9-2021

Evaluating Testing Procedures for OpenFlow Controller Network Re-Provisioning Time

Steven J. Jensen

EVALUATING TESTING PROCEDURES FOR OPENFLOW CONTROLLER NETWORK RE-PROVISIONING TIME

THESIS

Steven J. Jensen, Master Sergeant, USAF

AFIT-ENG-MS-21-S-009

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-21-S-009

EVALUATING TESTING PROCEDURES FOR OPENFLOW CONTROLLER NETWORK RE-PROVISIONING TIME

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyberspace Operations

Steven J. Jensen, B.S., M.I.S.

Master Sergeant, USAF

September 16, 2021

AFIT-ENG-MS-21-S-009

EVALUATING TESTING PROCEDURES FOR OPENFLOW CONTROLLER NETWORK RE-PROVISIONING TIME

THESIS

Steven J. Jensen, B.S., M.I.S.
Master Sergeant, USAF

Committee Membership:

Mark G. Reith, Ph.D.
Chair

Scott R. Graham, Ph.D.
Member

Timothy H. Lacey, Ph.D.
Member

AFIT-ENG-MS-21-S-009

Abstract

Software-Defined Networking (SDN) has advantages over traditional networks. SDN can perform monitoring, management, load balancing, network virtualization and orchestration, and policy enforcement across a variety of networks. However, SDN has seen low adoption, for various reasons, including difficulty in evaluation. In 2018, the Internet Engineering Task Force (IETF) released a benchmarking methodology for SDN controllers in Request for Comments (RFC) 8456. Aside from throughput and latency measures, there is little methodology validation for metrics defined in the RFC. Furthermore, the RFC's guidance provides no standardized analysis techniques and thus misses an opportunity to foster comparable results among researchers.

To validate that the RFC methodology is capable of detecting and evaluating SDN controller Network Re-provisioning Time (NRT), a small, three switch Local Area Network (LAN) is emulated on a single physical server. All hosts and devices within are virtualized with open-source tools. Five SDN controllers are evaluated with 30 tests runs of network traffic in the test environment. Network traffic captures and reports are inspected to identify possible errors and analyze NRT with related packet loss differences between controllers.

The RFC NRT methodology did not always detect the NRT, and was not able to detect the NRT in any OpenDaylight (ODL) controller test run. The methodology was able to detect NRTs for the remaining controllers under test. The two of the remaining controllers are proactive and consistently produced NRTs near one second or lower, but the test environment was not sensitive enough to detect NRTs under 20 milliseconds. The remaining two reactive controllers had NRTs of 12 seconds or longer, with greater variability.

RFC 8456 is a step in the right direction, with detailed specifications for required evaluation parameters. There are a few edge cases where the methodology struggles, and adding a network re-convergence time metric may account for those cases. The experiment showed how different the NRTs are based on controller implementation and configuration. During testing, the environment exhibited errors that require further investigation.

Table of Contents

List of Figures

List of Tables

EVALUATING TESTING PROCEDURES FOR OPENFLOW CONTROLLER NETWORK RE-PROVISIONING TIME

I. Introduction

1.1 Problem Background

The United States Air Force (USAF) uses traditional networking concepts, with long life cycles for hardware, and various mechanisms for managing heterogeneous network devices. Network defenses and traffic control are independently spread out across those devices with limited coordination. Independence is a short-term resiliency advantage, as networks can operate in a degraded state when certain components are non-functional. However, the lack of fully synchronized management, defenses, and traffic control, put traditional networks at a long-term disadvantage.

Software-Defined Networking (SDN) has the potential to extend network hardware's useful life and improve management by relying on general purpose servers and network devices. A key component of SDN is a centralized view of the entire network. This holistic view of the network enables better traffic control, and application of security across the entire network instead of at a few key points. This centralization does create a potential single point of failure, which warrants risk assessment.

Despite the advantages, SDN has seen neither wide adoption nor robust standardization, making it difficult to assess as an option for implementation. To identify situations where SDN deployment is advantageous, clear evaluation procedures are needed. These procedures should increase the opportunities to implement SDN and identify which SDN frameworks are most appropriate.

1.1.1 Problem Statement

Tools specifically designed to assess SDN controller throughput and latency metrics, namely Cbench [1] and its variants, have existed since 2010. However, these are not the only metrics that are relevant to network design. In 2018, the Internet Engineering Task Force (IETF) published Request for Comments (RFC) 8456, Benchmarking Methodology for SDN Controller Performance. The RFC expanded the metrics to assess SDN controllers, but there is little validation of the methodology. Any discussion of deploying SDN technology on military networks is predicated on a robust evaluation framework. The RFC needs further validation before it becomes that framework.

1.2 Research Objectives and Hypothesis

The primary research goal is to validate a portion of the RFC 8456 methodology, namely Network Re-provisioning Time (NRT), while the secondary goal is to determine if the controllers under test are significantly different in NRT metrics. A supporting goal is reliable NRT testing environment development, to provide valid results for analysis. The research hypothesis is that the RFC 8456 NRT methodology framework is able to reliably reveal meaningful differences in the observed metrics between the SDN controllers under test.

1.2.1 Approach

There are three approaches for SDN testing: simulation, physical, and emulation. Simulation is fast and tightly controlled but lacks real world factors. Physical equipment testing can provide real world results for small networks, but the cost to mirror large networks is prohibitive. Emulation of physical devices in software scales with larger networks and provides results closer to physical networks with less upfront cost.

This research adopts the emulation approach, with a single physical server containing the entire experiment, with all hosts and devices virtualized. The virtualized systems form a three-switch ring topology with a traffic generating host on each end. A script generates test traffic flows between the hosts, and disables the link over which it flows at a set point, forcing the network to reroute the traffic. Examination of packet captures determines time between events, and traffic reports measure packet losses. The choice of network controller is the single factor, and each controller is tested thirty times in random order.

1.2.1.1 Assumptions and Limitations

RFC 8456 covers many possible tests, this work is focused on the NRT methodology. NRT is one of several RFC 8456 metrics, and arguably one of the least evaluated. Meaningful evaluation of the NRT is critical to the methodology's maturity. The developed test environment only emulates a small three-switch mesh topology Local Area Network (LAN) and does not produce generalized results for other network topologies. At present, the test environment does not have traditional network emulation for a baseline comparison. It can only compare SDN controllers with other SDN controllers.

1.2.1.2 Contributions

This thesis extends SDN controller evaluation research. Specifically, it provides validation of and refinements to RFC 8456. It also provides a statistical comparison of NRT metrics in a small test network.

1.3 Document Overview

Six chapters cover the performed research. Chapter II defines SDN and SDN architecture with its applications. Then, it covers SDN controller performance evaluation, common tools to perform those evaluations, and earlier research in this area. Chapter III details the experiment test bed design and development tools. Chapter Chapter IV covers RFC 8456's methodology, its implementation in the experiment, and result analysis methods. Chapter V supplies the analysis of the methodology and the produced controller metrics. Finally, Chapter VI summarizes the research and findings with details of future work.

II. Background and Literature Review

2.1 Overview

This chapter discusses the information needed to understand general SDN concepts and SDN controller performance evaluation. Section 2.2 outlines traditional networks and devices. Next, Section 2.3 defines the characteristics of SDN in comparison to traditional networking. Section 2.3.1 defines the modern SDN architecture. Finally, Sections 2.3.4 and 2.4 cover SDN tools and related research for controller performance evaluation.

2.2 Traditional Networking Concepts

Network devices perform one to many network packet manipulation and forwarding operations. Such devices include routers, switches, firewalls, load balancers, Intrusion Detection Systems (IDSs), and other network security systems. The aggregated packet forwarding and manipulation resources across a group of network devices are known as the forwarding plane. Similarly, the collective functions instructing the devices to manipulate and forward packets are called the control plane. Last, the application plane is all applications and services used to control network behavior orchestrated by the control plane [2].

In traditional networks, each network device has a fragment of the entire forwarding, control, and application planes. The control plane logic on each device depends on limited information, and makes forwarding decisions based on their functionality and information it learns directly from neighboring devices, without a picture of the network as a complete system. The control plane logic also integrates tightly with the forwarding and application fragments on the devices, often with proprietary management interfaces. All these factors increase complexity, cause unpredictable behavior, and make rapid change implementation difficult. Traditional networking has reasonable resiliency, with each networking device able to work independently if needed.

2.3 Software Defined Networking

The simplest definition of Software Defined Networking, according to the Open Networking Foundation, is the "physical separation of the network control plane from the forwarding plane, and where a control plane controls several device" [3]. IETF RFC 7426 defines it as a "programmable networks approach that supports the separation of control and forwarding planes via standardized interfaces" [2]. A more complete definition identifies the following characteristics [4]:

- Plane separation: The forwarding plane uses logic and data tables to forward, drop, consume, or replicate incoming packets. The control plane has logic that programs and/or configures the forwarding plane. In traditional networking, individual network devices hold portions of the control plane and forwarding plane. In SDN, the control plane does not reside on network devices. It is logically centralized on a control device or controller, which gives it a global view of the network to programming forwarding devices.

- Simplified devices and centralized control: Building off plane separation, the network devices are simpler. The controller takes the load of converting high-level policy into simple, lower-level instructions, which are then sent to the forwarding devices for quick execution.

- Network automation and virtualization: SDN provides three abstractions for programmatic control of the network. Distributed state abstraction provides a whole network view instead of dealing with the state of each individual device.

Forwarding abstraction handles vendor specific device hardware, and configuration abstraction allows programming an overall goal without specifying how it will be implemented. Application Programming Interfaces (APIs) enable communication between and within the planes. They abstract the underlying topology and network protocols. Additionally, they supply a path for programming forwarding devices. Then, applications can rapidly automate network changes without direct access to those devices. This is like modern computer operating systems, which provide an abstraction of the physical hardware to applications. In combination, these abstractions present a virtual network as a service to host devices, separate from the actual physical topology.

- Openness: Having open, well documented standards for SDN APIs should promote a larger number of individuals and organizations to develop the technology.

2.3.1 Software Defined Networking Architecture

The SDN architecture has four major components: network devices, controllers, applications, and interfaces. This section details their sub-components and interconnections as shown in Fig. 1.

2.3.1.1 Interfaces

The first piece of architecture is the various interfaces within and between each plane that enable communications. The primary interfaces are APIs, aided by network service interfaces when located on separate devices. RFC 7426, SDN: Layers and Architecture Terminology, classifies the interfaces between planes as North, South, East, and West bound[5]. North Bound Interfaces (NBIs) connect the application plane to the control/management planes. There is no standardization across NBI implementations, as network applications can utilize many different APIs, but Rep-

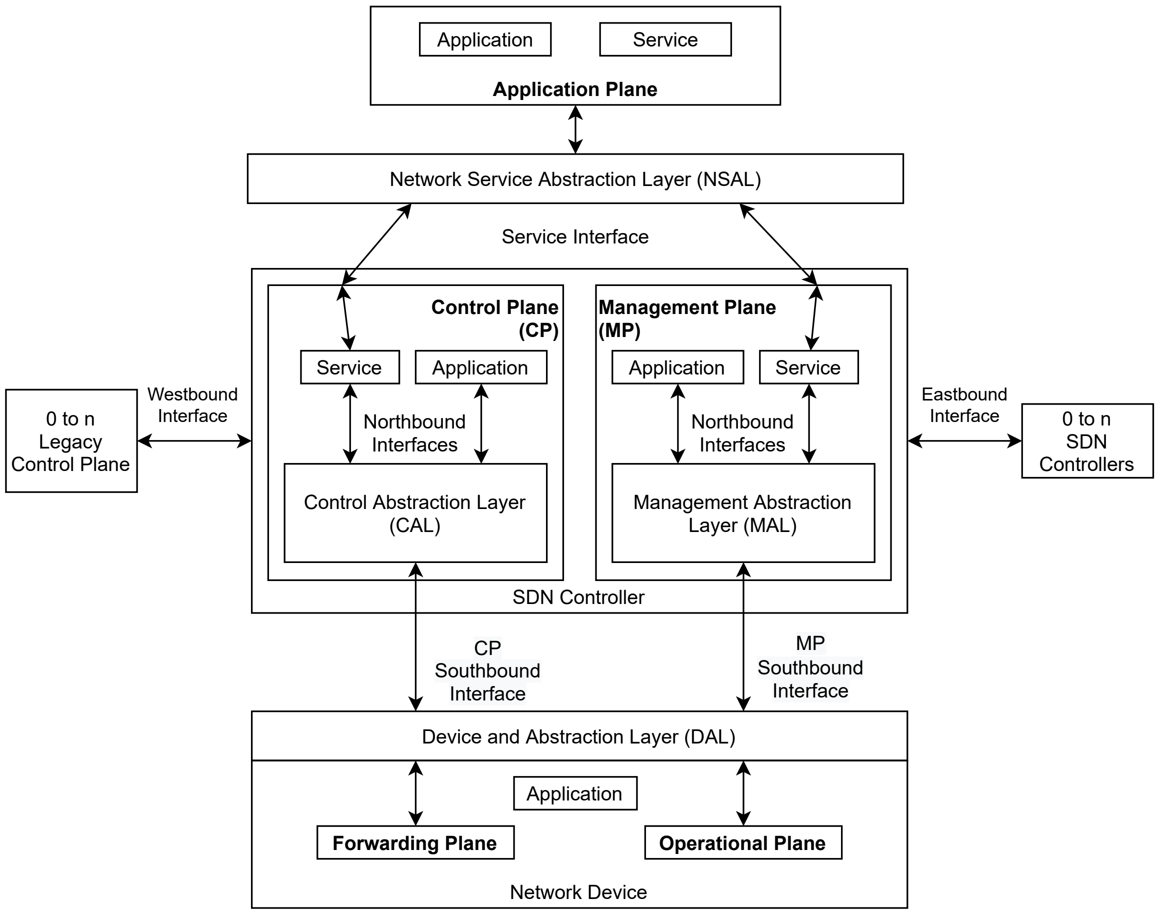


Figure 1. SDN Layered Architecture [5][2]

resentational State Transfer (REST)ful APIs are common due Hypertext Transfer Protocol (HTTP) integration and their simplicity. Distributed controllers use East Bound Interface (EBI)s to coordinate and synchronize between their physical systems. Like NBIs, there is no standardization in EBIs. West Bound Interface (WBI)s communicate with traditional devices using compatible protocols and give controllers access to information where they SDN interface with legacy networks.

The South Bound Interfaces (SBIs) are the communication path from controllers to network devices. There are two classes of SBIs associated with their respective planes. The first is Control Plane Southbound Interface (CPSI) which connects the control plane to the forwarding plane in the network devices. CPSIs have high

	Control Plane Interface Protocols	**Management Plane Interface Protocols**
OpenFlow Dependent Protocols	POF, OpenState, ROFL, HAL, PAD, DevoFlow	OvSDB, OF-Config
OpenFlow Independent Protocols	ForCES, OpFlex, BGP, P4, XMPP	SNMP, NetConf

Table 1. Southbound Interface Protocols

throughput, low latency, and redundancy to ensure prompt updates to flow tables. The other class is Management Plane Southbound Interface (MPSI), which passes configuration, fault, and monitoring data between the management plane and the operational plane in the network devices. The information is less time sensitive and does not require a high-performance connection. Some SBI protocols incorporate elements of both, but just enough to support their primary role [2].

The most common and standardized CPSI is OpenFlow, and it is the primary driver of SDN development, sponsored by the Open Networking Foundation. It is so prolific that researchers can classify other SBIs by their dependence on it [5]. Table 1 shows a cross section of SBI protocols based on primary plane and OpenFlow dependency.

OpenFlow switches use match action tables to make forwarding decisions. The switch compares the packet against these tables when the packet enters and can match on multiple defined characteristics. The switch performs the assigned action or actions when it finds a match. The actions include dropping the packet for security or other concerns, modifying it to implement a service or policy, queuing it for later transmission, and forwarding out a designated port [6].

If the switch does not find a match for the packet in the flow table, it sends it to an OpenFlow controller via a PACKET_IN message. The controller determines the actions relevant network devices should take based on its algorithms and policies set by its applications. The matches and actions required of each device create a traffic flow. The controller then sends the flow instructions to each network device which changes its match action flow table accordingly. There are several message options from the controller to the devices to conduct the flow table modifications. The first packet of a traffic stream without a match received by the forwarding device is queued or dropped until the device receives the modification message. This causes an initial delay in the delivery of the flow's traffic while the forwarding device waits for instructions. Afterwards, all matching packets will have the proper action taken as long as the flow remains in the device's flow table. If there are multiple matches, the forwarding devices apply actions based on highest flow priority set by the controller. The controller can also make modifications to traffic flows based on other events, such as a change in the physical network topology [6].

2.3.1.2 Network Devices

Networking devices oversee packet manipulation and forwarding from one device to the next. They have a forwarding plane with resources to process the rules they receive from the controller. Implementing those rules on traditional devices with highly specialized hardware can be difficult, as they lack flexibility. General-purpose hardware has increased in performance, making SDN workable, and can run alongside the specialized hardware as hybrid switches. SDN rules can be complex and large, increasing memory requirements [7]. Another component within networking devices is the operational plane, which manages all other aspects of the network device not related to forwarding, i.e., configuration, resources, and state. Applications can exist on the devices, which may depend on the forwarding and/or operational plane, but do not connect to external services. Both the forwarding and operational planes communicate with the controller through a Device and resource Abstraction Layer

(DAL) via Southbound Interface protocols [2].

2.3.1.3 Network Controller

The controller provides APIs connecting all components in the architecture. Its primary components are a control and/or management plane. The control plane takes topology information and forwarding requests from the networking devices through the Southbound interface protocol and Control Abstraction layer. It also receives external service requests through a Northbound interface protocol and Network Services Abstraction Layer (NSAL) [2]. The applications with the control plane use this information to generate data flow instructions, pushed through the SBI to the network devices' Forwarding Planes for installation. The Management plane monitors, configures, and maintains networking devices. It interacts with the operational plane through a Management Abstraction Layer (MAL) to control networking device state [2]. Examples include the state of network ports, the size, and number of forwarding tables, and remote access.

Controller scalability and flow processing performance are key factors influencing overall SDN performance. Physical controller placement also links to network scalability. As the physical distance between the controller and network increases, the time required for communication between them increases. This propagation delay influences the controller's ability to respond to requests, negatively affecting overall network performance [7].

