before testing the topology
 - (b) Run the Mininet topology using Connection 1
 - (c) Run the SDN controller using Connection 2
 - (d) Execute `pingall` and wait for it to finish
 - (e) Run `r1h1 ping r4h4 -c 100` four times
 - (f) Run `r1h1 ping r4h4 -f -c 100000`

The commands I used to run the topology and controller were as follows:

- Topology: `sudo -E mn --custom my_basic.py --topo mybasic --mac --switch ovs --controller remote`
- Controller: `ryu run my_controller.py`

6 Results

My results were intended to include results from both topologies that I implemented in Mininet. Unfortunately, when I attempted to collect performance data for the second topology, Mininet repeatedly froze. Similarly, because my implementation was broken and I spent most of my time debugging it, I was restricted to analyzing basic debugging statistics. So these results are necessarily incomplete. Similarly because my implementation was broken and I spent most of my time debugging it, my analysis is limited to the very basic debugging data that I was collecting.

My theory is that the extreme reduction in performance that I experienced in Mininet while trying to collect statistics for my second topology is related to the loop that the topology contains. I was expecting to sidestep these problems by using a STP-based controller, but unknown problems still occurred. To expand on my theory, I think it is possible that OpenFlow messages were causing extreme congestion by all repeatedly passing through the single bottleneck in the system during the topology discovery phase. I attempted to debug this using Wireshark on one of the switches, but did not find enough evidence to confirm this hypothesis. It is also quite possible that there were issues with the flow rules that my controller was installing. My novice Ryu programming skills similarly limited me in troubleshooting this possibility, and the lack of detail in the Ryu documentation did not help. I did however have some success asking ChatGPT to explain some of the undocumented functionality.

In fig. 6 you can see that even after executing `pingall` within Mininet and waiting some time, ping times are initially much larger before experiencing a sharp reduction in latency. Subsequent pings do not have the same issue, even if time passes between pings. If I am not mistaken, this is because it takes some time for the network topology to stabilize and the appropriate flow rules to be installed on all of the switches. Because there a large number of hosts, I believe

¹⁵<https://medium.com/@abdulkaderhajjouz/implementing-different-network-topologies-e2bbafea2edb>

First vs. second ping (μs)

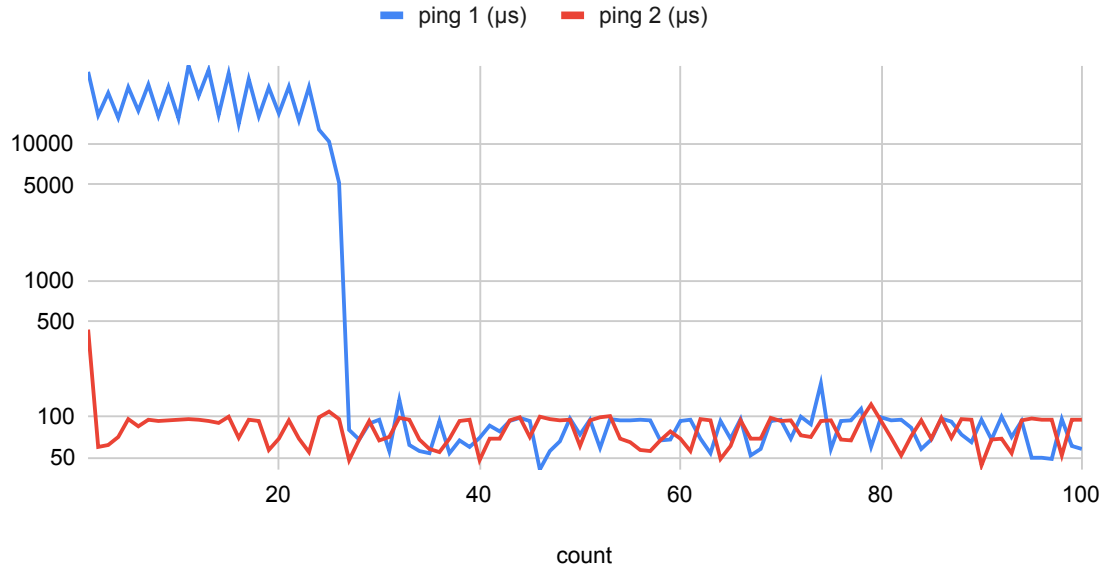


Fig. 6. Comparison of first versus second ping in experimental topology.

there is quite a bit of overhead passing OpenFlow messages across the links, since there are a lot of bottlenecks within the topology.

