PROGRESSIVE NETWORK DEPLOYMENT, PERFORMANCE, AND CONTROL WITH SOFTWARE-DEFINED NETWORKING

DISSERTATION

Daniel J. Casey, Major, USAF

AFIT-ENG-DS-18-M-017

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-DS-18-M-017

PROGRESSIVE NETWORK DEPLOYMENT, PERFORMANCE, AND CONTROL WITH SOFTWARE-DEFINED NETWORKING

DISSERTATION

Presented to the Faculty

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 Doctor of Philosophy

Daniel J. Casey, MS

Major, USAF

March 2018

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-DS-18-M-017

PROGRESSIVE NETWORK DEPLOYMENT, PERFORMANCE, AND CONTROL WITH SOFTWARE-DEFINED NETWORKING

DISSERTATION

Daniel J. Casey, MS
Major, USAF

Committee Membership:

Barry E. Mullins, Ph.D., P.E.
Chairman

Robert F. Mills, Ph.D.
Member

Michael R. Grimaila, Ph.D., CISM, CISSP
Member

ADEDJI B. BADIRU, Ph.D., P.E.
Dean, Graduate School of Engineering and Management

AFIT-ENG-DS-18-M-017

Abstract

The inflexible nature of traditional computer networks has led to tightly-integrated systems that are inherently difficult to manage and secure. To remedy this, new designs have moved low-level network control from hardware into software creating software-defined networks (SDN) that are centrally controlled, programmable, and dynamic. However, augmenting an existing network with these enhancements can be expensive and complex with unpredictable results. This research investigates solutions to these problems through hardware and software.

For hardware, it is hypothesized that an add-on device, or "shim" could be used to make a traditional switch behave as an OpenFlow SDN switch while maintaining reasonable performance. A cost-saving design is proposed to enable experimentation on existing networks and the ability to quantify benefits of upgrading before investing. A design prototype is implemented, tested, and found to cause approximately 1.5% reduction in throughput for one flow and less than double increase in latency, showing that such a solution may be feasible for gaining insight into the value of network upgrades before committing to the costs.

For software, it is hypothesized that a new design built on event-loop and reactive programming may yield a controller that is both higher-performing than existing research-based controllers and easier to program. The contribution is in three blocks that build on each other: a schema, a library, and a framework. The schema provides a means to standardize OpenFlow application programming in any language, an important step in creating a standardized northbound interface and in making network applications portable between platforms. The library node-openflow extends high-performance message encoding and decoding to the Node.js platform. The framework rxdn applies the reactive programming paradigm in a novel way to create network applications that are functional, modular, and intuitive.

The library and framework are benchmarked using the popular Cbench tool and compared with peer projects. The library node-openflow is found to have performance approaching that of professional controllers, however it exhibits higher variability in response rate. The framework rxdn is found to exceed performance of two comparable controllers by at least 33% with statistical significance in latency mode with 16 simulated switches, but is slower than the library node-openflow or professional controllers (e.g., Libfluid, ONOS, and NOX).

Collectively, this work enhances the tools available to researchers, enabling experimentation and development toward more sustainable and secure infrastructure.

Table of Contents

Page

Page

List of Figures

Figure Page

Figure Page

List of Tables

List of Abbreviations

Abbreviation		Page
SDN	Software-Defined Networking	1
AFRL	Air Force Research Laboratory	1
NSA	National Security Agency	1
LTS	Laboratory for Telecommunication Sciences	1
ASIC	application specific integrated circuit	8
CLI	command-line interface	9
API	Application Programming Interface	10
AN	Active Networking	13
DARPA	Defense Advanced Research Projects Agency	13
IETF	Internet Engineering Task Force	14
RFC	Request for Comments	14
SNMP	Simple Network Management Protocol	14
NETCONF	Network Configuration Protocol	14
OF-CONFIG	OpenFlow Configuration and Management Protocol	14
OVS	Open vSwitch	14
OVSDB	Open vSwitch Database	14
ForCES	Forwarding and Control Element Separation	15
MAC	Media Access Control	16
IP	Internet Protocol	16
VLAN	Virtual Local Area Network	19
MPLS	Multiprotocol Label Switching	19
DSL	domain-specific language	23

PROGRESSIVE NETWORK DEPLOYMENT, PERFORMANCE, AND CONTROL WITH SOFTWARE-DEFINED NETWORKING

I. Introduction

1.1 Background

Software-Defined Networking (SDN) has become a popular vehicle for research in recent years, with its own dedicated conferences, books, and coverage by top-tier journals. In its 2014 list of cyber research and development challenges, 24th Air Force specifically asked for investigations into the security implications of deploying SDN across the enterprise network. The Air Force Research Laboratory (AFRL) Rome Laboratory, the National Security Agency (NSA) Research Directorate, and the Laboratory for Telecommunication Sciences (LTS) all have ongoing research and have expressed a desire to work with other researchers in this area.

The field of SDN-related research is broad, but the protocols, hardware, and tools to develop, research, and use such networks are limited and immature. The resources are also costly, as the hardware and software which support SDN are still specialized compared to traditional network hardware. Even once the hardware and software are procured, it is still a difficult undertaking to properly program applications to control the network in line with operational needs.

These factors contribute to slowed adoption of SDN technologies across the industry. As SDN shows the most promise for future networking advancements, slowed SDN adoption translates into stagnation in research innovation.

This work seeks to provide resources to promote SDN adoption. Two areas of particular interest are first, lowering the barrier to entry for SDN and second, improving the means of its deployment, performance, and scalability.

1.2 Problem Statement

SDN is a tool that can be used to solve problems related to networking; however, SDN itself introduces a host of new problems. Some of the key problems include:

1. Hardware: Upgrading hardware to support SDN is expensive and risky, as it is difficult to understand the effects of switching an existing network to SDN in advance.
2. Controllers: There is a lack of easy-to-program, high-performance controller software suitable for testing and research.
3. Applications: Existing network control applications are complex and not modular.

The problem is the combination of these hurdles make it prohibitively difficult for network researchers and system administrators to adopt SDN.

1.3 Research Goals and Hypotheses

The core problem that this work seeks to address is improving the resources available to ease and encourage adoption of SDN. Enhancing the tools available will enable experimentation and development of more sustainable and secure network infrastructure.

The key problems listed above are explored separately with the following hypotheses:

1. Hardware: It is possible to create an external device that could be added to existing network infrastructure that would be inexpensive (relative to the cost of new hardware) and perform well enough to enable experimentation and step-by-step adoption in SDN.

2. Controllers: It is possible to use newer programming paradigms to create a library—the foundation of a controller—that will be easier to program and higher performing than existing research-oriented controllers.

3. Applications: It is possible to adopt programming paradigms from web application development to enable intuitive and modular programming of control applications for an SDN controller.

1.4 Approach

Again, the approach is broken across the three domains:

1. Hardware: Real network hardware is used to design, implement, and test the best-case performance of such an external device that would allow SDN functions to be added to existing hardware. The system is tested in terms of latency increase and bandwidth reduction versus a switch without the add-on device.

2. Controllers: A new, fully-functional OpenFlow library is written and tested against its peers using a popular benchmarking tool with $n = 36$ replications. The results are analyzed graphically, with Welch's F test, and with *post hoc* methods, and shown to be statistically significant.

3. Applications: A new framework for developing network control applications in a consistent, modular, and intuitive way is designed. Its performance is measured alongside and in the same way as the library developed above.

1.5 Assumptions/Limitations

For all three aspects, while the term SDN is used broadly, the designs and programs developed focus only on the OpenFlow protocol. While there are other suitable protocols, OpenFlow is currently the representative and accessible (non-proprietary) protocol for this work. The concepts would translate to most other SDN-related protocols, but not without porting all code.

Domain-specific assumptions and limitations include:

1. Hardware: The design tested is a minimal prototype to determine the best-case for increased latency and decreased bandwidth due to the addition of such a device to a traditional switch. Therefore, the device designed does not implement OpenFlow, but makes the connected switch behave as an OpenFlow switch that has been preconfigured with forwarding rules by an OpenFlow controller. It is assumed that such a design may be useful in low-to-medium traffic environments, test labs, and so on, but will not hold up to high levels of network traffic.

2. Controllers: The library implements OpenFlow protocol versions 1.0 and 1.3 only. Version 1.0 is needed for compatibility with the benchmarking tool (Cbench), and 1.3 is currently the most widely-adopted version. Professional controllers are expected to support these and more, including non-OpenFlow protocols.

3. Applications: As the framework is developed on top of the library, the same assumptions and limitations apply.

1.6 Research Contributions

Contributions by domain include:

1. Hardware: A design for a device and configuration for almost any traditional switch is developed such that the device will usurp all control from the switch, allowing that system to behave like an OpenFlow switch. Performance measurements show that bandwidth reduction for a single host is minimal and additional latency is lower than expected, with approximately 1.5% reduction in throughput less than double increase in total latency.

2. Controllers: A fast OpenFlow library is contributed that is easy to develop for in a variety of accessible languages and is shown to perform significantly better than several existing controllers.

3. Applications: A novel framework is contributed that features single-threaded programming concepts with familiar language semantics while exhibiting moderately fast performance and highly modular design.

1.7 Overview

Chapter II provides a background on SDN and related technologies relevant to this work.

The foundation of computer networking is hardware. The capabilities and limitations of hardware dictate what is possible at all layers of the protocol stack. SDN is a new paradigm for networking that permeates the entire stack, even requiring new hardware. Restructuring an enterprise network to transition to SDN can be extremely difficult even if the hardware is already capable. The added costs and difficulty of replacing hardware simultaneously with a restructure puts an SDN transition out of the realm of possibility for all but the best-resourced organizations. Innovation is needed to overcome these difficulties and bring the advantages of SDN to existing networks. Chapter III explores the challenge of transitioning a traditional network to a modern SDN architecture in a controlled, non-disruptive manner.

From the other end of the network stack, SDN controllers are high-level programs that control the behavior of network devices. Existing controllers tend to fall into one of two categories: commercial controllers designed for enterprise use and research controllers created by graduate students. Those in the former category tend to be high performing, but costly and difficult to understand and manage. Those in the latter category tend to be flexible, but lacking in performance and limited to a specific research objective. There is a void left between these categories, and Chapters IV-VI explore the challenges of creating a new controller that is general-purpose, flexible, research-oriented, easy-to-understand, and yet performs well under load.

Chapter IV contributes a schema for the OpenFlow protocol that provides for message creation and validation, allowing verified, interoperable interfaces that are language- and platform-agnostic. Chapter V explains the choice of language and platform as well as the design, testing, and performance of the new OpenFlow protocol library. Chapter VI describes the design of the controller framework, its modularity, distribution, testing, and performance results. Chapter VII covers the testing methodology and results for performance testing of the library and framework separately and against other similar projects.

Chapter VIII concludes by reviewing the current state of SDN and motivation for this work. It also restates research conclusions, contributions, limitations, and recommendations for future work.

II. Software-Defined Networking

2.1 Introduction

This chapter provides an introduction to SDN, including its core principles, precursors, and components. The definition and terminology of SDN are given in Section 2.2, with motivations for SDN in Section 2.3. A few related works that preceded SDN are described in Section 2.4, with OpenFlow explained in Section 2.5.

2.2 Definition and Terminology

2.2.1 Definition.

SDN is a popular topic both in research and industry, with hundreds of papers and products reiterating basic concepts of the architecture. As a result, the terminology is not always consistent. This section briefly covers these introductory concepts using the most popular terminology and widely-accepted concepts.

A software-defined network is typically defined as a network in which the management, control, and data planes have been separated. This definition implicitly refers to the network hardware on which these functions exist, like routers and switches, collectively called forwarding devices. This separation enables forwarding devices to be simplified, programmable, and centrally-controlled.

2.2.2 Traditional Networks.

In a traditional network, forwarding devices contain the logic for rapidly transferring data (the data plane), the logic for determining how data is to be forwarded (the control plane), and the interfaces or protocols to specify configuration and policy (the management plane). Figure 1 depicts these three planes and the interactions between them.

2.2.2.1 Data Plane.

The data plane is implemented in hardware as an application specific integrated circuit (ASIC). As an ASIC, its functions are relatively simple but extremely fast. Its job is to move data through the device and on to its destination according to rules in its forwarding table. A set of related traffic is referred to as a flow. Any flows not covered by the forwarding table are redirected to the control plane.

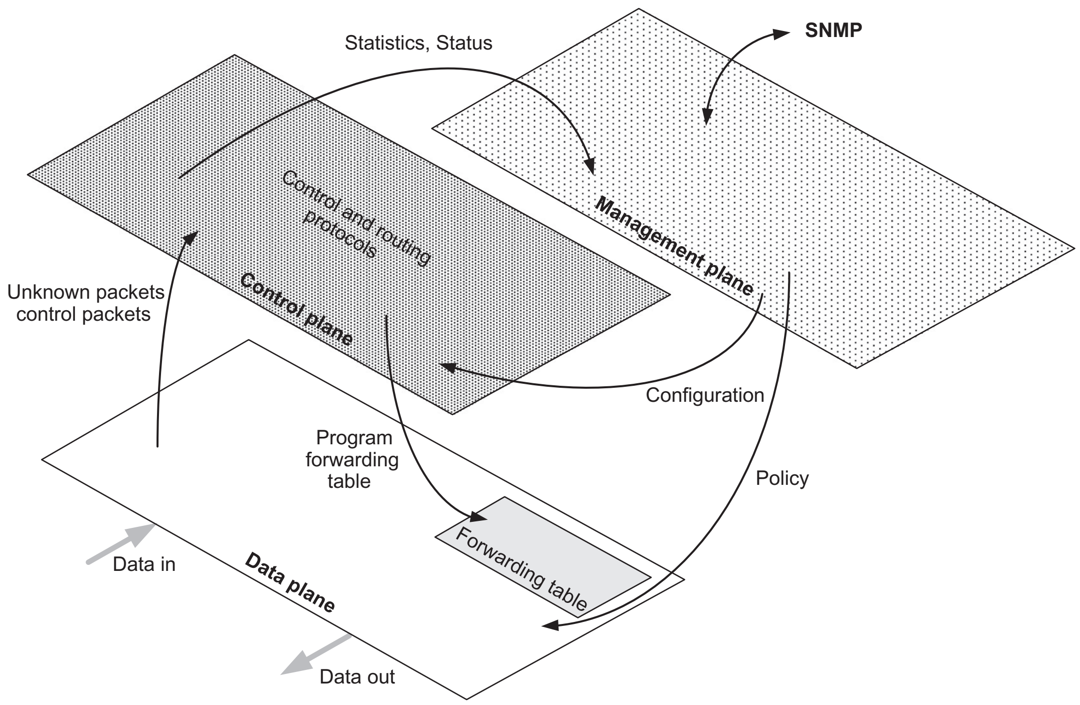


Figure 1. The three network planes and the interactions between them [1]

2.2.2.2 Control Plane.

The control plane is implemented with a general-purpose processor, making it much slower than the data plane, but much more flexible and able to perform complex computations. It is responsible for populating the forwarding table, determining proper actions for flows not handled by the forwarding table, and reporting status and statistics to the management plane.

2.2.2.3 Management Plane.

The management plane consists of any methods by which a network administrator specifies configuration and policy to the forwarding device. This includes the command-line interface (CLI) and network management protocols. Regardless of method, this configuration is relatively static; that is, it does not dynamically update with changing network conditions or other inputs.

Several network management protocols were developed before the advent of SDN, a few of which are described in Section 2.4. Being part of the management plane, these protocols may query and configure certain runtime parameters of the devices, but do not operate directly on flows like the control plane.

2.2.3 Software-Defined Networks.

One of the distinct characteristics of SDN is the separation of data and control planes such that control and management of the network can be accomplished independently of the forwarding devices [1]. This new architecture enables centralization of control, programmatic control, and simplification of forwarding devices.

2.2.3.1 Data Plane in SDN.

The forwarding devices communicate with the controller via a standardized protocol, which is called the southbound protocol (or southbound interface), and these links are normally illustrated below the controller (hence the term "south"). These devices can be much simpler than a traditional switch or router, as they do not require forwarding algorithms or routing protocols. Instead, forwarding devices maintain a flow table that is populated by the controller. Most forwarding devices available today are hybrid devices, meaning they contain all the logic of a traditional device with SDN functions added.

2.2.3.2 Control Plane in SDN.

In SDN, the control plane is removed from forwarding devices, implemented in software on server (or set of servers), and centralized [2]. One SDN controller is responsible for controlling many forwarding devices. The controller is the focal point of the network, and in diagrams is normally drawn in the center, as shown in Figure 2, with applications above and forwarding devices below. More sophisticated controllers are sometimes called "network operating systems." This title implies analogies to computer operating systems, as the forwarding devices are like the computer hardware, the interfaces to the forwarding devices are like device drivers, and the management applications that run on the controller are like computer applications running on an operating system.

2.2.3.3 Management Plane in SDN.

The management plane contains the network applications that program or configure the controller. The interface between the controller and these applications is referred to as the northbound interface and is usually a well-documented Application Programming Interface (API) specific to the controller.

2.3 Motivation

By leveraging this new architecture, networks can be made more dynamic to changing conditions, less costly to acquire and maintain, and more flexible to experimentation. Traditional networking, while effective and time-tested, has not significantly changed in decades. This stagnation in innovation has led to increasingly complex networks. Benson et al., in attempting to quantify network complexity, observe that university and enterprise networks tend towards higher complexity as the network grows over time, which "generally leads to significant manual intervention when managing net-

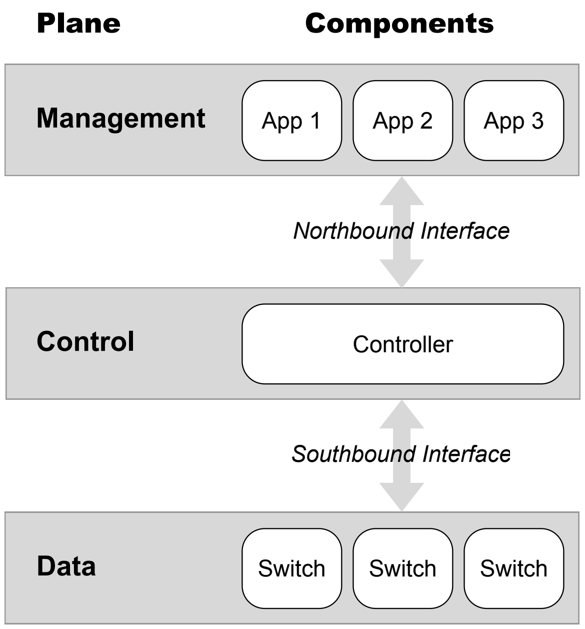


Figure 2. Typical SDN architecture [adapted from 2]

works", resulting in networks that "are more prone to failures, and are difficult to upgrade and manage" [3].

Traditional networking is highly distributed. Each switch and router has its own set of networking protocols and algorithms to determine proper forwarding of frames or packets. To be high-performing, configurable, and independent, each device contains an enormous amount of complexity. Because devices are independent, network operators must perform the manually-intensive task of writing configuration files specific to each device, connecting to each device individually, and loading and testing those configurations. Depending on a number of factors (e.g., routing protocols, active subnets), a small change to one portion of a network may require connecting to many devices across the larger network to update configurations, as depicted in Figure 3. While newer protocols and technologies can ease these transitions, Benson et al. found that due to costs, these newer technologies are often not available to network administrators [3]. By centralizing the control plane, SDN advocates argue that the new architecture is capable of eliminating these classes of problems.

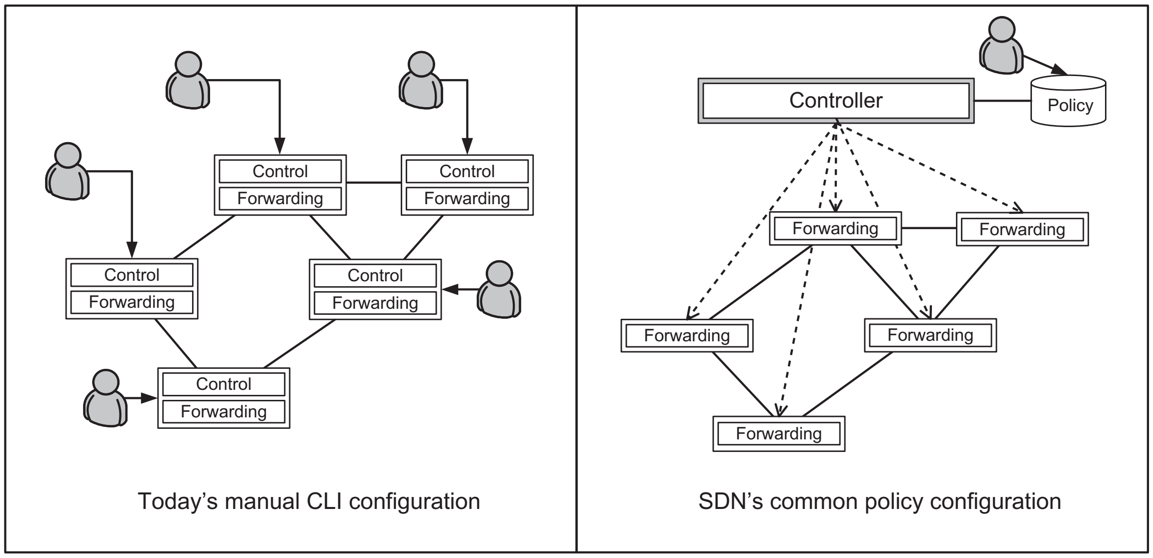

Control
Forwarding
Today's manual CLI configuration
Controller
Policy
Forwarding
SDN's common policy configuration

Figure 3. Network management: traditional versus SDN [1]

The growing costs of network hardware are also a target problem for SDN. In the SDN architecture, traditional switches and routers become simple forwarding devices, greatly reducing their complexity, development timelines, and costs. By moving to a standardized set of features and protocols, entrenched industry leaders are starting to face competition from small specialists providing incredible value for customers on commodity hardware. These cost gains were first established in data centers, as explained in [4]: "the cost of hiring engineers to write sophisticated control programs to run over large numbers of commodity switches proved to be more cost-effective than continuing to purchase closed, proprietary switches that could not support new features without substantial engagement with the equipment vendors." This historical survey goes on to explain that the desire of such customers to break free from overpriced, locked-in platforms provided the initial momentum to make SDN a powerful trend in the industry.

2.4 Precursors to SDN

Two main research themes in networking predated and contributed to the development of SDN: network programmability and separation of control and forwarding.

2.4.1 Active Networking.

In the mid-to-late 1990s, a technology called Active Networking (AN) was developed as a joint research effort among research universities and the Defense Advanced Research Projects Agency (DARPA) [5]. The goals of AN were to increase the rate of evolution and customizability of networks by making them programmable. AN was never widely deployed or adopted by the community at large, in part due to its complexity. One of the proposed implementations of the technology was to have small, executable bits of code called "capsules" that would be written by network users in Java or Tcl and executed dynamically by switches or routers in the network [6]. Another was via an out-of-band API. With either approach, instead of passively forwarding data, the network would become a flexible, programmable mechanism. A strong use case for AN was the consolidation of the functions of ad hoc network devices, including firewalls, proxies, and gateways to simplify network architecture and reduce the administrative burden. In this way, the foundational ideas and use cases of AN are similar to those of SDN.

SDN also provides network programmability, but by different means [4]. Instead of programming functionality to be executed dynamically on routers or switches, programmatic control in SDN is achieved at the centralized controller that can dynamically affect forwarding device behavior. Rather than making distributed devices more complex, as in Active Networking, SDN allows them to be greatly simplified.

2.4.2 Management Plane Protocols.

The Internet Engineering Task Force (IETF) has released a series of memoranda in their Request for Comments (RFC) format which describe protocols and data formats for monitoring and controlling networked devices. While these operate on the management plane and not the control plane, they are notable as SDN precursors for attempting to solve the need for more centralized control of network devices by providing some degree of network programmability. In some cases, they provide a foundation for SDN systems or an alternative means of control by multilingual SDN controllers.

Simple Network Management Protocol (SNMP) was first developed by the IETF as RFC 1067 in 1988 [7]. This protocol has been extended over the years and remains one of the most ubiquitous networking protocols in use today. The protocol was initially developed to provide monitoring and control features to any network connected device, including servers, but a lack of vendor support has mostly relegated it to router monitoring roles [8].

The Network Configuration Protocol (NETCONF) was developed by an IETF working group in 2006 as RFC 4741 to be a modern and extensible replacement for SNMP [9]. The OpenFlow Configuration and Management Protocol (OF-CONFIG), published in 2011, is built on top of NETCONF [10].

Open vSwitch (OVS) is a popular software switch implementation which supports OpenFlow, but provides its own management plane protocol, Open vSwitch Database (OVSDB), which is independent from OpenFlow [11, 12].

2.4.3 ForCES.

Some of the earliest work in centralized network control goes back to the design of telephony networks and Signalling System No. 7 in the 1970s [13]. In 2004, an

IETF working group developed Forwarding and Control Element Separation (ForCES), which contributed a standardized interface between separated control and data planes to make each more flexible to change. By making the two planes dependent upon a standard interface instead of locked to each other, it was thought that ForCES would enable rapid innovation of both [14]. While there was research that built on top of ForCES, it failed to catch on with device vendors, and therefore never made it beyond research prototypes. Around 2008, OpenFlow was developed by a joint team at Stanford and Berkeley.