Though the controller is logically centralized, its physical hardware is either centralized or distributed across multiple locations. If physically centralized, any failure in the controller has a large, negative impact on network operation. Network engineers must consider this single point of failure when implementing SDN. Increasing the size of the network necessitates physically distributing the controller to overcome the latency issues described previously, but also increases reliability. Distributed controllers organize in flat relationships as peer-to-peer systems, or a hierarchical structure depending on the application [7].

Control granularity also affects controller performance. Control granularity is how many requests the controller chooses to process, and how much information it uses to find a solution [7]. Finer control induces more overhead traffic between the controller and forwarding devices. In consideration of this, controllers can make reactive or proactive decisions. With reactive, the controller must make decisions for every new flow, incurring a slight delay for the first data packets waiting at the forwarding device. However, a high volume of requests, or a physical greater distance from the forwarding devices amplifies the delay. The proactive method pre-populates entries the forwarding device tables or leaves basic level policy decisions to the forwarding devices.

2.3.1.4 Application Plane

The last major component is the Application plane. Applications and services within the plane communicate with the controller through the NBI to refine network behavior [2]. These can range from topology discovery, authentication, policy, security, and Quality of Service (QoS) to virtual machine orchestration. It can be on the controller or distributed across interconnected systems.

2.3.2 SDN Applications

SDN controllers provide a platform for improvements in four broad categories of applications beyond management: monitoring, load balancing, virtualization/orchestration, and policy enforcement [8].

- Monitoring and Management: the centralized nature of controllers provides a global view of the topology and other aspects of network activity. Host tracking also supplies a record as devices join, leave, or move to different locations in the network. They serve as an oracle for activity on the network, although the more active and granular the monitoring, the greater the overhead on the controller(s).

- Load balancing: the global view also provides the means to optimize low-level resource allocation, latency, and throughput across the network.

- Network Virtualization and Orchestration – SDN enables virtualization or slicing of physical network resources, acting as a network hypervisor. This also makes it suited for coordinating with cloud orchestrations systems, such as OpenStack. These systems constantly create, destroy, and move Virtual Machine (VM)s within them. In combination, SDN and orchestration systems guarantee network availability to VMs without manual re-configuration or complicated topologies.

- Policy enforcement – The global view of a controller is also effective for enforcing policy. The controller can apply authentication and access controls across the entire network. Applying firewall and intrusion detection policies implements network wide security. Finally, a controller can dictate many forms of traffic engineering/shaping for QoS or policy-based routing.

Researchers initially applied these capabilities to networks of limited geographical size, such as LANs, Campus Area Networks (CANs), and data center networks. It has expanded to carrier and service providers in the form of optical networks, Software Defined Wide Area Networks (SD-WANs), and network slicing for small consumers. The latest developments have been in cloud, fog, and edge computing, including Internet of Things (IoT), wireless infrastructure, Wireless Sensor Networks (WSNs), unmanned aerial vehicles (UAVs)s, and Vehicle Area Networks (VANs) [8].

2.3.3 Controller Performance Evaluation

Comparing the performance between software-defined networks is difficult, as the applications are varied as well as optimization objectives. Many combinations of control and data plane performance including different combinations of hardware and software make the problem hard [9]. The one required SDN component is the controller, so most evaluations center on its characteristics. However, comparisons tend to evaluate qualitative over quantitative metrics. The broad categories of quantitative performance benchmarks/metrics for controllers include the following [8]:

- Throughput – These tests measure the number of flow requests the controller can process over time.
- Latency – Measures the delay or duration of time required for the controller to service a request from network devices.
- Flow – Flow metrics measure the efficiency of installing flows across multiple devices and characteristics of the traffic between the data source and destination.
- Topology – These benchmarks measure the time to discover the topology, including network devices, network hosts, and the links between them. They also include change detection time and maximum detection size.
- Threading – Threading tests evaluate the controller's usage of operating system and hardware resources.
- Other – The tests that do not fit into the above categories include the controller's flow table capacity, ping delay, electrical energy use, network re-provisioning

time, and controller fail-over time.

There are few pre-built tools for evaluating controller performance. The original open-source tool for evaluating OpenFlow controllers is Cbench [1]. It performs only throughput and latency tests with OpenFlow 1.0, which limits its usefulness. Many recent open-source extensions and tools expand the test types, but only expand to version 1.3 and are limited to specific controller implementations. One of these, OFCProbe, attempts to reduce virtualization overhead by simulating the environment in Java. Proprietary tools such as OFCbenchmark, OFBench, OFnet, and Pkt Blaster also have situational constraints [8]. Most have not seen development in the past five years. In addition, variation in tool implementations produces different results with the same controllers, making it difficult to evaluate them. Manual packet capture and custom analysis gives an equal playing field for comparison.

2.3.4 Software Defined Networking Tools

SDN emulators create networks running real switch and application code within a single system environment. They use various forms of virtualization to provide SDN controllers, switches, and host systems networked together within the host operating system. The virtualization methods are sufficiently lightweight, enabling scalable emulation of network devices without significant investment in physical hardware.

Mininet is an emulator developed in 2010 to rapidly develop and test SDN networks on a single laptop [10] and the most common emulator used by the SDN research community. It uses Linux container virtualization and therefore limited to that Operating System (OS) platform. More recent SDN emulation development has focused on transitioning to Docker container virtualization. These implementations have the emulated hosts in Docker, and some are extensions of Mininet. Examples include ContainerNet [11], vSDNemul [12], NestedNet [13], and ComNetsEMU [14]. Docker has increased scalability and thus makes these tools suitable for large wireless and sensor network emulation. Docker solutions are also dependent on the host operating system. In contrast, Network EMulatOr (NEMO) [15] emulates the network entirely within a Java virtual machine. This frees the operating system constraint, as Java is portable to many operating systems.

2.4 Related Research

The IETF has codified SDN controller evaluation in RFC 8456, Benchmarking Methodology for SDN Controller Performance [16]. It proposes four test categories that are protocol and controller independent, though there are OpenFlow specific references in its examples and figures.

The first category is performance and includes the following tests:

1. Network Topology Discovery Time – The time taken by the controller to completely discover the topology.

2. Asynchronous Message Processing Time – How long the controller takes to respond to an unsolicited message from a network device.

3. Asynchronous Message Processing Rate – Measures the number of non-trivial requests the controller responds to over a period.

4. Reactive Path Provisioning Time – The time to set up a path measured from the first flow provision request to controller to the last provision response sent from it.

5. Proactive Path Provisioning Time – The path set up time measured from the first northbound interface (application layer) request to the last path provisioning response sent to the network devices.

6. Reactive Path Provisioning Rate – The number of paths a controller can establish per second from southbound interface requests.

7. Proactive Path Provisioning Rate – The number of paths a controller can establish per second from northbound interface requests.

8. Network Topology Change Detection Time – The time delta from a controller receiving a topology change notification and sending a topology rediscovery request to the network devices.

The scalability category tests include:

1. Control Sessions Capacity – The number of sessions a controller can accept and maintain with network devices.

2. Network Discovery Size – The maximum discoverable number of nodes, links, and hosts.

3. Forwarding Table Discovery Size – The maximum number of flow table entries stored in the controllers forwarding table.

The two security tests are:

1. Exception Handling – Captures the effects of error handing and notifications during the Path Provisioning Rate, Path Provisioning Time, and Network Topology Change Detection Time tests.

2. Handling Denial of Service Attacks – Captures the effects of Denial-of-Service (DoS) attacks during the Path Provisioning Rate, Path Provisioning Time, Network Topology Change Detection Time, and Network Discovery Size tests.

The final category, reliability tests are:

1. Controller Failover Time – Used for distributed controllers and measures the time between active controller failure and the backup controller receiving the first rediscovery message.

2. NRT – Time taken by the controller to reroute flows due to traffic path failure measured from the first failure notification received to the last path-provisioning message sent to the network devices.

Most earlier controller performance evaluations performed throughput and latency tests [8]. An exception is Tello and Abohasan [17], who performed six of the RFC 8456 tests when it was still in draft status. They selected POX [18], a basic controller, OpenDaylight (ODL) [19], Open Network Operating System (ONOS) [20], Ryu [21], and Floodlight [22] for evaluation. Their performance portion included Asynchronous Message Processing Time and Rate, Network Topology Change Detection Time, and Reactive Path Provisioning Time. Scalability evaluations included Network Topology Discovery Time and Discovery Size. They used Cbench, Mininet, Docker, custom Python script, and WireShark packet capture. The parameters for the experiment are not well defined, such as controller, application, and protocol versions and settings. The results are presented in bar graphs, but the number of runs per test are not stated, nor statistically analyzed or compared.

Bah et al. compared ONOS and ODL without referencing RFC 8456, but performed tests equivalent to throughput or Asynchronous Message Processing Rate, Network Topology Discovery Time, and NRT [23]. Mininet emulated network topologies, Cbench generated control traffic, and iPerf [24] generated Transmission Control Protocol (TCP) test traffic. The NRT measurement tool was iPerf bandwidth reports, but the RFC implies using User Datagram Protocol (UDP) or other connectionless traffic to measure packet loss and performing packet analysis for NRT. Again, the specific parameters for experiment software components, number of runs, and statistical analysis are not well defined.

Silva et al. performed throughput and network discovery tests for Floodlight, ONOS, and Ryu working with home consumer grade switches [25]. They used RFC 8456 as a baseline and Mininet and iPerf3 for their test environment. Their reporting included many of the parameters required by the RFC. All tests were run at least 10 times as required by the RFC but the results did not state the exact number. The only test results were arithmetic means, with no further statistical analysis or comparison.

Using a single run, not specifying the number of runs, or not creating a model gives little statistical power to and confidence in the results. The arithmetic means may be numerically different but does not prove they are from different controller groupings or show how much variance a controller exhibits. Statistical analysis of larger samples creates higher confidence with predicted values and comparisons.

2.5 Background Summary

SDN technology offers advantages over traditional networks. Controllers are the heart of SDNs, and their interfaces to other components define their operation. The capabilities of SDNs of monitoring, management, load-balancing, network virtualization and orchestration, and policy enforcement were developed in smaller scale networks, but now apply to a wider range of network types including data center, Wide Area Network (WAN)s, and IoT. Platforms for emulation of SDN are prevalent, but the variation in interfaces and applications makes controller performance evaluation difficult. The IETF has created an RFC in an effort to standardize controller testing. Some, but not all, of the tests have been tried, and opportunities to validate statistical results appear promising.

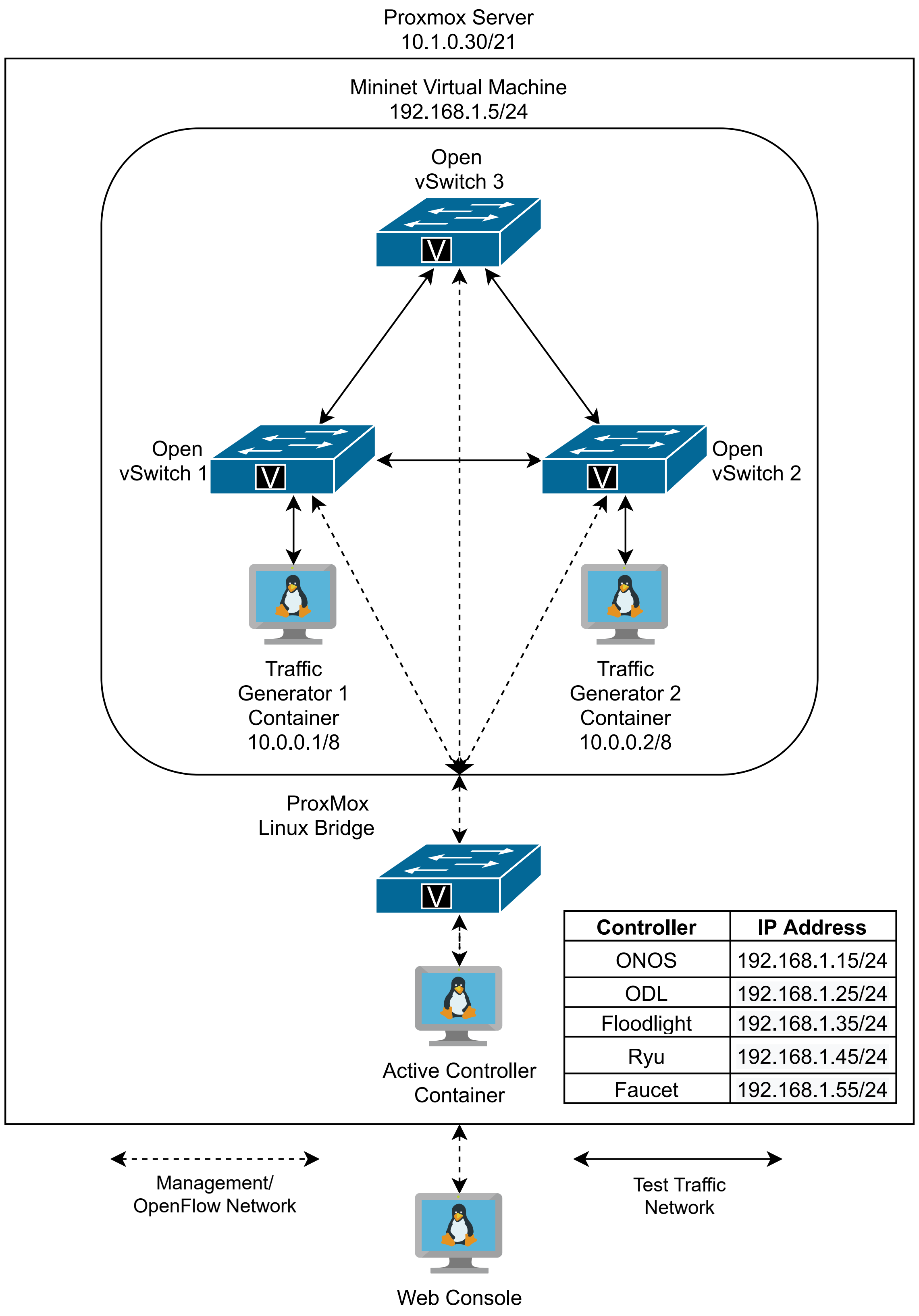
III. Network Re-Provisioning Test Environment Design

3.1 Overview

This chapter provides a detailed description of the test environment developed to support OpenFlow SDN controller NRT evaluation thesis research. A major component is a virtualization server containing the entire test environment for all other components. Second, Mininet VM has the tested network topology, providing the network devices and interfaces as defined in Fig. 1. The Mininet VM also contains the experiment scripts. The final components are selected OpenFlow controller VMs, providing the controller and application plane roles as shown in Fig. 1. Finally, the chapter describes tools used in script development.

3.2 System Summary

The experimental system purpose is evaluating open source OpenFlow network controller network re-provisioning time methodology under RFC 8456 specifications. A single server using Linux container and Kernel-based Virtual Machine (KVM) virtual services contains the entire experiment. A web browser equipped computer connects to the server and provides VM management access. A Mininet virtual machine hosts the test network topology including three interconnected virtual switches. It also hosts two network traffic generating VMs, each connected to a virtual switch in the test topology. The topology, as illustrated in Fig. 2 creates two alternate paths for traffic between the generators. One is the primary, and the other is the alternate target for re-provisioning. It provides a separate control network between the switches and OpenFlow controllers as shown in Fig. 2 for control communication. Packet capture software on the Mininet VM allows capture of both the control and data traffic as needed. Individual Linux containers host and isolate each controller.



Controller	IP Address
ONOS	192.168.1.15/24
ODL	192.168.1.25/24
Floodlight	192.168.1.35/24
Ryu	192.168.1.45/24
Faucet	192.168.1.55/24

Figure 2. Test System Design

3.3 Detailed Description

The following describes major system components in detail. They are the ProxMox Virtual Environment (VE) server, Mininet virtual network environment, and the evaluated OpenFlow controllers.

3.3.1 ProxMox Server

ProxMox VE 6.3 is a Debian Linux based platform with two different operating system virtualization methods. The first is Linux Container (LXC), which is very lightweight, and allows for rapid re-initialization of the controllers. It also has a traditional hyper-visor, KVM. The Mininet system subsystem is run in a KVM VM and has internal Linux containers for devices in the virtual topology. Nesting containers is a possibility, but not implemented at this time. A Hewlett Packard Z840 workstation with an Intel Xeon E5-2687W Central Processing Unit (CPU) at 3.10GHz with twenty cores and 64 GB of Error Correction Code (ECC) RAM is the host system. This supplies enough resources to prevent system starvation, which could affect the results.

3.3.2 Network Virtual Machine

The network VM uses Ubuntu Linux 20.04.2 Server Long Term Support (LTS) for the operating system. The server version has fewer default packages installed than the desktop version, consuming fewer resources. Ubuntu LTS versions are the most stable versus non-LTS releases with new features that may have unknown bugs. The network VM has eight virtual CPU cores, 32GiB of RAM, and 1TiB of hard drive storage assigned. This provides enough computing resources to perform the test, store large packet captures, and capture analysis.

The virtual machine hosts the Mininet application, test script, virtual switches, hosts, and links between them. Appendix A provides the test script and configuration information. It also connects the virtual switches to the controllers under test. The Mininet application is a Command Line Interface (CLI) Python script and is a standard in the SDN community. It is used for rapid prototyping of software-defined networks by emulation on a single system. The latest stable version, 2.3.0, is used for compatibility with Python 3.8.5 installed on the system.

3.3.2.1 Switches

Open vSwitch is an open-source virtual switch that is OpenFlow compatible, cutting the need for hardware, and can work with the test controllers. It also allows for rapid stand up and teardown of virtual network topologies. The switches use the latest stable version released for Ubuntu, 2.13.1.

3.3.2.2 Traffic Generating Hosts

As traffic generating hosts are containers within the Mininet VM, they use the same Ubuntu 20.04.2 Server LTS operating system. The iPerf3 bandwidth testing CLI utility generates the test traffic. It provides UDP traffic with sequence numbers for measuring and reporting packet loss. The configured IPerf3 reporting also shows traffic bandwidth at 0.1-second intervals, the smallest capable for the tool, to see fluctuations and help detect re-provisioning.