On subsequent pings (as seen in fig. 7), we can see that the first ping always takes additional time. My controller instructs all switches in the topology to behave as learning switches, so once the flow rules are installed no messages should pass between the switch and the controller. My controller logs confirmed that this was the case during pings, so I am not sure why there is a slight spike in latency for the first ping received in each sequence. Comparing with fig. 8, we can see that excluding the first ping time leaves us with relatively uniform results across all three ping sessions.

7 Future Work

This project left much to be desired. I barely scratched the surface of what is possible with SDN, and there is much to be done. In addition to fixing my second implementation, the third simple topology discussed by Liu et al. (enforcing policy via ToR switches) could be compared alongside the others. My experimental prototype is also fairly inflexible and would be difficult to be used in practice: a real-world implementation would read the policy for each F-switch from a YAML configuration file, or use some similar approach. Most importantly, the policy update mechanism for the topology combining F- and S-switches would need to be implemented.

Future work might extend this by incorporating game theory extensions of this work, similar to what Liu et al. express in their recent 2024 paper [17]. Adversarial situations often arise in networks where individual operators are

Subsequent pings

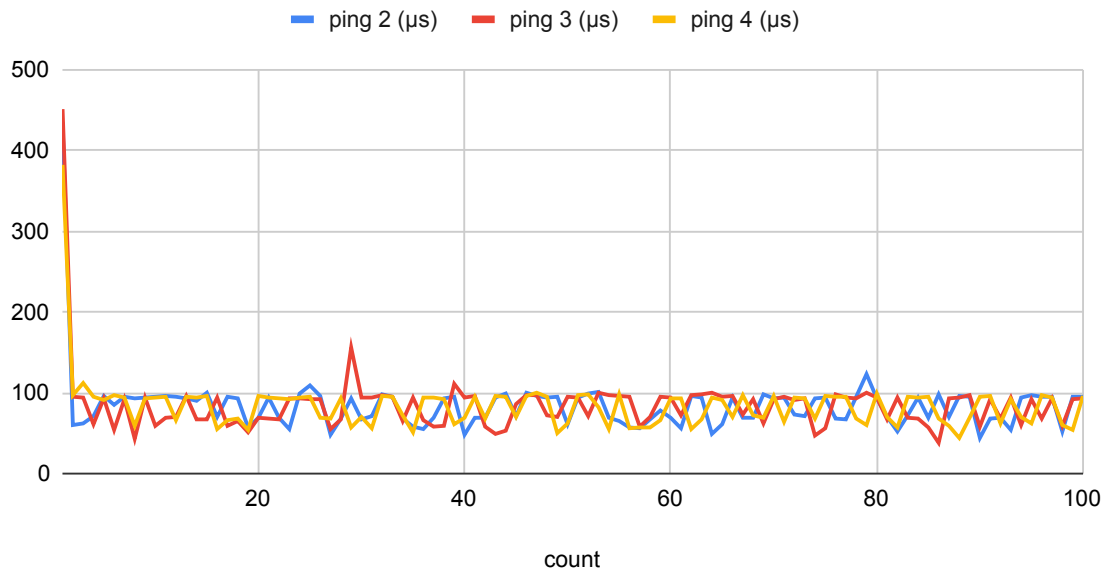


Fig. 7. Comparison of 2nd, 3rd, 4th pings in experimental topology.

competing or collaborating with each other. Plus, game theory's applications to wireless networks have already been explored previously [3, 9].

Again taking a page from Liu et al.'s book, the Internet Topology Zoo [11] could help determine whether the topologies explored in this paper are feasible at a large scale. However, care would need to be put into deciding whether implementing policy enforcement for a backbone makes sense. Since the primary use-case is for a corporate or data center network (DCN), it may instead make more sense to extend the topology using some kind of fat-tree approach [1].

Lastly, the strategy proposed by Liu et al. could also be extended to classify and react to burst traffic. For example, by extending the controller logic developed in this paper to create traffic classes based on live metrics such as the number of packets received from a single source.