2.5 OpenFlow

The OpenFlow team learned the lessons of Active Networks, ForCES, NETCONF, and other projects that failed to catch on, and developed the first open interface between data and control planes to be widely and fully adopted by vendors [4]. OpenFlow is currently the most popular "southbound interface" protocol for SDN, the means by which a controller communicates with and configures forwarding devices. The three main components of an OpenFlow-based network include its hardware, software, and the protocol itself.

2.5.1 OpenFlow Hardware.

OpenFlow was first proposed in 2008 as a compromise to bridge the gap between researchers and vendors. Researchers needed a mechanism for low-level flow control on real switching hardware to be able to rapidly develop and test new protocols and architectures. Vendors, however, were understandably reluctant to expose the inner workings and trade secrets of their devices, which they saw as contributing greatly to their hard-earned market shares. Determined to avoid the stalemate of ForCES, OpenFlow was designed to expose the minimum-needed control in a platform-agnostic way without requiring the opening of vendors' hardware. This allowed greater steps in research, as prior options were all unsuitable: "the commercial solutions are too closed and inflexible, and the research solutions either have insufficient performance or fanout, or are too expensive" [15].

An OpenFlow switch provides at least one flow table and a control channel to at least one controller (via TCP or TLS) [15]. Figure 4 depicts the basic components of an OpenFlow switch and its control channel to an external OpenFlow controller.

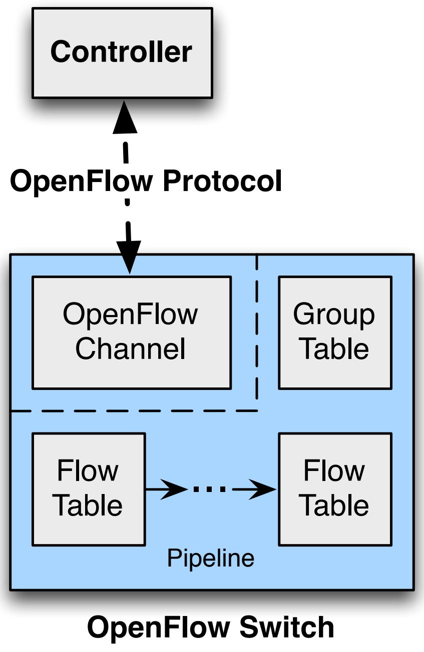


Figure 4. The basic components of an OpenFlow switch [16]

Flow tables are configured by the controller and contain flow entries. Each flow entry specifies a match that ties the entry to the header fields of a given flow. For example, one entry may match any flows where the first packet is destined to a particular Media Access Control (MAC) or Internet Protocol (IP) address, while another may match on the source TCP port number. Each flow entry may also provide counter mechanisms for the controller to query and gain statistical information about the number of packets matching each flow entry. Finally, each entry contains an action to be performed on matching flows, such as forwarding the packets out of a given switch port, sending matching packets to the controller, or (in versions 1.1 and later) specifying an alternative flow table pipeline. The control channel protocol is the OpenFlow switch protocol itself [16].

OpenFlow 1.0 (2009) and earlier versions specified a single flow table to be supported in hardware, with 1.1 (2011) and later providing the option for multiple tables to be supported by hardware. Multiple tables, when supported by the hardware, act as a one-way pipeline to provide more sophisticated flow control and better performance. Figure 5 depicts a pipeline of n flow tables inside an OpenFlow switch, with each packet that passes through the switch potentially picking up new metadata and actions at each table. At the end of the pipeline, any actions attached to that packet are applied.

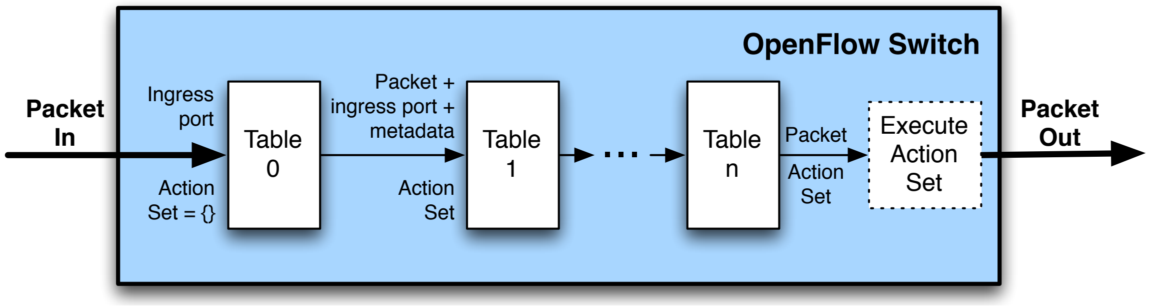


Figure 5. OpenFlow packet matching through multiple tables [16]

Figure 6 is a flow chart showing the order in which matches are examined for each packet and table, and if a match exists, how updates may occur to possibly modify the packet before forwarding it. Figure 6 should be viewed as the logic that occurs when a packet is sent to any of the tables of Figure 5. The first decision in Figure 6 is to find whether there exists a match for this particular packet in the table. If there is a match, any actions are applied, and the packet may continue to be processed by other tables if a "goto-table" statement exists. If there is no match in the current table, the last decision of the flow chart is to check the table for a miss flow entry. The miss flow entry is a catchall action that matches any unmatched packet. Note that the default action if no match exists is to drop the packet; an unconfigured OpenFlow switch, by specification, forwards no traffic.

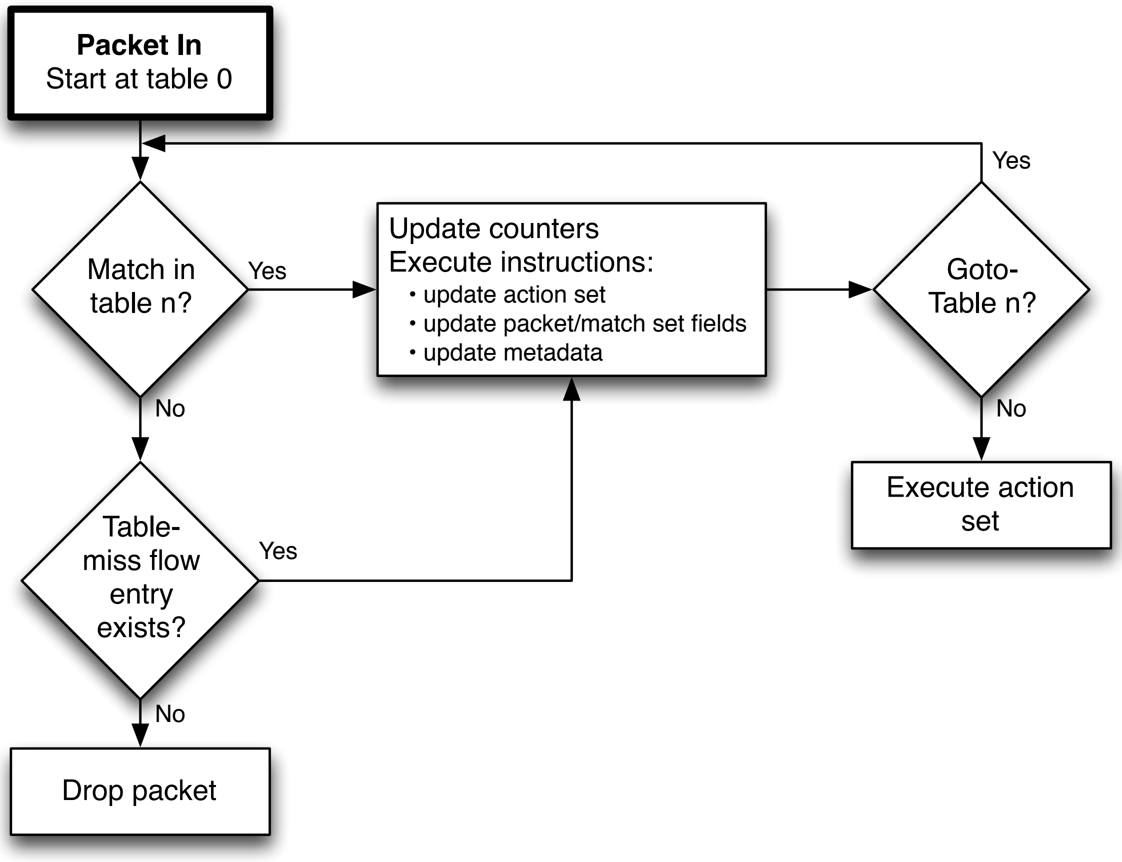


Figure 6. Flow chart for OpenFlow's Match, Action, and Instruction [16]

2.5.2 OpenFlow Protocol.

OpenFlow protocol versions are generally released every 12–18 months, as shown in Table 1, and the number of header field matches and message types has been steadily increasing since version 1.0. Each protocol specification starts by defining the concepts and requirements for a complying switch. Certain features are optional, and these are queryable by the controller through various features-request messages. Some examples of optional features include multiple flow tables, certain virtual switch ports (e.g., to specify "normal" or non-OpenFlow based processing of a matching flow), certain match fields (e.g., pushing and popping Virtual Local Area Network (VLAN) tags and Multiprotocol Label Switching (MPLS) headers is supported but not required), and certain counters for statistics gathering by the controller. A controller that tries to use an optional feature that is not implemented should be sent a corresponding `Error` message by the switch.

Table 1. OpenFlow switch protocol versions [derived from 16]

OpenFlow version	Match fields	Messages	Release date
1.5.1, 1.4.1, 1.3.5	36, 35, 30	45, 42, 40	April 2015
1.5.0	36	45	December 2014
1.3.4	30	40	March 2014
1.3.3	30	40	September 2013
1.4.0	35	42	October 2013
1.3.2	30	40	April 2013
1.3.1	30	40	September 2012
1.3.0	30	40	June 2012
1.2	26	36	December 2011
1.1.0	24	15	February 2011
1.0.0	22	12	December 2009

The protocol specifies the initial handshake procedure whereby a controller and switch establish a connection and negotiate their highest common operating version by way of `Hello` messages. After session establishment and initial configuration, the most important messages are `PacketIn`, `PacketOut`, and `FlowMod`. While these can vary in their implementation from version to version, each version has and makes extensive use of them to provide SDN-style control by the controller over the switch. The switch may notify the controller of a new flow with a `PacketIn` message. This message includes details about the packet, including a unique transaction ID (xid), header fields, and possibly all or part of the actual frame. A buffer ID may be given that the controller may use to reference the particular packet in future messages. Similarly, the controller may send a `PacketOut` message to the switch in order to have the switch send the packet out one of its ports. This `PacketOut` message may include the full packet itself or a buffer ID referring to a previous packet stored in the switch's memory, as well as other information like the output port or pipeline the packet should be processed through. The `PacketOut` message is shown as a C struct in Figure 7 and as a byte diagram in Figure 8.

```
/* Send packet (controller -> datapath). */
struct ofp_packet_out {
  struct ofp_header header;
  uint32_t buffer_id;   /* ID assigned by datapath (OFP_NO_BUFFER if none). */
  uint32_t in_port;     /* Packets input port or OFPP_CONTROLLER. */
  uint16_t actions_len; /* Size of action array in bytes. */
  uint8_t pad[6];
  struct ofp_action_header actions[0]; /* Action list - 0 or more. */
  /* The variable size action list is optionally followed by packet data. */
  /* This data is only present and meaningful if buffer_id == -1. */

  /* uint8_t data[0]; */
  /* Packet data.  The length is inferred from the length field in the header. */
}
```

Figure 7. PacketOut as a C struct [16]

The most interesting message is the `FlowMod`, short for *modify flow entry*, which is shown as a byte diagram in Figure 9. Rather than specify a single packet, this message is sent from the controller to the switch to match any number of header fields or other packet metadata, and includes actions or instructions (depending on the version) of how such matching flows should be processed by the switch. The `FlowMod` may also indicate a soft or hard timeout period, indicating when the switch should remove the rule from its tables after a given period of inactivity or after an absolute period from the rule installation, respectively.

These simple messages provide the basic building blocks upon which all OpenFlow control is based, allowing centralized and programmatic control of thousands of commercially-available networking products today.

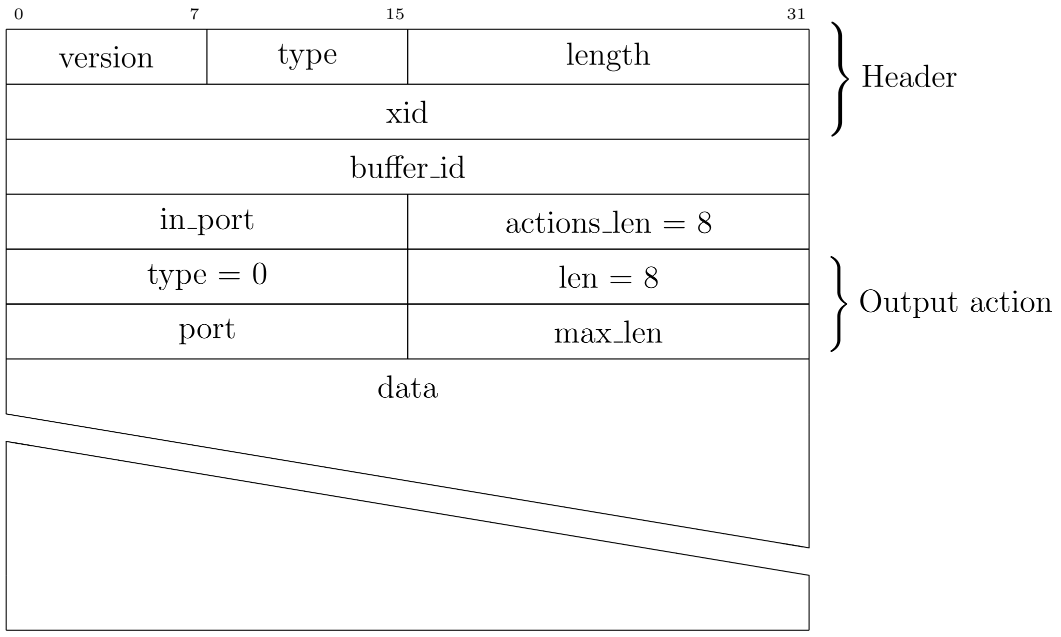


Figure 8. OpenFlow PacketOut with output action, version 1.0 [derived from 16]

2.5.3 OpenFlow Software.

2.5.3.1 The role of the controller.

OpenFlow software includes compatible controllers and any applications written for such controllers. In SDN, a controller is a focal point of the architecture. In its simplest form, it receives a policy, model, or set of instructions from a network administrator or network application, translates these into rules, and installs these rules on network switches. In more complex instances, the controller forms a distributed layer over the physical network and provides a unified, but *logically centralized* view of the network to network applications. While many research controllers are built specifically for OpenFlow, many commercial and enterprise controllers support other southbound protocols and therefore tend to provide abstractions over OpenFlow details at the northbound interface. Either way, as shown in Figure 2, the controller itself represents some form of abstraction from the raw forwarding devices (data layer) to the network

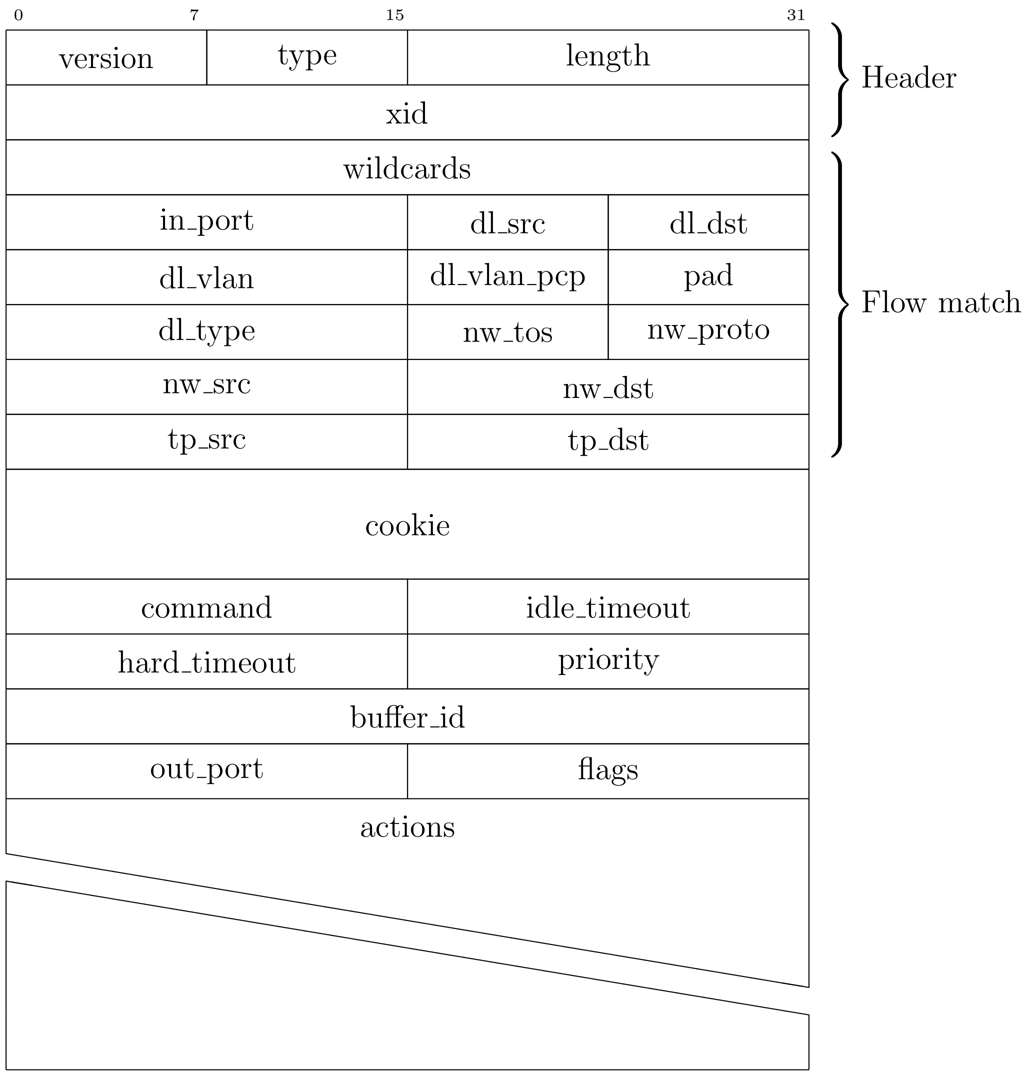


Figure 9. OpenFlow FlowMod, version 1.0 [derived from 16]

applications that define the network policy (management layer). This is summarized well in [17]:

> Because the control platform simplifies the duties of both switches (which are controlled by the platform) and the control logic (which is implemented on top of the platform) while allowing great generality of function, the control platform is the crucial enabler of the SDN paradigm.

There is currently no standard northbound interface, though some have been proposed (e.g., [18]). Some researchers have even argued against the development of such a standard, saying such constraints will provide little benefit and hamper innovation [19]. Consequently, there is little-to-no portability among network applications from one controller to another, and therefore selection of a controller for a particular network demands a comprehensive survey. There is no one way to build a controller, and many open- and closed-source implementations have been defined.

2.5.3.2 First controllers.

NOX [20] and POX [21] were two of the first controllers, and both expose a simple API in C++ and Python, respectively. The programmer uses this API to describe the desired network behavior as a set of functions. These functions are called by the runtime when certain types of messages are received from a switch, and used to react to changing conditions of the network. These controllers were developed at the same time as OpenFlow at Stanford and Berkeley to prove the feasibility and explore the consequences of the SDN architecture [22].

2.5.3.3 Research controllers.

Frenetic-OCaml [23] and Pyretic [24] also use a runtime, but instead of a traditional API, they expose a domain-specific language (DSL) to the programmer in OCaml and a subset of Python, respectively. Therefore, the programmer defines network behavior as a set of compositions in that DSL. The runtime translates these abstract policies to specific flow modification messages that are sent to the switches connected to the controller. From the programmer's viewpoint, the network devices are abstracted into one giant switch [25]. These two controllers are ongoing, joint research efforts at Cornell and Princeton.

2.5.3.4 Enterprise controllers.

Beacon [26], Floodlight [27], and OpenDaylight [28] are Java-based controllers that target an enterprise network environment. As Java is considered to be a language of high developer productivity, the APIs of these controllers are written with developer ease-of-use in mind. OpenDaylight can support various southbound protocols as well as northbound APIs. Besides OpenFlow, southbound protocols may include NETCONF, SNMP, OVSDB, and even proprietary CLI commands and protocols. Similarly, OpenDaylight's northbound APIs support many different programming languages and network protocols.

2.5.3.5 Controller goals.

The controllers listed above were all designed with specific goals in mind. NOX and POX were created as the first control programs for the developing OpenFlow protocol. Frenetic and Pyretic were developed to further the notions of modularity and composition for network control, and the Java-based controllers were developed to provide enterprise-ready, high-performance controllers.

2.5.3.6 Controller themes.

Upon reviewing the various types of controllers, there are a few themes to notice. Research controllers tend to be specialized to a specific area of research (e.g., domain-specific languages) and are generally low-performing. Enterprise controllers tend to place a premium on performance, but are more complex and difficult to modify. In most controllers, much of the OpenFlow protocol specifics are abstracted away. Furthermore, learning one controller does not translate into sufficient knowledge necessary to effectively use another. In other words, programming a routing algorithm in POX, Frenetic, and Floodlight result in vastly different programs, each requiring a substantial investment in learning the particularities of the chosen platform.

III. OpenFlow Shim: Reducing Costs and Disruptions in Hardware Upgrades

3.1 Introduction

While SDN is an evolution of networking that came from research dating back over 20 years, its most widely-used implementation today is through the OpenFlow protocol, which facilitates communication between the controller and switch [4]. Early versions of OpenFlow were very simple, supporting only a few of the typical layer 2 and 3 protocol header fields, and the OpenFlow agents running on switch hardware were crude [15]. These agents were created as custom firmware for select models of traditional switches.

This approach worked as a research vehicle and helped turn the industry in the direction of SDN because it required little or no modification to hardware. However, as SDN concepts have advanced, the community has realized that status quo hardware will not be sufficient.

The prototype design presented here begins the exploration of custom hardware by augmenting an existing hardware switch with newer, SDN-enabling features. The goals are to ascertain if such an idea performs well enough for implementation, such that it mitigates the difficulty of moving an existing network to an SDN architecture.

A secondary interest is to explore dynamic offload of network processing from an SDN controller to its switches. This requires custom switch hardware, as existing devices have no such capabilities. The goal will to be explore how the existing SDN architecture could be improved by rethinking the allocation of computing resources in a network.

This design represents a working foundation upon which many other projects can be built. It is also general enough that it can easily be ported to other development boards, including field-programmable gate array (FPGA) boards from other vendors (e.g., the Altera Stratix 10).

3.2 Background

Upgrading an enterprise network to leverage SDN can be difficult and expensive. The most widely-used SDN protocol, OpenFlow, only reached its 1.0 release in 2011 [4] and has only been available in commercial switches since 2012. Furthermore, the protocol itself is rapidly evolving, with other SDN protocols under active development. These factors add cost and risk to SDN upgrades, which hinder adoption and growth. Moreover, legacy networking equipment may still provide full functionality. As described in [29], "Certainly, rip-and-replace is not a viable strategy for the broad adoption of new networking technologies." Providing cost-effective means to adopt SDN technologies without completely replacing existing infrastructure benefits researchers and end-users alike.

An inexpensive hardware device that usurps flow-level control from a legacy switch is proposed. This "shim" layer can provide SDN features on legacy switches to enable pre-purchase testing and cost-effective infrastructure upgrade planning.

The goals are to determine a viable shim design, where the shim is best deployed and if, despite performance limitations, such a device is practical. The specific performance metrics are throughput and latency of the device while connected to a switch. The design is implemented on a NetFPGA-1G-CML development board which uses a Xilinx Kintex-7 FPGA connected to four 1 Gigabit per second (Gbps) Ethernet PHY chips and ports. The switch is a Cisco Nexus 3048T. It is not a "legacy" switch as it includes support for OpenFlow and other SDN protocols, but was configured to behave as one for the experiments. It is equipped with forty-eight 1 Gbps Ethernet ports and four 10 Gbps enhanced small form-factor pluggable (SFP+) transceiver ports.

3.3 Related work

There are a variety of proposed techniques for gaining control of legacy network devices in an SDN network. They generally fall into the categories of controller-based legacy control, intermediate devices, and network architecture planning.

An example of controller-based legacy control is OpenDaylight, a relatively new and open-source SDN controller. It includes a service adaptation layer which aims to abstract the southbound protocol details from the higher layers by use of protocol plugins [30]. In addition to OpenFlow, NETCONF, etc., plugins can be created to cover various legacy switches. Depending on the particular target switch and control devices exposed (CLI, SNMP, web, etc.), this could be difficult. A heterogeneous switch environment would further complicate the effort. Even a well-written plugin for a specific device might not provide the required resolution of control to cover certain use cases, making the investment questionable.