3.3.2.3 Topology

RFC 8456 Section 4.1 strongly suggests using spine and leaf topology for testing, which has multiple, equal cost redundant paths between network switches in a partial mesh. The controller under test can select any path for the initial traffic flow in either direction. This requires either automatically detecting the primary path the controller selected or selecting a different topology with unequal cost paths to force path selection. Custom interfaces for each controller are needed to implement the first option, due to nonstandard NBIs. This experiment uses OpenFlow enabled switches in a small mesh topology option as shown in Fig. 2. It should force the link between switches 1 and 2 as the primary traffic path with fewer network hops between the traffic generators. Links interconnecting the switches to each other and from the switches to hosts have typical 1 Gbps LAN backbone speeds.

3.3.2.4 Experiment Scripts

The main experiment script written in Python 3, `test.py`, relies on Mininet Python 3 classes and methods. It requires the Internet Protocol (IP) address of the controller under test and run number as arguments, for logging the test start including the date and time. Then it generates the virtual network topology and connects it to the controller. Next, it calls a host location advertisement script, `gratuitousARP.py`. Normal Address Resolution Protocol (ARP) operation has a host requesting the address of a destination, to which it receives a reply from the destination with it's address. A gratuitous ARP broadcasts an unsolicited reply with the host's address, which advertises the traffic generators' presence to the OpenFlow controller. The Python 2 host advertisement script comes from the ONOS controller tutorial VM and was not refactored in Python 3. The current version of the test and host advertisement scripts are in Appendix A and B. After verifying connectivity between the traffic generators, the experiment script initiates packet capture and starts test traffic on the traffic generators, and triggers network re-provisioning. Finally, it stops traffic generation, packet captures, Mininet topology, logs the end of the test, and cleans up any leftover topology.

3.3.3 Controllers

Running all controllers in separate containers avoids resource competition. The containers also allow for fast re-initialization from virtual snapshot backups, quickly re-initializing controllers without topology data from earlier test runs. This forces a reset to the primary traffic path, after the controller re-provisioned to the alternate in the previous run. The containers are built off a ProxMox Ubuntu 20.04.2 Server LTS template. Each is given four virtual CPUs, 8GiB of Random Access Memory (RAM), and 16GiB of storage for mid-range capability. The only differences are the controller software and supporting language packages.

ONOS 2.5.1, ODL 8 (Oxygen), Floodlight 1.2, Ryu 4.34 and Faucet 1.9.53 [26] are the test controllers. All are open source and freely available for testing. Additionally, they are capable of OpenFlow protocol version 1.3 or higher, as 1.3 is a de-facto standard for OpenFlow switches and have wide market availability. Most have commercial backing, updates within the past three years, and maintain fair to good documentation [8]. These characteristics are desirable for a SDN controller in operational status for many years.

The test requires Open Systems Interconnection (OSI) model layer 2 switching capabilities, and SDN controller ability to address switching loops. Also, the controllers must be capable of ARP, and Internet Control Message Protocol (ICMP) ping requests, verifying controller flow creation between the traffic generating hosts. Last, they need a topology tracking method for re-provision event detection. Each controller has additional applications enabled to support these features, if not enabled by default. ODL Oxygen is two years older than the current Aluminum release during testing but is the newest with the required layer 2 switching application available.

3.4 Development Tools

The test script was developed in Visual Studio Code 1.5.3, and Microsoft Python and Pylance extensions for Studio Code ensured proper syntax and aided debugging.

3.5 Design Summary

This chapter presents the overall design of the network re-provisioning test bed in line with RFC 8456. Individual components are detailed with justifications for selection and configuration. It also presents the software tools used for script development.

IV. Methodology

4.1 Objective

The primary purpose of this methodology is evaluating RFC 8456's published protocol for measuring for measuring and reporting SDN controller NRT and network re-provisioning packet loss. The secondary objective is identifying and comparing those metrics between the tested controllers. The first supporting objective involves setting up and verifying a virtual SDN test topology. The second is to capture 60 seconds of UDP test traffic before, during, and after a SDN re-provision event on that topology and measure total UDP packet loss for the duration. The third objective is to parse the capture files, and identify the metrics needed to determine the network re-provisioning times. The fourth is to determine the expected NRT, Forward Direction Packet Loss (FDPL), and Reverse Direction Packet Loss (RDPL) for the controllers in the test and determine if significant differences exist between them.

4.2 System and Component Under Test

The network re-provisioning methodology is the system under test as shown in Fig. 3. The component and single factor under test is the OpenFlow SDN controller. The parameter categories are divided into system and iPerf3 traffic workload. Controller and test environment parameters fall under the system category. Finally, the controller NRT and packet loss are the desired metrics produced by the test.

4.2.1 Assumptions

The experiment design has two important assumptions. First, every controller under test selects the link between Open VSwitch 1 and 2 as the primary path. The test topology makes that link desirable as the primary but does not guarantee

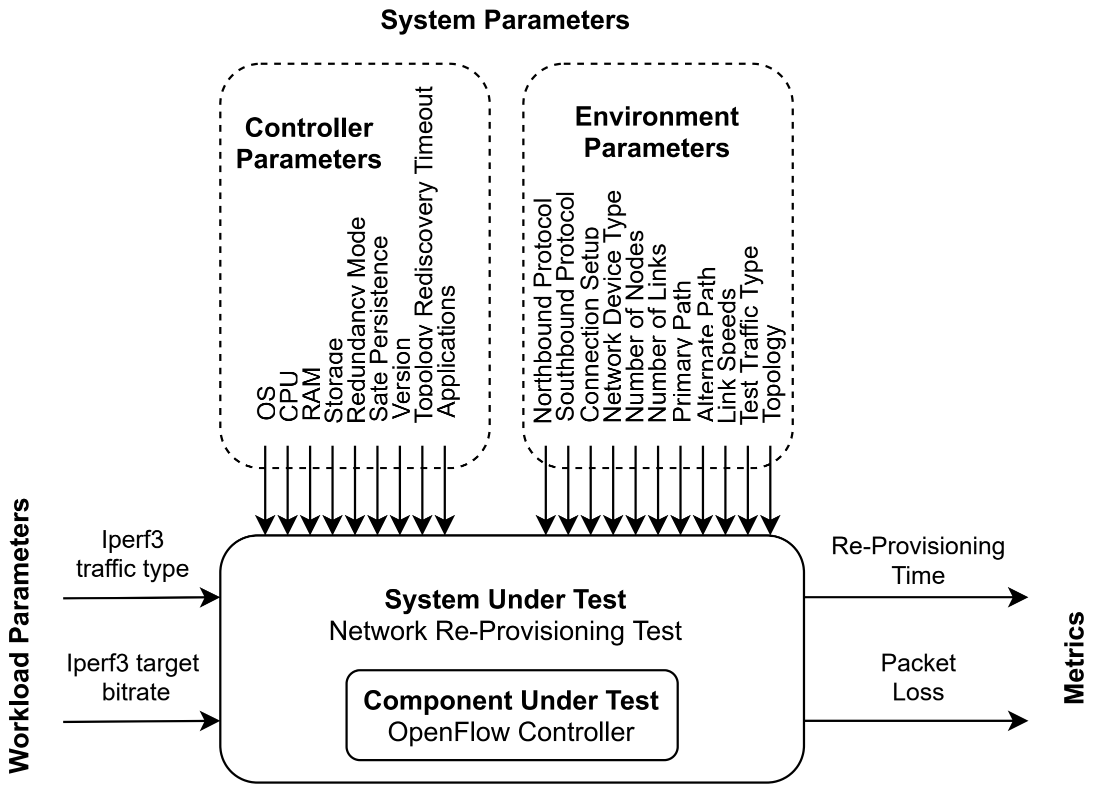


Figure 3. System Under Test

selection as required by the RFC. The second assumption is that the default topology timeouts, for controllers that have them, are acceptable. The test topology is small enough for quick discovery. The rediscovery timeout becomes relevant in larger and complex topologies.

4.3 Metrics

As specified in RFC 8456 NRT objective, the experiment measures "the time taken by the controller to reroute traffic when there is a failure in existing traffic paths, defined as the interval starting with the first failure notification message received by the controller and ending with the last flow re-provisioning message sent by the controller at its southbound interface" [16]. However, NRT measurement in the methodology is through test traffic response, and not control messaging traffic. The time between the last frame successfully received by Traffic Generator 2 from Traffic Generator 1 before path loss and the first successfully received frame after path restoration is the Forward Direction Path Re-Provisioning Time (FDRT). The Reverse Direction Path Re-Provisioning Time (RDRT) is the opposite: the time between the last successful frame received by Traffic Generator (TG)1 from TG2 before path loss, and the first successful frame after path loss. Averaging the FDRT and RDRT calculates the NRT. $NRT = (FDRT/RDRT)/2$ This accounts for asymmetrical or equally practical forward and reverse paths. Missing frame sequence number counts between received UDP packets determines the RFC required FDPL and RDPL

4.4 Factors

The key factor in this experiment is the specific controllers under test, including their underlying programming languages as shown in Table 2. Each controller is a treatment for the experiment.

Controller	**Language**
ONOS	Open JDK 11
ODL (Oxygen)	Open JDK 8
Floodlight	Open JDK 8
Ryu	Python 3.8
Faucet	Python 3.8

Table 2. Experiment Factors

4.5 Uncontrolled Variables

The primary uncontrolled variables include the resource load of the operating systems of the virtual environment, console system, virtual machines, and containers. Unrelated system tasks may interfere with controller processing. To mitigate those potential effects, the environment is given abundant physical resources to take care of such tasks, and not starve the experiment processes.

4.6 Experiment Parameters

The mandatory experiment parameters required by RFC 8456 separate into test environment and controller configuration categories. Table 3 lists the test environment parameters. All the controllers selected use REST as NBI protocol, and OpenFlow 1.3 (the de facto standard) for SBI protocols. The connections between the controllers and switches are un-encrypted to simplify the experiment, and not interfere with the capture of traffic. Virtual switches allow for fast setup and tear down of the virtual topology between multiple runs. The small switch mesh topology creates two traffic paths for the re-provisioning test, and one less link for one path, encouraging its selection as the primary.

The generated iPerf3 traffic defines the workload parameters. IPerf's UDP bandwidth test provides packets with sequence number increments in the payload, and missing sequence number counts for packet loss measurement. Link speeds are constrained to one Gbps to keep the virtual switches and hosts from consuming the entire internal host bandwidth, preventing fluctuations that can affect the test. The iPerf3 UDP bandwidth test is set for one Gbps for 100% link saturation. This should increase packet loss detection during the re-provisioning event.

Table 4 lists the common parameters shared by all the controllers. The system configuration gives each controller an average amount of computing power. To minimize the time required for start up and shutdown of controllers between experiments, controller implementation is non-distributed. This means there is no controller redundancy. Controller state persistence is disabled to prevent information from previous experiments carrying over to subsequent experiments. The controllers re-detect the topology for every run without prior knowledge.

Environment Parameter	**Setting**
Northbound Protocol	REST
Southbound Protocol/Version	OpenFlow 1.3
Connection Setup	un-encrypted
Network Device Type	Virtual Switch
Number of Nodes	5
Number of Links	5
Number of Links in Primary Path	3
Number of Links in Alternate Path	4
Link Speeds	1 Gbps
Forwarding Plane Test Traffic Type	1 Gbps iPerf UDP bandwidth test
Topology	3 switch mesh

Table 3. Environment Parameters

Controller Parameter	**Setting**
System Configuration	Ubuntu Linux 20.4.2 container with 4 CPU, 8G RAM, 16 GB hard drive, virtual network interface
Redundancy Mode	None
State Persistence	Disabled

Table 4. Common Controller Parameters

Table 5 lists parameters specific to each controller and applications activated in addition to those installed by default to enable required capabilities. The controllers' documentation does not always specify the rediscovery timeout value. ONOS has a Logical Link Discovery Protocol (LLDP) probe interval of 3 seconds. ODL's built-in topology discovery and layer 2 switch application documentation indicate LLDP used for loop removal, but not active probing. The ODL lldp-topology-discovery application defaults to a 5-second default timeout but was not active during testing. Floodlight's link discovery code [27] and status messages indicate a 15 second LLDP probe rate. The Ryu simple switch application uses Spanning Tree Protocol (STP), which is purely reactive, and STP can take 30 to 50 seconds to converge on a solution. Faucet documentation states it has LLDP but does not use it for topology detection.

It does claim to use another unspecified method. Faucet requires the topology defined in its configuration files. It is based on the Ryu framework, likely uses the Ryu STP library to handle changes in a similar manner. Open vSwitch 1 was set as the primary switch to avoid Open vSwitch 3 becoming the root and defining the alternate path as primary.

Controller	**Version**	**Rediscovery Timeout**	**Applications**
ONOS	2.5.1	3 s	Default Apps, Host Location Provider, LLDP Link Provider, OpenFlow Base Provider, OpenFlow Provider Suite, Proxy ARP/NDP, Reactive Forwarding
ODL	8	Not Specified (5 s when LLDP enabled)	Default Apps, ODL Layer 2 Switch, DLUX
Floodlight	1.2	15 s	Default Apps
Ryu	4.34	Not Specified	Default Apps, Simple Switch STP 1.3, Web Socket Topology
Faucet	1.9.53	Not Specified	Default Apps

Table 5. Specific Controller Parameters

4.7 Experiment Design

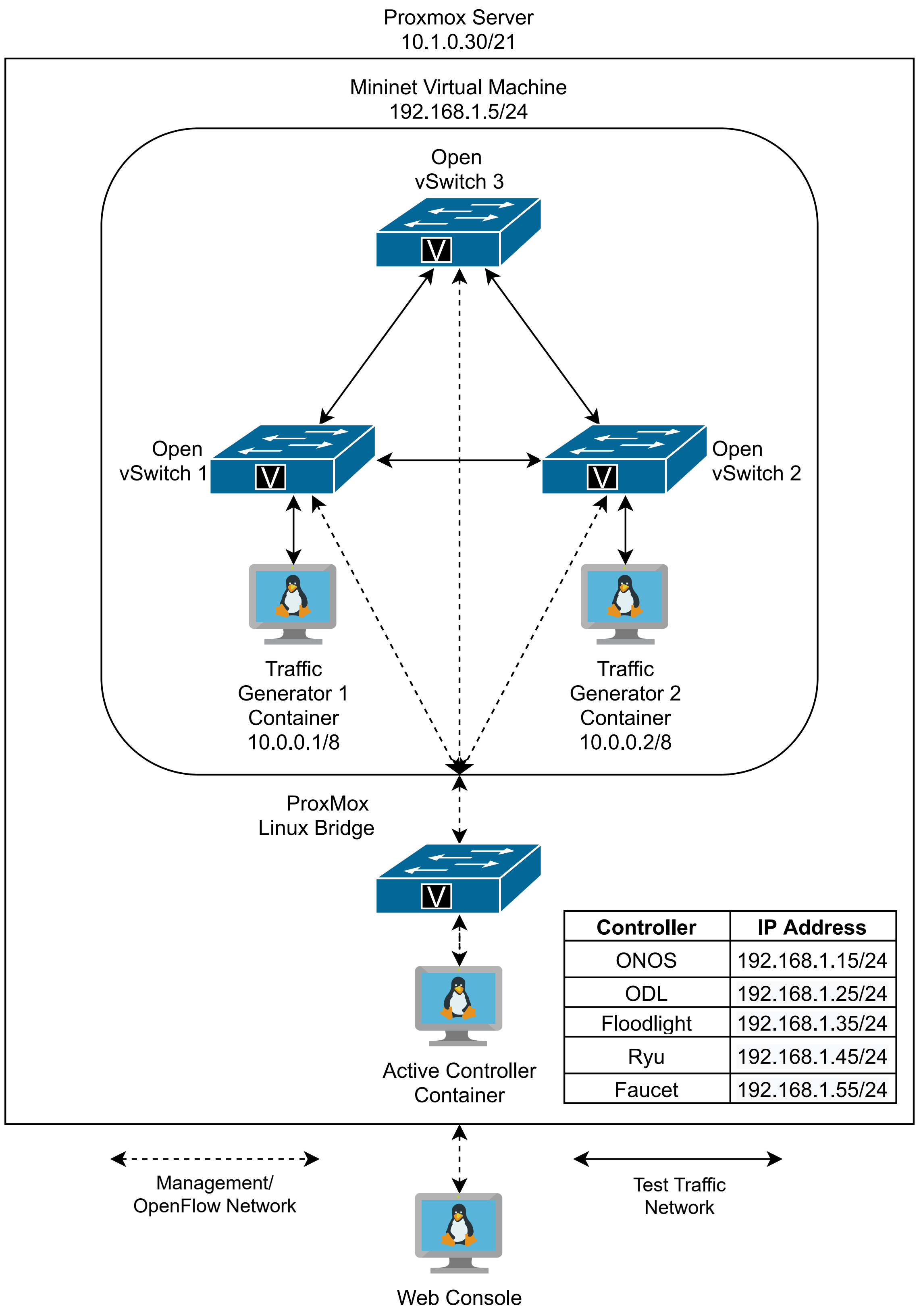
This section outlines RFC 8456's general methodology requirements, and how the implemented experiment design in Fig. 4 and methodology meet those requirements. The RFC has no requirements on data analysis.

4.7.1 RFC 8456 Network Re-provisioning Time Methodology

RFC 8456 specifies three prerequisites for measuring NRT. The first is that a network with specified number of nodes and redundant paths must be deployed. The second is that the controller must know the location of the traffic generating hosts. Last, the controller must not pre-provision the alternate-path as the primary.

The required steps are:

1. Send bidirectional traffic continuously with unique sequence numbers from test traffic generators TP1 and TP2.
2. Disable a switch or link in the primary traffic path.
3. Stop the trial after receiving the first frame after re-convergence.
4. Record the time of the last received frame prior to the frame loss at test traffic generator TP2 (TP2-Tlfr) and the time of the first frame received after the frame loss at test traffic generator TP2 (TP2-Tffr). According to the RFC, there must be a gap in sequence numbers between these frames.
5. Record the time of the last received frame prior to the frame loss at test traffic generator TP1 (TP1-Tlfr) and the time of the first frame received after the frame loss at test traffic generator TP1 (TP1-Tffr).



Controller	IP Address
ONOS	192.168.1.15/24
ODL	192.168.1.25/24
Floodlight	192.168.1.35/24
Ryu	192.168.1.45/24
Faucet	192.168.1.55/24

Figure 4. Test System Design

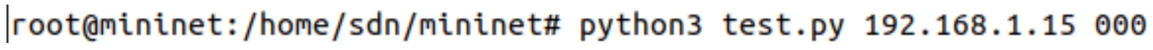
4.7.2 Implemented Design

JMP 15 Pro [28] statistical software generates a random run order with 30 replicates for each controller. The random run order helps mitigate random unknown variables, and 30 replicates increases the accuracy by the central limit theorem over the 10-minimum specified in the RFC.