8 Conclusion

This report briefly explored my project's beginnings, overviewed the background related work on SDN that led me to attempt to replicate components of Liu et al.'s 2016 paper, and performed a new experiment on the resulting topology that I emulated. Surmountable challenges (programming in Ryu with subpar documentation) and insurmountable challenges (completing this project on time) were encountered along the way. I learned more than I ever dreamed I would about SDN, traffic engineering, network topologies, and L2/L3 of the OSI model. I am proud to say that although in the beginning I was not comfortable engaging in any area of this project, I learned a lot, and know more about networks as a result.

Subsequent pings (first reply excluded)

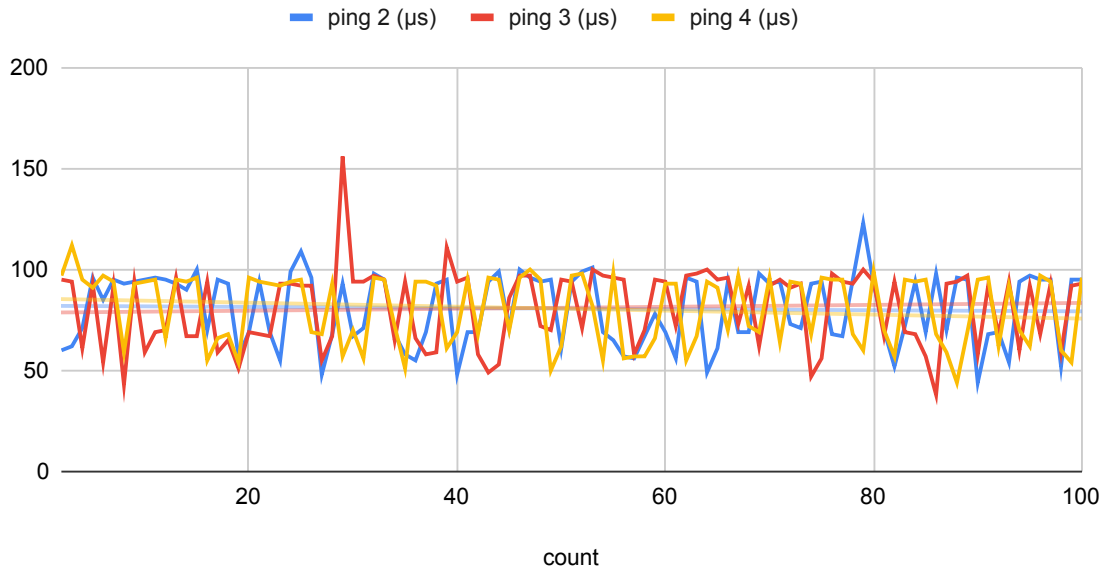


Fig. 8. Comparison of first versus second ping in experimental topology, with the first reply received excluded.

One last note on the first research contribution outlined as a goal at the beginning of this paper: How easy it is to relate ITS and/or teletraffic engineering approaches with SDN? Hard. My conclusion was that the lack of existing literature indicates that publishing an introduction to the links between ITS and SDN was outside of the scope of a class research project (after all, one would expect many papers to exist if it were easy). In contrast, I have stumbled upon examples of the reverse (principles commonly used in communications networks which influence ITS) [4, 14, 15, 18]. However, my primary interest was not in exploring ITS directly,¹⁶ so it seemed most prudent that I worry less about the ITS component and more about producing some limited experimental results.

Acknowledgments

Thanks to Jianping Pan, Jinwei Zhao for the excellently run class that I was fortunate to participate in. Special thank you to Emily Martins for multiple suggestions! Shiyi Zhang, Siyu Liang, Ekaba Bisong, Liam Calder, Jonathan Ami, Alpar Arman, and Justin Drastil for commenting on my project and providing peer review on my project updates and final presentation. Rena for catching my spelling mistakes and helping me with her draw.io skills when I needed to represent my constructed topologies. ChatGPT 4o for both helpful and terrible debugging advice in Mininet/Ryu. And to whoever you are: thank you; I hope you enjoyed what you read.

¹⁶A strange admission when one factors this report's title into account.