An example of an intermediate device is given in [31], where the authors propose a system to translate OpenFlow messages on-the-fly into legacy command directives for non-OpenFlow switches. They test this on three different vendors' switches, using command-line, SNMP, and web service configuration to modify the switch behavior in accordance with the OpenFlow messages from the controller. While successful at translating messages for the targeted hardware, this approach has its own drawbacks. Again, it would be necessary to customize the method of configuration control for every vendor, and possibly every model or even software version of a switch. Also, as explained in [29], "One fundamental restriction of this approach is sacrificing the reactive mode of operation of OpenFlow, which packets without a matching rule are forwarded to the controller via packet-in events." As the OpenFlow protocol itself must be altered, many network applications could not run unmodified. Finally, they also found their approach requires substantial modifications to the controller, further distancing their solution from a standard OpenFlow deployment.

ClosedFlow is a more recent effort to include legacy switches in an OpenFlow environment. In [32], the authors target 10-year old Cisco switches with the goal of being able to run unmodified SDN applications. They repurpose the layer 3 routing protocols on the switches, specifically Open Shortest Path First (OSPF). This approach was successful at finding a rough approximation of OSPF to OpenFlow commands to enable SDN functionality on older hardware. The obvious drawback to this approach is that it requires the legacy switches to be multilayer switches that support routing protocols. These switches are much more expensive and typically less abundant than layer 2-only switches.

An example of network architecture planning as an SDN-enabler is given in [33], where the authors propose careful placement of SDN-enabled switches in a network. As long as any flow traversing the network is handled by at least one SDN switch, many of the benefits of SDN can be gained while requiring only a subset of devices be upgraded. Given certain assumptions and a typical, 3-tier enterprise network architecture, they suggest that as few as 10% of the distribution layer switches can be upgraded to achieve SDN management capabilities over the whole network. This approach comes closest to solving the problem and could be used simultaneously with the proposed shim.

3.4 Hardware Shim Design

An OpenFlow "shim" layer implemented with relatively inexpensive hardware could be successful in certain scenarios. The envisioned device will connect to an

OpenFlow controller, present itself to the controller as a regular OpenFlow switch, and control flows on the connected legacy switch in accordance with messages from the controller. An example physical configuration is shown in Figure 10. Design goals include utilizing ubiquitous switch features, supporting all OpenFlow features, minimizing latency, maximizing throughput, matching the frame rate from the switch, and keeping the hardware design simple (and therefore inexpensive).

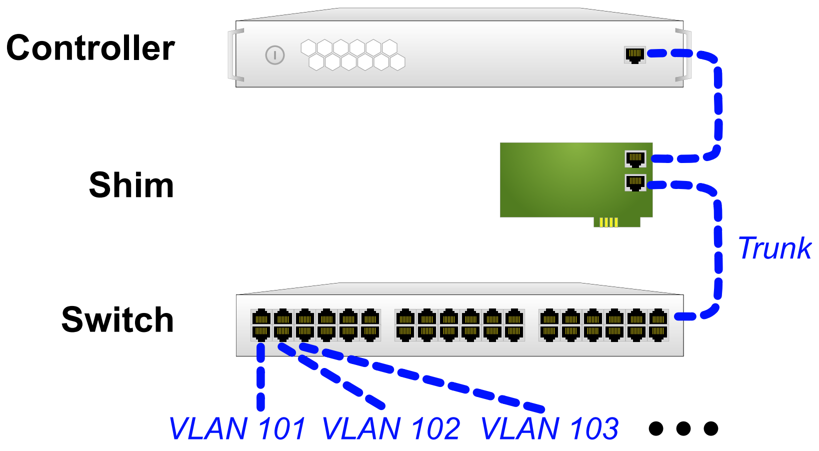


Figure 10. Physical switch/shim configuration showing the insertion of an OpenFlow shim

The only prerequisite switch feature is support for IEEE 802.1Q VLAN tags. This makes the design widely applicable, as VLANs are supported on the vast majority of business class switches manufactured in the last 10 years. A configuration in Cisco IOS format is shown in Figure 11. The legacy switch must be preconfigured with a VLAN trunk to the shim and access ports on unique VLANs. While there are 4094 VLAN IDs available (12 bits with first and last reserved: $2^{12} - 2$), some switches only support a smaller subset of active VLANs. However, this smaller subset is usually more than the number of ports on the device. Switch-connected ports should be configured as trunk ports, but the selection of VLANs for these trunks should be distinct from those used for the host-connected access ports.

```
interface Ethernet1/1
  no lldp transmit
  no cdp enable
  switchport mode access
  switchport access vlan 301
  spanning-tree port type edge
  spanning-tree bpdufilter enable
interface Ethernet1/2
  ...
  switchport access vlan 302
  ...
interface Ethernet1/48
  description netfpga-shim
  ...
  switchport mode trunk
  switchport trunk allowed vlan 301-347
```

Figure 11. Example switch configuration in Cisco format

As each host port resides on its own VLAN, the switch will never pass frames from one port to another, only to the shim. The shim receives all switch traffic, and is able to manage whether each packet is delivered out another port, delivered to the controller via an OpenFlow `PacketIn` message, dropped, modified, or rewritten. The shim therefore contains the primitives of a true OpenFlow switch, with the legacy switch providing a physical extension of ports to the FPGA device. This approach should work even on a network where VLANs are already in use, as long as separate VLAN IDs are chosen and the VLANs allowed on each trunk are carefully controlled.

In contrast with other approaches, this design does not require any modification to the SDN controller, and should be able to support all OpenFlow features. It is widely applicable to legacy switches, requiring only VLAN support, and does not have to be customized to the switch make or model. It can be implemented with relatively inexpensive hardware; suitable FPGA boards are less than \$1,000, while replacing a switch can cost \$20,000.

3.5 Implementation

The system is written in VHSIC Hardware Description Language (VHDL) with Vivado Design Suite 2014.4, targeting the Xilinx Kintex-7 XC7K325T-1FFG676 FPGA on the Digilent/CML NetFPGA-1G-CML [34]. To keep the design small and fast, the existing Digilent/CML code base was not used and new components were created. The design is comprised of three top-level components: the receive and transmit interfaces and the shim wrapper. Secondary top-level components include the clock generator, global reset, and PHY reset, along with debugging components (integrated logic analyzer and virtual I/O).

The VHDL components are depicted in Figure 12. The shim wrapper creates a shim component for each activated interface (the number of active shim interfaces is configurable). Each shim component includes an Ethernet parser (parse) which extracts header details, a modifier (mod) which manipulates the header, and an Ethernet generator (tx) which combines the modified header with the frame payload into a Gigabit Media Independent Interface (GMII) stream. The modifier references the shim configuration memory to map the incoming frame VLAN to the input port of the external switch, which is analogous to OpenFlow's `uint8_t in_port` portion of a `ofp_packet_in` message. Based on the other header parameters and the configuration memory, the modifier selects whether to transmit a modified version of the frame or drop it. If selected for transmission, the output port is encoded as the new VLAN (again analogous to the OpenFlow `uint16_t out_port`), and the Ethernet generator is signaled to generate the modified GMII stream for transmission. The FIFO queue is used to hold the frame payload.

If the frame must be broadcast or output on multiple ports (`OFP_FLOOD` or `OFP_ALL` output ports in OpenFlow), the modifier instead signals the broadcast generator, which keeps a two-stage FIFO buffer for generating n copies of the frame without

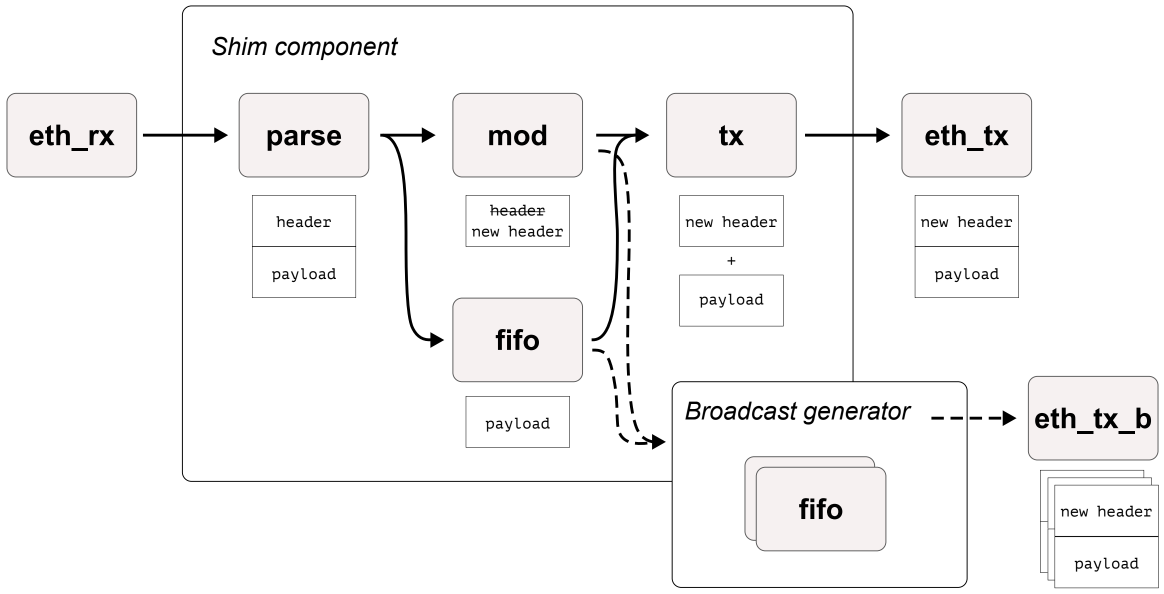


Figure 12. Design VHDL components

blocking the primary pipeline. Of course, given enough sequential broadcasts and a smaller parameterized FIFO for the broadcast generator, the system can drop some broadcast frames. However, dropping frames due to contention is allowed in Ethernet.

This system is designed to match the switch's outgoing frame rate. As the shim is a hardware pipeline design, it is able to track one-for-one with frames coming from the switch. Implementing a single port of the design on the NetFPGA-1G-CML uses only a small portion of the FPGA, as shown in Table 2 (without broadcast offload and using a static VLAN configuration). A custom shim could be developed on a much smaller (and inexpensive) FPGA.

Table 2. FPGA resource utilization (1 port, static flow, Kintex-7)

Resource	Utilization	Available	Utilization %
FF	870	407600	0.21
LUT	582	203800	0.29
Memory LUT	7	64000	0.01
BRAM	15	445	0.34

3.6 Measurements and Results

In order to estimate the design's practicality and limitations in terms of performance, tests were run to find the affect on throughput and latency of traffic going through a shim-connected switch.

The hardware used for testing included:

- Shim: the shim design synthesized and loaded onto one NetFPGA-1G-CML
- Switch: a Cisco Nexus 3048 (N3K-C3048TP-1GE) connected to the shim
- Test server: an Aberdeen Superserver with 384 GB RAM, two 2.30 GHz 8-core Intel Xeon E5–4610 CPUs, and two 4-port Intel 82576 1 Gbps Ethernet cards running Fedora 21 and Docker 1.5.0
- Test FPGA: a second NetFPGA-1G-CML loaded with a custom latency test

No controller was used in testing as the tests were to find the loss of performance in the best-case, isolated to the interaction between switch and shim. Therefore, the shim was pre-loaded with forwarding rules. This mimics an SDN with proactive flow rule installation, or one in which rules for these particular flows have already been installed by a controller. It is clear without testing that in reactive flow handling there would be significant costs to performance, especially latency of the initial packet of a flow.

Throughput and latency were measured with and without the shim. The goal of testing throughput was to estimate the decreased capacity for one stream in terms of bits of data per second. The goal of testing latency was to estimate by how much an individual packet would be slowed by having to transition through the shim, even when there is otherwise no contention for resources.

The expectation is for throughput to be significantly decreased—by 10% or more—and for latency to effectively double. The expectation of double latency stems from the assumption that the processing and re-transmission of packets by the shim takes about the same amount of time as the switch. These expectations are each evaluated with an independent, two-sample t-test.

3.6.1 Test Setup.

Because of the nature of these tests, Transmission Control Protocol (TCP) was used for testing throughput and User Datagram Protocol (UDP) was used for testing latency. TCP is a more realistic measure of throughput as it is a connection-oriented protocol that is typically used when transmitting large amounts to ensure no data is lost or received out-of-order. UDP, a connectionless protocol, has lower overhead than TCP, and is typically used in situations where low latency is a priority over guaranteed message delivery.

Test 1 provides a baseline for the performance of the switch without the shim device. In Test 1, the two switch ports are configured as access ports of the same VLAN. This effectively makes traffic pass normally through the switch from one port to another. Test 2 connects the shim device to the switch as a VLAN trunk, and the two access ports are set to separate VLANs. This forces the switch to forward traffic to the shim only, as described in Section 3.4.

Switch features that may interfere (e.g., STP, CDP, and LLDP) were disabled for all tests. It is assumed that the performance characteristics are the same within the switch regardless of which physical ports were used and which VLAN a port is assigned to.

3.6.2 Throughput test.

Throughput was measured with iperf3, a widely-used, open-source software tool written by Lawrence Berkeley National Laboratory and its Energy Sciences Network. It is designed to measure throughput over IP links. It is implemented as a single-threaded, client-server application and can test TCP, UDP, or SCTP throughput [35].

A pair of containers was used to perform each throughput test with iperf3, with one container attached to its own Ethernet port on the test server, connected to the switch. One container ran as an iperf3 client while the other ran as the server. Containers were pinned to separate CPUs to reduce contention and overhead. The throughput test configurations are depicted in Figure 13. Note that the areas in gray are not part of the test. As stated above, the shim was pre-loaded with flow rules to forward packets.

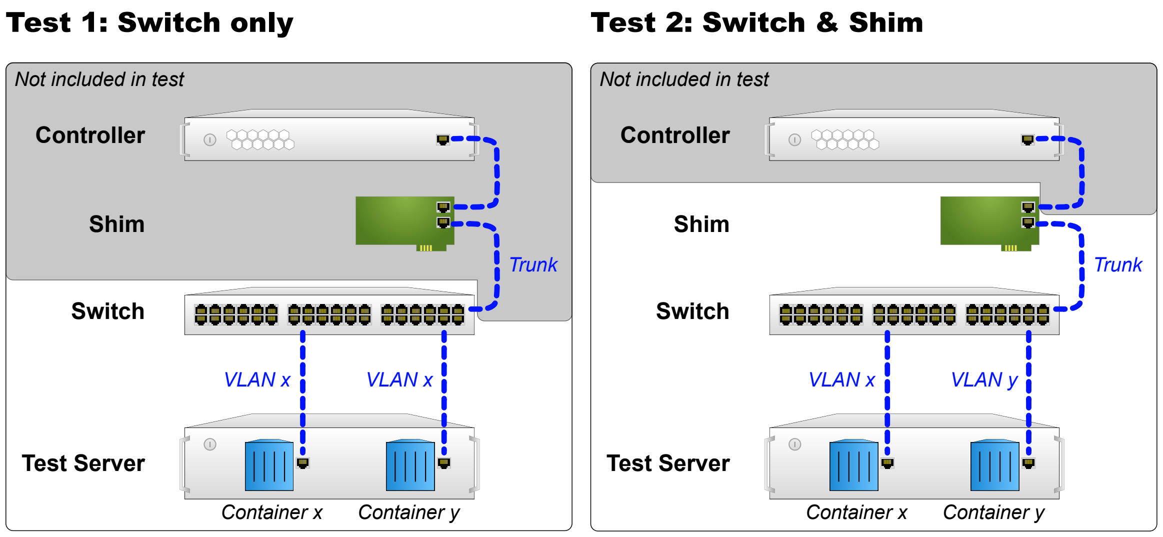


Figure 13. Overview of throughput test configurations

One run of iperf transmits data from the client to the server for one second, ten times, pausing for one second between each transmission, and gives a simple average of the results at the end in megabits per second (Mbps). The version of iperf3 available at time of testing did not report statistics beyond the overall average. However, each

1-second result is printed, and it was these intermediate results which were used for analysis.

Preliminary results using iperf3 indicated an average throughput of the switch alone at approximately 945 Mbps with a sample standard deviation of approximately 6. Given these values, a desired accuracy, and a confidence level, Jain provides an equation for sample size for determining mean

$$n = \left(\frac{100zs}{r\bar{x}} \right)^2 \tag{1}$$

where n is the sample size, z is the z-score for the confidence level, s is the sample standard deviation, $\bar{x}$ is the sample mean, and r is the desired accuracy as a percentage [36]. Jain states that a desired accuracy of r percent implies that the confidence interval is $(\bar{x}(1 - r/100), \bar{x}(1 + r/100))$. For example, a confidence level of 95% would require an r of 5 and z of 1.96.

Applying this equation to the preliminary values and a desired confidence level of 99% results in $\lceil n \rceil = 3$. This low value results from the sample mean being so large relative to the sample standard deviation. From this, it was determined that attaining 20 samples would be more than sufficient.

3.6.3 Latency test.

Latency was measured with a separate NetFPGA-1G-CML in order to get higher resolution timing than possible on a conventional server. This is because the FPGA runs only and exactly the synthesized VHDL, while a computer is interrupt-driven and therefore can make no guarantees on precise timing of events.

The board was loaded with custom VHDL to send traffic on one port and receive it on the other, measuring the transfer time at a resolution of tens of nanoseconds. In order to measure latency independent of congestion, transmissions were separated by one second, similar to the throughput test. Unlike the throughput test, the hardware nature of the FPGA prevented a preliminary trial, so 40 samples was determined to be sufficient regardless of the resulting sample mean and standard deviation due to the central limit theorem. One UDP packet was sent from the sending port per second, 40 times. The time delta from the time sent to the time received on the receiving port was recorded. The latency test configurations are depicted in Figure 14.

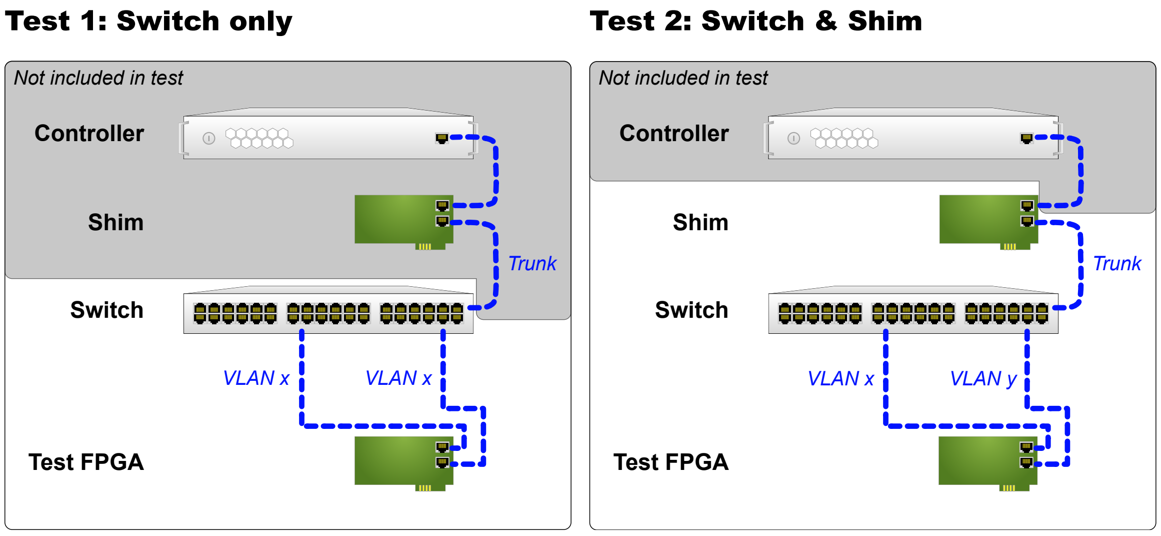


Figure 14. Overview of latency test configurations

3.6.4 Results and Analysis.

Table 3 summarizes the results of each test, with subscript T denoting throughput and subscript L denoting latency. Raw results are given in Appendix A.

Table 3. Performance results

	$\bar{x}_T$	s_T	$\bar{x}_L$	s_L
Test 1: Switch	943.7 Mbps	5.4	3894.6 ns	39.6
Test 2: Switch & Shim	935.7 Mbps	9.3	7567.6 ns	36.7

Welch's t-test is an appropriate test to compare two means, which provides the same results as an independent t-test while correcting for a lack of homogeneity of variance, if necessary [37].

On average, throughput for the switch alone ($\bar{x} = 943.7$, SE $= 1.197$) was significantly higher than the switch with shim ($\bar{x} = 935.7$, SE $= 2.083$), $t(30.3) = 3.33$, $p < 0.01$, $r = 0.512$. Latency was significantly lower for the switch alone ($\bar{x} = 3894.6$, SE $= 6.3$) than the switch with shim ($\bar{x} = 7567.6$, SE $= 5.8$), $t(77.5) = -430.3$, $p < 0.01$, $r = 0.9998$.

The 99% confidence interval ($\alpha = 0.01$) for difference in means for throughput is $(1.4, 14.6)$. This interval does not include zero and is positive, indicating significance at this confidence level and a higher average throughput for the switch alone. This matches with expectations. Additionally, the higher-bound of the interval is 14.5, which indicates a percent difference of only $\frac{14.6}{943.7} = 0.015$, or about 1.5% reduction in throughput when adding the shim under these conditions, which is less than expected.

The 99% confidence interval ($\alpha = 0.01$) for difference in means for latency is $(-3695.539, -3650.461)$. This interval does not include zero and is negative, indicating significance at this confidence level and a lower average latency for the switch alone. This matches with expectations.

However, with a sample mean of 3894.6 ns for the switch alone, the additional average latency of up to 3695.539 ns for adding the shim is *less* than double the original latency. This indicates that it is possible that the shim in its current configuration is able to forward packets faster than the switch. Alternatively, the assumption that the switch behaves the same in both test conditions may be false. In either case, the total system of switch with shim resulted in less additional latency on average than expected.

The latency measurements met expectations based on the switch specifications, which indicate a latency between 2.7 and 7.2 microseconds.

While the increased latency and reduced throughput may be negligible for this simple set up, it is expected to be a larger issue with contention in the switch when there are multiple simultaneous streams that all need to be fed through the shim.

3.7 Potential Issues and Mitigations

The most significant issue is decreased throughput as the configuration creates a bottleneck at the switch port leading to the shim. A single switch port simply cannot handle the traffic of all the other ports at the same time. While this design would not be the best fit for a high-traffic, throughput-sensitive network, there are a few approaches that can help alleviate the issue.

If available, order-of-magnitude higher-bandwidth ports can be used on both the shim device and switch. For example, many switches have a few higher-speed uplink ports, e.g., 10 Gbps ports on an otherwise 1 Gbps switch, or 40 Gbps ports on a 10 Gbps switch. A variety of 10 Gbps SFP+ FPGA boards are available for less than $3000. As another approach, many FPGA development boards have two or four network ports; the bandwidth of these could be combined with a link aggregation protocol, albeit to the detriment of the number of available switch ports.

The design also introduces added latency to each flow. The amount of latency is proportional to the amount of traffic and number of ports the shim is required to handle. The mitigating approaches outlined above also help reduce added latency. Additionally, the choice of FPGA as a platform and the specific implementation of that design keep added latency to a minimum, as opposed to a purely software-based design.

While the design introduces some additional latency, it is less than 10 microseconds for most flows. However, the added latency is more profound with broadcast or multicast traffic. Since each port is on a separate VLAN, there is no shared broadcast domain. As a result, when the shim must deliver a frame to multiple ports, it must sequentially produce a copy of that frame for each destination port. The amount of time required to transmit one broadcast frame is more than n times that of a traditional switch, where n is the number of ports. This also results in skew between the broadcast times of the first and last ports, which could negatively affect some applications. The time the shim spends sending all these frames precludes it from sending any other traffic, which could result in a backlog of traffic and potentially many dropped frames.

One technique to avoid the broadcast latency issue is to use a feature that retains isolation between host ports while providing a shared broadcast domain, like protected or isolated ports or Private Virtual LANs (PVLANs). These features are generally the same, but have different names and nuances across vendors and models. Since the shim is on an unprotected or promiscuous port of a primary VLAN, it could send broadcasts to all other ports with a single frame. The issue with using PVLANs is that it removes the ability of the shim to distinguish between individual ports. The shim could potentially target certain ports by keeping track of MAC addresses and relying on the MAC learning of the switch, but the device could no longer be a drop-in replacement supporting OpenFlow.

Another technique to mitigate broadcast latency, resulting backlog, and dropped frames is to utilize a separate port and processing pipeline of the shim for sending broadcasts. This keeps the primary port of the shim dedicated to unicast transmissions only, which are higher priority in most situations. When a broadcast is needed, it is sent to a separate pipeline within the FPGA that transmits packets out a secondary port connected to the switch. The delay for the primary pipeline is identical to that of a unicast frame.

While these approaches can mitigate the issue, the design will always result in some reduction in bandwidth. Whether this is acceptable depends upon the nature of the network; the shim may not be an acceptable choice for a network operating near its maximum capacity. On the other hand, an existing network could be safely partitioned to allow experimental deployment of the SDN architecture on the existing hardware.

3.8 Future work

Continued design development is planned to make the design more relevant to high-traffic environments. One approach is to move development to a 10 Gbps board and use 10 Gbps switch uplink ports for the shim trunk.