4.7.2.1 Test Runs

Before each test run, the selected controller manually boots from a container snapshot backup via the ProxMox web console. The test holds until ProxMox server indicates the container is online. If the controller is not set up as a service on boot, recalled commands from the CLI history start it. This ensures the same launch commands start the controllers during their runs.

The experiment script, `test.py`, is invoked in Python 3 on the Mininet VM within the mininet directory, as shown in Fig. 5 with the static controller IP address and run number as arguments. The script runs under root privileges to ensure permissions to necessary resources. Before executing, the controller and run number are verified against the run list in JMP. The script logs the run number, controller IP address, and start date/time for verification during analysis. Next, it constructs the Mininet VM virtual topology as shown in Fig. 4. Mininet waits until all switches connect to the controller, then pauses for 30 seconds to allow for topology detection, and switch configuration. Initial testing showed the controllers using STP algorithm took approximately 30 seconds to detect the switching loop and determine the spanning tree. The topology meets the first RFC prerequisite of a set number of nodes with redundant paths.



```
root@mininet:/home/sdn/mininet# python3 test.py 192.168.1.15 000
```

Figure 5. Test.py Launch

The experiment script then invokes,`gratuitousARP.py`, the host advertisement script, on both TG hosts, to make the switches and controller host tracker aware of their presence for flow building. The script then invokes the Mininet `pingAllFull` method, causing every pair of hosts to ping each other three times. This verifies the controller successfully set up flows between the hosts, before test traffic begins, and serves as a backup the host advertisement script. This satisfies the second RFC prerequisite. If the pings are not successful by the third attempt, something has gone wrong with the topology or controller, and the run is manually aborted. The setup does not fully meet the third RFC prerequisite, as one path is more desirable, but does not guarantee its selection as primary.

Otherwise, the script terminates any hung iPerf processes from the previous run. Sixty-five second TShark packet captures are initiated on the traffic generators with a capture filter for UDP port 5001 on TG1 and port 5002 on TG2. The filter minimizes the capture file sizes and focuses on the stream of packets received by the iPerf3 servers. The 65 second capture period encapsulates the entire 60 second UDP streams. Packet capture file names contain run number and TG host names for verification during analysis.

An iPerf3 server is started on UDP port 5001 for TG1, as well as on port 5002 for TG2 to differentiate the traffic streams in the packet captures. The iPerf3 servers log the bandwidth for the connections at the shortest interval available in the tool, 0.1-seconds. The bandwidth logs aid in identifying the re-provision event and recording the connection packet loss. The script saves the output to filenames with the run and port numbers. This satisfies the required traffic generation step.

The switches communicate with the controller to determine the best path for the traffic. The path between OpenFlow switches 1 and 2 is more desirable for the controller to select, due to fewer hops, and becomes the primary path. A 3-second pause allows for flow installation and the connections to reach a steady state across the network before triggering re-provision. It triggers when the script administratively disables the link between OpenFlow Switches 1 and 2 after the 3-second pause, meeting the second required step. This forces both switches to notify the controller of the topology change. The controller determines the best alternate path and pushes new flow table entries to the switches to restore traffic flow. Previous testing with TCP traffic in the test topology identified 3 seconds as the approximate time for the slow start process to finish and reach link saturation. UDP traffic reaches link saturation faster than TCP; maintaining the pause allows for both types of traffic in future testing.

The script pauses for 60 seconds, allowing the iPerf3 traffic to end and TShark to complete packet capture. This does not fully satisfy the requirement of ending the run upon the first frame received after re-provision. If re-provisioning happens after the packet capture stops, or not at all, then the test does not detect the NRT. The script then tears down the topology and runs a cleaning process to remove any lingering traces before the next run. Finally, the script logs the controller IP address, run number, and test stop date/time. The controller container under test is rolled back to the previous clean snapshot. The rollback procedure also shuts down the container, which keeps only one controller active on the network at a time. Only one controller is online during each run to prevent other controllers' interference.

Manual inspection of the packet capture file sizes after each run help detect errors. If the capture is 384 bytes, then it is empty, invalidating the run. The capture file size from both TGs should be similar. Large file size differences indicate possible errors and are noted for further analysis. Checking the run log against the JMP list verifies correct controller usage. The packets in the capture are date/timestamped which verifies they fall between the start and end times in the log. Performing a virtual snapshot of the Mininet VM every five runs, provides data backup.

IPerf3 report file sizes are inspected after each run. The sizes for the forward and reverse directions should be similar in size. If not, something went wrong with iPerf3 or the re-provision. Inspection of the report content after the runs complete, also helps find any execution errors.

4.7.2.2 Metric Extraction

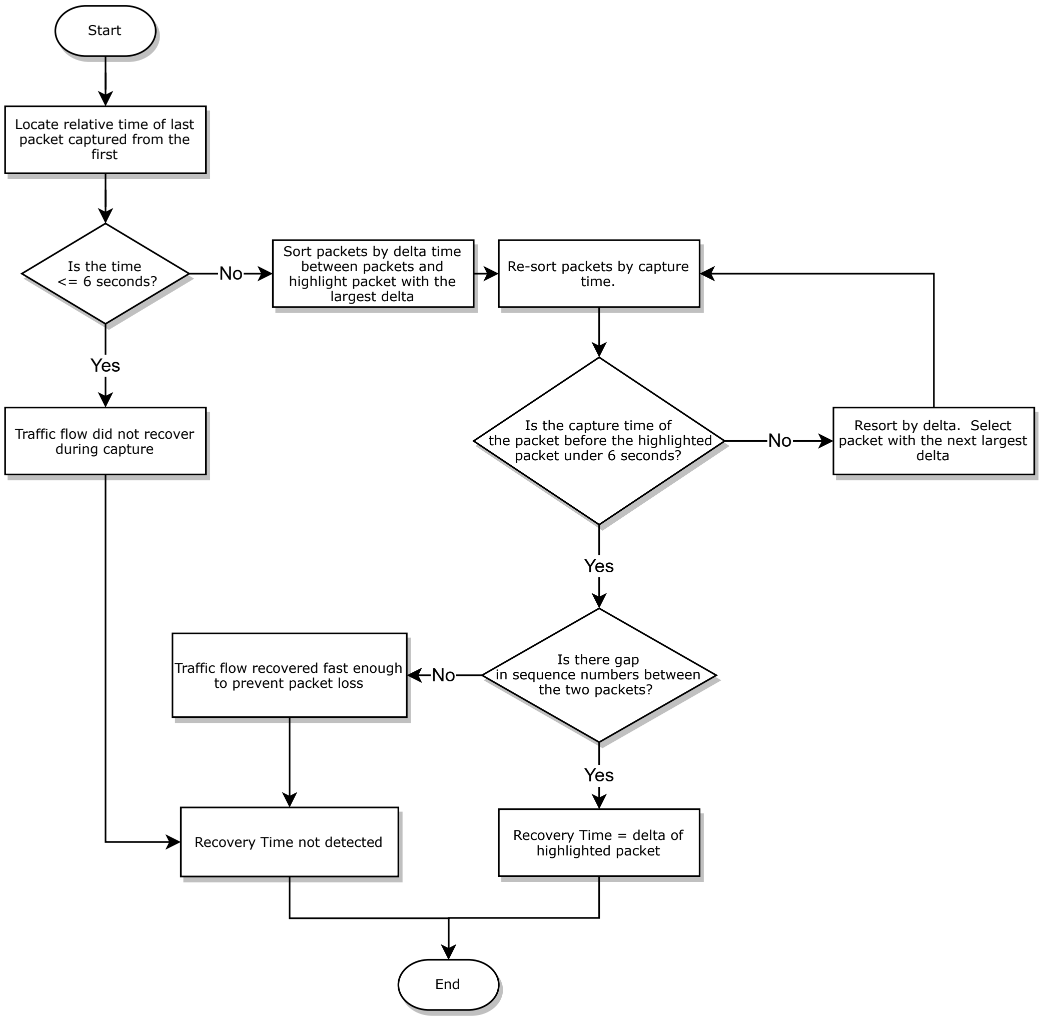
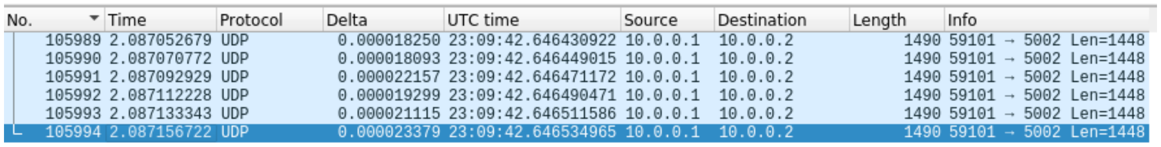

Start
Locate relative time of last packet captured from the first
Is the time <= 6 seconds?
No
Sort packets by delta time between packets and highlight packet with the largest delta
Re-sort packets by capture time.
Yes
Traffic flow did not recover during capture
Is the capture time of the packet before the highlighted packet under 6 seconds?
No
Resort by delta. Select packet with the next largest delta
Yes
Traffic flow recovered fast enough to prevent packet loss
No
Is there gap in sequence numbers between the two packets?
Yes
Recovery Time not detected
Recovery Time = delta of highlighted packet
End

Figure 6. Manual Recovery Time Extraction Process

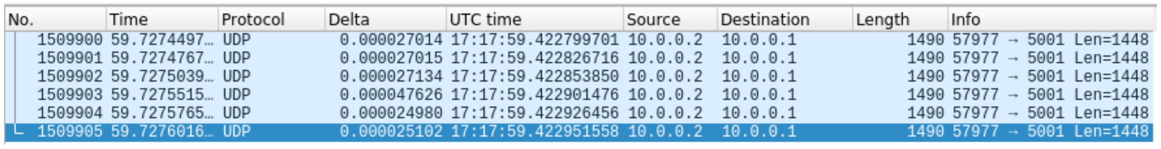
The FDRT and RDRT are extracted for steps 4 and 5 through manual inspection of packet captures in WireShark. The process flow is shown in Fig. 6 and performed for both directions of traffic flow for each run. Adding a column displaying the *delta* in capture time between packets is key to the process. First, the time difference between the first and last packet is checked. If under 6 seconds, the traffic flow did not recover before the end of the 60-second packet stream or did not recover at all. An example of the last packet captured in WireShark for a run is highlighted in Fig. 7. The packet's capture time relative to the first packet captured, is approximately 2 seconds. The re-provision event should normalize around 3 seconds in the packet capture, as the script waits to trigger it 3 seconds after traffic starts.



```
No.      Time        Protocol  Delta        UTC time            Source    Destination  Length  Info
105989 2.087052679 UDP        0.000018250 23:09:42.646430922 10.0.0.1  10.0.0.2      1490 59101 → 5002 Len=1448
105990 2.087070772 UDP        0.000018093 23:09:42.646449015 10.0.0.1  10.0.0.2      1490 59101 → 5002 Len=1448
105991 2.087092929 UDP        0.000022157 23:09:42.646471172 10.0.0.1  10.0.0.2      1490 59101 → 5002 Len=1448
105992 2.087112228 UDP        0.000019299 23:09:42.646490471 10.0.0.1  10.0.0.2      1490 59101 → 5002 Len=1448
105993 2.087133343 UDP        0.000021115 23:09:42.646511586 10.0.0.1  10.0.0.2      1490 59101 → 5002 Len=1448
105994 2.087156722 UDP        0.000023379 23:09:42.646534965 10.0.0.1  10.0.0.2      1490 59101 → 5002 Len=1448
```

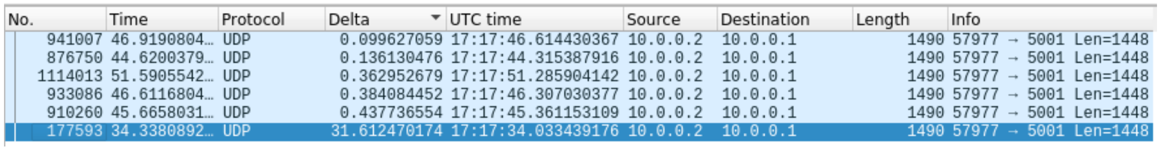
Figure 7. Re-provision Failure in Capture

If the difference is near 60 seconds, the flow recovered. Fig. 8 shows an example run with the final packet capture around 59 seconds. The capture is then sorted on the *Delta* column. The packet with the highest *delta* is highlighted in Fig. 9. In the example, the difference between time of capture and the *delta* from the previous packet is approximately 3 seconds.



```
No.      Time          Protocol  Delta        UTC time            Source    Destination  Length  Info
1509900 59.7274497… UDP        0.000027014 17:17:59.422799701 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
1509901 59.7274767… UDP        0.000027015 17:17:59.422826716 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
1509902 59.7275039… UDP        0.000027134 17:17:59.422853850 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
1509903 59.7275515… UDP        0.000047626 17:17:59.422901476 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
1509904 59.7275765… UDP        0.000024980 17:17:59.422926456 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
1509905 59.7276016… UDP        0.000025102 17:17:59.422951558 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
```

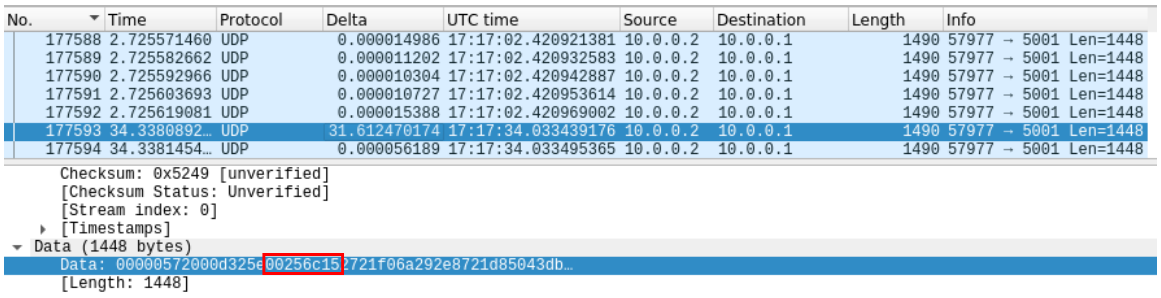
Figure 8. Re-provision Success in Capture



```
No.      Time          Protocol  Delta         UTC time            Source    Destination  Length  Info
 941007 46.9190804… UDP        0.099627059 17:17:46.614430367 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
 876750 44.6200379… UDP        0.136130476 17:17:44.315387916 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
1114013 51.5905542… UDP        0.362952679 17:17:51.285904142 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
 933086 46.6116804… UDP        0.384084452 17:17:46.307030377 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
 910260 45.6658031… UDP        0.437736554 17:17:45.361153109 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
 177593 34.3380892… UDP       31.612470174 17:17:34.033439176 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
```

Figure 9. Delta Selection

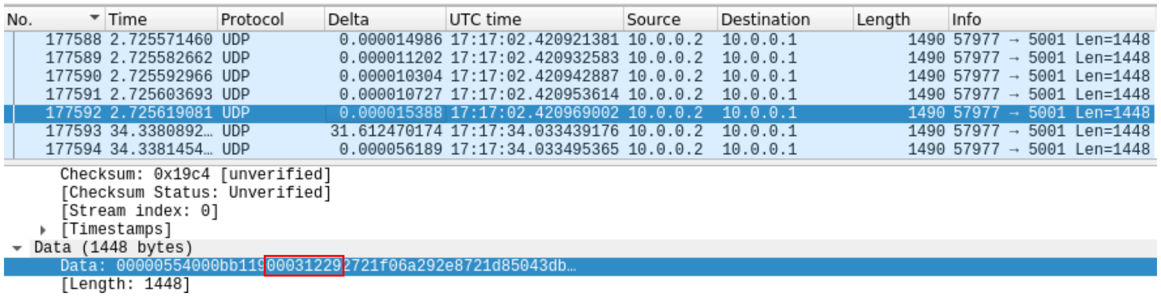
The capture is then re-sorted by time. The hexadecimal sequence number in the data field is noted as shown in Fig. 10. The capture time of the previous packet is checked against the 6 second threshold. In Fig. 11, the capture time is around 3 seconds as expected. If not under the threshold, the capture is resorted by *delta*, and the next highest *delta* packet selected. The process is repeated until previous packet capture time falls under the threshold. The hex sequence number in the data field of both packets is compared for a difference greater than zero, indicating a sequence gap. In Fig. 10 and 11 the gap is $0x256c15 - 0x31229 = 0x2259 = 2,251,244$ packets. If a gap is found, the *delta* is recorded as the re-provision time. If there is no gap, either the re-provision happened fast enough there was no sequence gap, or the alternate route might have been chosen as the primary, requiring no re-provision for that traffic flow direction. In these cases the NRT is not detected.



```
No.      Time         Protocol  Delta        UTC time              Source    Destination  Length  Info
177588 2.725571460  UDP      0.000014986 17:17:02.420921381 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
177589 2.725582662  UDP      0.000011202 17:17:02.420932583 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
177590 2.725592966  UDP      0.000010304 17:17:02.420942887 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
177591 2.725603693  UDP      0.000010727 17:17:02.420953614 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
177592 2.725619081  UDP      0.000015388 17:17:02.420969002 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
177593 34.3380892…  UDP     31.612470174 17:17:34.033439176 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
177594 34.3381454…  UDP      0.000056189 17:17:34.033495365 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448

        Checksum: 0x5249 [unverified]
        [Checksum Status: Unverified]
        [Stream index: 0]
      ▸ [Timestamps]
    ▾ Data (1448 bytes)
        Data: 00000572000d325e00256c152721f06a292e8721d85043db…
        [Length: 1448]
```

Figure 10. First Packet After Re-provision



```
No.      Time         Protocol  Delta        UTC time              Source    Destination  Length  Info
177588 2.725571460  UDP      0.000014986 17:17:02.420921381 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
177589 2.725582662  UDP      0.000011202 17:17:02.420932583 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
177590 2.725592966  UDP      0.000010304 17:17:02.420942887 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
177591 2.725603693  UDP      0.000010727 17:17:02.420953614 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
177592 2.725619081  UDP      0.000015388 17:17:02.420969002 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
177593 34.3380892…  UDP     31.612470174 17:17:34.033439176 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448
177594 34.3381454…  UDP      0.000056189 17:17:34.033495365 10.0.0.2  10.0.0.1      1490 57977 → 5001 Len=1448

        Checksum: 0x19c4 [unverified]
        [Checksum Status: Unverified]
        [Stream index: 0]
      ▸ [Timestamps]
    ▾ Data (1448 bytes)
        Data: 00000554000bb119000312292721f06a292e8721d85043db…
        [Length: 1448]
```

Figure 11. Last Packet Before Re-provision

4.7.2.3 Statistical Analysis

The metrics are assessed with JMP 15 Pro statistical software. The mean, standard deviation, and upper and lower 95% confidence limits for each controller are calculated and displayed with box plots and tables. Parametric comparison of the metrics has more statistical power than non-parametric, but the data must pass assumption of normality checks. The JMP Fit Model tool generates a predicted value for each metric, and then saves the difference between the metric value from each run and the predicted value called the residual. The residuals must be normally distributed to pass assumption of normality. The residuals are displayed in a normal quantile plot for a visual check of normality and followed up with the distribution fit tool and goodness-of-fit tests for statistical verification. If the data does not pass assumption of normality, non-parametric tests are used instead of parametric for comparison and controller rankings. Tables, box plots, and test results are displayed accordingly.