References

- [1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. *SIGCOMM Comput. Commun. Rev.* 38, 4 (Aug. 2008), 63–74. doi:10.1145/1402946.1402967
- [2] Franco Callegati, Walter Cerroni, and Carla Raffaelli. 2023. *Traffic Engineering: A Practical Approach*. Springer International Publishing, Cham. doi:10.1007/978-3-031-09589-4
- [3] Dimitris E. Charilas and Athanasios D. Panagopoulos. 2010. A Survey on Game Theory Applications in Wireless Networks. *Computer Networks* 54, 18 (Dec. 2010), 3421–3430. doi:10.1016/j.comnet.2010.06.020
- [4] Nan Cheng, Ning Lu, Ning Zhang, Xiang Zhang, Xuemin Sherman Shen, and Jon W. Mark. 2016. Opportunistic WiFi Offloading in Vehicular Environment: A Game-Theory Approach. *IEEE Transactions on Intelligent Transportation Systems* 17, 7 (July 2016), 1944–1955. doi:10.1109/TITS.2015.2513399
- [5] D. Clark. 1988. The Design Philosophy of the DARPA Internet Protocols. *SIGCOMM Comput. Commun. Rev.* 18, 4 (Aug. 1988), 106–114. doi:10.1145/52325.52336
- [6] Epic Networks Lab. 2020. Software Defined Networks & OpenFlow - IP Network Layer | Computer Networks Ep. 5.5 | Kurose & Ross.
- [7] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The Road to SDN: An Intellectual History of Programmable Networks. *SIGCOMM Comput. Commun. Rev.* 44, 2 (April 2014), 87–98. doi:10.1145/2602204.2602219
- [8] Fazal Hadi, Muhammad Imran, Muhammad Hanif Durad, and Muhammad Waris. 2018. A Simple Security Policy Enforcement System for an Institution Using SDN Controller. In *2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*. 489–494. doi:10.1109/IBCAST.2018.8312269
- [9] Zhu Han, Dusit Niyato, Walid Saad, Tamer Başar, and Are Hjørungnes. 2011. *Game Theory in Wireless and Communication Networks: Theory, Models, and Applications*. Cambridge University Press, Cambridge. doi:10.1017/CBO9780511895043
- [10] Norbert L. Kerr. 1998. HARKing: Hypothesizing After the Results Are Known. *Personality and Social Psychology Review* 2, 3 (Aug. 1998), 196–217. doi:10.1207/s15327957pspr0203_4
- [11] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (Oct. 2011), 1765–1775. doi:10.1109/JSAC.2011.111002
- [12] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (Jan. 2015), 14–76. doi:10.1109/JPROC.2014.2371999
- [13] James F. Kurose and Keith W. Ross. 2017. *Computer Networking: A Top-down Approach* (7. edition ed.). Pearson Education, Boston Munich.
- [14] Khalilollah Raeisi Lejji, Esmail Amiri, Emad Alizadeh, and Mohammad Hossein Rezvani. 2020. A Game Theory-based Mechanism to Optimize the Traffic Congestion in VANETs. In *2020 6th International Conference on Web Research (ICWR)*. 217–222. doi:10.1109/ICWR49608.2020.9122324
- [15] Caixia Li, Sreenatha Gopalarao Anavatti, and Tapabrata Ray. 2013. A Game Theory Based Traffic Assignment Using Queueing Networks. In *2013 13th International Conference on ITS Telecommunications (ITST)*. 210–215. doi:10.1109/ITST.2013.6685547
- [16] Jiaqiang Liu, Yong Li, Huandong Wang, Depeng Jin, Li Su, Lieguang Zeng, and Thanos Vasilakos. 2016. Leveraging Software-Defined Networking for Security Policy Enforcement. *Information Sciences* 327 (Jan. 2016), 288–299. doi:10.1016/j.ins.2015.08.019
- [17] Yangyang Liu, Jingyu Hua, Yuan Zhang, and Sheng Zhong. 2024. GameTE: A Game-Theoretic Distributed Traffic Engineering in Trustless Multi-Domain SDN. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. 1248–1259. doi:10.1109/ICDCS60910.2024.00118
- [18] Athanasios Maimaris and George Papageorgiou. 2016. A Review of Intelligent Transportation Systems from a Communications Technology Perspective. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. 54–59. doi:10.1109/ITSC.2016.7795531
- [19] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74. doi:10.1145/1355734.1355746
- [20] V. Srivastava, J. Neel, A.B. Mackenzie, R. Menon, L.A. Dasilva, J.E. Hicks, J.H. Reed, and R.P. Gilles. 2005. Using Game Theory to Analyze Wireless Ad Hoc Networks. *IEEE Communications surveys and tutorials* 7, 4 (2005), 46–56.
- [21] Fei Tong. 2016. *Protocol Design and Performance Evaluation for Wireless Ad Hoc Networks*. Ph.D. Dissertation. University of Victoria.
- [22] Amin Vahdat, David Clark, and Jennifer Rexford. 2015. A Purpose-built Global Network: Google’s Move to SDN: A Discussion with Amin Vahdat, David Clark, and Jennifer Rexford. *Queue* 13, 8 (Oct. 2015), 100–125. doi:10.1145/2838344.2856460
- [23] Xipeng Xiao, A. Hannan, B. Bailey, and L.M. Ni. 2000. Traffic Engineering with MPLS in the Internet. *IEEE Network* 14, 2 (March 2000), 28–33. doi:10.1109/65.826369