Another approach, which can be combined with the move to 10 Gbps, is to implement handoff between output ports on the shim; the switch will be partitioned into three or four sets of ports, with each set being handled by a different port on the shim. Frames that cross partitions will be transparently handed off in the FPGA. It is expected that this will improve contention by a factor of the number of partitions.

Finally, updating the design to include an OpenFlow agent will make the shim a stand-alone, drop-in SDN enabler. An OpenFlow agent is the intermediary between an OpenFlow controller and the data plane. An agent on the shim would make the combination shim and switch appear and act as a standard OpenFlow switch to a controller. The agent may be implemented as a soft-core processor (MicroBlaze on Xilinx FPGAs), an ARM-based mezzanine card, or on a host CPU as in [38]; these options will be compared.

The upgrade path to a modern SDN network architecture can be daunting. Equipment costs are formidable, benefits may be unclear, and networking programming is unfamiliar territory for many network engineers. A simple shim device that can add a layer of SDN functionality to legacy equipment may go a long way toward easing adoption pains for network practitioners. Certainly, there will be trade-offs in such a design in terms of performance. However, these compromises may be acceptable in testbeds as the shim can help SDN newcomers understand how they might leverage new networking technologies, perform more controlled upgrades of their equipment, and get the most long-term value for their investment.

IV. OpenFlow Schema: Message Creation, Exchange, and Validation

4.1 Introduction

By choosing to adopt SDN, a network practitioner may be seeking to reduce their organization's dependence on a single company. However, without interoperability between SDN software offerings, that dependence simply moves up the stack from hardware to software.

4.1.1 Hardware Independence.

Through OpenFlow, the hardware switches in the network lose much of their importance; in a sense, they become a commodity. As long the switches implement the right versions of the chosen protocols and have the desired performance characteristics, it does not matter which vendor produced them. So-called "whitebox switches" aim to fill this need in the market today. They are often available at a fraction of the price of traditional hardware and offer total flexibility with regard to software and protocols.

This is in contrast to traditional networking hardware, where the choice of vendor is often extremely important to the practitioner. In these traditional devices, the control and data planes are vertically integrated. In other words, the configuration, upgrades, and maintenance of the devices varies significantly from one vendor to the next, and sometimes even by product line of the same vendor. The result is that the network practitioner ends up having to repurchase from the same vendor when upgrading or expanding the network, or else face exorbitant switching costs, a situation referred to as "lock-in" [39].

4.1.2 Software Dependence.

Whitebox switches may be immune to hardware vendor lock-in, but they do not function without software. As explained in Chapter II, the controller is the most important component of an SDN architecture [17]. Once a controller is chosen, a practitioner will need to develop or acquire network control applications that interface with that controller to make network behavior conform to policy. This interface between controller and application is often called the "northbound interface." With no standard northbound interface, portability between controllers is impossible. This is shown conceptually in Figure 15, where though there is total compatibility between switches and controllers, the applications written for controller A are incompatible with controller B, and vice-versa.

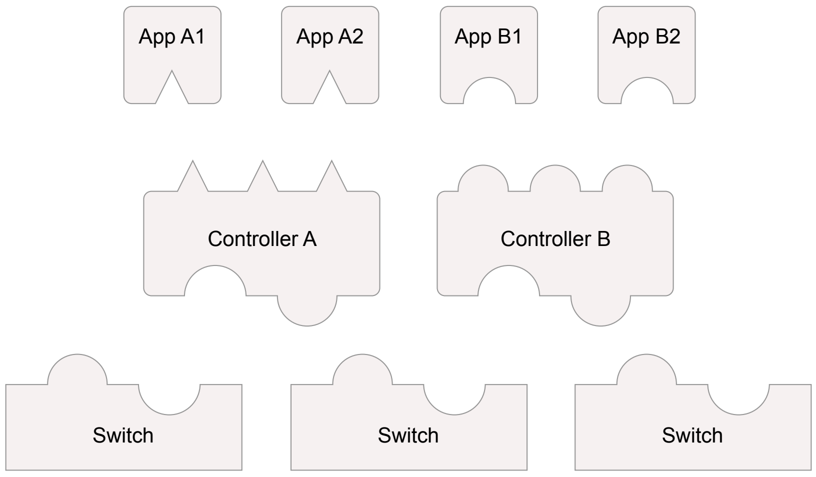


Figure 15. Conceptual diagram showing non-interoperability of SDN applications between controllers

4.1.3 Schema as a *Lingua Franca*.

One solution to this problem would be to standardize the northbound bound interface on a single API or set of APIs. As described in Section 2.5.3, while OpenFlow has become the standard southbound interface for SDN, there does not currently exist a standard northbound interface. While some standardized northbound interfaces have been proposed, others advocate against premature standardization of the northbound interface, arguing it will stifle innovation [18, 19, 39].

Instead of a standardized northbound interface, this chapter discusses the implementation of a schema for creating, validating, and sharing messages between the controller and its applications. A schema in this sense is a formal description of the format and types of data that can be contained in a document or object. For example, an Extensible Markup Language (XML) schema can be used to constrain an XML document to certain elements and contents. This approach allows for total flexibility of API while constraining OpenFlow-related data to a known valid set of names and values.

OpenFlow, the most widely-used SDN switch protocol, is standardized and developed by the Open Networking Foundation (ONF). Besides describing the characteristics of OpenFlow switches and controllers, a major portion of each version's specification is devoted to describing the wire protocol format, along with a nominal C language header file. While this dictates the format for messages passed "on the wire" (between switches and controllers), it does not specify a format for OpenFlow message data while in memory or as it is passed between the controller and applications.

4.1.4 Schemas and Serialization.

The term "serialization" is used to describe the translation of in-memory objects or structures to a stream of bytes. There are several existing, high-quality serialization libraries such as Google's Protocol Buffers, Apache Avro, and Apache Thrift; however, these libraries solve a different problem than the type of serialization required here. These libraries are made primarily to ease development of custom remote procedure call (RPC) systems, where the in-memory format is specified but the serialized binary format is flexible. In this case, the OpenFlow binary format is already published, while the in-memory format is flexible.

Therefore, the schema created here aims to prove the utility of a standard, documented format for the in-memory representation of OpenFlow messages, while Chapter V discusses serialization and deserialization of messages.

4.2 Related work

Instead of a schema, most existing OpenFlow controllers utilize a custom, language-specific format for these messages. For example, the Java-based controllers Floodlight and OpenDaylight use Java classes, Python-based controllers Ryu and Pyretic use Python objects, and the ONF-maintained Libfluid library uses C++ classes. A notable exception is the proprietary Hewlett Packard Enterprise (HPE) Virtual Application Networks (VAN) SDN controller, which uses an HPE-branded schema to validate messages and other data. This approach makes it easier for developers to write applications for this proprietary controller; however, these applications and the schema exposed by the controller are still specific to that one controller and not intended to be universal nor to aid portability.

4.3 Benefits

An independent, non-proprietary OpenFlow schema would be an important addition to the SDN ecosystem. It would decrease developer effort and could allow a degree of interoperability between network applications built for different controllers. It could even promote inter-controller communication between different systems.

In this controller project, the schema has greatly reduced the complexity and redundancy of the OpenFlow library by separating the encoding and decoding functions from those functions that check for correctness of required fields and values. The schema also provides a means for creation of a new message instance with valid default values. As the schema is language-agnostic, it can be extended and updated for new OpenFlow versions independently of the controller and OpenFlow library. The controller API documentation is greatly simplified by referring the format of all OpenFlow messages to the schema itself.

4.4 Implementation

4.4.1 JavaScript Object Notation (JSON).

JavaScript Object Notation (JSON) is a common format for exchange of data between applications. While based initially on JavaScript Objects, JSON is widely used to exchange data between many languages, including Java, Python, Ruby, C, and others [40]. JSON Schema can be used to check for validity by providing a standard way to describe required properties, types, and allowed values in any structure. JSON Schema is described in a series of IETF drafts, and provides a format (itself JSON) for validating the structure and content of any JSON document [41].

4.4.2 YAML Ain't Markup Language (YAML).

While JSON is widely used and suitable to large schemas, it is tedious to write. All key-values and strings must be quoted with double-quotes, all sub-structures must be bracketed with curly braces, arrays must be bracketed with square braces, and comments are not allowed. To ease this effort and improve readability, the OpenFlow schema created here is written in YAML. YAML, like JSON, is a common format for exchanging data between applications, or as a configuration file format, but aims to be more human-friendly. As of version 1.2, YAML is an official subset of JSON, making it easy to translate into either JSON or in-memory JavaScript objects, while being easy to read and write [42]. Unlike JSON, comments are allowed, formatting is unambiguous yet less strict, whitespace is significant, and strings do not require quoting.

4.4.3 YAML-JSON Translation.

The schema of the OpenFlow 1.0 header in both YAML and JSON is given as an example in Figure 16. In this figure, the left side shows how simply a message can be documented using JSON Schema in YAML, and the right side shows the simple, one-to-one translation from YAML into JSON that is done with the YAML library parser. Instead of translating the result to JSON, the library can instead instantiate the input YAML into JavaScript objects in memory. Whether read from the resulting JSON or directly from YAML, these objects can then be fed into a JSON Schema validation library or object instantiation library to validate or create new objects according to the schema, respectively.

4.4.4 OpenFlow Schema Structure.

The schema is structured at the top level by OpenFlow version number, with versions 1.0 and 1.3 currently implemented. This allows the consumer of the schema to select and validate against all versions or a specific version of the protocol, as required. Each OpenFlow schema version consists of a collection of YAML documents. A generic message is defined first in the schema, and each OpenFlow message extends this generic message. A separate document defines common structures which are referenced in the message definitions.

4.4.5 References.

While the entire set of schemas could be held in one single document, splitting it by message across several documents makes it easier to write and maintain. The

```yaml
# Header common to all OpenFlow messages
ofp_header:
  description: OpenFlow header
  type: object
  properties:
    version:
      description: OpenFlow version
      type: integer
      minimum: 1
      maximum: 0xFF
      default: 1
    type:
      $ref: "#/ofp_type"
    length:
      description: Message length in bytes
      type: integer
      minimum: 8
      maximum: 0xFFFF
      default: 8
    xid:
      description: Transaction ID
      type: integer
      minimum: 0
      maximum: 0xFFFF_FFFF
      default: 0
  required:
    - version
    - type
    - length
    - xid
```

```json

"ofp_header": {
  "description": "OpenFlow header",
  "type": "object",
  "properties": {
    "version": {
      "description": "OpenFlow version",
      "type": "integer",
      "minimum": 1,
      "maximum": 255,
      "default": 1
    },
    "type": {
      "$ref": "#/ofp_type"
    },
    "length": {
      "description": "Message length in bytes",
      "type": "integer",
      "minimum": 8,
      "maximum": 65535,
      "default": 8
    },
    "xid": {
      "description": "Transaction ID",
      "type": "integer",
      "minimum": 0,
      "maximum": 4294967295,
      "default": 0
    }
  },
  "required": [
    "version",
    "type",
    "length",
    "xid"
  ]
}
```

Figure 16. OpenFlow header schema in YAML (left) and JSON (right)

separate files refer to structures in the other by Uniform Resource Identifier (URI) references (defined in a similar IETF draft). While these references are currently relative, they can be expanded to “absolute” URIs. A standards body like the ONF could use this to enable applications to update their supported schemas at run time, keeping every implementation on the exact same schema format.

The example in Figure 16 is one of the common structures referenced by every message, as every message starts with a header. By using references, common structures can be defined once, reducing redundancy. An example of this can also be seen in Figure 16, left side, lines 12–13 (and respectively right side, lines 13–15), where the type of the message links to the definitions of `\` in the same document (`#/` indicates the root of the document). The definition of `ofp_type` is shown in Figure 17, which shows that the type of the message is defined as an enumeration of strings. In other words, the type must be the string value of one of the message types defined for that version of OpenFlow.

4.4.6 Example Message Definitions.

An example message definition is given in Figure 18, which shows how succinctly the echo request message can be defined. The definition starts with required header information in the first three lines, then goes on to describe the echo request message object. As `header` is the only element in the `required` array, the `data` property is optional. The `header` property consists of a merge of the `ofp_header` definition and the type `OFPT_ECHO_REQUEST`. Use of the JSON Schema keyword `allOf` allows the same header definition to apply to every OpenFlow message by overriding the `type` property.

A longer example is given in Figure 19, which shows how the OpenFlow version 1.3 `PacketIn` message, containing several required and optional fields, is defined. Note that the definition starts with an enumerated list of the required properties; any

```
ofp_type:
  description: Message type (OFPT_HELLO, OFPT_ERROR, etc)
  type: string
  enum:
    - OFPT_HELLO
    - OFPT_ERROR
    - OFPT_ECHO_REQUEST
    - OFPT_ECHO_REPLY
    - OFPT_VENDOR
    - OFPT_FEATURES_REQUEST
    - OFPT_FEATURES_REPLY
    - OFPT_GET_CONFIG_REQUEST
    - OFPT_GET_CONFIG_REPLY
    - OFPT_SET_CONFIG
    - OFPT_PACKET_IN
    - OFPT_FLOW_REMOVED
    - OFPT_PORT_STATUS
    - OFPT_PACKET_OUT
    - OFPT_FLOW_MOD
    - OFPT_PORT_MOD
    - OFPT_STATS_REQUEST
    - OFPT_STATS_REPLY
    - OFPT_BARRIER_REQUEST
    - OFPT_BARRIER_REPLY
    - OFPT_QUEUE_GET_CONFIG_REQUEST
    - OFPT_QUEUE_GET_CONFIG_REPLY
```

Figure 17. JSON Schema definition of the OpenFlow 1.0 message type field as an enumerated string value in YAML

```
$schema: http://json-schema.org/draft-04/schema#
description: Schema describing an OpenFlow echo request message, version 1.0
id: of10/ofp_echo_request.json

type: object
required: [header]
properties:
  header:
    allOf:
      - $ref: definitions.json#/ofp_header
      - properties:
          type:
            enum: [OFPT_ECHO_REQUEST]
            default: OFPT_ECHO_REQUEST
  data:
    $ref: definitions.json#/optional_data
```

Figure 18. JSON Schema definition of the OpenFlow 1.0 Echo Request message in YAML

properties not listed here are optional. Also note the definition of the `cookie` property, which defines a string that is limited by a regular expression. The resulting string describes a hexadecimal value that is 16 characters long, which is an 8-byte (or 64-bit) integer. 64-bit integers are defined this way in order to prevent loss of precision when storing values in JavaScript objects. (As described in Chapter V, JavaScript only stores numbers in 64-bit IEEE 754 representation, so such string-based workarounds are necessary; however, this will not hinder implementations in languages with native 64-bit integer support).

4.4.7 Schema Distribution.

The OpenFlow Schema codebase is intended to be distributed as a set of YAML documents. However, as YAML support is not as widespread as JSON, the codebase includes a script to generate JSON Schema documents from the YAML source. This may help for languages which do not have a YAML library or parser, as JSON support is nearly ubiquitous.

Covering OpenFlow versions 1.0 and 1.3, the schema consists of about 3800 lines of YAML which generates about 4000 lines of JSON. For distribution, the code can be published to the JavaScript package management system npm [43], making it fast and easy to install.

4.4.8 Message Instantiation.

In order to facilitate creation of new message instances based off the schema, a separate function was written. This function utilizes the popular JavaScript-based JSON Schema validator, Ajv [44]. While Ajv can provide limited object creation, it does not support this feature using `default` keywords in subschemas or the `allOf` keyword, both of which are used in the OpenFlow schema. The function creates

```yaml
$schema: http://json-schema.org/draft-04/schema#
description: Schema describing an OpenFlow packet in message, version 1.3
id: of13/ofp_packet_in.json

type: object
required:
  - header
  - buffer_id
  - total_len
  - reason
  - table_id
  - cookie
  - match
properties:
  header:
    allOf:
      - $ref: definitions.json#/ofp_header
      - properties:
          type:
            enum: [OFPT_PACKET_IN]
            default: OFPT_PACKET_IN
  buffer_id:
    type: integer
    minimum: 0
    maximum: 4294967295
    default: 0
  total_len:
    type: integer
    minimum: 0
    maximum: 65535
    default: 0
  reason:
    $ref: definitions.json#/ofp_packet_in_reason
  table_id:
    type: integer
    minimum: 0
    maximum: 255
    default: 0
  cookie:
    type: string
    pattern: ^[a-fA-F0-9]{16}$
    default: '0000000000000000'
  match:
    $ref: definitions.json#/ofp_match
  data:
    $ref: definitions.json#/optional_data
```

Figure 19. JSON Schema definition of the OpenFlow 1.3 Packet In message in YAML

an instance of Ajv, loading the schema into it, and extracts a fully-resolved schema object. It then implements Algorithm 1 to allow any message, denoted by `id`, to be instantiated with default values. Comments in Algorithm 1 are shown right-aligned, prefixed with ▷.

Algorithm 1 instantiator

```
schema ← retrieved schema from Ajv by id
procedure RECURSIVEINSTANTIATE(schema)
    if schema has property $ref then            ▷ Resolve and merge any refs; recurse
        resolved ← merged, resolved reference from Ajv
        return recursiveInstantiate(resolved)
    else if schema has property type then              ▷ Properly instantiate default
        if type is object then                  ▷ Recurse into each required property
            if schema has property required then
                for each required property p do
                    result[p] ← recursiveInstantiate(p)
                end for
            end if
            return result
        else if type is integer, array, or string then             ▷ Return default value
            if schema has property default then
                return default
            else
                return null
            end if
        else return null
        end if
    else if schema has property allOf then                 ▷ Merge each allOf; recurse
        merged ← merged schemas
        return recursiveInstantiate(merged)
    end if
end procedure
```

As the code recursively iterates through the schema for each call, the use of this package in the library is done completely at application startup. The library uses this package to precompute the defaults for each object and stores those as a constant, avoiding many recursive calls and improving performance.

An example of using the package from a JavaScript environment (e.g., Node.js as described in Chapter V or a modern web browser) is shown in Figure 20. In this example, the lines starting with `// =>` give the result of the `console.log` statements printed to the console. At the start of the file, the `Instantiator` class from the library is loaded, and then a short example schema is defined. This example schema is unrelated to OpenFlow or SDN. The schema is loaded into the library on line 29 and a new instance of the object is created on line 30. Line 31 prints out the result, which shows that each of the properties and its default value have been created on the new object.

4.5 Future Work

A future goal is to add schemas for other popular versions. The ONF releases new versions of the OpenFlow switch protocol approximately every 12–18 months, as shown in Table 1, and coverage of additional versions would greatly increase the usefulness of the schema.

Another goal is to extend the schema to the point where it (1) can become the documentation for the protocol, in place of the C header file, and (2) makes it possible to write a single encoding/decoding library that covers all versions by referencing different schema. This latter goal would allow the use of new OpenFlow versions by just updating the referenced schema version. However, this would be non-trivial as it would require examining all published versions of the protocol to determine how to describe each construct in a way that makes it correct and minimizes redundancy. There is some prior art in this latter objective by the project Loxigen by Big Switch Networks, which uses a simplified, proprietary description of the protocol to generate library functions for C, Java, and Python [45].

```
/* Example use of Instantiator */
const mySchemas = [
  {
    "$schema": "http://json-schema.org/draft-04/schema#",
    "id": "theSchemaId",
    "type": "object",
    "required": [
      "firstName",
      "lastName"
    ],
    "properties": {
      "firstName": {
        "type": "string",
        "default": "Foo"
      },
      "lastName": {
        "type": "string",
        "default": "Bar"
      },
      "optionalProperty": {
        "type": "string",
        "default": "Hello"
      }
    }
  }
]

/* Instantiate with all properties */
let ins = new Instantiator(mySchemas);
let myDefaultInstance = ins.instantiate('theSchemaId');
console.log(myDefaultInstance);

// => { firstName: 'Foo', lastName: 'Bar', optionalProperty: 'Hello' }

/* This time, only with `required` properties */
ins.requiredOnly = true;
let myRequiredInstance = ins.instantiate('theSchemaId');
console.log(myRequiredInstance);

// => { firstName: 'Foo', lastName: 'Bar' }
```

Figure 20. Example use of instantiator function

V. node-openflow: High-Performance OpenFlow Protocol Library

5.1 Introduction

This chapter describes the design, implementation, and testing of the OpenFlow library `node-openflow`. The library is a cornerstone to any OpenFlow-related SDN work on the Node.js runtime system. While other libraries and controllers exist, this library is intended to satisfy a use case that existing options do not adequately address: exhibiting high performance while being easy to use and modify.

5.1.1 Foundation of a Controller.

As explained in Chapter II, an SDN controller is made of software, and because of the nature of the SDN architecture, this particular piece of software is the central focal-point of the network. As the focal-point, the controller must manage a multitude of sockets and states for the hardware switches below it (its “southbound” connections) and it must expose a powerful and consistent API for the applications above it (its “northbound” interface). It must also do its job quickly and without error, as the functioning of the entire network depends on it.

The problem of effectively and efficiently managing connections to hundreds or even thousands of devices is the focus of this chapter. The library is flexible enough to be usable both on its own and as a foundation to other projects; as a library, node-openflow provides all the foundational components that a fully-featured controller can be built upon. However, node-openflow may also be directly used as a controller. While this chapter builds upon the schema developed in Chapter IV, it is also used as the foundation to the framework in Chapter VI. The performance of this library as a controller itself and the performance of the framework built upon it are examined in Chapter VII.

5.1.2 Goals.

The library created here provides an implementation of the OpenFlow switch protocol with the following goals:

1. Easy to understand, modify, and extend
2. Expose the underlying semantics of OpenFlow
3. High-performance (at least an order-of-magnitude better than Ryu)

The hypothesis in the development of this new controller is that choosing Node.js as the runtime system for the library will allow for meeting these goals while also contributing a useful tool to the community for further SDN research.

5.1.3 Chapter Outline.

Related works are covered in Section 5.2, Node.js and its choice as the target runtime system for the library are described in Section 5.3, and the choice of TypeScript as the implementation language is described in Section 5.4. Section 5.5 details the implementation, structure, correctness testing, and use of the library on its own or as a foundation for other projects. Finally Section 5.6 examines future directions for the library. The performance of the library is evaluated in Chapter VII.

5.2 Related work

5.2.1 Existing Controllers.

Existing controllers were first considered as possible starting points, however none provided a suitable foundation to align with the goals expressed above.

5.2.1.1 Enterprise Controllers.

Existing controllers which are enterprise-focused are high-performing; however, they abstract away many of the details of OpenFlow. These controllers are often designed this way as they may support many other southbound protocols according to the needs of a large enterprise network. Enterprise controllers include commercial products like HP VAN and NEC ProgrammableFlow as well as open source efforts like OpenDaylight and the Open Network Operating System (ONOS).

As an example, OpenDaylight's architecture in Figure 21 depicts many layers of abstraction. In a controller like this, southbound protocol details do not emerge beyond the lower-layers, and the controller's high-level APIs are unique to itself. As a result, implementing the same routing algorithm on two different controllers may result in vastly different code, each requiring a substantial investment in learning the particularities of the chosen platform.

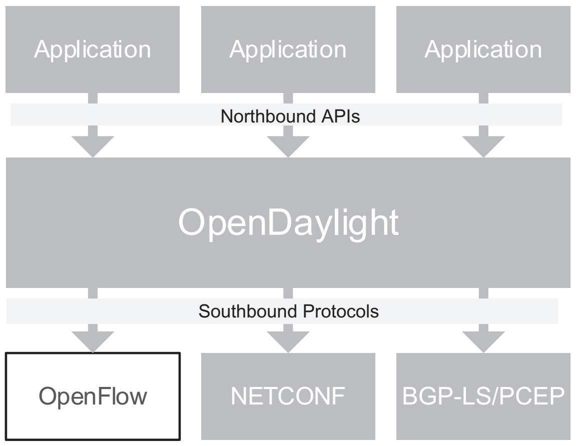


Figure 21. Architecture of OpenDaylight [28]

While suitable for the enterprise, this has negative consequences for research. A student or researcher using the controller must become very familiar with both the underlying protocol and the top-level API, which may have little semantic resemblance to the underlying southbound protocols. Any experimentation on lower layers may require extensive changes to middle layers in order to expose protocol changes in the API. Finally, the increased complexity and size compared to research controllers can be a substantial barrier to entry for a graduate student, who may not have extensive software engineering experience.

5.2.1.2 Research Controllers.

Many research-oriented controllers are specialized for a specific area of research (e.g., domain-specific languages, policy-based control, etc.), and may not be suitable for general use. These controllers tend to exhibit poor performance when compared to enterprise controllers, on the level of several orders of magnitude (as supported by the results in Chapter VII). This makes it difficult to perform experiments related to quantified performance, and makes some experiments on large physical or simulated networks impossible. The Ryu controller is in fact general-purpose while keeping the semantics of OpenFlow; however, its poor performance and complex internals make it unsuitable for these types of experiments.

In [22], the authors were able to make a few simple improvements to the NOX controller and gain a performance increase of over 30 times. NOX was a popular research controller for years, and the success of their contributions shows that performance was never a critical factor for its developers. The authors assert that this trend applies broadly to research controllers, stating "most published results were gathered from systems that were never optimized for performance", and conclude that this has led to the incorrect perception that the SDN architecture as a whole is inefficient.