4.8 Methodology Summary

This chapter covered the objective to evaluate RFC 8456 NRT methodology with four OpenFlow SDN controllers. The metrics, factors, uncontrolled variables, parameters, and design were defined with assumptions, deviations, and additions. The metric extraction and analysis processes were outlined.

V. Results and Analysis

5.1 Overview

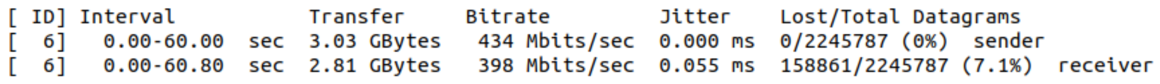
This chapter evaluates the implemented methodology and the statistical analysis of the controller metrics. Section 5.2 reviews the initial screening of iPerf reports, which eliminated one controller from further analysis. Section 5.3 details findings from packet capture metric extraction. Finally, Section 5.4 and 5.5 analyze the NRT, and associated packet loss for the four remaining controllers.

5.2 IPerf Reports

During the experiment, iPerf collected metrics and produced a summary report for each run as shown in Figure 12. From Figure 12a, observe that controllers with fast re-provisioning times tended to have lower packet loss within the 60-second test period. Lower packet loss also correlated with larger packet capture file sizes. As iPerf transmitted more packets successfully, TShark captured more packets. Controllers with slower re-provisioning times tended to have more packet loss with less data, as seen in Fig. 12b, and smaller packet capture file sizes. ODL did not follow these trends. During test runs, ODL's packet capture file sizes were on the smaller end of the spectrum. This initially suggested long re-provisioning times. However, ODL's iPerf reports revealed lower packet loss, suggesting shorter re-provisioning times. Further investigation of the iperf3 reports, showed the other controllers exhibited 400-500 Mbps bitrate for their test runs. For ODL, the bitrate started at similar levels to the other controllers, but dropped to 200-300 Mbps within 3 seconds, suggesting a re-provision event occurred. However, the bandwidth did not recover to the prior bitrate, and remained at that lower level throughout each run. Inspection of the packet captures showed no sequence number gaps between the last packet received before re-provision and the first packet after. Sequence gaps did start appearing near the end of the runs. These conditions did not meet the NRT definition, therefore the NRT remained undetected for all ODL runs.

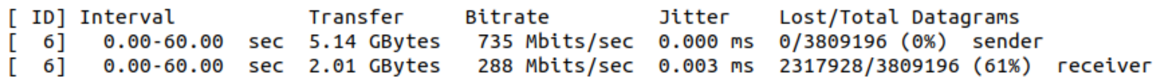
The iperf3 client reported fewer packets sent, as shown on the sending side of Fig. 12c, indicating possible traffic shaping. Because these indicators were found in every test run, with the NRT not detected, ODL was excluded from further analysis. Bah et al. noted ODLs bandwidth limitations [23], and Tello et al. stated the L2Switch application does not install flows to the switches [17], in contrast to the flow installation process described in Section 2.3.1.1. The ODL controller receives, processes, and instructs the switch how to forward each packet, instead of installing flow instructions so the switch knows how to forward subsequent packets.

Before re-provision, a packet travels to the first switch, to ODL, back to the first switch, to the second switch, back to ODL, back to the second switch, and on to the destination. After re-provision, the third switch is added to the loop, increasing the delay, and cutting the bitrate. ODL buffers the packets, ensuring there is no packet loss, until the buffer is full, and it begins dropping incoming packets. This explains the sequence gap absence in the beginning, then appearing later in the capture. The sequence gaps were tied to the buffering effect and not the re-provision time.



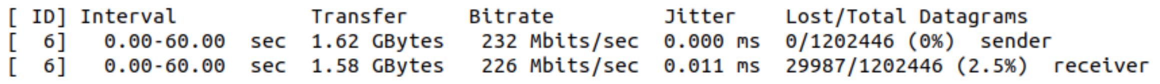
```
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-60.00  sec  3.03 GBytes   434 Mbits/sec  0.000 ms  0/2245787 (0%)  sender
[  6]   0.00-60.80  sec  2.81 GBytes   398 Mbits/sec  0.055 ms  158861/2245787 (7.1%)  receiver
```

(a) Floodlight: High Bit Rate/Low Packet Loss



```
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-60.00  sec  5.14 GBytes   735 Mbits/sec  0.000 ms  0/3809196 (0%)  sender
[  6]   0.00-60.00  sec  2.01 GBytes   288 Mbits/sec  0.003 ms  2317928/3809196 (61%)  receiver
```

(b) Ryu: Low Bit Rate/High Packet Loss



```
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-60.00  sec  1.62 GBytes   232 Mbits/sec  0.000 ms  0/1202446 (0%)  sender
[  6]   0.00-60.00  sec  1.58 GBytes   226 Mbits/sec  0.011 ms  29987/1202446 (2.5%)  receiver
```

(c) ODL: Low Bit Rate/Low Packet Loss

Figure 12. IPerf Report Summary Examples

5.3 Network Re-provisioning Time Metric Extraction

Seven NRTs were not detected in 120 runs of the remaining controllers, with a detection rate of $\frac{113}{120} = 94.1\overline{6}\%$. The manual inspection of captures and recording metrics increased the risk of human error. In two instances, once with Floodlight and once with Faucet, the traffic flow did not recover in one direction within packet capture duration. In the remaining five instances, the NRT sequence gap was undetected for Floodlight in the forward, reverse, or both directions. Floodlight either performed fast enough to prevent packet loss, or pre-provisioned the alternate route as primary, making the NRTs undetectable. The small size of the test topology may not have induced enough delay in combination with the controller to induce packet loss. The quickest NRT recorded from Floodlight was 0.22 seconds and is a probable detection floor for the tested topology.

5.4 Controller Re-provisioning Time

The controllers' NRT means, as calculated by JMP, are shown in Table 6. The data did not pass distribution normality checks, and statistical comparison was accomplished with non-parametric tests. The tests concluded all controllers are significantly different from each other. The JMP normality checks and non-parametric test outputs are shown in Appendix C.

Both Java controllers, Floodlight and ONOS, had the fastest NRTs with high consistency as shown in Table 6, and Fig. 13. T Bah et al. noted that ONOS pre-calculates all alternative paths [23], and might explain how well it performed. Floodlight is a fork of the Beacon controller, which was designed with high performance in mind [29]. Floodlight is lightweight and might pre-calculate paths too, and may have given an edge over ONOS.

Both Python controllers had significantly longer NRTs than the Java controllers did. They also had greater variability, with large whiskers for their box plots in Fig. 13. Faucet is based on Ryu, and both relied on Ryu's STP libraries in testing. STP is a loop prevention method used in traditional networking, and likely contributed to longer NRTs over the Java controllers. Faucet might have multi-threading support enabled by default, giving it an advantage over the simpler Ryu switching application.

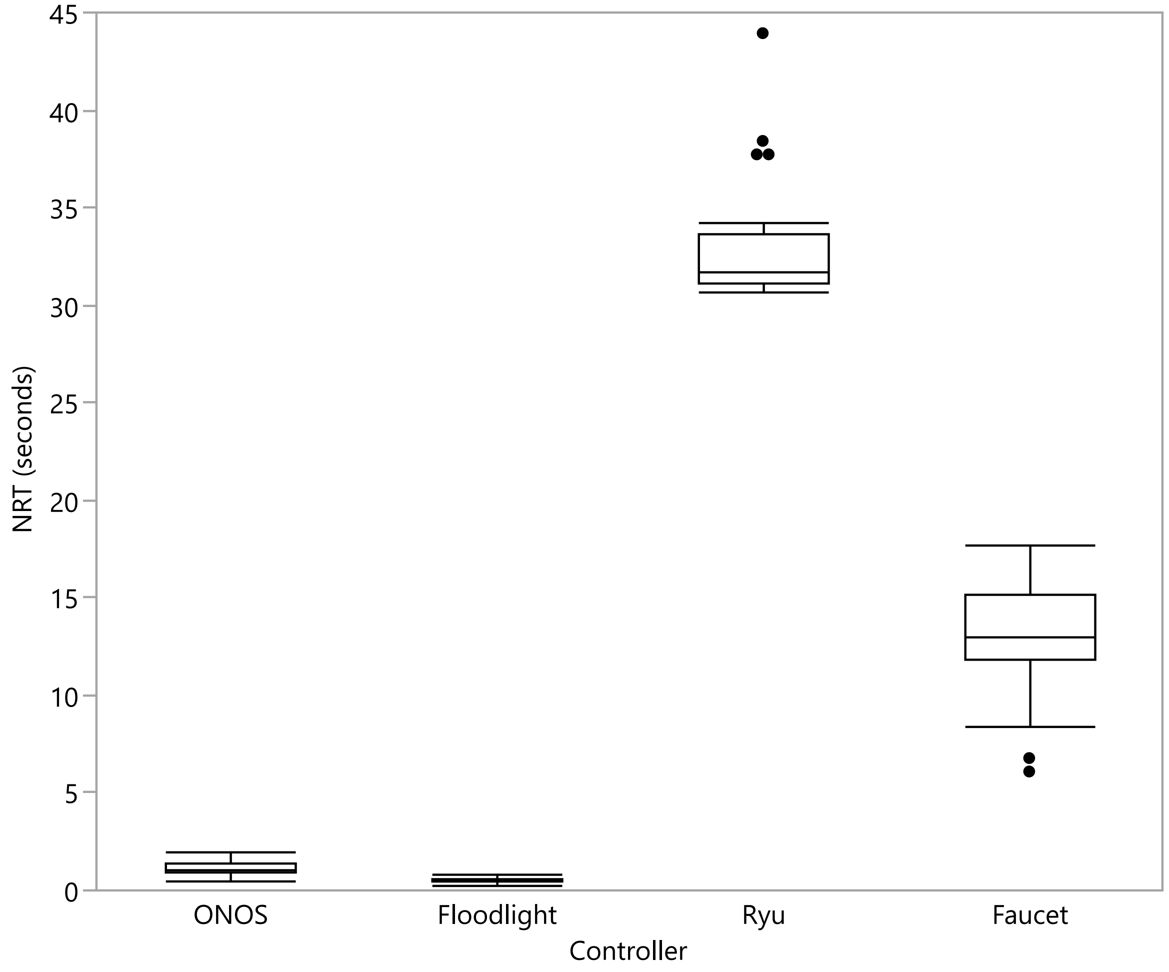


Figure 13. NRT Box Plots

Controller	Mean NRT	Std Dev	Upper 95%	Lower 95%
Floodlight	0.45 s	0.13 s	0.51 s	0.40 s
ONOS	1.09 s	0.35 s	1.22 s	0.96 s
Faucet	12.98 s	2.69 s	14.00 s	11.95 s
Ryu	32.92 s	2.96 s	34.03 s	31.82 s

Table 6. Controller NRT Statistics

5.5 Controller Packet Loss

The controllers' FDPL and RDPL followed similar patterns to their NRT suggesting a correlation. Again, the data did not pass normality checks and comparisons were made non-parametrically, showing all were significantly different. The JMP normality check and non-parametric test outputs are shown in Appendix C. The FDPL and RDPL box plots are similar, as shown in Fig. 14 and 15. The controllers ranked the same, as show in Table 7 and 8.

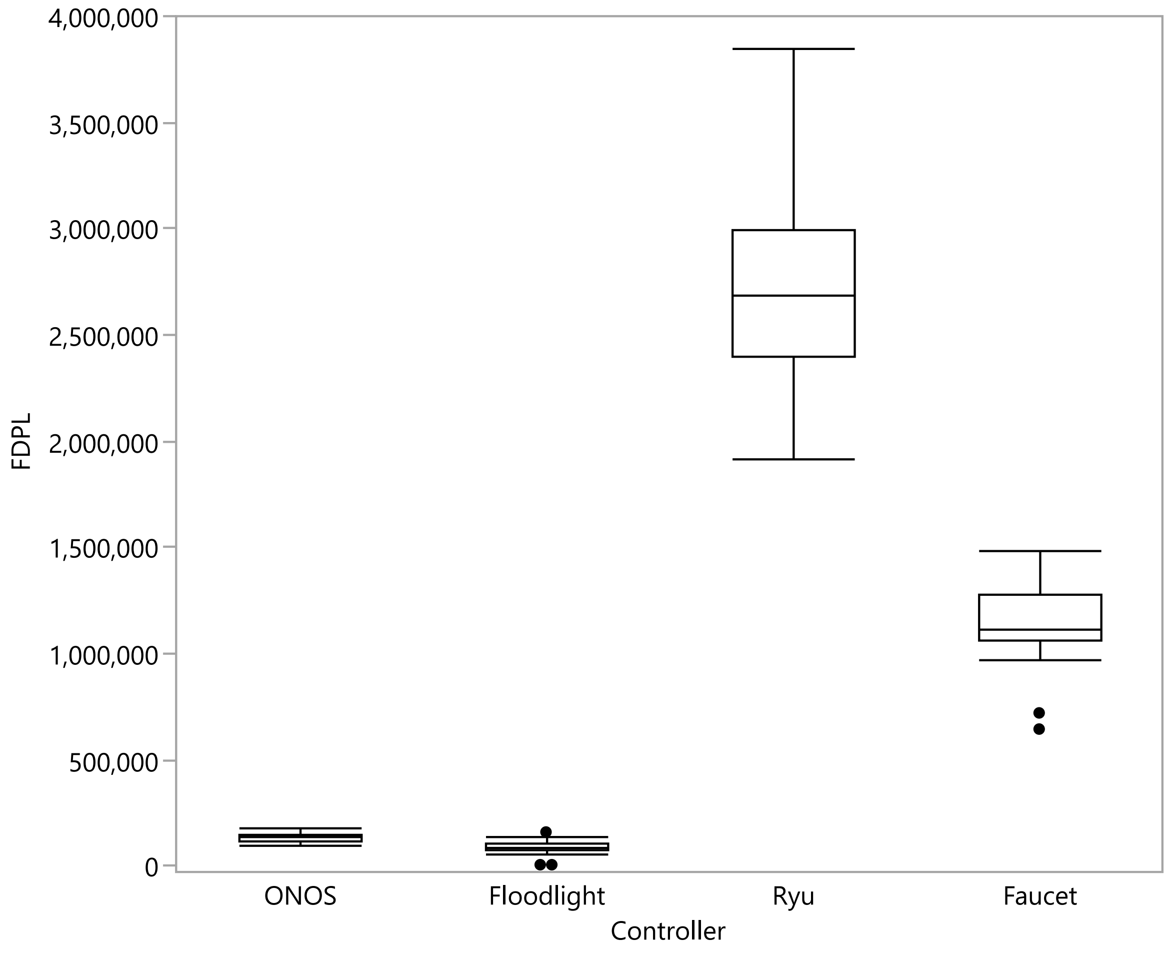


Figure 14. FDPL Box Plots

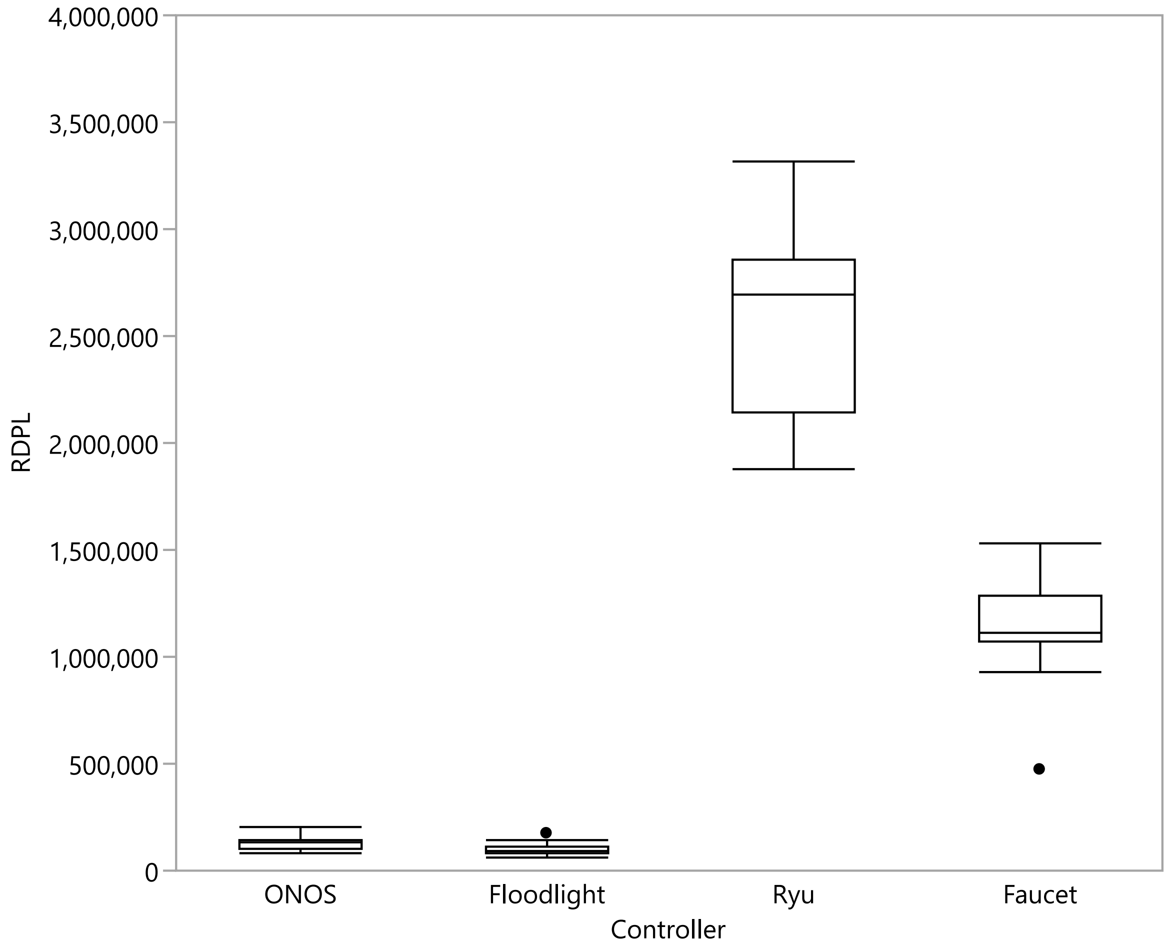


Figure 15. RDPL Box Plots

Controller	Mean FDPL	Std Dev	Upper 95%	Lower 95%
Floodlight	85,242 pkts	33,426 pkts	97,723 pkts	72,760 pkts
ONOS	135,525 pkts	24,653 pkts	144,731 pkts	126,320 pkts
Faucet	1,137,896 pkts	186,346 pkts	1,208,778 pkts	1,067,014 pkts
Ryu	2,688,923 pkts	424,531 pkts	2,847,445 pkts	2,530,401 pkts