A Topology 1 in Mininet

```

1 def build(self):
2     # Main topology
3     tor_switches = []
4     for rack in range(1, 5):

```

```

5      # Create ToR switch
6      tor = self.addSwitch('s%d' \% rack, dpid='%x' \% rack)
7      tor_switches.append(tor)
8      # Each host in rack
9      for host in range(1,5):
10         host = self.addHost('r%dh%d' \% (rack, host), ip='10.0.%d.%d' \% (rack-1,
11                                host))
12         # Add links between each host and ToR
13         self.addLink(tor, host)
14
15     # Extra switches required by construction
16     # Create a S-switch, dpid always starts with '1'
17     s1s = self.addSwitch('s1s', dpid='11')
18
19     # Create two F-switches, dpid always starts with '2'
20     s1f = self.addSwitch('s1f', dpid='21')
21     s2f = self.addSwitch('s2f', dpid='22')
22     s3f = self.addSwitch('s3f', dpid='23')
23     s4f = self.addSwitch('s4f', dpid='24')
24
25     # Connect S and F switches to topology
26     self.addLink(s1s, s1f, port1=1, port2=1)
27     self.addLink(s1s, s2f, port1=2, port2=1)
28     self.addLink(s1s, s3f, port1=3, port2=1)
29     self.addLink(s1s, s4f, port1=4, port2=1)
30
31     self.addLink(s1f, tor_switches[0], port1=2)
32     self.addLink(s2f, tor_switches[1], port1=2)
33     self.addLink(s3f, tor_switches[2], port1=2)
34     self.addLink(s4f, tor_switches[3], port1=2)

```

Listing 2. S-switch and F-switch topology proposed by Liu et al.

B Topology 2 in Mininet

```

1 def build(self):
2     # Main topology
3     # Create filtering switches
4     s1f = self.addSwitch('s1f', dpid='21')
5     s2f = self.addSwitch('s2f', dpid='22')
6     s3f = self.addSwitch('s3f', dpid='23')
7     s4f = self.addSwitch('s4f', dpid='24')
8     # Link filtering switches
9     self.addLink(s1f, s2f, port1=1, port2=1)
10    self.addLink(s2f, s3f, port1=2, port2=1)

```

```
11 self.addLink(s3f, s4f, port1=2, port2=1)
12
13 for rack in range(1, 5):
14     # Create ToR switch
15     tor = self.addSwitch('s%d' \% rack, dpid='\%x' \% rack)
16     # Link rack to s1f (for outgoing traffic)
17     self.addLink(s1f, tor, cls=TCLink, port2=1)
18     # Link s4f to rack (for incoming traffic)
19     self.addLink(s4f, tor, cls=TCLink, port2=2)
20     # Each host in rack
21     for host in range(1,5):
22         host = self.addHost('r%dh%d' \% (rack, host), ip='10.0.%d.%d' \% (rack-1,
23                                     host))
24         # Add links between each host and ToR
25         self.addLink(tor, host)
```

Listing 3. “Straightforward solution” topology from Liu et al. used in comparison.