5.2.2 OpenFlow Libraries.

After determining existing libraries did not align with the goals, a number of existing OpenFlow libraries were considered. There are three main tasks of any OpenFlow library:

1. to enable instantiation and building of OpenFlow messages in a structured way,
2. to decode binary OpenFlow message data from the network into usable structures, and
3. to encode OpenFlow structures into binary messages that can be sent to a socket.

The three libraries considered were LoxiGen, Libfluid, and Ryu. At this point, the Node.js runtime had been selected as a viable system to build on, so part of the evaluation was determining how well the existing library could integrate with Node.js.

5.2.2.1 LoxiGen.

The LoxiGen project (Eclipse Public License) is not itself a library, but a language compiler that accepts a set of language definitions (customized to the LoxiGen project) for a specific version of the OpenFlow protocol, and outputs encoding and decoding library functions for the structures as source code in C, Python, and Java [45]. Extending this to also output structures for JavaScript (or one of its variants) would be on the same level of difficulty as a direct implementation, but would not include the benefits of having readable and readily modifiable code and language-homogeneity, which are both important to the goals of the project. Another possibility would be to utilize the C language bindings (called Loci) by encapsulating them within a native Node.js module, but this would have similar drawbacks; it is difficult and unintuitive to translate between C/C++ and Node.js. Furthermore, unlike JavaScript modules, C/C++ modules are not portable between platforms unless their system calls are extensively wrapped in precompiler directives. In summary, use of LoxiGen would not be a good fit for a project whose goal is to be easily adopted and extended by graduate students.

5.2.2.2 Libfluid.

Libfluid is an open-source (Apache 2.0 license) set of C++ classes for building, encoding, and decoding OpenFlow messages which is written in C++, supporting OpenFlow versions 1.0 and 1.3 [46]. Use of this library would also require encapsulation as a native module, bringing the same drawbacks as use of LoxiGen. Furthermore, since its creation several years ago, there has been almost no work or maintenance done on the library, nor extension into newer versions of the protocol, indicating possible abandonment by its authors.

5.2.2.3 Ryu.

Ryu is an open-source (Apache 2.0 license) OpenFlow controller and OpenFlow library created by Nippon Telegraph and Telephone (NTT) Communications Corporation which is written in Python [47]. It is widely used, high-quality, and actively maintained. However, its use creates performance problems. Using it with Node.js would require running one or more Python processes with inter-process communication (IPC) to carry data back and forth between the Python and Node.js processes. Besides limiting performance to that of Ryu itself, the IPC required introduces additional overhead, which would reject the goal of high-performance.

Ryu's internal implementation is not as simple as a direct Node.js implementation due to the nature of Python. In order to achieve reasonable performance, Ryu uses the greenlet and eventlet third-party libraries to gain high concurrency with asynchronous communication. Use of these libraries enhance Ryu, as they increase concurrency without the complications of shared memory management, but add their own drawbacks. As they are not part of the Python standard library, they must wrap related standard library functions during run-time to ensure compatibility. This technique, possible in many dynamic languages (including JavaScript), is referred to as “monkey-patching”; it complicates understanding, maintenance, and use of affected APIs, and is therefore discouraged except as a last resort. In the case of greenlet, additional complexity is introduced (beyond standard Python and Node.js) by its explicit event loop instantiation and requirement that the programmer include jumps between “micro-threads” (a.k.a. coroutines) in order to handle asynchronous events [48, 49]. In contrast, the single, implicit event loop and non-blocking standard API of Node.js results in asynchronous code which is easier to read and write while maintaining better performance in I/O-bound applications [50].

5.2.2.4 Language Homogeneity.

As none of the above options are written for Node.js, the use of their libraries with Node.js would have the drawback of breaking language and platform homogeneity. This would reduce flexibility in choice of data structures for how to represent OpenFlow messages, as they all (necessarily) impose their own design decisions on the structure of messages. It would also represent a larger barrier to adoption by students with limited time, as it would require some level of mastery of multiple runtime systems and languages.

5.3 Runtime System: Node.js

The library is built on the Node.js runtime system. This system was chosen for its cross-platform design, relative high performance, and ease of use, all of which align well to the stated goals for this library. Node.js is a cross-platform runtime system built on Google's V8 engine and the libuv asynchronous I/O library. Node.js is implemented with a combination of C++ and JavaScript [51].

5.3.1 V8.

V8 is an engine to run JavaScript developed by Google primarily for use in the Chrome web browser. It features a Just-In-Time (JIT) compiler supporting speculative optimizations. While all JavaScript is JIT compiled into an intermediate bytecode and run by an interpreter, functions that are run frequently are selectively further compiled into optimized machine code to increase overall performance [52].

5.3.2 libuv.

libuv is a cross-platform fork of the Libev event library for Unix. The developers of Node.js created libuv in order to extend Node.js to support Microsoft Windows. Libev has since been removed from Node.js, and libuv is used in Node.js to handle event-driven I/O for Node.js while keeping the same API across all platforms. libuv is significant in that its event loop architecture is the basis for the design paradigm of Node.js [53].

The libuv architecture is shown in Figure 22. In this figure, the boxes drawn near the bottom of the figure represent lower-level, OS-interfacing components while components near the top represent APIs that interface with Node.js.

5.3.3 Cross-Platform Asynchronous Programming.

Each operating system supported by libuv provides its own asynchronous event notification mechanism, and these are shown as components in Figure 22. epoll, kqueue, event ports, and IOCP are the asynchronous event notification mechanisms of Linux, FreeBSD/macOS, Solaris, and Windows, respectively. While these features can

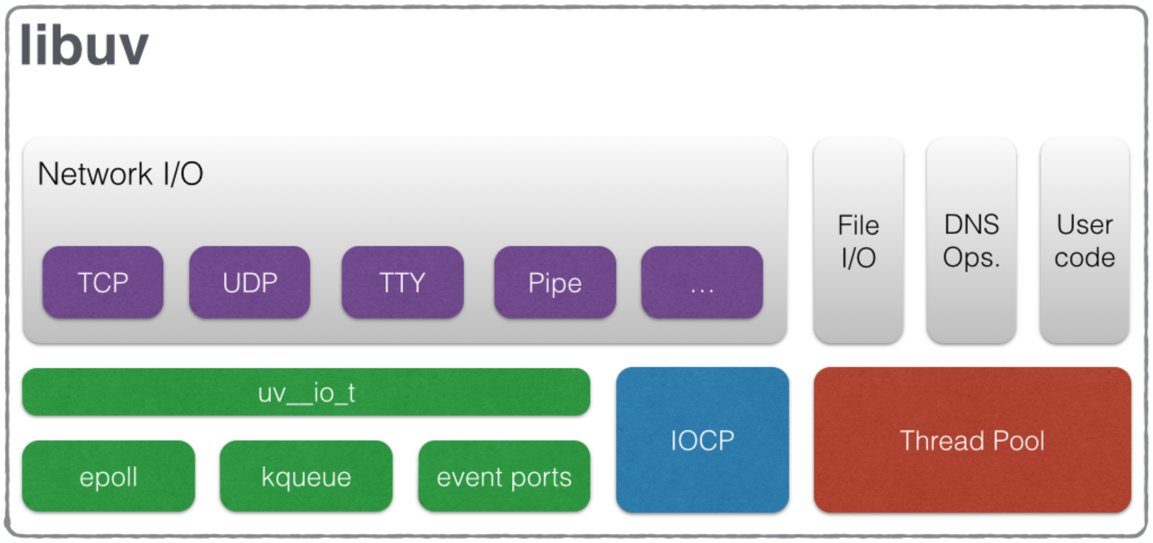


Figure 22. The libuv architecture [53]

be used to the same effect, each one varies both syntactically and semantically from the others. This illustrates the main feature of the libuv library, which is to abstract over these operating system details and present one unified, cross-platform API for asynchronous I/O.

However, these asynchronous event notification mechanisms only apply to certain system calls. Other system calls, such as those related to the file system and Domain Name System (DNS) lookups, are only provided as blocking calls. To handle these synchronous, blocking calls in an asynchronous manner, libuv maintains a thread pool. By default, this thread pool is made up of four threads. This is shown at the bottom right of Figure 22. Whenever such a call is made, it is assigned to a thread which can block while the rest of the program executes in the event loop.

5.3.4 Event-Based Programming.

Node.js is built on libuv, which handles asynchronous programming with an event loop, as shown in Figure 23.

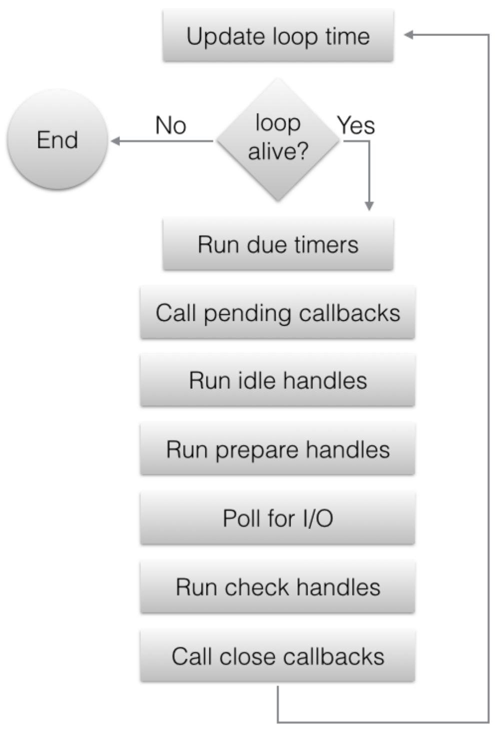


Figure 23. The libuv event loop [53]

While traditional network servers (e.g., Apache) make heavy use of multithreaded programming to serve many requests concurrently, Node.js eschews this model for event-based, asynchronous programming. The motivation for this is a simpler programming model while maintaining a high degree of concurrency over I/O-bound workloads. In comparing the approaches, one author expands on the difficulties of the former:

> Even though many developers have successfully used multithreading in production applications, most agree that multithreaded programming is anything but easy. It's fraught with problems that can be difficult to isolate and correct, such as deadlock and failure to protect resources shared among threads. [54]

I/O tasks, by nature, run orders of magnitude slower than tasks involving pure computation. By implementing non-blocking calls for these tasks, the main loop of the program can continue to process other events while the I/O-related task continues in the background, on a separate thread. When a task completes, the main loop gives any results to "listening" functions by way of a callback or event notification. libuv cooperates with the operating system to manage background work through polling or threads, depending on the platform, in a way which keeps the platform-specific details transparent to the user of the library.

libuv implements a single-threaded event loop which continually checks for new events and handles callbacks associated with those events. libuv's event-loop pattern (and therefore Node.js) can yield programs that are higher-performing, simpler, and less prone to error than traditional patterns. By abstracting away the details of handling multi-threaded logic, the library shields the user from having to use semaphores, mutexes, or other safety mechanisms for shared memory, which makes code both simpler and less prone to error. This method also avoids the memory overhead incurred in the traditional model by having a static number of threads or processes ready to handle the expected workload from connecting clients.

libuv embraces asynchronous, non-blocking calls for I/O-intensive tasks in order to keep from blocking the event loop and maintain good performance. I/O-intensive tasks typically include tasks such as reading or writing a file from a disk, sending or receiving data over the network, or printing data to the screen.

5.4 Programming Language: TypeScript

5.4.1 JavaScript Alternatives.

By choosing Node.js as the platform, the de facto and eponymous choice for language is JavaScript, or more precisely ECMAScript 2015 (ES2015) [55]. While the language is being actively developed and extended as the ECMAScript standard by the standards committee "TC-39", JavaScript still contains many peculiarities that are often confusing to newcomers. Some examples include prototypal inheritance, implied global variables, global scoping, dynamic "duck" typing, and unintuitive type checking semantics (a.k.a. "truthiness"). These have been well-documented in

Douglas Crockford's book, *Javascript: The Good Parts* [56]. A host of languages that compile to JavaScript have been developed to attempt to address these issues, including CoffeeScript, Elm, Google's Dart, and Microsoft's TypeScript.

5.4.2 TypeScript Features.

TypeScript is a strict superset of JavaScript developed by Microsoft. Typescript can help to bridge the gap to JavaScript for programmers who are already familiar with languages like C/C++, C#, and Java by providing explicit types with type inference, static type checking, generics, enumerations, and interfaces. It also brings many of the latest features from ES2015 which may not yet be implemented in browsers or Node.js, like traditional class-based inheritance, iterators, generators, modules, decorators, and async/await function keywords. It also provides a "compile-down" feature to target any of several older versions of JavaScript, allowing the developer to choose the required level of compatibility for older runtime systems, at the cost of some features which cannot be compiled down (e.g., generators).

5.4.3 TypeScript Components and Tools.

TypeScript provides a compiler and language server which work to translate TypeScript source into plain JavaScript and provide realtime error-checking and autocompletion hints, respectively. In the simplest cases, the "compiled" TypeScript is simply JavaScript with the types removed. During compilation, some non-JavaScript features (e.g., enumerations) are implemented in JavaScript with helper functions. While types and some other features have no run-time component and will simply be removed from the output, the type-checking, autocompletion, and other features provided by them through the language server during development are a tremendous aid to the programmer. The error-checking and autocompletion features have been integrated into many prominent editors and development environments, including Microsoft's suite of Visual Studio products.

5.4.4 TypeScript Interoperability.

Since TypeScript compiles to readable JavaScript, it is interoperable with any other JavaScript or Node.js language; the API of the framework can be consumed by applications written in any of the languages listed above, making it easy to extend. Furthermore, Node.js provides an API and build tool to include modules written in C or C++ which opens another avenue for extension of the framework.

5.4.5 TypeScript Benefits.

Anders Hejlsberg is the creator of TypeScript and the lead C# architect at Microsoft, and previously created Turbo Pascal and Delphi at Borland. He promotes TypeScript as a useful JavaScript development tool that becomes more useful as the project grows in size, eventually becoming a necessity for code maintainability [57]. The first version of this library was written in CoffeeScript. CoffeeScript is another compile-to-JavaScript language with a simple and functional style that promotes readability [58]. However, as the library grew in size it soon became unmanageable. Adopting TypeScript meant a non-trivial amount of extra work to rewrite the code base, but it was proven worthwhile. Switching to TypeScript significantly improved the quality of the code, eliminated classes of errors, and noticeably increased productivity in extending and building on the project.

5.5 Implementation

5.5.1 OpenFlow Versions.

The library implements the two most popularly used versions of the OpenFlow switch protocol, versions 1.0 and 1.3. Each of the OpenFlow messages and major structures that are used by those messages are implemented as classes. There are 22 message types in version 1.0, and 30 in version 1.3. The important structures include Match, Action, Port, Queue, Instruction (1.3 only), and OpenFlow eXtensible Match (OXM) (1.3 only). Each message and structure class has its own encode and decode methods.

5.5.2 Encoding and Decoding OpenFlow Messages.

Encode methods read the structured data in the `message` property of the instance and return a raw buffer ready to be written to a socket. Decode methods are static methods that take a raw buffer and parse it into the `message` property. The library exposes a top-level `decode` function which takes a buffer of either version and any message and returns an instance of that decoded message. Sub-structures of messages (Match, Action, Port, Queue, Instruction, or Oxm) are instantiated as required when decoding, and have their encode methods called by the parent's encode method.

5.5.3 Instantiating and Validating OpenFlow Messages.

Messages and structures can be built by calling `new` on the class, in which case it will be populated with default values from the schema, or given default values in the constructor. Structures can be built first and then attached to the parent message; for example, a program requiring a new flow modification message (`FlowMod`) can first instantiate and configure the Match, Action, and Instruction structures and then attach those to a new `FlowMod` message. The message can also be validated against the schema before being encoded. As a result, building complex OpenFlow structures is relatively simple. For example, Figure 24 shows the creation of a `FlowMod` as well as its associated `Match`, `Action`, and `Instruction`, along with the attachment of these pieces at the end to form a complete `FlowMod` message. The last line of the figure then encodes the message into a binary buffer, ready to be transmitted on a socket.

```
// Create FlowMod
let fm = new of13.FlowMod();
fm.message.header.xid = 0x01;
fm.message.buffer_id = 0x1234;
fm.commandVal = of13.OFPFC_ADD;
fm.flagsVal = of13.OFPFF_SEND_FLOW_REM;
fm.message.hard_timeout = 0;
fm.message.idle_timeout = 30;
fm.message.priority = 10;

// Create match
let ma = new of13.Match();
ma.oxm_fields = [new of13.Oxm({
  oxm_field: "OFPXMT_OFB_ETH_DST",
  oxm_value: sm.dstmac,
})];

// Create instruction, action
let ins = new of13.Instruction();
ins.typeVal = of13.OFPIT_APPLY_ACTIONS;
let act = new of13.Action();
act.typeVal = of13.OFPAT_OUTPUT;
act.port = sm.dstport;
act.max_len = of13.OFPCML_NO_BUFFER;

// Attach instantiated match, action, instruction
fm.message.match = ma;
ins.actions = [act];
fm.message.instructions = [ins];

let buffer = fm.encode();
```

Figure 24. Construction and encoding of an OpenFlow 1.3 FlowMod message

Figure 25 shows the directory structure of the library source code. In this figure, `openflow.ts` is the top-level file which is automatically included when the library is imported. That top-level file then includes the individual versions, `of10.ts` and `of13.ts`, which import all enumerations, messages, and structures for that version. Note that as TypeScript is a compile-to-JavaScript language, the actual files loaded by Node.js end with `.js` instead of `.ts`. This is transparent to the user of the library, as all files in the package are already compiled to JavaScript.

5.5.4 Testing Library Conformity to Specifications.

The OpenFlow switch specification documents are sparing in explanations and examples, instead relying mostly on excerpts from the C header file openflow.h to provide most of the details. Additionally, conventions like property names, padding around variable-sized structures, and whether padding is included in `length` values are inconsistent, even within the same OpenFlow version.

As a result, extensive and detailed tests are required to ensure that the library properly instantiates, encodes, and decodes the messages so that it will actually be able to communicate with OpenFlow switches, whether they be hardware or software. Each message and structure for each version contains tests for instantiation, encoding, and decoding. Additionally, there are tests to ensure proper error-handling for malformed buffers, schema validation errors, and version or message code mismatch.

The term *test coverage*, common in the test driven development process of software development, refers to the percentage of code that is covered by the tests in aggregate. The idea is to ensure that tests cover all code paths, to the extent possible, such that no portion of the software is left untested, reducing the chances of bugs. Overall, the `node-openflow` library has 94% coverage of code statements and 95% of functions. The portions of code without tests are mostly related to the `Error` message class which is

```
src
|-- benchmark.ts
|-- decoders.ts
|-- encoders.ts
|-- of10
|   ... (36 lines omitted)
|-- of10.ts
|-- of13
|   |-- enumeration.ts
|   |-- messages
|   |   |-- AsyncConfig.ts
|   |   ... (32 lines omitted)
|   |   |-- TableMod.ts
|   |    -- instantiate.ts
|   |-- messages.ts
|   |-- structures
|   |   |-- Action.ts
|   |   |-- Instruction.ts
|   |   |-- Match.ts
|   |   |-- Oxm.ts
|   |   |-- Port.ts
|   |   |-- Queue.ts
|   |    -- QueueProperty.ts
|    -- structures.ts
|-- of13.ts
|-- openflow.ts
|-- test
|   ... (12 lines omitted)
|-- utilities
|   |-- bitutil.ts
|    -- readerwriter.ts
 -- utilities.ts

10 directories, 98 files
```

Figure 25. Tree view of library source directory

only partially implemented at this time. While improving test coverage may increase code quality, it would not necessarily be advantageous to bring these numbers to 100%; a better goal would be to improve the quality of the tests that already exist, as there are several cases where a function or method is tested with only one or two inputs of the range of inputs it could possibly be expected to handle.

An example of testing the proper encoding of the OXM class is shown in Figure 26. The test asset is declared in lines 2–11, which is an OXM class that describes and IPv4 address, in this case, 192.168.0.12. Lines 14–23 run the current battery of tests. Line 18 instantiates the test object, line 19 encodes it to a buffer and translates it into a hexadecimal string, and line 20 compares the results of this encoding to the correct answer stored in the test.

Output of all tests running is shown in Figure 27. The encoding and decoding methods of each class are tested against stored correct values, and a statement is supplied denoting the test coverage of the overall codebase.

5.6 Future Work

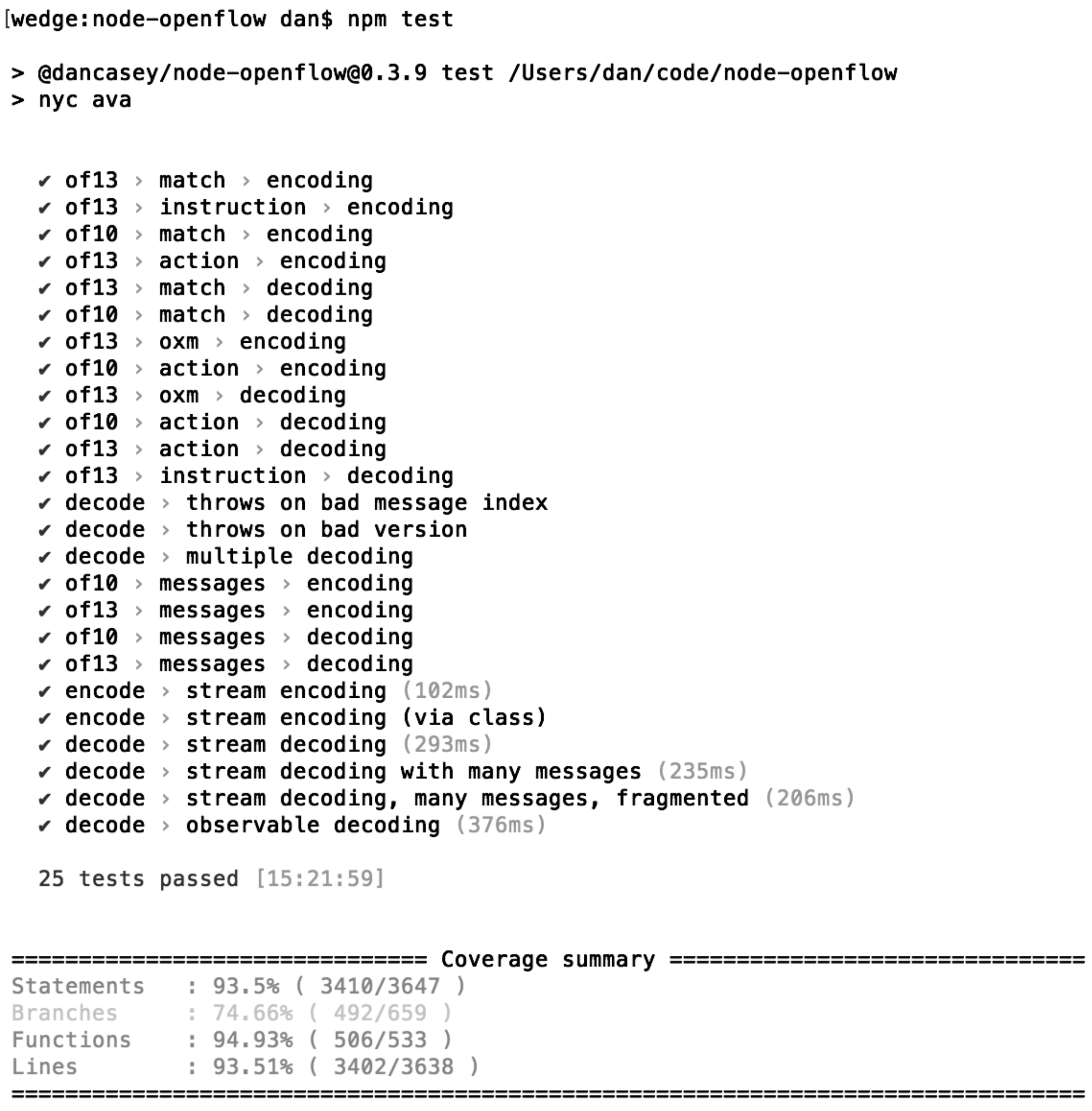
An obvious improvement to the library would be to support more OpenFlow versions. This would not be difficult, as the structures and methods are already in place. However, each specification is fairly long, with each newer one becoming longer than the previous, so it is tedious. The work would have to start by updating the schema (Chapter IV). The next versions to target would be versions 1.4 and 1.5 (the latest). There would not be a reason to implement 1.1 or 1.2, as any systems that old would fall back to 1.0, and newer systems at least implement 1.3.