Table 7. Controller FDPL Statistics

Controller	Mean RDPL	Std Dev	Upper 95%	Lower 95%
Floodlight	95,072 pkts	24,619 pkts	104,436 pkts	85,708 pkts
ONOS	127,551 pkts	34,273 pkts	140,348 pkts	114,753 pkts
Faucet	1,150,611 pkts	196,449 pkts	1,228,323 pkts	1,072,898 pkts
Ryu	2,583,586 pkts	424,171 pkts	2,741,974 pkts	2,425,199 pkts

Table 8. Controller RDPL Statistics

5.6 Discussion

When selecting controllers to test, Tello et al. selected Ryu over Faucet, as Faucet is based on Ryu [17]. However, in this limited experiment, the NRT results show Faucet was much faster than Ryu. However, both used reactive re-provisioning techniques, placing them behind the proactive controllers, Floodlight and ONOS. Both Floodlight and ONOS are written in Java, but Floodlight edged slightly ahead in testing. Implementation seems to be an important factor and specifying the controller version and enabled applications is vital for repeatability. Stating that one controller is better than the other without the software implementation parameters is not a fair comparison, and the RFC does well in requiring parameter specifications.

However, there are issues with the methodology. First, the RFC NRT methodology objective specifies it evaluates controllers which install flows to network devices [16]. ODL with L2Switch application does not install flows, and the triggering event does not directly produce the RFC required packet loss with an associated gap in sequence numbers. Second, the significant re-provisioning delay from the reactive controllers stood apart from the normal network delay, made packet identification for metric calculation easier. However, the proactive controllers had re-provisioning delays closer to normal network delay, making identification harder. Floodlight also had instances of delay short enough to prevent associated packet loss and sequence gaps. Third, the methodology specifies terminating the test after the first successful packet received after re-convergence, but this does not account for complete recovery failure, as the sequence gap would be infinite. To correct these issues, the following changes are recommended.

First, network re-convergence time could be added as a metric. It measures the time a network takes to re-converge on a traffic path solution, without limiting it to a single SDN convergence implementation. Network re-convergence could then compare ODL and similar controllers to those that install flows. Also, network re-convergence is a metric used in traditional networks, and allows direct comparison for SDN transition viability.

Second, correlating the switches' messages to a drop in bandwidth and a return to previous bandwidth levels may be an effective method for detecting NRT with fast proactive controllers. The packet loss and sequence gap requirements work well for detecting NRT with reactive, flow installing controllers, but might have issues with proactive controllers, depending on the network configuration. Regardless of re-convergence method, the switches send messages to the controller in the event of a link failure. Capturing the traffic between the controller and switches in addition to test traffic would attach a timestamp to the messages. The timestamp is an alternate marker for packet identification in the test traffic capture, but all clocks must be in sync.

Finally, other metrics in the RFC use trial duration in their methodology. To account for unacceptable re-convergence time and failure, a maximum trial duration should be specified as well.

The statistical metric analysis produced useful information beyond simple means. Faucet and Ryu showed greater NRT and packet loss variability over Floodlight and ONOS, which is a consideration for network design and optimization. The controllers under test were not found to be part of the same sample distribution, but the analysis gives more confidence than single runs or listing the arithmetic means.

5.7 Results Summary

ODL was removed from analysis, due to non-detection of the key NRT metric. Normality of the metric distributions could not be assumed, so all metrics were assessed with non-parametric tests. Technical issues and methodology gaps were identified with possible solutions. Despite the issues, the NRT, FDPL, and RDPL estimates were found for all controllers. In all tests, Floodlight ranked first, ONOS second, Faucet third, and Ryu last. All controllers were found to be significantly different from each other, concerning the tested metrics.

VI. Conclusions

6.1 Introduction

The current state of SDN controller evaluation relies on throughput and latency metrics. The IETF has published a methodology expanding the number of metrics but has little validation from the research community. Section 6.2 details the research conclusions towards this validation, and Section 6.3 its significance. Section 6.4 describes the limitations of the research and Section 6.5 lists future efforts. Finally, Section 6.6 summarizes the conducted research.

6.2 Research Conclusions

The research was partially successful in achieving its research objectives. RFC 8456 does not pass validation, due to the conflicting NRT definition and how it is measured. Additionally, the requirement for a sequence number gap and complete re-provisioning failure makes NRTs detection problematic and in this experiment, eliminated one controller from comparative analysis.

The objective to determine if the tested controllers are significantly different in NRT metrics is partially successful. Even though ODL is ineligible for comparative analysis, the other controllers provided sufficient data for comparison. In their test configuration, the qualifying controllers are significantly different in their metrics for the test topology. Floodlight provides the fastest and most consistent NRT, with the least packet loss. Implementation matters with the controllers. Faucet performs better than Ryu, despite the common lineage. Floodlight and ONOS are both built on Java, but Floodlight performs better in testing.

The test environment build objective to produce reliable results was partially successful. The environment does not guarantee the initial provisioning of the primary path, and iperf3 reporting has errors invalidating some runs. There was sufficient data for non-parametric comparative analysis.

Due to the partial success of the research objectives, the research hypothesis that the RFC 8456 NRT methodology framework is able to reliably reveal meaningful differences in the observed metrics between the SDN controllers under test is not proven true.

6.3 Research Contributions

The research found limitations in RFC 8456 NRT methodology, which make comparison of OpenFlow SDN controllers problematic. To overcome these limitations, the research suggests adding the NRT metric to permit comparison of controllers with different traffic pathing implementations. Adding bandwidth based NRT may improve NRT detection for proactive SDN controllers, and specifying a maximum trial duration should identify intolerable times for user implementations.

The research also created a process for identifying NRT from packet captures. It is a manual and inefficient process but provides a baseline for refinement and automation in further research. In addition, measuring the bandwidth during the tests verified ODL's L2Switch application bandwidth limitations and identified the probable source.

The metric extraction processes, proposed changes to the NRT methodology, and statistical analysis method should provide a stronger framework for OpenFlow SDN comparison. With the improved framework, the research community, USAF, and Department of Defense (DoD) can evaluate SDN controllers for implementation in networks to provide management and/or defensive benefits.

6.4 Research Limitations

The first limitation of the research stem from iPerf3 errors. Definite errors were identified in the reports, casting some doubt on the rest of the results. Investigation and correction of the errors or building custom traffic generation is required.

The second set of limitations comes from the test environment. It only evaluates OpenFlow protocol version 1.3 capable controllers. The OpenFlow protocol is the most prominent SBI, but there are other versions and protocols available. The experiment script only builds one test topology and needs additional coding to emulate traditional networks for comparison and handle topologies of various types and sizes.

The final set comes from the metric extraction and analysis processes. The manual nature of extraction may increase error and requires a script or real time analysis. The non-parametric analysis shows the tested controllers are significantly different but are less statistically powerful than parametric tests. The source of non-normality needs identification and mitigation to perform parametric comparisons.

6.5 Future Work

Opportunities to refine and expand the research are:

- Matching the control traffic messages to the test traffic may eliminate the need for sequence gaps to identify NRT. This requires capture and inspection of the control traffic in addition to the test traffic.

- Reducing errors requires automating the metric extraction process through parsing scripts or real time analysis. Another avenue to reduce packet loss metric errors requires eliminating iperf3 errors or creating custom method of packet generation.

- Performing manual pre-provisioning or automatic path detection should guarantee the primary path disruption across multiple network topologies.

- The contributing factors to NRT need analysis. Controller asynchronous message and path provisioning times should influence NRT. The degree of influence is of interest, as well as which controller methods effectively minimize these times and their trade-offs.

- Investigating how effectively SDN controllers re-provision with links or devices frequently cycling up and down is useful for network design considerations.

6.6 Summary

To evaluate an SDN controller, a solid methodology framework is required. The IETF RFC 8456 could benefit from refinement with regard to NRT. The limitations of the RFC make the NRT undetectable in certain cases. The test environment needs refinement as well, to compensate for errors in experiment setup, utilized tools, and metric extraction. As configured, the Floodlight controller had the best NRTs and lowest packet losses, in comparison to the other controllers in the experimental topology. Identifying the specific factors with the greatest the greatest influence on NRT are a future area of research.

Appendix A. Test Script

```
#!/usr/bin/python3


from mininet.link import TCLink

from mininet.node import Controller, Node, OVSSwitch,
   RemoteController

from mininet.net import Mininet

from mininet.cli import CLI

from mininet.topo import Topo

from mininet.log import setLogLevel

from datetime import datetime

from time import sleep

import os

import sys


"""

Test topology with two alternate paths

"""

class nRTopo(Topo):


    # Make OpenFlow 1.3 the default protocol for virtual
       switches

    def addSwitch( self, name, **opts ):

        kwargs = { 'protocols' : 'OpenFlow13' }

        kwargs.update( opts )

        return super(nRTopo, self).addSwitch( name, **
            kwargs )


    # Create testing topology
    def build( self ):

        # Create and link switches
        switches = []
        for i in range(1,4):
            switches.append(self.addSwitch('s%s' % i))

        self.addLink( switches[0], switches[1], bw=1000)
        self.addLink( switches[1], switches[2], bw=1000)
        self.addLink( switches[2], switches[0], bw=1000)


        # Add and link traffic generators
        for i in range(1,3):
            host = self.addHost('h%s' % i)
            self.addLink(host, switches[i-1], bw=1000)

"""
Run network re-provisioning test
"""
def run(ctrlIP=None, runNum=0):
```

```
# Create network topology"
topo = nRTopo()
net = Mininet(topo=topo,
              controller=RemoteController('c0', ip=
                 ctrlIP),
              link=TCLink,
              autoSetMacs=True,
              waitConnected=True )
net.start()

# wait for controller to set up switches
print("+++␣Waiting␣30␣seconds␣for␣switch␣configuration
   ")
sleep(30)

# gratuitous ARP from every host to make the
   controller aware
print("+++␣Sending␣gratuitous␣ARPs␣from␣hosts")
for h in net.hosts:
    h.cmd('./gratuitousArp.py')

# Ping all hosts 3 times to test reachability.
print("+++␣Starting␣host␣reachability␣tests")
net.pingAllFull()
net.pingAllFull()
net.pingAllFull()
```

```
# enter mininet command line
#cli = CLI
#cli(net)

# get refenences to traffic generators
h1 = net.getNodeByName('h1')
h2 = net.getNodeByName('h2')

# clear any iperf running on traffic generators
print("+++␣Killing␣any␣running␣iperf␣traffic␣processes
   ")
h1.cmd( 'killall␣-9␣iperf' )
h2.cmd( 'killall␣-9␣iperf' )

# start iperf services
print("+++␣Starting␣iperf␣servers␣on␣traffic␣
   generators")
h1.cmd('iperf3␣-s␣-p␣5001␣-D')
h2.cmd('iperf3␣-s␣-p␣5002␣-D')

# wait for iperf server init
sleep(3)

# begin 60s packet capture on traffic generators, run#
   _hostname
```

```
print("+++␣Starting␣tshark␣packet␣captures␣on␣traffic␣
   generators")
h1.cmd(f'tshark␣-f␣"udp␣port␣5001"␣-w␣/root/pcap/{
   runNum}_h1.pcapng␣-a␣duration:65␣&')
h2.cmd(f'tshark␣-f␣"udp␣port␣5002"␣-w␣/root/pcap/{
   runNum}_h2.pcapng␣-a␣duration:65␣&')

# start 60s iperf3 1 Gbps UDP traffic, save output for
    bw at 0.1s intervals
# and packet loss metric: run#_port.iperf
print("+++␣Generating␣iperf␣UDP␣traffic")
h1.cmd(f'iperf3␣-u␣-b␣1G␣-c␣{h2.IP()}␣-p␣5002␣-t␣60␣-i
   ␣0.1␣--logfile␣/root/iperf/{runNum}_5002.iperf␣&')
h2.cmd(f'iperf3␣-u␣-b␣1G␣-c␣{h1.IP()}␣-p␣5001␣-t␣60␣-i
   ␣0.1␣--logfile␣/root/iperf/{runNum}_5001.iperf␣&')

# wait 3 seconds for traffic stabilization
sleep(3)

# down link in primary path to trigger re-provision
print("+++␣Removing␣link␣in␣primary␣path")
net.configLinkStatus('s1', 's2', "down")

# wait 60 seconds for reprovision
print("+++␣Waiting␣for␣re-provision␣and␣packet␣capture
   ")
```

```
    sleep(60)

    # enter mininet command line
    #cli = CLI
    #cli(net)

    # stop mininet
    print("+++␣Stopping␣Mininet")
    net.stop

    # clean up any lingering Mininet topology
    os.system("mn␣-c")

if __name__ == '__main__':
    setLogLevel('info')

    # Get controller IP and run number
    IP = sys.argv[1]
    runNum = sys.argv[2]

    # Log start of test run
    f = open( "/root/runlog.csv", "a")
    now = datetime.now().strftime("%m/%d/%Y,␣%H:%M:%S")
    f.write(f"{runNum},␣{IP},␣{now},␣started\n")

    # run test
```

```
run(IP, runNum)


# Log end of test run
now = datetime.now().strftime("%m/%d/%Y,␣%H:%M:%S")
f.write(f"{runNum},␣{IP},␣{now},␣finished\n")
f.close()
```

Appendix B. Host Location Advertisement Script

```
#!/usr/bin/python2
"""
Sends a gratuitous ARP from every network interface on a
   host
From ONOS Tutorial VM
"""

import sys
import os
import fcntl
import socket
from struct import pack

def getIPAddress(intf):
    #Borrowed from:
    #http://stackoverflow.com/questions/24196932/how-can-i
       -get-the-ip-address-of-eth0-in-python
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    return socket.inet_ntoa(fcntl.ioctl(
            s.fileno(),
            0x8915,  # SIOCGIFADDR
            pack('256s', intf[:15])
    )[20:24])

def gratuitousArp(intf, ip=None, mac=None):
```

```
#Adapted from:
#https://github.com/krig/send_arp.py/blob/master/
   send_arp.py
sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW
   )
try:
    sock.bind((intf, socket.SOCK_RAW))
except:
    print ('Device␣does␣not␣exist:␣%s' % intf )
    return

if not ip:
    try:
        ip = getIPAddress(intf)
    except IOError:
        print ('No␣IP␣for␣%s' % intf)
        return
packed_ip = pack('!4B', *[int(x) for x in ip.split('.'
   )])

if mac:
    packed_mac = pack('!6B', *[int(x,16) for x in mac.
       split(':')])
else:
    packed_mac = sock.getsockname()[4]
```

```
bcast_mac = pack('!6B', *(0xFF,)*6)
zero_mac = pack('!6B', *(0x00,)*6)
eth_arp = pack('!H', 0x0806)
arp_proto = pack('!HHBBH', 0x0001, 0x0800, 0x0006, 0
   x0004, 0x0001)
arpframe = [
    ## ETHERNET
    # destination MAC addr
    bcast_mac,
    # source MAC addr
    packed_mac,
    # eth proto
    eth_arp,

    ## ARP
    arp_proto,
    # sender MAC addr
    packed_mac,
    # sender IP addr
    packed_ip,
    # target hardware addr
    bcast_mac,
    # target IP addr
    packed_ip
]
```

```
    # send the ARP packet
    sock.send(''.join(arpframe))

if __name__ == "__main__":
    if len(sys.argv) > 1:
        intfs = sys.argv[1:]
    else:
        intfs = os.listdir('/sys/class/net/')

    for intf in intfs:
        gratuitousArp(intf)
```