Another improvement would be to reduce redundancy by further consolidating frequently-used structures into common classes. Before doing this, versions 1.4 and

```
# YAML test asset
OFPXMT_OFB_IPV4_SRC:
  obj:
    oxm_class: "OFPXMC_OPENFLOW_BASIC"
    oxm_field: "OFPXMT_OFB_IPV4_SRC"
    oxm_hasmask: false
    oxm_length: 4
    oxm_value: "c0a8000c" # 192.168.0.12
    oxm_mask: ''
  buf:
    "80001604c0a8000c"

// Test runner
test("encoding", t => {
  for (let name in tests) {
    if (tests.hasOwnProperty(name)) {
      let test = tests[name];
      let o = new Oxm(test.obj);
      let buf = o.encode().toString("hex");
      t.is(buf, test.buf);
    }
  }
});
```

Figure 26. Test asset and runner for proper OXM encoding


```
[wedge:node-openflow dan$ npm test

> @dancasey/node-openflow@0.3.9 test /Users/dan/code/node-openflow
> nyc ava


  ✔ of13 › match › encoding
  ✔ of13 › instruction › encoding
  ✔ of10 › match › encoding
  ✔ of13 › action › encoding
  ✔ of13 › match › decoding
  ✔ of10 › match › decoding
  ✔ of13 › oxm › encoding
  ✔ of10 › action › encoding
  ✔ of13 › oxm › decoding
  ✔ of10 › action › decoding
  ✔ of13 › action › decoding
  ✔ of13 › instruction › decoding
  ✔ decode › throws on bad message index
  ✔ decode › throws on bad version
  ✔ decode › multiple decoding
  ✔ of10 › messages › encoding
  ✔ of13 › messages › encoding
  ✔ of10 › messages › decoding
  ✔ of13 › messages › decoding
  ✔ encode › stream encoding (102ms)
  ✔ encode › stream encoding (via class)
  ✔ decode › stream decoding (293ms)
  ✔ decode › stream decoding with many messages (235ms)
  ✔ decode › stream decoding, many messages, fragmented (206ms)
  ✔ decode › observable decoding (376ms)

  25 tests passed [15:21:59]


=============================== Coverage summary ===============================
Statements   : 93.5% ( 3410/3647 )
Branches     : 74.66% ( 492/659 )
Functions    : 94.93% ( 506/533 )
Lines        : 93.51% ( 3402/3638 )
================================================================================
```

Figure 27. Output from all tests running on library

1.5 should be examined for similarities and differences to the already implemented 1.0 and 1.3.

Finally, the most interesting improvement to the library would be to determine how to decorate the schema (discussed in Chapter IV) such that the library can be reduced to a series of function-generating functions for encoding and decoding each message and structure. The benefit would be a substantially reduced codebase to maintain as well as making it nearly trivial to implement new versions of the protocol by simply writing decorated YAML specs.

VI. rxdn Framework: Modular Network Control with Reactive Programming

6.1 Introduction

This chapter builds on the library developed in Chapter V. The node-openflow library is usable by itself as a foundation on which to create a controller application, and a few examples are included in its repository. However, such a bare-bones approach is not very productive, as it only provides a means of encoding, decoding, and checking validity of OpenFlow messages. Therefore, this chapter contributes "rxdn", a framework which enhances productivity in writing network control applications. The name "rxdn" is a portmanteau combining the "Rx" prefix used by Microsoft's Reactive Extensions (ReactiveX) and the last two letters of the acronym SDN, highlighting the framework's design on the ReactiveX programming pattern and its embodiment of the basic principles of SDN.

Similar to the library node-openflow, this framework is written on top of the library to try to fulfill the following goals:

1. Easy to understand and extend
2. Provide intuitive means of network application modularity
3. Throughput and latency performance exceeding that of Ryu

Approaches to dealing with asynchronicity and modularity are covered as related works in Section 6.2. The application of these works to the design and implementation of the rxdn framework is covered in Section 6.3. Finally, Section 6.4 examines future directions for the library. The performance of the library is evaluated in Chapter VII.

6.2 Related work

This section describes approaches used by other OpenFlow controllers to solve the same design challenges of rxdn. It also covers some building blocks previously unrelated to SDN which were used directly—or as inspiration for—its design in dealing with the problems of asynchronicity and modularity.

6.2.1 Asynchronicity.

Where some platforms handle concurrency through parallelization (multiple threads or processes), Node.js is built for handling concurrency with non-blocking callback functions. This has important consequences to any code written for Node.js in which performance is at all important.

It is easier to write code which will run on a single thread than code which will run on multiple threads or processes, as there is no need to handle the intricate details of thread-safe memory access, including semaphores, mutexes, or in the case of processes, inter-process communication. However, it must be understood by the programmer that there is a *single* event loop, and if any code in the user's program blocks without relinquishing control, the entire program will block. Instances where blocking may occur include reading from or writing to slow resources, such as hard disks and network sockets. CPU-intensive functions must be written such that they periodically relinquish control or they will similarly stall the event loop.

6.2.1.1 Yielding, Processes, and Threads.

Node.js is designed specifically for I/O-bound and not CPU-bound problems. While most network control applications are not CPU-bound problems, it is possible to envision an advanced SDN application that would be. For example, an advanced intrusion prevention system (IPS) that analyzes and blocks traffic in real time could require a large amount of computing power to classify network traffic. For this reason, it is worth briefly exploring the possible options of the platform to handle such problems.

The easiest solution is to have the long-running function periodically yield control back to the event loop. In Node.js, this can be done with the built-in, global function `setImmediate(callback[, ...args])`. This could be used within a CPU-bound function such that it runs a certain number of iterations before putting itself as the callback with some argument to indicate how to resume. While better than blocking the entire application, this both increases the overall time the CPU-bound function takes to complete and slows the number of iterations per time period of the event loop.

There are two built-in means to perform multiprocess execution in Node.js. With these, a separate process can be used to handle CPU-bound portions while not blocking the event loop of the main process. They are the API functions `child_process.fork` and `cluster`. They are slightly different versions of the same functionality, which is to spawn a new process, load it with some predetermined program, and then use serialized interprocess communication to move data or signals back and forth. There is overhead to starting a new process, there is no shared memory (which would be fast), and each communication between the processes is additionally burdened by having to serialize and deserialize any objects that need to be passed.

Outside of built-in approaches, there are a few native modules (modules which are at least partially written in C++ and bound to the V8-Node API) which can be used to implement a multithreaded application in Node.js. The most popular of these is `node-webworker-threads` which conforms to the relatively new Web Worker standard made by the World Wide Web Consortium (W3C) [59]. The Web Worker standard specifies a public API for web browsers to allow web page JavaScript scripts to spawn background worker threads separate from the main thread. As with Node.js multiprocess execution, this standard also requires all data to be serialized and passed via IPC.

In summary, Node.js is not currently the correct choice of platform for CPU-bound problems, and increasing concurrency for I/O-bound problems in Node.js is better solved through one of the methods described below.

6.2.1.2 Callback-based.

As described in Chapter V, Node.js is a good framework on which to build SDN applications, as such programs are by nature I/O bound and therefore require a high degree of concurrency. Node.js naturally encourages concurrent programming with its asynchronous, callback-driven API. For most non-blocking API calls, this takes the form of `action(arguments, callback)`, where `callback` is a function that takes two arguments: an error object and a result. Within the callback, if the error argument is undefined, then the callback may assume that the action completed successfully. Otherwise, the error object is populated and the callback function may chose to throw, print, or ignore it [51]. This style is often called the "error-first" callback style. As a model for concurrency it is both simple and effective. This is not a new idea, as it was very nearly described as continuation passing style (CPS) as early as 1975 in the implementation of the SCHEME language interpreter [60].

However, simply attaching nested callbacks as anonymous functions for a series of computations which rely on the results of asynchronous computation yields code that is difficult to understand and maintain. Overly-nested blocks make error handling tedious, as it is difficult to see where and if errors are handled. For these reasons, such code has ominously been referred to as "Callback Hell" or the "Pyramid of Doom" [61]. An example of overly-nested code is shown in Figure 28. While somewhat contrived, this example is not far from typical error-first callback style code written in Node.js to connect several asynchronous functions or servers together. Notice in the example how the code for an unmatched ID (lines 22–24) is separated from the place the lookup happens (line 8), confusing the flow of the block among a myriad of braces.

```
 1  // Read a file, check an ID in database, write to a socket
 2  fs.read(file, (err, contents) => {
 3    if (err) {
 4      console.error("Error reading file", err);
 5      process.exit(1);
 6    } else {
 7      let {id, value} = parse(contents);
 8      db.has(id, (err, result) => {
 9        if (err) {
10          console.error("Error connecting to database", err);
11          process.exit(1);
12        } else {
13          if (result) {
14            socket.write(value, (err) => {
15              if (err) {
16                console.error("Error writing to socket", err);
17                process.exit(1);
18              } else {
19                console.log(`Sent ${value}`);
20              }
21            })
22          } else {
23            console.log("ID not found");
24          }
25        }
26      })
27    }
28  })
```

Figure 28. An example of "callback hell" [**derived from 61**]

While the confusing pyramid structure can be mitigated by separating out and naming all anonymous callback functions, error handling in such situations is still overly complex. A number of language changes, API changes, and libraries have been written to try to better address the problem. These include the addition of generators and promises to JavaScript in ECMAScript 2015 and the proposed addition of `async` and `await` function keywords which build on promises [55].

6.2.1.3 Promises.

The term "promise" was used to mean a *future but possibly unresolved value* as early as 1976 [62]. Any callback function can be wrapped in a promise. To overcome nesting and its related error-handling problems, promises can be chained together, forwarding the asynchronous result from one function to the next. Each of these functions is called with the result of the previous when it is ready, returning its own promise, ensuring properly-ordered, non-blocking, asynchronous execution [63]. An example of this is given in Figure 29 (from the popular promise library Bluebird.js). Lines 3–16 show the traditional version with `try` and `catch`, while lines 20–28 show the same code using promises. Notice how line 20 chains together sequential functions with the method `then`, while lines 23 and 26 continue chaining by adding error handling functions.

While promises do expose a `catch` method and improve upon error-first callback style, this type of error-handling is still less powerful than traditional, synchronous error-handling using `try`, `catch`, and `finally`. The `async` and `await` keywords were designed specifically to allow synchronous-style function calls and error handling for asynchronous code.

6.2.1.4 Async/Await.

While promises are a refinement of the error-first callback pattern, the keywords `async` and `await` are further refinements to JavaScript promises. These keywords were added to C#.NET 5.0 in 2012, and due to their popular adoption there, have been

```
// Error-first callbacks: read a JSON file, parse it, handle errors

fs.readFile("file.json", function (err, val) {
    if (err) {
        console.error("unable to read file");
    }
    else {
        try {
            val = JSON.parse(val);
            console.log(val.success);
        }
        catch (e) {
            console.error("invalid json in file");
        }
    }
});

// Promise version:

fs.readFileAsync("file.json").then(JSON.parse).then(function (val) {
    console.log(val.success);
})
.catch(SyntaxError, function (e) {
    console.error("invalid json in file");
})
.catch(function (e) {
    console.error("unable to read file");
});
```

Figure 29. Comparing error-first callback to promises [64]

proposed for inclusion into ECMAScript 2017 with TypeScript 2.0 already supporting them [55, 65, 66].

The `async` keyword is used to decorate a function declaration, indicating that the function may include asynchronous `await` function calls. In other words, only those functions marked with `async` may include `await` keywords, and including them in other functions is an error.

In this pattern, the `async` keyword is added before any function call which returns a promise when the result of that promise is required before any further lines of code in the function are executed. As a result, the program does not block, but *asynchronously waits* for the resolution of the promise at that line. An example is shown in Figure 30. The function on line 2 is declared with the `async` keyword, so the keyword `await` can be used within the function body (lines 4, 10, and 12). Line 19 calls this function and can chain together `then` and `catch` keywords exactly as if the function were declared using promises.

The largest benefit in this style comes from being able to use the traditional `try`, `catch`, and `finally` error handling keywords, as if the entire function were synchronous. Just as promises will silently "swallow" errors if no `catch` method is added, this pattern is built on promises and will do the same; in fact any `async` function returns a promise.

While this greatly simplifies code, making it appear synchronous, users of this style still need to be fluent in the underlying promises upon which it is built in order to use it properly.

6.2.1.5 Reactive Extensions.

Reactive Extensions is a set of libraries first designed and implemented by Microsoft with the goal of making asynchronous programming easier and more powerful. It combines concepts from observer and iterator patterns with functional programming

```
// Read a file, check an ID in database, write to a socket
async function readCheckWrite() {
  try {
    let contents = await readFileAsync(file);
  } catch (err) {
    console.error("Error reading file", err);
    process.exit(1);
  }
  let {id, value} = parse(contents);
  let dbResult = await dbHasAsync(id);
  if (dbResult) {
    let socketResult = await socketWriteAsync(value);
    console.log(`Sent ${value}`);
  } else {
    console.log("ID not found");
  }
}
// Output "Complete" or log any uncaught error
readCheckWrite().then(console.log("Complete")).catch(console.error);
```

Figure 30. An example of using async and await for asynchronous chaining

concepts, making these available to a variety of languages [67]. One of its creators summarizes its purpose as follows

> The goal of Rx is to coordinate and orchestrate event-based and asynchronous computations such as low-latency sensor streams, Twitter and social media status updates, SMS messages, GPS coordinates, mouse moves and other UI events, Web sockets, and high-latency calls to Web services using standard object-oriented programming languages such as Java, C#, or Visual Basic. [68]

The collection of Rx libraries are together described as a "polyglot implementation", as they are available to the languages listed in the 2012 publication above and more recently have been extended to Clojure [69], JavaScript [70], Scala [71], Swift [72], and others. Each implementation is written to fit well with the idioms of the target language. As a result, the concepts are easily carried from one language to another, but the classes, interfaces, and operator names may change. Each implementation is named with the prefix "Rx" followed by the name of the language; the implementation for JavaScript is called RxJS [70]. The official API for RxJS gives an interesting example motivating its use within a web browser by showing how to write code that will get the x and y values of mouse clicks *at most* once per second, which is shown in Figure 31. Notice the dearth of local variables in the latter block (lines 16–20) compared to plain JavaScript (lines 3–12).

```
1  // Plain JavaScript:
2
3  var count = 0;
4  var rate = 1000;
5  var lastClick = Date.now() - rate;
6  var button = document.querySelector('button');
7  button.addEventListener('click', () => {
8    if (Date.now() - lastClick >= rate) {
9      console.log(`Clicked ${++count} times`);
10     lastClick = Date.now();
11   }
12 });
13
14 // With RxJS:
15
16 var button = document.querySelector('button');
17 Rx.Observable.fromEvent(button, 'click')
18   .throttleTime(1000)
19   .scan(count => count + 1, 0)
20   .subscribe(count => console.log(`Clicked ${count} times`));
```

Figure 31. Capturing throttled mouse click locations with and without RxJS [70]

Drawbacks of using RxJS (or any Rx library) include the initial burden of becoming familiar with its very large API. The Observable class of the current version of RxJS includes over 20 static and 100 class methods. To be proficient, a user of the library should be familiar with most of these, as only knowing a few will result in inefficient hammer-nail style programming (the tendency to treat every problem as a nail if the only tool you have is a hammer). Marble diagrams included in the documentation are a great aid to the Rx novice, as they visually represent the changes each operator performs to the stream and are quickly understandable.

Marble diagrams are explained by the legend in Figure 32, and examples are shown in Figures 33 through 36. Note that while all these examples are shown as emitted integers, Observables are generic classes and may emit items of any type, including complex objects or even other Observables. The way to read a marble diagram is to picture the top line or lines as an initial stream of events that happen as time progresses from left to right. Below this is an operator drawn as a box. This operator takes events from the stream or streams and translates them into new events or new streams of events. These results of the operator are shown below the box.

Figure 33 shows how the interval operator can be used to emit a sequence of events sequentially as given by the time parameter. Figure 34 shows how an arrow function can be used to transform source events (shown above the map function), in this case multiplying them by 10 to give the result (shown below the map function). Figure 35 shows how state can be incorporated into a stream of events with the scan operator. The accumulator function is provided the last emitted event value, x, and the current incoming value, y. It uses the specified function to combine them—in this case, addition—and emits the resulting intermediate value. This operator can also take an optional seed value. Figure 36 is an example of an operator that combines two Observables into one. In this case, it concatenates the string values of the incoming events as its output.

Debugging code that passes through Observables can be problematic, as each line of the user's code is interpolated with many lines of Rx library code. To mitigate this, some debuggers, like the one in Visual Studio Code, include an option to debug "only my code", skipping over third-party and platform libraries.

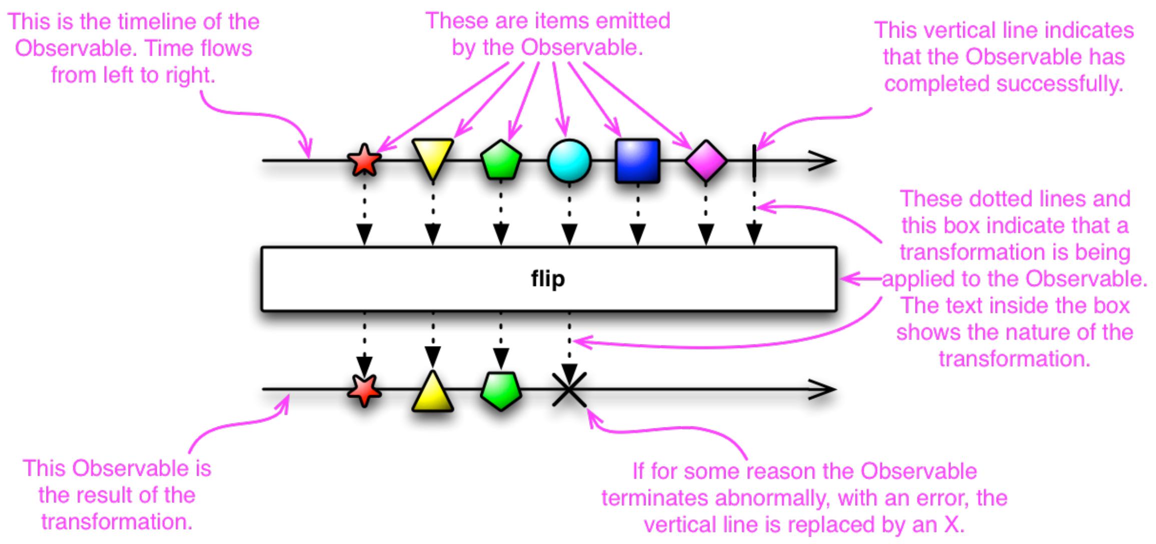

This is the timeline of the Observable. Time flows from left to right.
These are items emitted by the Observable.
This vertical line indicates that the Observable has completed successfully.
These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.
flip
This Observable is the result of the transformation.
If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

Figure 32. Rx Marble Legend [67]

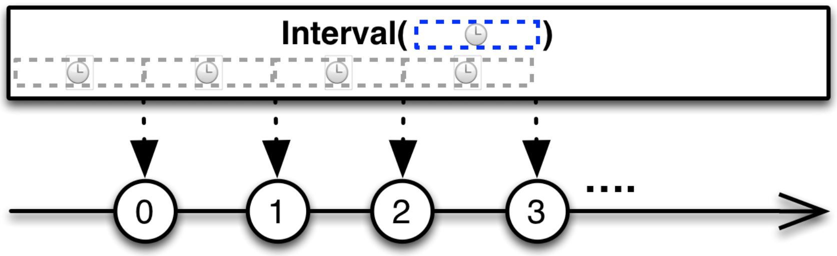


Figure 33. Rx interval operator creates an Observable that emits a sequence of integers spaced by a given interval [67]

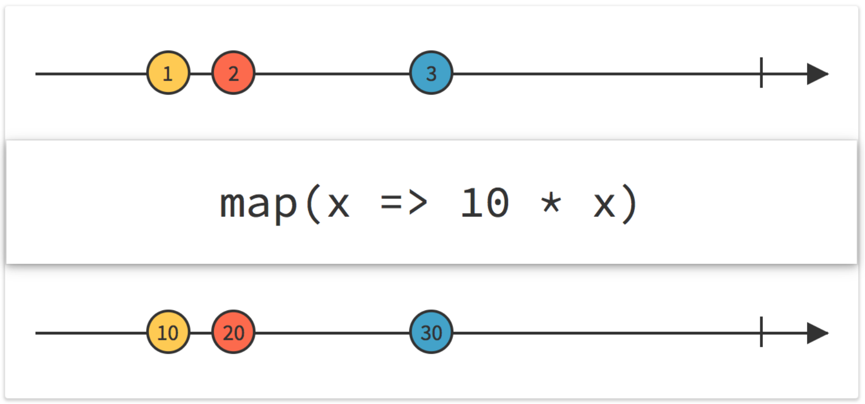


Figure 34. Rx map operator transforms items according to a function [67]

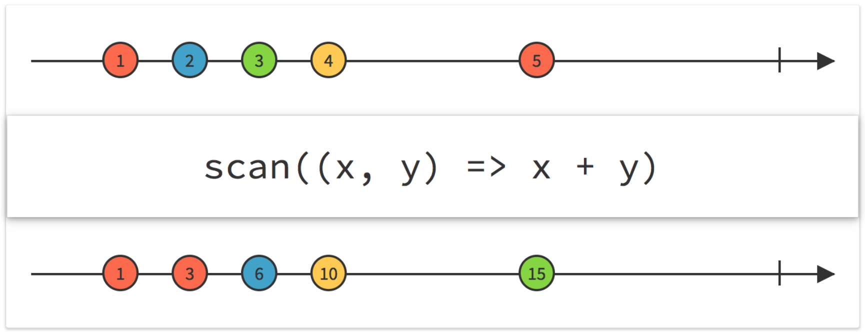


Figure 35. **Rx scan operator applies an accumulator function over the source [67]**

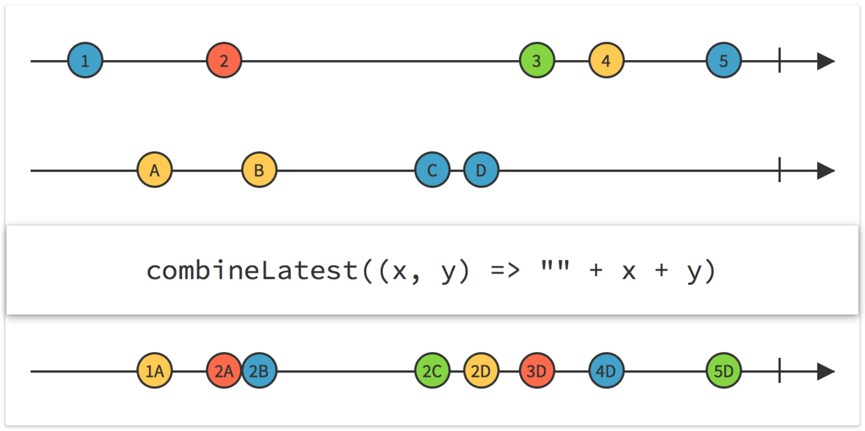


Figure 36. **Rx combineLatest operator combines the latest item emitted by each source based on the given function [67]**

Finally, stack traces printed from errors in code executing in an Observable can be so long as to be unusable. In recognition of this, the latest version of RxJS (version 5) has a specific design goal to shorten stack traces.

6.2.2 Modularity.

Network application modularity refers to the facilities provided by the controller framework to enable multiple, independent applications to be run at the same time. The user of the controller is the network administrator or operator who is writing or applying prewritten modules of functionality as a set of network control applications applied to the controller.

For example, for a given controller, a user may have a layer 2 switch module that remembers switch port locations of MAC addresses and uses this information to send `FlowMod` messages so that traffic is efficiently switched to the proper ports. The user may also have an access control list (ACL) module that prohibits certain traffic as specified by MAC address on certain links. The user may wish to install these and other modules on the switch to get the functionality of both, without having to rewrite both into a single switch/ACL module. The user probably also needs the ability to specify, for times where the output of the two may conflict, which module has precedence. Here, it makes sense to apply the ACL first, and not waste resources capturing the locations of blacklisted MAC addresses. With any number of possible modules it is important to expose a clean, consistent method for prioritization to the user.

6.2.2.1 Existing controllers.

Some controllers simply yield the matter of modularity completely to the user, expecting them to handle the nuances of instantiation, configuration, and prioritization of separate pieces of functionality. This is the case for the Onix distributed controller:

> Just to be clear, we only imagine a single "application" being used in any particular deployment; this application might address several issues, such as routing and access control, but the control platform is not designed to allow multiple applications to control the network simultaneously. [17]

Other controllers simply utilize the module features of the host language and platform, expecting users to be familiar with constructing modules in that language and importing and connecting them together in a single application given to the controller. This is the means of modularity for Python-based controllers like POX and Ryu, as well as some Java-based controllers. In most cases, each module provides a set of callback functions that specify a certain type of packet, message, or event to which they should be applied. This approach is effective but naive; multiple callback functions may be called for a given message, each applying opposing actions to the message. It is left to the user to wrap these in a parent callback function to prioritize and deconflict the modules. This does not scale well when many modules are needed, as the control flow is not explicitly defined but hidden in a series of messy parent modules and chained callbacks.

Finally, some controllers make use of language features like operator overloading to provide an explicit means of modularity. The "Frenetic family" of controllers, Frenetic-OCaml and Pyretic, solve the problem by implementing a DSL that provides sequential and parallel composition operators. For example, in Pyretic [24], the sequential (`>>`) and parallel (`+`) operators along with the Boolean "and" (`&`) can be used in a single expression to create a policy that performs destination IP routing across two switches:

```
(match(switch=A) & match(dstip='10.0.0.1') >> fwd(6)) +
(match(switch=B) & match(dstip='10.0.0.1') >> fwd(7))
```

These two lines are concatenated with a + so that they work in parallel. Each one matches on two specific conditions: an originating switch and a destination IP address. When those two conditions match, the sequential operator (>>) is used to perform an action, which in this case is to forward the matching flow out to the specified port (port 6 for switch A, and port 7 for switch B).

This method is effective and interesting from a research perspective, as it redefines modules as functional building-blocks that can be composed using a specific algebra. The main drawback to such an approach is that it does not resemble traditional programming, forcing a detailed understanding of a totally new paradigm on potentially novice students and researchers.

The library node-openflow (described in Chapter V) can be used directly to construct a controller, and provides simple examples for applications or benchmarking in a few different styles, including callback functions (via the Node.js `EventEmitter` API), Reactive Extensions (Rx) Observables, and the Node.js Transform Stream interface. Of these, callbacks are the easiest to implement for simple projects, with Observables being a better choice for larger projects, while transform streams provide the best performance. While a set of transform streams can be placed in a pipeline with high performance, the flow control structure is very rigid, and degrades to poor performance and/or becomes difficult to program when multiple parallel pipelines are required.

Instead of these approaches, rxdn uses RxJS Observables, but wraps them into a structured design which makes them more straightforward to use, similar to the Cycle.js web application framework.

6.2.2.2 Cycle.js.

Cycle.js is a web application framework initially designed around RxJS version 4, but has since been updated and expanded to support version 5 as well as several other reactive programming libraries. While RxJS could be used itself or within other frameworks, Cycle.js demands a more strict approach by abstracting the entire web application into a single `run` function which ties together a `main` function and a set of drivers. Side effects, including network communication, console logging, or outputting HyperText Markup Language (HTML) to the document object model (DOM) are separated as drivers. To conform to the framework, the main function and any functions it calls should be side-effect free functions [73]. This structure is represented in Figure 37.

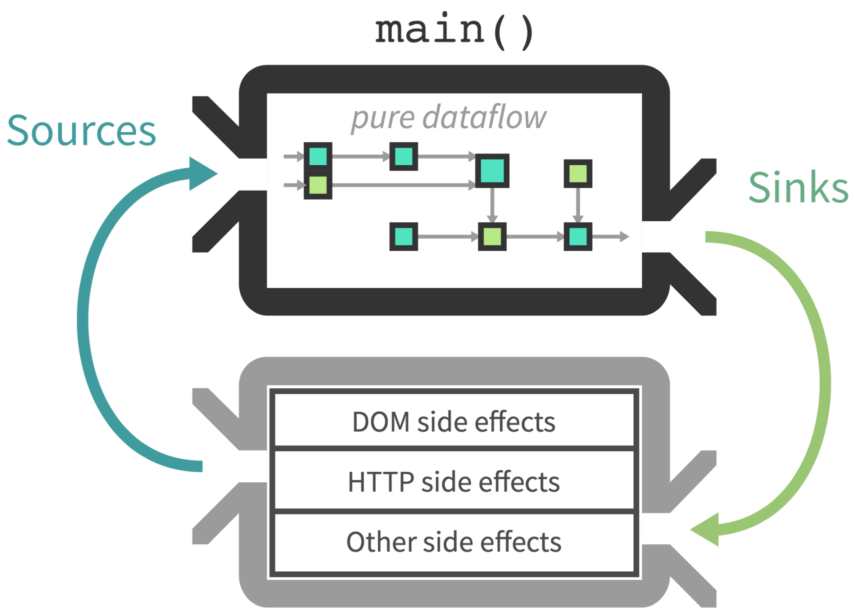


Figure 37. Cycle.js high-level application structure [73]

This separation of logic and effects is inspired by functional languages like Haskell. Haskell applies concepts from category theory to try to make a programming environment where code is safer, easier to test, and more powerful. Haskell uses concepts like referential transparency (side-effect free functions), immutability, functors, and monads to achieve these goals. Rx, Cycle.js, LINQ, and other libraries and frameworks inspired by the functional programming paradigm borrow one or more of these concepts, applying them to more traditional imperative or object-oriented programming languages.

Without going into a tutorial on functional programming in Haskell, it is clear that some of these concepts, while initially constraining, yield immediate benefits to a large code base. For example, referential transparency can allow chaining together functions in powerful ways, while also enabling lazy evaluation. Lazy evaluation has many uses, including being able to perform computations on just a subset of a large collection without iterating over the entire data set. Concepts like partial application and monads can be used to perform powerful generic queries over collections [74]. Also, by having a clear delineation between code which contains side effects or mutable objects and functions that do not, the latter group is far easier to test for correctness. Test methodologies like test driven development (TDD) are often difficult to implement due to the laborious process of creating test fixtures. These fixtures are predefined state of instantiated classes, used to test correct behavior of methods in the class when given some existing conditions. None of that is needed when testing a referentially-transparent function.

The web development community is littered with half-formed and abandoned frameworks, but Cycle.js has developed a small but dedicated following in a relatively short time. Cycle.js builds on Rx and other functional concepts to contribute composable, nested components with a clear and traceable data flow in a way that is appealing to novice and experienced programmers. A simple and idiomatic "Hello World" style web application in Cycle.js is shown in Figure 38. It clearly shows the cycle of how inputs (here, via the `sources.DOM` Observable) are used to create outputs.

```
import Cycle from '@cycle/xstream-run';
import {div, label, input, hr, h1, makeDOMDriver} from '@cycle/dom';

function main(sources) {
  return {
    DOM: sources.DOM.select('.myinput').events('input')
      .map(ev => ev.target.value)
      .startWith('')
      .map(name =>
        div([
          label('Name:'),
          input('.myinput', {attrs: {type: 'text'}}),
          hr(),
          h1(`Hello ${name}`)
        ])
      )
  };
}

Cycle.run(main, {
  DOM: makeDOMDriver('#main-container')
});
```

Figure 38. Example "Hello (name)" web application in Cycle.js [73]

6.3 Design and Implementation

6.3.1 Structure.

Cycle.js is a web application framework, but rxdn is not a web application; it is an OpenFlow controller written for Node.js. While the languages are the same and the APIs are similar, there are significant differences between a web application running in a web browser and a server application running in Node.js. Therefore, Cycle.js cannot be used directly, as it makes many assumptions about the code running in a browser and its primary focus is on efficiently writing HTML to the DOM. Therefore, a tailored structure was created based on the core concepts of Cycle.js to be more applicable to a server framework which accepts connections from clients (in this case, switches) and implements behavior based on a set of modules.

The primary abstractions provided by Cycle.js are the `run` function, the separation of logic into referentially transparent functions composed by a single `main` function, and the separation of side effects (e.g., network communication) into a set of drivers. The `run` function is called with `main` and the set of drivers as its arguments. Each driver is a function which takes an Observable called a "sink" and returns an Observable called a "source." The `main` function takes the set of sources as an argument passed from `run` and returns a set of `sinks`. These sets are objects where the key is equal to the name of the driver and the value is the Observable corresponding to that driver. Finally, the `run` function ties the sources input and `sinks` output of `main` together to the sink input and source output of each driver. This is represented in Figure 39, specifically with the `main` function in a cycle with the DOM driver, which is responsible in Cycle.js for writing HTML to the DOM, the primary output to the user (this driver is specific to a web application and is not present in rxdn).

These abstractions clearly delineate the location and structure of each piece of functionality. It also removes much of the confusion of using RxJS directly. For

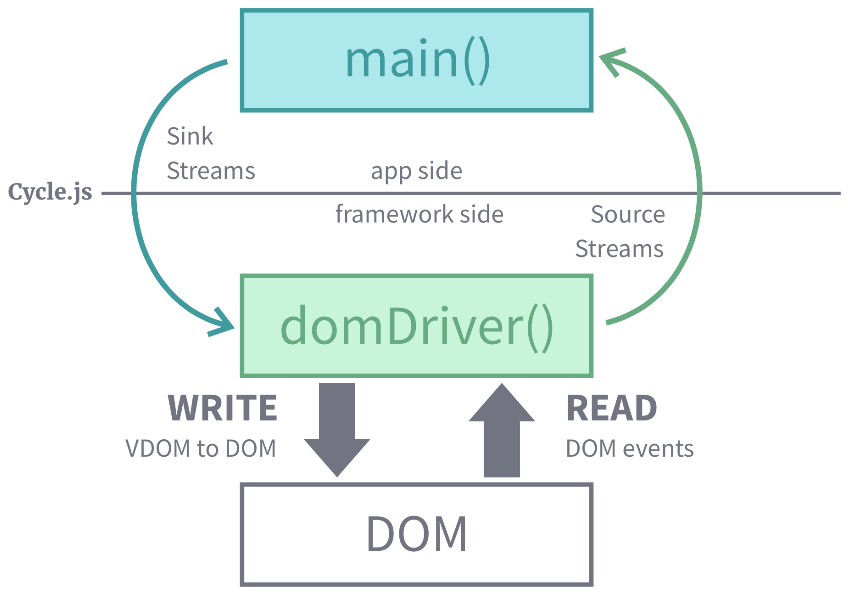


Figure 39. Cycle.js structure with DOM driver [73]

example, RxJS provides Observables, but does not inherently provide answers to questions like the following:

- At what point should an Observable be subscribed to?
- When should an Observable be unsubscribed?
- Should a given Observable be created as cold, hot, or multicast?

These questions may be straightforward for a simple application that merely requests some data from a server and displays it, but can become confusing in a large framework.

This structure creates a cycle, which at first seems to present a chicken-and-egg problem: what does `run` call first, the `main` function or each of the driver functions, and how can the outputs of one function be given as the inputs to another function when the first function has not yet been called? Simplifying drivers into a single function, the pseudocode could be:

```
let sinks = main(sources)
let sources = drivers(sinks)
```

This cannot work, as `sources` is referenced before it is created. In more mathematical terms, this could be written

$$a = f(b) \tag{2}$$

$$b = g(a) \tag{3}$$

When combined, this yields $a = f(g(a))$. By understanding that `sources` and `sinks` are not single-assignment variables, but Observables which represent a stream of events, it becomes clear that intermediate or *proxy* Observables can be used to tie together these function input and output streams (where p represents a proxy):

$$p = \text{(initialize)} \tag{4}$$

$$a = f(p) \tag{5}$$

$$b = g(a) \tag{6}$$

$$p = f(b) \tag{7}$$

Or in pseudocode,

```
let sinks = main(sources)
let sources = drivers(sinks)
```

This problem is solved by Cycle.js (and rxdn) with the use of Subjects. A Subject is a class which implements both the Observer and Observable interfaces. This means it can both subscribe to an Observable and itself be subscribed to. As all inputs and outputs for these functions are Observables, the Subject class can be used to create a set of proxies which act as intermediaries to tie these functions together. The actual `run` used in rxdn is clear and concise and is shown in Figure 40. The interfaces defined in rxdn are also clear and concise, and help in understanding the `run` function; these are shown in Figure 41.

```
import {Subscription, ReplaySubject} from "rxjs";
import {Collection, Component, Drivers} from "./interfaces";

interface SubjectCollection {
  [name: string]: ReplaySubject<any>;
}

function makeProxies(drivers: Drivers): SubjectCollection {
  const proxies: SubjectCollection = {};
  const names = Object.keys(drivers);
  names.forEach(name => { proxies[name] = new ReplaySubject(1); });
  return proxies;
}

function callDrivers(drivers: Drivers, proxies: Collection): Collection {
  const sources: Collection = {};
  const names = Object.keys(drivers);
  names.forEach(name => {
    let source = drivers[name](proxies[name]);
    sources[name] = source;
  });
  return sources;
}

function subscribeAll(
  sinks: Collection, proxies: SubjectCollection
): Subscription {
  const subscription = new Subscription();
  const names = Object.keys(sinks);
  names.forEach(name => {
    subscription.add(sinks[name].subscribe(proxies[name]));
  });
  return subscription;
}

export function run(main: Component, drivers: Drivers): Subscription {
  const proxies = makeProxies(drivers);
  const sources = callDrivers(drivers, proxies);
  const {sinks} = main(sources);
  const subscription = subscribeAll(sinks, proxies);
  return subscription;
}
```

Figure 40. Run function of rxdn

```typescript
import {Observable} from "rxjs";

/** A collection of Observables indexed by key */
export interface Collection {
  [name: string]: Observable<any>;
}

/**
 * A Component is a function which accepts a source of Observables
 * indexed by key (a Collection) and returns sources as inputs to
 * composed Components and sinks as inputs to Drivers. A Component
 * should not create side-effects, as this is the function of a Driver.
 */
export interface Component {
  (sources: Collection): {sources: Collection, sinks: Collection};
}

/**
 * A Driver is a function which takes an Observable (a Sink) and returns
 * an Observable (a Source).
 * The Driver is the place to acquire events or data from external sources
 * and to create side-effects.
 */
export interface Driver<Sink, Source> {
  (sink: Observable<Sink>): Observable<Source>;
}

/** A collection of Drivers indexed by key */
export interface Drivers {
  [name: string]: Driver<any, any>;
}
```

Figure 41. rxdn interfaces

6.3.2 Drivers.

A driver is simply a function with accepts an Observable of any type as its `sink` and returns an Observable of any type as its `source`. The `run` function is responsible for calling each driver with its corresponding `sink` and collecting all drivers' `source` Observables into the `sources` object which is the input to `main`.

There are three drivers implemented in rxdn, and it is a simple pattern to follow to add more. The simplest is the console logging driver. As logging is a side-effect, this functionality can be implemented as a driver. Note that there is nothing technically stopping a component from including a `console.log` statement; however, starting with a trivial console logging is useful to show how components and drivers interact.

The source of the console driver is given in Figure 42. In it, observe that it (and every driver):

- Subscribes to the input `sink` Observable
- Performs some side-effect, most likely related to the input `sink` events
- Returns an Observable, most likely related to the result of the performed side-effect

In this case, there is no "result" from printing to the console, so the returned Observable is `Observable.never()`. This special static instantiation method returns an Observable which never emits any events. Failing to return an Observable is an error, as the TypeScript compiler will report, since it does not adhere to the interface of a driver. Notice that this driver is declared of type `Driver<string, void>`, which (in TypeScript notation) denotes that the input is an Observable of type `string` and the output is an Observable of type `void`, the "never" Observable. (Reference Figure 41 for the declaration of the `Driver` interface).

```
import {Driver} from "../interfaces";
import {Observable} from "rxjs";

/**
 * Logs sink to the console
 */
export const consoleDriver: Driver<string, void> = sink => {
  /* tslint:disable-next-line:no-console */
  sink.subscribe(msg => console.log(msg));
  return <Observable<any>> Observable.never();
};
```

Figure 42. The rxdn console driver

A driver for state is also included. While it is possible to have accumulated state via the `scan` operator within a component, the state driver shows how state can be separated as a side-effect and lays a foundation for distributed state.

The source of the state driver is given in Figure 43. Rather than directly exporting the driver function, this driver wraps the driver function within a creation function, allowing the application to provide an initial state with which to kick off the Observable. As shown in Figure 40, the Subjects tying together the `main` function input with the output of the drivers are of type `ReplaySubject(1)`, which means the last emitted item will be cached and accessible to any subscribers, and for state to be added to by a component, requires an initial value. The state driver cannot have a hard-coded initial value as the driver itself is generic, as shown in Figure 43 by its type of `Driver<T, T>`. This indicates that the type of the input Observable is the same as the type of the output Observable.

With this simple implementation, a component can get and set state from one or more instantiated state drivers. This driver could also be extended as a database client. For example, if state is kept as a JavaScript object type, this can be serialized with JSON and sent to a document store database like MongoDB.

```
import {Driver} from "../interfaces";
import {Observable} from "rxjs";

/**
 * Generic state driver
 * @param {T} initialState The initial state to use
 * @returns {Driver<T, T>}
 *
 * @example
 *     const updateState = otherObservable
 *       .map(value => state => state.set("key", value));
 */
export const makeStateDriver: <T>(initialState: T) =>
  Driver<T, T> = <T>(initialState: T) =>
    (sink: Observable<(state: T) => T>) =>
      sink
        .scan((state, changeFn) => changeFn(state), initialState)
        .startWith(initialState)
        .share();
```

Figure 43. The rxdn state driver

Finally and most importantly, rxdn includes an OpenFlow driver. The OpenFlow driver accepts connections from OpenFlow switches and maintains a map of socket IDs to socket objects, allowing it to send any OpenFlow message to any connected switch, as dictated by the components through its `sink` input. The type of the OpenFlow driver is `Driver<OFEvent, OFEvent>`, where `OFEvent` is defined in Figure 44; the `id` field is a concatenation of the socket remote address and remote port as a string, the event is an enumerated type indicating why the event is occurring (e.g., a switch connected, or a message was received), and where appropriate, the type may include an error object or OpenFlow message object. This is the interface used by each component to send and receive messages to and from OpenFlow switches. The algorithm for this driver is given in Algorithm 2.

```
export enum OFEventType {
  Connection,
  Disconnection,
  Error,
  Message,
}

export type OFEvent =
  {id: string, event: OFEventType.Connection} |
  {id: string, event: OFEventType.Disconnection} |
  {id: string, event: OFEventType.Error, error: Error} |
  {id: string, event: OFEventType.Message, message: OF.OpenFlowMessage};
```

Figure 44. The OFEvent type

Algorithm 2 rxdn OpenFlow driver

```
subscribe to sink input
for each message event in sink do
    socket ← lookup id in sockets map
    buffer ← call encode on message
    write buffer to socket
end for
source ← create new Observable of type OFEvent
for each new connection do
    add socket to socket map
    emit new connection
    for each new buffer from socket do
        messages ← call decode with buffer
        for each message in messages do
            emit message
        end for
    end for
end for
return source
```

6.3.3 Components.

Components in rxdn are similar to the driver pattern; each component implements the `Component` interface declared in the top-level interfaces. As a shortcut, OpenFlow components can import and implement the `OFComponent` interface from the OpenFlow driver. Both interface declarations and their corresponding collections are shown in Figure 45.

```
// Declared in interfaces.ts

/** A collection of Observables indexed by key */
export interface Collection {
  [name: string]: Observable<any>;
}

/**
 * A Component is a function which accepts a source of Observables
 * indexed by key (a Collection)and returns sources as inputs to composed
 * Components and sinks as inputs to Drivers. A Component should not
 * create side-effects, as this is the function of a Driver.
 */
export interface Component {
  (sources: Collection): {sources: Collection, sinks: Collection};
}

// Declared in drivers/openflow.ts

export interface OFCollection extends Collection {
  openflowDriver: Observable<OFEvent>;
}

export interface OFComponent {
  (sources: OFCollection): {sources: OFCollection, sinks: OFCollection};
}
```

Figure 45. Component interfaces

A very simple component is `Hello10`, which sends an OpenFlow 1.0 `Hello` message to any switch upon its connection to the controller. This component would not normally be used by itself, but in conjunction with other small modules to enable the overall controller functionality. In particular, it is wrapped into a larger `Core10` component providing other specification-required functionality, which is in-turn used by the `Switch10` component to provide layer 2 switch functionality. (If OpenFlow 1.3 is desired as the default message type, `Hello13`, `Core13`, and `Switch13` components are also provided in the framework). The source for `Hello10` is shown in Figure 46.

```
import {OFComponent, OFEventType, OFEvent} from "../../drivers/openflow";
import {of10, of13} from "@dancasey/node-openflow";

/** Sends an OpenFlow 1.0 Hello message upon connection */
export const Hello10: OFComponent = sources => {
  const hello = sources.openflowDriver
    .filter(m => m.event === OFEventType.Connection)
    .map(m => <OFEvent> {
      event: OFEventType.Message,
      id: m.id,
      message: new of10.Hello(),
    });

  return {sources, sinks: {openflowDriver: hello}};
};
```

Figure 46. Component which sends Hello message upon switch connection

The `Hello10` component first filters the source Observable from the OpenFlow driver (`sources.openflowDriver`) to find any events of type `OFEventType.Connection`. It then applies the `map` operator to transform these incoming events into a new OpenFlow message event with a new OpenFlow version 1.0 message attached to the event. The driver will receive the event, find the socket object in its map based on the `id` string, encode the attached message object into buffer, and send the buffer on the corresponding socket. If a component needs to send a message to another switch, it only needs to reference the appropriate `id` string for the target switch.

A component does not need to depend on an incoming event in order to send a message to a switch. For example, a topology discovery component may need to send periodic discovery messages; this can easily be accomplished with the static Observable creation method `interval` (shown as a marble diagram in Figure 33). Furthermore, while the component interfaces requires a single Observable output returned to each driver, multiple concerns can easily by addressed by combining Observables with any of a host of static and class Observable-combining methods methods, including `merge`, `combineLatest`, `race`, `zip`, and more. This provides a huge amount of flexibility while still constraining the pattern such that each component can be small and simple to understand.

This driver/component pattern, adapted from Cycle.js and RxJS, is at once simple, consistent, and easy to extend. Combining this with TypeScript's powerful code analysis, correctness checking, and autocompletion ("intellisense") features makes rxdn easy to write for and experiment with. It also is a natural fit for modular composition of functionality, allowing the user to build small, testable, and reusable blocks of functionality and connect them in a consistent, intuitive way.

6.3.4 Composition.

As alluded to, more advanced components are simply a result of composition of smaller components. As a top-down example, the layer 2 switch application is a composition of the components `Core10`, `Switch10`, and `OFLog` (which logs OpenFlow-related events to the console). The `Core10` component is itself a composition of `Echo` (which replies to `EchoRequest` messages), `Hello10`, and `Features` (which sends a `FeaturesRequest` message to newly-connected switches). The `Switch10` component is a composition of `SwitchMemory`, `PacketOut`, and `FlowMod`. The layer 2 switch application is shown in Figure 48 and graphically in Figure 47. The `Compose` function is shown in Figure 49.

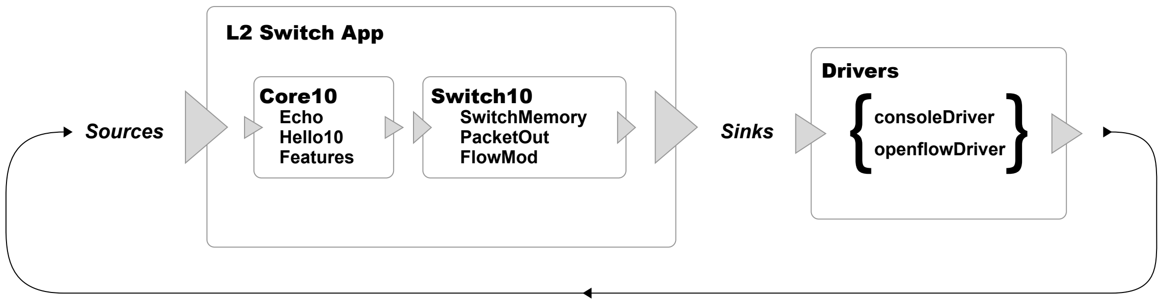


Figure 47. Layer 2 switch application component composition

Figure 48 demonstrates how `Compose` simply takes an array of components and a `sources` collection and returns a `sources` collection and a `sinks` collection, which means the result conforms to the component interface and can be dropped-in to the `main` function or another component. The reason for including a `sources` collection in the output of each component is now clear: it allows *upstream* components to filter out certain events from the stream which have already been handled. Alternatively, events can be modified or new events added to the `sources` stream. However, this type of modification should be limited, as it may begin to break the pattern of keeping side-effects in drivers, potentially leading to a confusing network application.