3.1 Network Re-provisioning Time

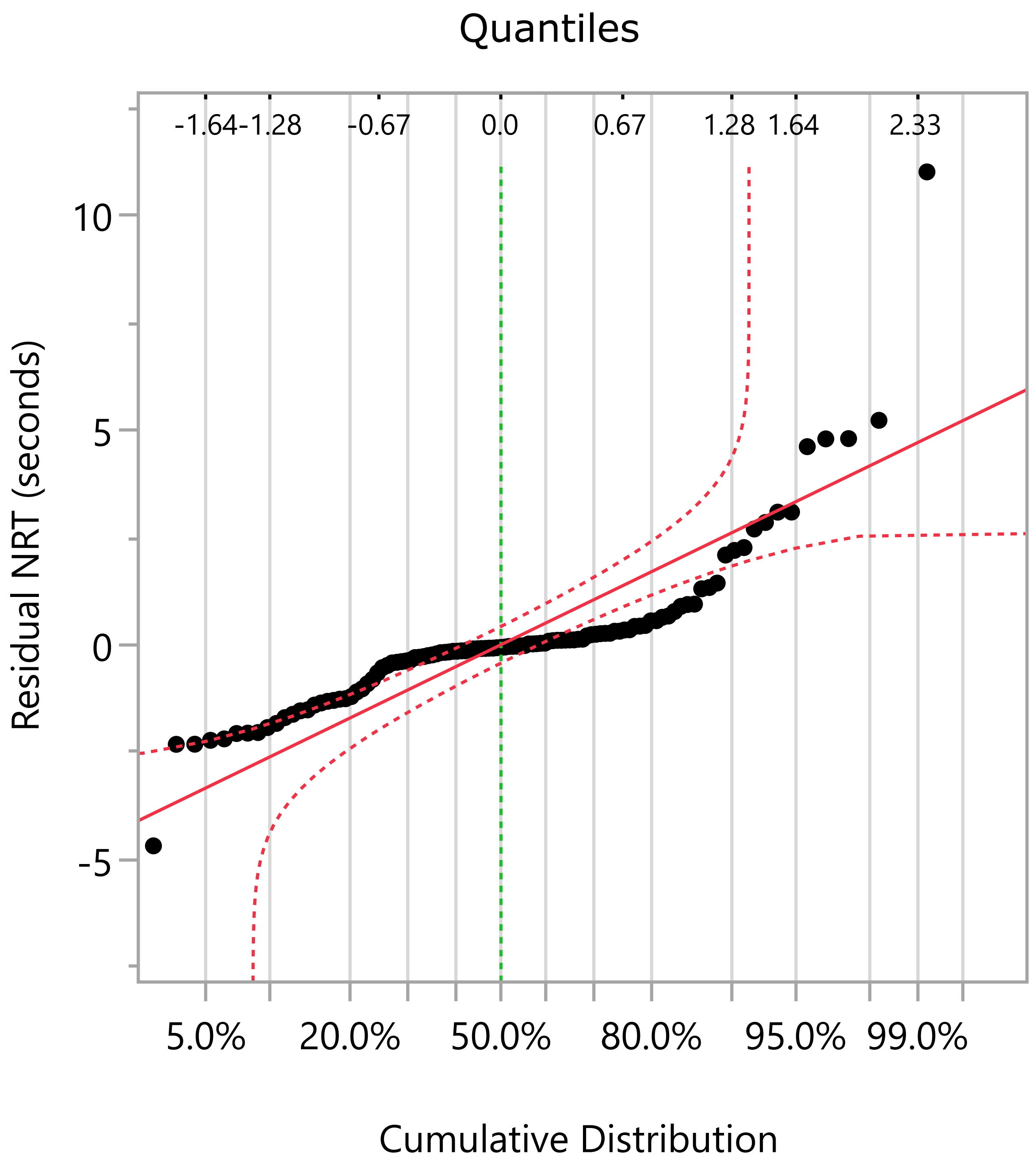
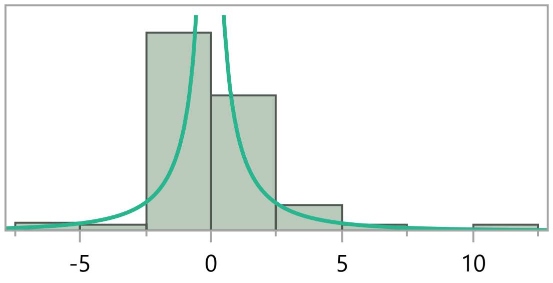


Figure 16. NRT Normal Quantile Plot

Distributions

Residual NRT



Compare Distributions

Distribution		AICc ^	AICc Weight	.2 .4 .6 .8	BIC	-2*LogLikelihood
SHASH	▬	388.00507	0.9927	▇	398.54425	379.6347
Cauchy	▬	398.94878	0.0042		404.29446	394.83968
Normal 3 Mixture	▬	400.54651	0.0019		416.11839	387.75406
Johnson Su	▬	401.41572	0.0012		411.9549	393.04535
Normal 2 Mixture	▬	465.15583	0		478.23202	454.59508
Normal	▬	483.58622	0		488.9319	479.47712

Fitted Normal Distribution

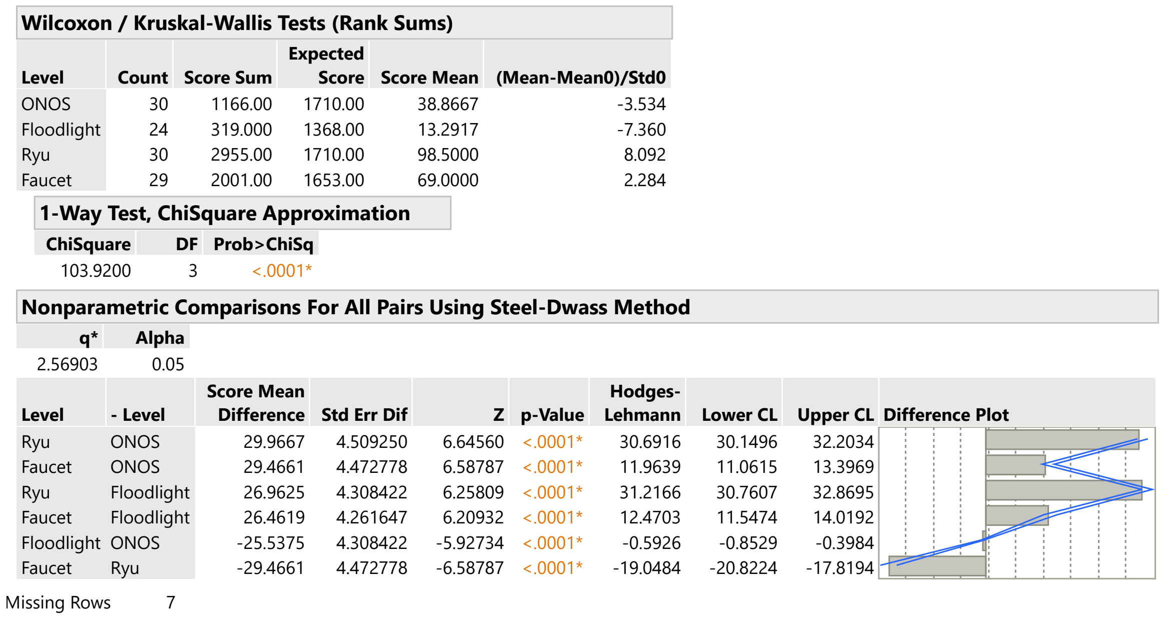
Parameter		Estimate	Std Error	Lower 95%	Upper 95%
Location	μ	1.254e-15	0.1907813	-0.373925	0.3739245
Dispersion	σ	2.0280332	0.0953641	1.8494774	2.0627995

Measures	
-2*LogLikelihood	479.47712
AICc	483.58622
BIC	488.9319

Goodness-of-Fit Test

	A2	Prob > A2
Anderson-Darling	7.108173	<.0001*

Figure 17. NRT Residual Distribution



Wilcoxon / Kruskal-Wallis Tests (Rank Sums)

Level	Count	Score Sum	Expected Score	Score Mean	(Mean-Mean0)/Std0
ONOS	30	1166.00	1710.00	38.8667	-3.534
Floodlight	24	319.000	1368.00	13.2917	-7.360
Ryu	30	2955.00	1710.00	98.5000	8.092
Faucet	29	2001.00	1653.00	69.0000	2.284

1-Way Test, ChiSquare Approximation

ChiSquare	DF	Prob>ChiSq
103.9200	3	<.0001*

Nonparametric Comparisons For All Pairs Using Steel-Dwass Method

q*	Alpha
2.56903	0.05

Level	- Level	Score Mean Difference	Std Err Dif	Z	p-Value	Hodges-Lehmann	Lower CL	Upper CL	Difference Plot
Ryu	ONOS	29.9667	4.509250	6.64560	<.0001*	30.6916	30.1496	32.2034	
Faucet	ONOS	29.4661	4.472778	6.58787	<.0001*	11.9639	11.0615	13.3969	
Ryu	Floodlight	26.9625	4.308422	6.25809	<.0001*	31.2166	30.7607	32.8695	
Faucet	Floodlight	26.4619	4.261647	6.20932	<.0001*	12.4703	11.5474	14.0192	
Floodlight	ONOS	-25.5375	4.308422	-5.92734	<.0001*	-0.5926	-0.8529	-0.3984	
Faucet	Ryu	-29.4661	4.472778	-6.58787	<.0001*	-19.0484	-20.8224	-17.8194	

Missing Rows 7

Figure 18. NRT Non-parametric Tests

3.2 Packet Loss

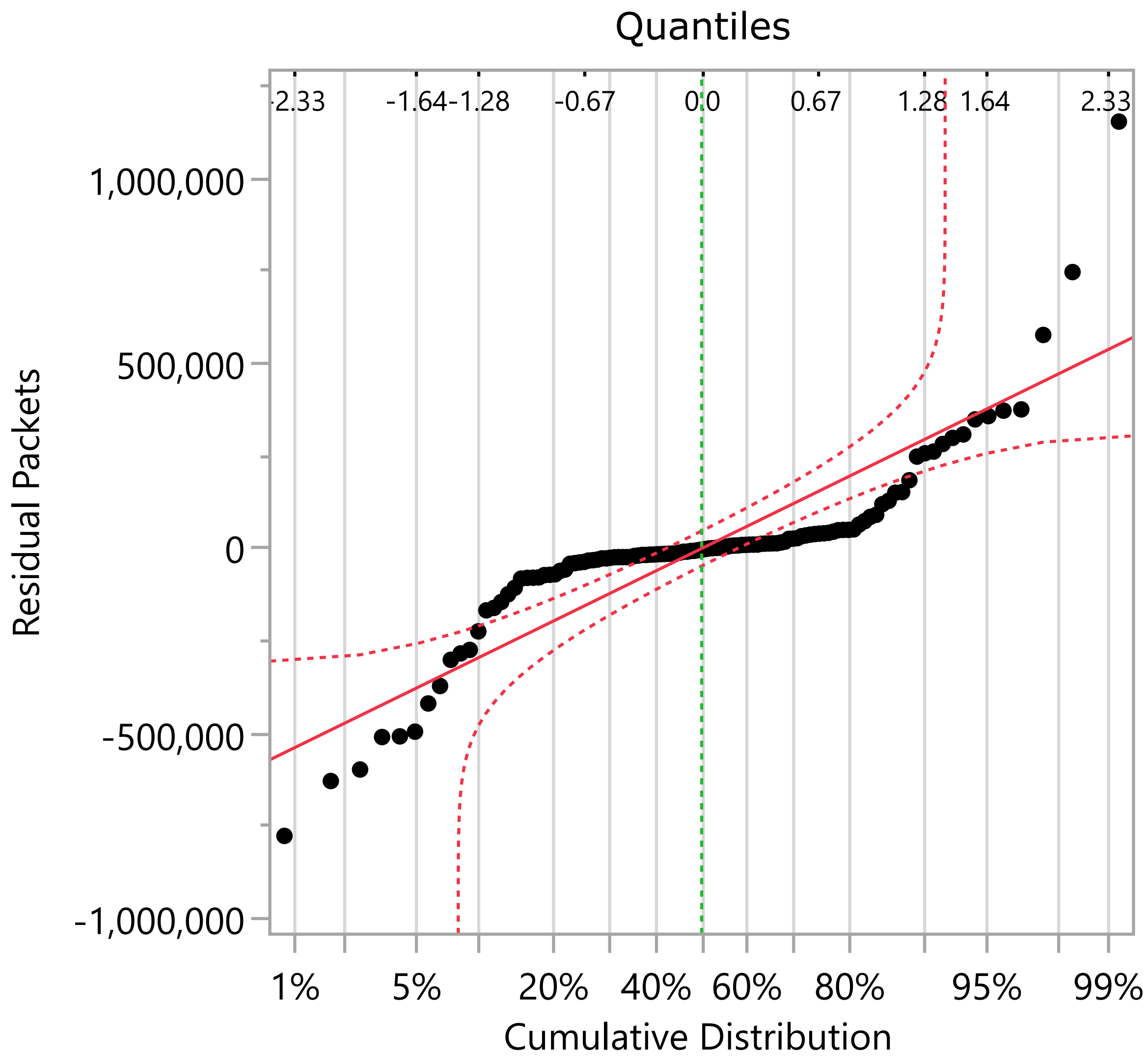
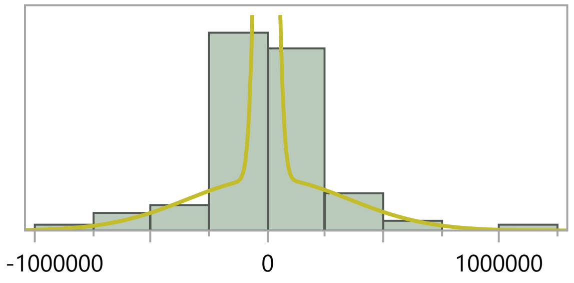


Figure 19. FDPL Residual Normal Quantile Plot

Distributions

Residual FDPL



Compare Distributions

Distribution		AICc ^	AICc Weight	.2 .4 .6 .8	BIC	-2*LogLikelihood
Normal 3 Mixture	—	3150.8611	0.7059		3166.7858	3138.1111
SHASH	—	3152.6234	0.2925		3163.389	3144.2725
Johnson Su	—	3163.0498	0.0016		3173.8154	3154.6989
Normal 2 Mixture	—	3266.6201	0		3279.9847	3256.0891
Normal	—	3279.2467	0		3284.7015	3275.1432
Cauchy	—	6151.4124	0		6156.8672	6147.3089

Fitted Normal Distribution

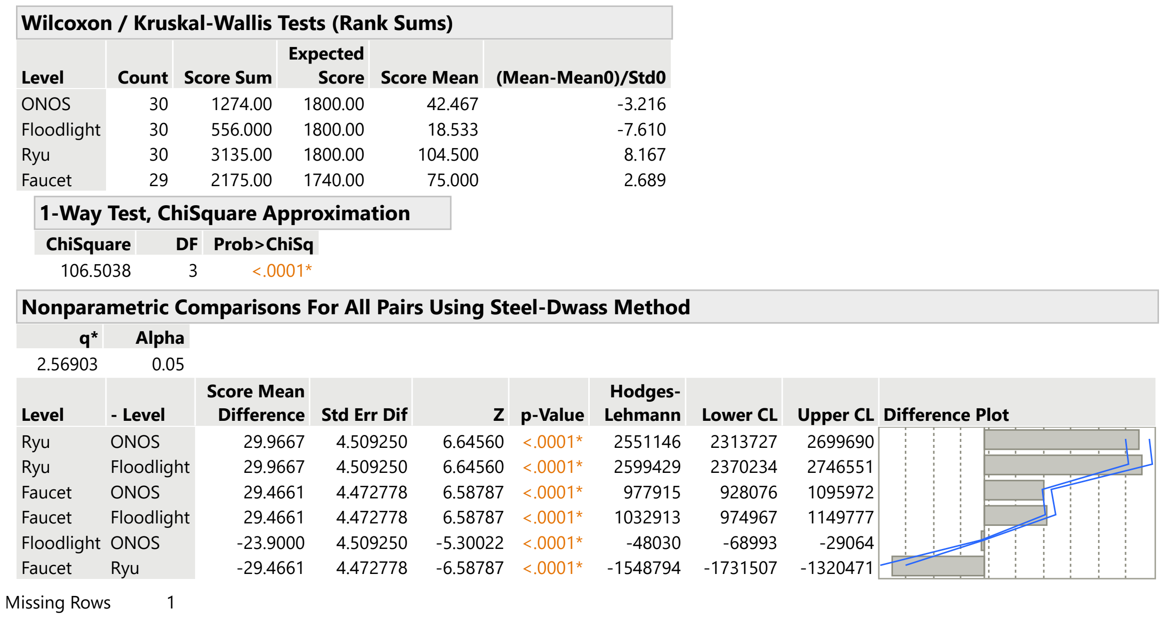
Parameter		Estimate	Std Error	Lower 95%	Upper 95%
Location	μ	4.696e-11	21095.337	-41346.1	41346.102
Dispersion	σ	230122.96	31.292917	230061.64	230184.3

Measures	
-2*LogLikelihood	3275.1432
AICc	3279.2467
BIC	3284.7015

Goodness-of-Fit Test

	A2	Prob > A2
Anderson-Darling	9.2925501	<.0001*

Figure 20. FDPL Residual Distribution



Wilcoxon / Kruskal-Wallis Tests (Rank Sums)

Level	Count	Score Sum	Expected Score	Score Mean	(Mean-Mean0)/Std0
ONOS	30	1274.00	1800.00	42.467	-3.216
Floodlight	30	556.000	1800.00	18.533	-7.610
Ryu	30	3135.00	1800.00	104.500	8.167
Faucet	29	2175.00	1740.00	75.000	2.689

1-Way Test, ChiSquare Approximation

ChiSquare	DF	Prob>ChiSq
106.5038	3	<.0001*

Nonparametric Comparisons For All Pairs Using Steel-Dwass Method

q*	Alpha
2.56903	0.05

Level	- Level	Score Mean Difference	Std Err Dif	Z	p-Value	Hodges-Lehmann	Lower CL	Upper CL	Difference Plot
Ryu	ONOS	29.9667	4.509250	6.64560	<.0001*	2551146	2313727	2699690	
Ryu	Floodlight	29.9667	4.509250	6.64560	<.0001*	2599429	2370234	2746551	
Faucet	ONOS	29.4661	4.472778	6.58787	<.0001*	977915	928076	1095972	
Faucet	Floodlight	29.4661	4.472778	6.58787	<.0001*	1032913	974967	1149777	
Floodlight	ONOS	-23.9000	4.509250	-5.30022	<.0001*	-48030	-68993	-29064	
Faucet	Ryu	-29.4661	4.472778	-6.58787	<.0001*	-1548794	-1731507	-1320471	

Missing Rows 1

Figure 21. FDPL Non-parametric Tests

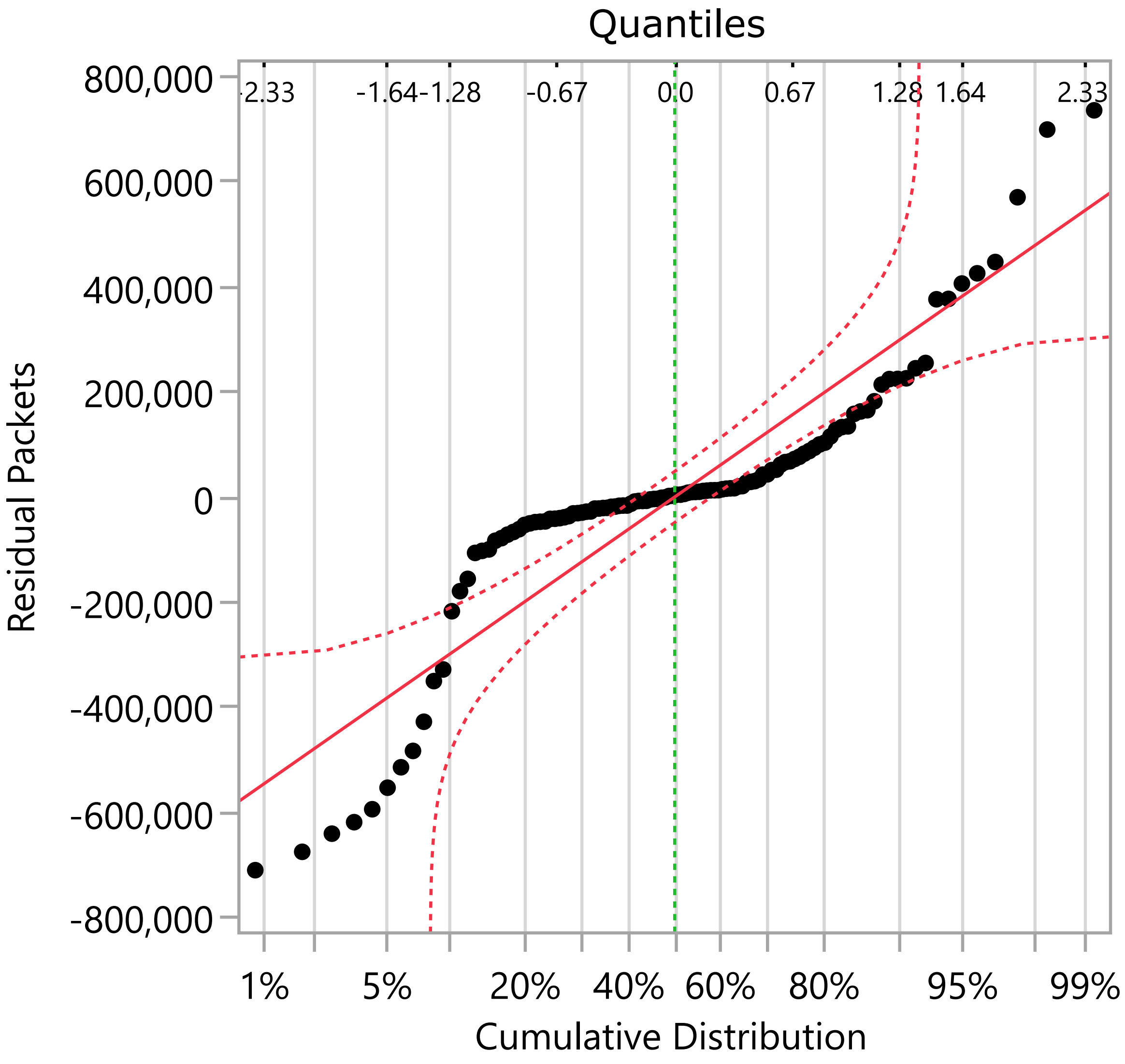
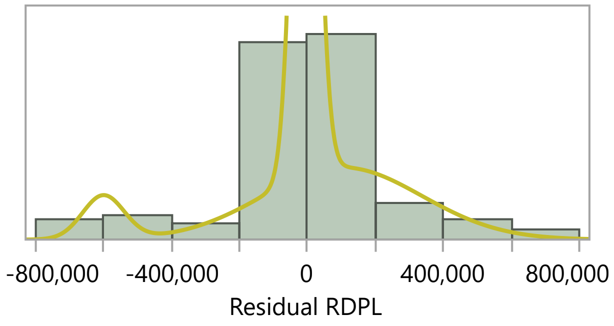


Figure 22. RDPL Residual Normal Quantile Plot

Residual RDPL



Compare Distributions

Distribution		AICc ^	AICc Weight	.2 .4 .6 .8	BIC	-2*LogLikelihood
Normal 3 Mixture		3105.3266	0.785		3126.0095	3087.9808
Normal 2 Mixture		3108.6157	0.1516		3121.8382	3098.0702
SHASH		3110.358	0.0634		3121.012	3101.9977
Johnson Su		3127.5424	1.2e-5		3138.1964	3119.1821
Normal		3200.0754	0		3205.4764	3195.9692
Cauchy		6003.2258	0		6008.6268	5999.1196

Fitted Normal Distribution

Parameter		Estimate	Std Error	Lower 95%	Upper 95%
Location	μ	7.728e-11	21683.482	-42498.84	42498.843
Dispersion	σ	233538.24	31.934584	233475.66	233600.84

Measures	
-2*LogLikelihood	3195.9692
AICc	3200.0754
BIC	3205.4764

Goodness-of-Fit Test

	A2	Prob > A2
Anderson-Darling	7.6580597	<.0001*

Figure 23. RDPL Residual Distribution

Wilcoxon / Kruskal-Wallis Tests (Rank Sums)

Level	Count	Score Sum	Expected Score	Score Mean	(Mean-Mean0)/Std0
ONOS	30	1143.00	1755.00	38.100	-3.856
Floodlight	29	627.000	1696.50	21.621	-6.816
Ryu	30	3045.00	1755.00	101.500	8.130
Faucet	27	1971.00	1579.50	73.000	2.554

1-Way Test, ChiSquare Approximation

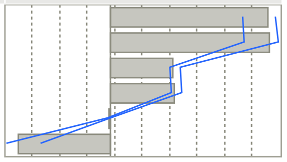
ChiSquare	DF	Prob>ChiSq
99.9770	3	<.0001*

Nonparametric Comparisons For All Pairs Using Steel-Dwass Method

q*	Alpha
2.56903	0.05

Level	- Level	Score Mean Difference	Std Err Dif	Z	p-Value	Hodges-Lehmann	Lower CL	Upper CL
Ryu	ONOS	29.9667	4.509250	6.64560	<.0001*	2567137	2126652	2669212
Ryu	Floodlight	29.4661	4.472778	6.58787	<.0001*	2601226	2154710	2703626
Faucet	ONOS	28.4648	4.403071	6.46476	<.0001*	1006051	947203	1118798
Faucet	Floodlight	27.9642	4.361682	6.41134	<.0001*	1030423	979989	1144733
Floodlight	ONOS	-16.4454	4.472778	-3.67678	0.0013*	-30920	-53048	-10483
Faucet	Ryu	-28.4648	4.403071	-6.46476	<.0001*	-1502359	-1693189	-1143768

Difference Plot



Missing Rows 4

Figure 24. RDPL Non-parametric Tests

Bibliography

[1] R. Sherwood and K. K. Yap, "Cbench controller benchmarker," *Last accessed, Nov*, 2011. [Online]. Available: `https://github.com/andi-bigswitch/oflops/tree/master/cbench`.

[2] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou, *Software-Defined Networking (SDN): Layers and Architecture Terminology*, RFC 7426, Jan. 2015. DOI: `10.17487/RFC7426`. [Online]. Available: `https://rfc-editor.org/rfc/rfc7426.txt`.

[3] Open Networking Foundation (ONF), *Software-Defined Networking (SDN) Definition - Open Networking Foundation*, 2018. [Online]. Available: `https://opennetworking.org/sdn-definition/`.

[4] T. Culver, C. Black, and P. Goransson, *Software Defined Networks , 2nd Edition*, ISBN: 9780128045794. [Online]. Available: `https://learning.oreilly.com/library/view/software-defined-networks/9780128045794/`.

[5] Z. Latif, K. Sharif, F. Li, M. M. Karim, and Y. Wang, "A comprehensive survey of interface protocols for software defined networks," *arXiv*, pp. 1–30, 2019, ISSN: 23318422. arXiv: `1902.07913`.

[6] ONF, "OpenFlow Switch Specification Version 1.3.5 (Protocol version 0x04)," Tech. Rep., 2015, p. 283. [Online]. Available: `http://www.opennetworking.org`.

[7] B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014, ISSN: 1553877X. DOI: `10.1109/SURV.2014.012214.00180`.

[8] L. Zhu, M. M. Karim, K. Sharif, C. Xu, F. Li, X. Du, and M. Guizani, “SDN Controllers: A Comprehensive Analysis and Performance Evaluation Study,” *ACM Computing Surveys*, vol. 53, no. 6, 2021, ISSN: 15577341. DOI: 10.1145/3421764.

[9] M. Ulema, *Vulnerabilities and Opportunities in SDN, NFV, and NGSON IEEE CQR 2014 International Workshop Emerging Technology Reliability Roundtable*, 2014.

[10] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: Rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM Workshop on Hot Topics in Networks, Hotnets-9*, 2010, ISBN: 9781450304092. DOI: 10.1145/1868447.1868466.

[11] M. Peuster, J. Kampmeyer, and H. Karl, “Containernet 2.0: A rapid prototyping platform for hybrid service function chains,” in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, pp. 335–337. DOI: 10.1109/NETSOFT.2018.8459905.

[12] F. N. Farias, O. Antônio de, L. B. da Costa, B. A. Pinheiro, and A. J. Abelém, *vSDNEmul: A software-defined network emulator based on container virtualization*, 2019. DOI: 10.5013/ijssst.a.20.04.07. arXiv: 1908.10980.

[13] X. Zhang, N. Prabhu, and R. Tessier, “NestedNet: A Container-based Prototyping Tool for Hierarchical Software Defined Networks,” in *Proceedings of the International Workshop on Rapid System Prototyping*, vol. 2020-Septe, 2020, ISBN: 9781728184661. DOI: 10.1109/RSP51120.2020.9244858.

[14] Z. Xiang, S. Pandi, J. Cabrera, F. Granelli, P. Seeling, and F. H. Fitzek, “An Open Source Testbed for Virtualized Communication Networks,” *IEEE Com-*

munications Magazine, vol. 59, no. 2, pp. 77–83, 2021, ISSN: 15581896. DOI: 10.1109/MCOM.001.2000578.

[15] L. Veltri, L. Davoli, R. Pecori, A. Vannucci, and F. Zanichelli, "NEMO: A flexible and highly scalable Network EMulatOr," *SoftwareX*, vol. 10, 2019, ISSN: 23527110. DOI: 10.1016/j.softx.2019.100248.

[16] B. Vengainathan, A. Basil, M. Tassinari, V. Manral, and S. Banks, *Benchmarking Methodology for Software-Defined Networking (SDN) Controller Performance*, RFC 8456, Oct. 2018. DOI: 10.17487/RFC8456. [Online]. Available: https://rfc-editor.org/rfc/rfc8456.txt.

[17] A. M. D. Tello and M. Abolhasan, "SDN Controllers Scalability and Performance Study," *2019, 13th International Conference on Signal Processing and Communication Systems, ICSPCS 2019 - Proceedings*, 2019. DOI: 10.1109/ICSPCS47537.2019.9008462.

[18] *GitHub - noxrepo/pox: The POX network software platform*, 2020. [Online]. Available: https://github.com/noxrepo/pox (visited on 05/24/2021).

[19] *Home - OpenDaylight*. [Online]. Available: https://www.opendaylight.org/ (visited on 05/24/2021).

[20] *Open Network Operating System (ONOS) SDN Controller for SDN/NFV Solutions*. [Online]. Available: https://opennetworking.org/onos/ (visited on 05/24/2021).

[21] *Ryu SDN Framework*. [Online]. Available: https://ryu-sdn.org/ (visited on 05/24/2021).

[22] *Floodlight Controller - Project Floodlight*. [Online]. Available: https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview (visited on 05/24/2021).

[23] M. Tbah, A. Azzouni, M. T. Nguyen, and G. Pujolle, "Topology Discovery Performance Evaluation of OpenDaylight and ONOS Controllers," *Proceedings of the 2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops, ICIN 2019*, no. Icin 2019, pp. 285–291, 2019. DOI: `10.1109/ICIN.2019.8685915`.

[24] Iperf, *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool*, 2016. [Online]. Available: `https://iperf.fr/` (visited on 05/23/2021).

[25] J. B. Silva, F. S. D. Silva, E. P. Neto, M. Lemos, and A. Neto, "Benchmarking of mainstream SDN controllers over open off-the-shelf software-switches," *Internet Technology Letters*, vol. 3, no. 3, pp. 1–6, 2020, ISSN: 2476-1508. DOI: `10.1002/itl2.152`.

[26] *Faucet SDN Controller*. [Online]. Available: `https://faucet.nz/` (visited on 05/24/2021).

[27] *floodlight/LinkDiscoveryManager.java at master · floodlight/floodlight · GitHub*. [Online]. Available: `https://github.com/floodlight/floodlight/blob/master/src/main/java/net/floodlightcontroller/linkdiscovery/internal/LinkDiscoveryManager.java` (visited on 05/26/2021).

[28] *Predictive Analytics Software — JMP Pro*. [Online]. Available: `https://www.jmp.com/en_us/software/predictive-analytics-software.html%20https://www.jmp.com/en_my/software/predictive-analytics-software.html` (visited on 05/26/2021).

[29] D. Erickson, *The Beacon OpenFlow Controller*, ISBN: 9781450321785.

Glossary

application plane The collection of applications and services that program network behavior [2]. 5

control plane The collection of functions responsible for controlling one or more network devices. CP instructs network devices with respect to how to process and forward packets. The control plane interacts primarily with the forwarding plane and, to a lesser extent, with the operational plan [2]. 5, 6

forwarding plane The collection of resources across all network devices responsible for forwarding traffic [2]. 5, 6

management plane The collection of functions responsible for monitoring, configuring, and maintaining one or more network devices or parts of network devices. The management plane is mostly related to the operational plane (it is related less to the forwarding plane) [2]. 11

operational plane The collection of resources responsible for managing the overall operation of individual network devices [2]. 11

Acronyms

APIs	Application Programming Interfaces. 7, 8, 11
ARP	Address Resolution Protocol. 24, 25, 32
CAN	Campus Area Network. 13
CLI	Command Line Interface. 23, 35
CPSI	Control Plane Southbound Interface. 8, 9
CPU	Central Processing Unit. 22, 25
DAL	Device and resource Abstraction Layer. 10
DoD	Department of Defense. 52
DoS	Denial-of-Service. 17
EBI	East Bound Interface. 8
ECC	Error Correction Code. 22
FDPL	Forward Direction Packet Loss. viii, x, 27, 29, 46, 47, 50, 69, 70, 71
FDRT	Forward Direction Path Re-Provisioning Time. 29, 38
HTTP	Hypertext Transfer Protocol. 8
ICMP	Internet Control Message Protocol. 25
IDS	Intrusion Detection System. 5
IETF	Internet Engineering Task Force. iv, 2, 6, 16, 19, 51, 54

IoT Internet of Things. 13, 19

IP Internet Protocol. 24, 35, 37

KVM Kernel-based Virtual Machine. 20, 22

LAN Local Area Network. iv, 3, 13, 24

LLDP Logical Link Discovery Protocol. 31, 32

LTS Long Term Support. 22, 23, 25

LXC Linux Container. 22

MAL Management Abstraction Layer. 11

MPSI Management Plane Southbound Interface. 9

NBI North Bound Interface. 7, 8, 12, 24, 30

NDP Network Discovery Protocol. 32

NEMO Network EMulatOr. 16

NRT Network Re-provisioning Time. iv, v, viii, x, 2, 3, 18, 20, 27, 28, 29, 33, 37, 40, 41, 42, 43, 44, 45, 46, 48, 49, 50, 51, 52, 53, 54, 66, 67, 68

NSAL Network Services Abstraction Layer. 11

ODL OpenDaylight. iv, 18, 25, 29, 31, 32, 42, 43, 48, 49, 51, 52

ONOS Open Network Operating System. 18, 19, 24, 25, 29, 31, 32, 44, 48, 49, 50, 51

OS Operating System. 15

OSI Open Systems Interconnection. 25

QoS Quality of Service. 12, 13

RAM Random Access Memory. 25

RDPL Reverse Direction Packet Loss. viii, ix, x, 27, 29, 46, 47, 50, 72, 73, 74

RDRT Reverse Direction Path Re-Provisioning Time. 29, 38

REST Representational State Transfer. 7, 30

RFC Request for Comments. iv, v, 2, 3, 4, 6, 7, 16, 18, 19, 20, 23, 26, 27, 28, 29, 30, 33, 35, 36, 41, 48, 49, 51, 52, 54

SBI South Bound Interface. 8, 9, 11, 30, 53

SD-WAN Software Defined Wide Area Network. 13

SDN Software-Defined Networking. iv, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 19, 20, 23, 25, 27, 41, 48, 49, 51, 52, 54

STP Spanning Tree Protocol. 31, 32, 35, 45

TCP Transmission Control Protocol. 18, 37

TG Traffic Generator. 29, 36, 37

UAV unmanned aerial vehicle. 14

UDP User Datagram Protocol. 18, 23, 27, 29, 30, 31, 36, 37

USAF United States Air Force. 1, 52

VAN Vehicle Area Network. 14

VE Virtual Environment. 22

VM Virtual Machine. 13, 20, 22, 23, 24, 35, 38

WAN Wide Area Network. 19

WBI West Bound Interface. 8

WSN Wireless Sensor Network. 13

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD–MM–YYYY)* 16–09–2020	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Sept 2019 — Sept 2021

4. TITLE AND SUBTITLE Evaluating Testing Procedures for OpenFlow Controller Network Re-Provisioning Time	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Jensen, Steven J, MSgt	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, Graduate School of Engineering and Management (AFIT/EN), 2950 Hobson Way, WPAFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENG-MS-21-S-009
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Intentionally Left Blank	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution Statement A:
Approved for Public Release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Software-Defined Networking promises several advantages over traditional networking architectures, but has seen little adoption. Recently codified controller evaluation methodologies have seen little validation without strong statistical analysis of the results. The research developed an environment testing implementing a published Network Re-provisiong Time methodology to evaluate five OpenFlow controllers. The methodology is strong with required parameters but had issues with some edge cases. Further refinement and adding a convergence metric may close the gaps.

15. SUBJECT TERMS

OpenFlow, Network Controller, Network Re-convergence, Network Re-provisiong Time, Software-Defined Networking

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Mark G. Reith, AFIT/ENV
U	U	U	UU	95	**19b. TELEPHONE NUMBER** *(include area code)* (312)785-7777 mark.reith.2.ctr@us.af.mil