The implementation of the `Compose` function (Figure 49) steps through the `sinks` outputs of each component, using the `merge` method to combine those with the same key. For `sources`, it chains the output of one component to the input of the next, such that earlier components receive the events first and may filter some out. For example, the `Echo` component filters out all `EchoRequest` message events, as it handles sending the required `EchoReply` message event, and no further component would need this message. This is depicted graphically in Figure 50.

```
import * as rxdn from "../rxdn";
import {Observable} from "rxjs";

interface OFConsoleCollection extends rxdn.OFCollection {
  consoleDriver: Observable<string>;
}

/**
 * Example L2 switch controller for OpenFlow 1.0.
 * Run with `node dist/examples/switch.js`.
 */
const main: rxdn.OFComponent = src => {
  return <{
    sources: rxdn.OFCollection,
    sinks: OFConsoleCollection,
  }> rxdn.Compose([
    rxdn.Core10,
    rxdn.Switch10,
    rxdn.OFLog,
  ], src);
};

const drivers: rxdn.Drivers = {
  consoleDriver: rxdn.consoleDriver,
  openflowDriver: rxdn.makeOpenFlowDriver({
    host: "0.0.0.0",
    port: 6633,
  }),
};

rxdn.run(main, drivers);
```

Figure 48. Layer 2 switch application

```typescript
import {Collection} from "../interfaces";

/**
 * Takes an array of components and a sources object and composes the
 * components such that sources output from one component flow as input
 * sources to the next, and sinks from each component are merged as
 * returned as a single sink object.
 */
export const Compose =
  <T extends Collection>(components: Array<(sources: T) =>
  {sources: T, sinks: T}>, sources: T) => {
    let nextSources: T = sources;
    let componentSinks: T;
    let sinks = <T> {};
    let result: {sources: T, sinks: T};

    components.forEach(component => {
      result = component(nextSources);
      nextSources = result.sources as T;
      componentSinks = result.sinks;

      // For any key of the nextSinks that is in the sinks, merge them.
      // Otherwise, add the new `key: stream` to sinks
      for (let sink in componentSinks) {
        if (sinks.hasOwnProperty(sink)) {
          sinks[sink] = sinks[sink].merge(componentSinks[sink]);
        } else {
          sinks[sink] = componentSinks[sink];
        }
      }
    });

    return {sources: nextSources, sinks};
};
```

Figure 49. Compose function

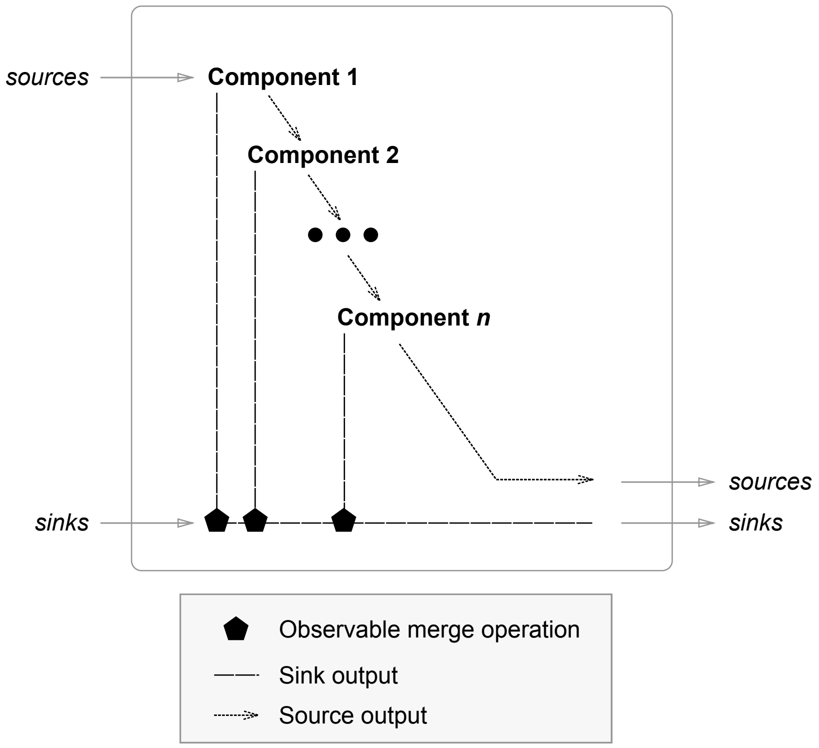


Figure 50. rxdn Compose function

6.4 Future Work

The two most lucrative areas for future work on this framework are in performance and distribution. The current performance of the framework and areas where it can be improved are covered together with the library performance in Chapter VII. Controller distribution is a complex topic and current major area of research in the SDN community [17, 75, 76].

This framework provides an interesting vehicle to begin to study controller distribution. As described in Section 6.3, the framework promotes a clear delineation between logic and side-effects by extracting state management from components and moving them into drivers. This separation is inherited from functional programming, and is a solid foundation for distribution, as one of the most difficult hurdles in distributed systems is state management and synchronization.

As described in Section 6.3.2, the simple state driver provided could easily be extended to include a client connection to a database, providing a common memory for distributed instances of the controller. Additionally, many document databases have built-in means for distribution, which would allow another independent variable for experimentation (i.e., multiple distributed controllers with one backing database as memory versus multiple distributed controllers with a distributed backing database).

VII. Controller Performance Testing

7.1 Introduction

Many of the qualities of an OpenFlow controller are subjective. For example, the ease of development of network applications, the simplicity of deployment, and code maintainability are all subject to the practitioner's exposure to various programming languages and paradigms and experience in network operations. However, a controller's ability to rapidly and reliability answer requests as the number of switches and network load increases is distinctly quantifiable.

This chapter explains the means by which OpenFlow controllers are traditionally measured in their performance, cover the testing methodology used to compare the products in this document to existing controllers, and finally give the results of the measurements.

7.2 Related work

There have been a number of notable controller performance testing tools and papers released since 2012. This section briefly describes these and their design.

The most widely-used controller benchmarking tool is Cbench [22]. Cbench emulates a configurable number of switches and hosts, the latter of which are represented as unique MAC addresses. For each switch, it makes a connection to the controller and starts sending `PacketIn` messages. In latency mode, Cbench waits between each `PacketIn` for the controller to respond with a `PacketOut` or `FlowMod`, and calculates the number of responses per second it receives from the controller. In throughput mode, Cbench sends as many `PacketIn` messages as will fit in its outgoing buffers and measures the total number of responses received to calculate the overall responses per second. By default, the program runs 16 times with 16 switch connections to the controller, discarding the first run as a warmup, but these parameters are all configurable via command line switches. Despite supporting only version 1.0 of the OpenFlow protocol, Cbench is widely used in the community as it is simple and effective. Cbench has also inspired several other controller benchmarking designs [77–79]. The Cbench algorithm is given in Algorithm 3.

Algorithm 3 Cbench benchmarking tool

```
-s command line switch becomes $s$, number of switches; default 16
-l command line switch becomes $l$, number of test loops; default 16
-m command line switch becomes $m$, duration of each test in ms; default 1000
Create $s$ OpenFlow sessions to the controller
if Latency mode (default) then
    for each session of $s$ sessions, each test of $l$ tests, over $m$ ms per test do
        Send PacketIn message
        Wait for matching FlowMod or PacketOut message
        Calculate number of messages received per second
    end for
else if Throughput mode (-t) then
    for each session of $s$ sessions, each test of $l$ tests, over $m$ ms per test do
        while Buffer not full do
            Queue PacketIn messages
            Count number of FlowMod or PacketOut messages received
        end while
    end for
end if
Calculate overall min/max/avg/stddev responses per second
```

OFCProbe [79] is a modular approach to controller benchmarking that was inspired by Cbench but written in Java. It is a redesign from the same author's previous work in benchmarking, OFCBenchmark [77]. The authors claim their updated tool offers a much higher degree of detail of results when compared to Cbench, being able to discern performance characteristics of individual virtual switches, making more insightful analysis possible. The separation of concerns into traffic generation, configuration, and statistics gathering makes the tool both more cross-platform and easier to extend than Cbench, which is written monolithically and requires the Linux kernel API.

The authors of hcprobe [78] adopt Cbench for performance testing and extend it into a larger framework which adds “functional/regression” and security tests. They argue that the criteria for selection of a controller extends beyond performance, and attempt to use their extensions to Cbench to quantify these additional criteria. Interestingly, they conclude that none of the controllers tested are truly ready for use in “enterprise”, which they define as any environment where security and reliability are the highest priorities.

More recently, papers including [80, 81] have utilized plain Cbench, unmodified since its 2012 release, to perform and report measurements of modern versions of popular controllers. The authors of these 2015 and 2016 papers generally concede that raw performance such as that measured by Cbench is by no means the only criteria for selecting a controller. They relegate Python-based controllers to the role of small, prototype networks due to their single-threaded nature and inability to scale. They conclude that the best performance is achieved by controllers written in C, C++, or Java.

7.3 Methodology

The problems of results gathering, test target orchestration, and Cbench parameter automation can be solved with a combination of scripting and containerization, and that is the main contribution of this methodology, detailed in Section 7.3.5.

The logical test configuration is depicted in Figure 51. This is a logical, not physical diagram, as all testing occurred internal to a single computer. The figure shows the test control program, Cbench runner, which creates and runs containers to perform testing. One container runs modified Cbench while the other runs the selected controller. The two containers are connected via virtual network. Results from the

Cbench container (from standard output, or `stdout`) are collected by the Cbench runner and appended to a results file in comma-separated values (CSV) format.

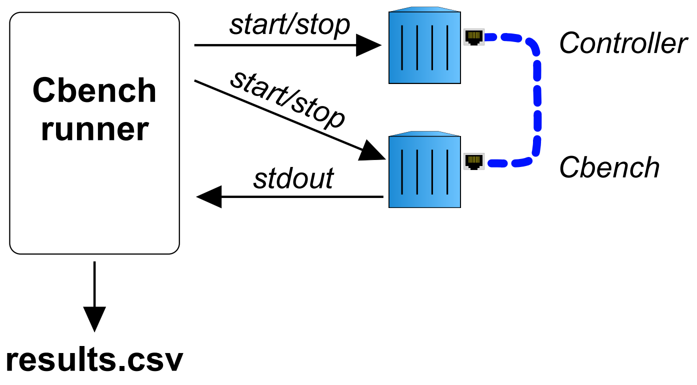


Figure 51. Test setup for Cbench runner controlling containers and collecting results

Each portion of Figure 51, including the software, hardware, parameters, and test suite are described in the rest of this section, with results and analysis following in the next section.

7.3.1 Cbench shortcommings.

There have been many published uses of Cbench and similar tools, with Cbench being the common standard for performance measurement. However, Cbench has a number of shortcomings, particularly in running multiple tests and gathering results. The program creates no files; connection and error-related messages are printed to standard error while performance related messages and statistics are printed to standard output. Example output of Cbench in action is given in Figure 52. Lines 11–26 show the results of each individual test run. Each of these lines start with a timestamp, followed by the number of simulated switches in that run. The four numbers in the middle represent the number of responses received for each of those four switches for the 1-second test interval. The rightmost number is a sum of these four numbers expressed in milliseconds. (Note that there is a small amount of error due to floating point representation; in the first line, $1332 \times 4/1000 = 5.328$ ms.) Line

27 is a summary statement of lines 11–26. While the results are clear for a single test, the program's lack of output options make running many tests across different controllers very labor-intensive.

```
1  cbench: controller benchmarking tool
2     running in mode 'latency'
3     connecting to controller at ryu:6633
4     faking 4 switches offset 1 :: 16 tests each; 1000 ms per test
5     with 100000 unique source MACs per switch
6     learning destination mac addresses before the test
7     starting test with 0 ms delay after features_reply
8     ignoring first 1 "warmup" and last 0 "cooldown" loops
9     connection delay of 0ms per 1 switch(es)
10    debugging info is off
11 18:22:05.564 4   switches: flows/sec:  1332  1332  1332  1332   total = 5.327995 per ms
12 18:22:06.666 4   switches: flows/sec:  1295  1294  1294  1294   total = 5.176995 per ms
13 18:22:07.770 4   switches: flows/sec:  1302  1302  1302  1302   total = 5.207995 per ms
14 18:22:08.876 4   switches: flows/sec:  1292  1292  1292  1291   total = 5.166969 per ms
15 18:22:09.977 4   switches: flows/sec:  1308  1307  1308  1308   total = 5.230995 per ms
16 18:22:11.079 4   switches: flows/sec:  1283  1282  1283  1283   total = 5.130995 per ms
17 18:22:12.182 4   switches: flows/sec:  1296  1295  1295  1296   total = 5.181995 per ms
18 18:22:13.283 4   switches: flows/sec:  1194  1193  1194  1193   total = 4.773995 per ms
19 18:22:14.384 4   switches: flows/sec:  1230  1230  1230  1230   total = 4.919995 per ms
20 18:22:15.485 4   switches: flows/sec:  800  794  794  796   total = 3.183997 per ms
21 18:22:16.588 4   switches: flows/sec:  1115  1116  1115  1115   total = 4.460960 per ms
22 18:22:17.689 4   switches: flows/sec:  1011  1010  1011  1010   total = 4.041996 per ms
23 18:22:18.789 4   switches: flows/sec:  670  669  669  668   total = 2.675997 per ms
24 18:22:19.890 4   switches: flows/sec:  1025  1025  1025  1025   total = 4.099996 per ms
25 18:22:20.990 4   switches: flows/sec:  1153  1152  1151  1152   total = 4.607995 per ms
26 18:22:22.093 4   switches: flows/sec:  611  607  610  608   total = 2.434566 per ms
27 RESULT: 4 switches 15 tests min/max/avg/stdev = 2434.57/5230.99/4419.70/918.86 responses/s
```

Figure 52. Example of unmodified Cbench output running against Ryu

Another problem with using Cbench to run multiple tests is the lack of test subject isolation across multiple tests. In other words, simple test scripting with Cbench along the lines of "run a test for each of the following parameters" is inadequate, because the same instance of a controller is running for each test, and state from a previous test may significantly alter the results of the current and future tests. To achieve uncontaminated results and a truer picture of performance, each controller must be restarted before each test. Cbench provides no means to do this, and it is unclear whether any of the published results papers took such considerations into account.

7.3.2 Software.

7.3.2.1 Cbench Modification.

The benchmark program in each case was Cbench, as it is the most widely-used and accepted tool for controller performance measurements. Cbench was modified only in its output (`printf` statements) such that its standard output could be easily concatenated into a CSV file. This allows the results to easily be read and plotted by programs like R and Excel. An example of unmodified Cbench output is given in Figure 52, and modified CSV output in Figure 53. In the latter, all text that is not CSV-style results is part of standard error, so it is not stored by the test suite. The CSV columns are, in order, the number of fake switches (parameter `-s`), a boolean value indicating whether this result is a warmup loop, and the remaining columns are the number of responses received for this iteration for each fake switch, in order. The changes to Cbench are included in patch format in Appendix D.

7.3.2.2 Controllers Tested.

A variety of controllers which are written for different languages and platforms were tested. They are listed in Table 4 along with their type and the version tested.

Table 4. Tested OpenFlow controllers

Controller	Version	Language/Platform
rxdn	0.1.6	TypeScript 2.1, Node.js 7.3.0
node-openflow	0.3.9	TypeScript 2.1, Node.js 7.3.0
Libfluid	0.1.0	C++ and libevent 2.0
Ryu	4.9	Python 2.7.12
ONOS	1.9.0	Java 1.8.0
NOX	0.9.2	C++ with libboost
Trema	0.10.1	Ruby 2.3.3

```
cbench: controller benchmarking tool
running in mode 'latency'
connecting to controller at ecstatic_jepsen:6633
faking 4 switches offset 1 :: 16 tests each; 1000 ms per test
with 100000 unique source MACs per switch
learning destination mac addresses before the test
starting test with 0 ms delay after features_reply
ignoring first 1 "warmup" and last 0 "cooldown" loops
connection delay of 0ms per 1 switch(es)
debugging info is off
4, 1, 1346, 1346, 1346, 1346
4, 0, 1272, 1271, 1271, 1272
4, 0, 1200, 1199, 1198, 1198
4, 0, 1301, 1301, 1300, 1300
4, 0, 1275, 1274, 1274, 1274
4, 0, 1299, 1299, 1298, 1299
4, 0, 1277, 1277, 1276, 1276
4, 0, 1301, 1301, 1301, 1301
4, 0, 1320, 1319, 1319, 1320
4, 0, 1315, 1316, 1316, 1315
4, 0, 1287, 1287, 1286, 1286
4, 0, 1317, 1316, 1317, 1315
4, 0, 1312, 1312, 1312, 1312
4, 0, 1309, 1309, 1309, 1308
4, 0, 1293, 1292, 1293, 1293
4, 0, 1302, 1302, 1302, 1302
```

Figure 53. Example of modified Cbench output (CSV)

7.3.2.3 Test Order.

The order of the tests was latency mode first, followed by throughput, with controllers tested in the order shown in Table 4 by increasing numbers of switches.

7.3.3 Test System.

The tests were run on a four-core, 2.6 GHz AMD Opteron 4130 CPU running Linux kernel 4.8.13 with 16 GB RAM. The tests wrote result data asynchronously and outside the execution of each test, making the specifications of the hard disk array irrelevant. The system was a headless, bare-minimum installation of Arch Linux, and was not running any active processes at the time of the test other than the test suite itself and its supporting processes. Similarly, the processes under test as well as the Cbench process itself were run as containers on the same system, making physical network interface card (NIC) specifications irrelevant.

7.3.4 Test Workload Parameters.

Cbench was run in each of its test modes (latency and throughput) with the number of fake switches increasing according to 2^n for $0 \leq n < 8$. The rest of the Cbench parameters were left at their default values.

Each of the controllers was tested using its included application for benchmarking. These applications are included in virtually every controller package, and are usually a simplified version of a switch or hub with logging and other unnecessary functions disabled. As Cbench simply counts replies and does not distinguish between `PacketOut` and `FlowMod` messages, some of the benchmarking applications would not be suitable for a real network as they take shortcuts, such as failing to include proper ports or MAC addresses in replies. For the purpose of finding a maximum performance measurement, this approach is simple and effective, but it is important to note that a network application could not perform better than this measurement without changes to the controller itself.

The number of replications for each test was calculated using the pwr package for the R programming language [82]. The package calculates parameters related to statistical power for a variety of tests based on guidance given in [83]. As the object of statistical analysis for the controllers is to compare their means of performance, analysis of variance (ANOVA) is an appropriate statistical tool to use. The function `pwr.anova.test` is used to calculate one missing parameter out of the set; in this case, the parameter being solved for is the number of replications, n. The number of groups, k, is equal to the number of controllers tested. The `sig.level` and `power` parameters are equal to α (the Type I error rate, or probability of detecting an effect that does not exist) and $1 - \beta$ (the power, which is the inverse of the Type II error rate, or probability of failing to detect an effect that does exist), respectively. Typical values for these parameters are $\alpha = 0.05$ and $1 - \beta = 0.8$, and there was not deemed to be a reason to deviate from these. The effect size can be calculated as the ratio of the standard deviation of the k means to the standard deviation of the k groups. In this case, these values were not known before hand, so conventional effect size to detect a "medium" effect as defined by [83] was used. This value is also given by a function of the pwr package. The output of both is shown in Figure 54. The ceiling of the value for n is the minimum number of replications for each test; to add a slight margin, the value $n = 36$ was chosen.

A summary of the test factors is listed in Table 5.

Table 5. Test factors

Factor	**Values**
switches	$1, 2, 4, 8, 16, 32, 64, 128$
mode	latency, throughput
controllers	rxdn, node-openflow, Libfluid, Ryu, ONOS, NOX, Trema

```
> library(pwr)
> cohen.ES(test = "anov", size = "medium")

     Conventional effect size from Cohen (1982)

           test = anov
           size = medium
    effect.size = 0.25

> pwr.anova.test(k = 7, f = 0.25, sig.level = 0.05, power = 0.8)

     Balanced one-way analysis of variance power calculation

              k = 7
              n = 32.05196
              f = 0.25
      sig.level = 0.05
          power = 0.8

NOTE: n is number in each group
```

Figure 54. Calculation of effect size and number of replications via pwr in the R programming language

A System Under Test (SUT) diagram is shown in Figure 55. Varied factors are shown as workload parameters on the left, with constant system parameters on top. The constant controller software parameters are listed in Table 4, with system software and hardware parameters described in Section 7.3.3. The output of the SUT are the metrics, which are each written as a single line into a results file, including which controller was tested, with which parameters, and the response counts for each simulated switch. (Warmup results are also written but are discarded in analysis).

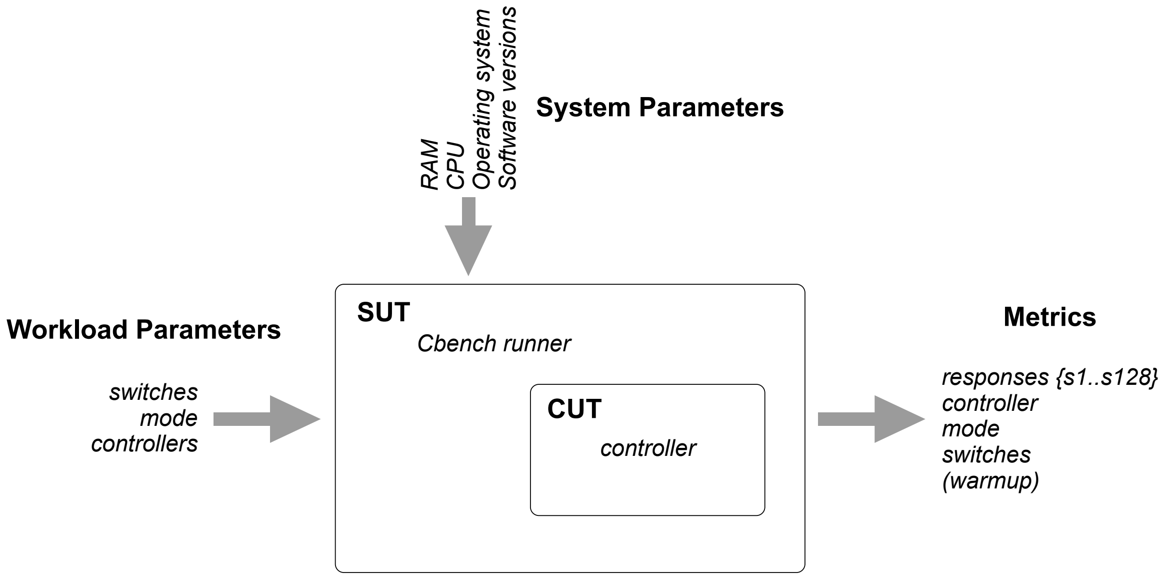


Figure 55. System under test diagram

7.3.5 Test Suite.

The test suite is a single Git repository called "Cbench runner" which contains all the components needed to install, build, and run tests across seven controllers, with a simple template to easily add others. The only prerequisites are Git [84], Node.js [51], and the Docker software containerization platform [85], all of which are freely available and easy to install on Linux, Mac, and Windows, allowing anyone to quickly and easily perform controller performance tests on virtually any platform.

The test suite contains a directory of Docker build files (called Dockerfiles): one for the customized, CSV-printing version of Cbench, and one for each of the seven controllers included. These consist of a series of build instructions which the Docker daemon uses to download the parent container image and install the controller upon. The Dockerfile for Libfluid is shown as an example in Figure 56.

```
FROM buildpack-deps:precise

RUN apt-get update
RUN apt-get install -y --no-install-recommends libevent-dev libssl-dev

RUN git clone https://github.com/OpenNetworkingFoundation/libfluid.git
WORKDIR libfluid
RUN ./bootstrap.sh
WORKDIR libfluid_base
RUN ./configure --prefix=/usr
RUN make
RUN make install
WORKDIR ../libfluid_msg
RUN ./configure --prefix=/usr
RUN make
RUN make install
WORKDIR ../examples/controller
RUN make all

# Suitable options here are:
# - msg_controller
# - raw_controller
# - secure_controller
ENTRYPOINT ["./msg_controller"]

# Suitable options here are:
# - l2
# - cbench
CMD ["cbench"]
```

Figure 56. Dockerfile for a Libfluid image for Cbench testing

The test suite also contains a Node.js program written in TypeScript which runs each test for each of the desired numbers of fake switches and each of the controllers.

The program handles restarting the controller between each test to provide isolation, prepending the controller name and test mode (latency or throughput), and writes all the results to a single CSV file. The algorithm for the program is given in Algorithm 4, and its usage in Figure 57. The full source code of the program is given in Appendix C.

Algorithm 4 Test suite runner

```
Open results file for appending
for each mode of throughput, latency do
    for each controller to be tested do
        for each of the numbers of fake switches given in -s do
            Start controller container
            Start Cbench container, linked to controller
            Wait for Cbench exit
            Stop and remove controller container
            Split Cbench stdout by line, prepend controller name, test mode
            Write results to file
        end for
    end for
end for
Close results file
```

```
Usage: node dist/start.js [options]

Options:
  -h, --help           Show help                                   [boolean]
  -a, --append         Append results                              [boolean]
  -f, --file           File for results              [default: "results.csv"]
  -j, --just           Specify a single test to run                 [number]
  -l, --loops          Number of loops                 [number] [default: 37]
  -M, --mac-addresses  Unique source MAC addresses [number] [default: 100000]
  -m, --ms-per-test    Test length in ms             [number] [default: 1000]
  -s, --switches       Numbers of switches
                                [number] [default: [1,2,4,8,16,32,64,128]]
  -t, --targets        YAML test specification      [default: "targets.yaml"]
```

Figure 57. Test suite runner usage

7.4 Results and Analysis

7.4.1 Boxplots.

In order to visualize how the increasing number of simulated switches impacts each controller, the number of switch responses per test was summed across all switches and the results graphed. Cbench's throughput and latency modes are discussed separately.

7.4.1.1 Throughput Mode.

As discussed, Cbench in throughput mode sends as many `PacketIn` messages as will fit in its outgoing buffers and measures the total number of responses received to calculate the overall responses per second. The result is an indication of how well a controller responds when overloaded. Figure 58 shows boxplots for each of the controllers where the total number of responses for each run is plotted on the y axis against the number of simulated switches on the x axis.

An immediate observation is the dominance of Libfluid over the other controllers tested. After Libfluid, it appears that ONOS and NOX perform well, and it is difficult to make out differences beyond that. The high performance of Libfluid can be attributed to the fact that it can be considered more of a framework upon which to build a controller than a fully-fledged controller, and that it is written in C++, so there is very little overhead. Furthermore, its use of libevent (similar to libuv used by Node.js) may contribute to its high performance.

A problem with this view is that responses across switches are cumulative so the controller's ability to scale with the number of switches artificially inflates its score on the plot. To gain further insight, a new column of data is created from the quotient of the number of responses by the number of switches, the result of which is shown in Figure 59. This plot illustrates that there is a decrease in performance to each

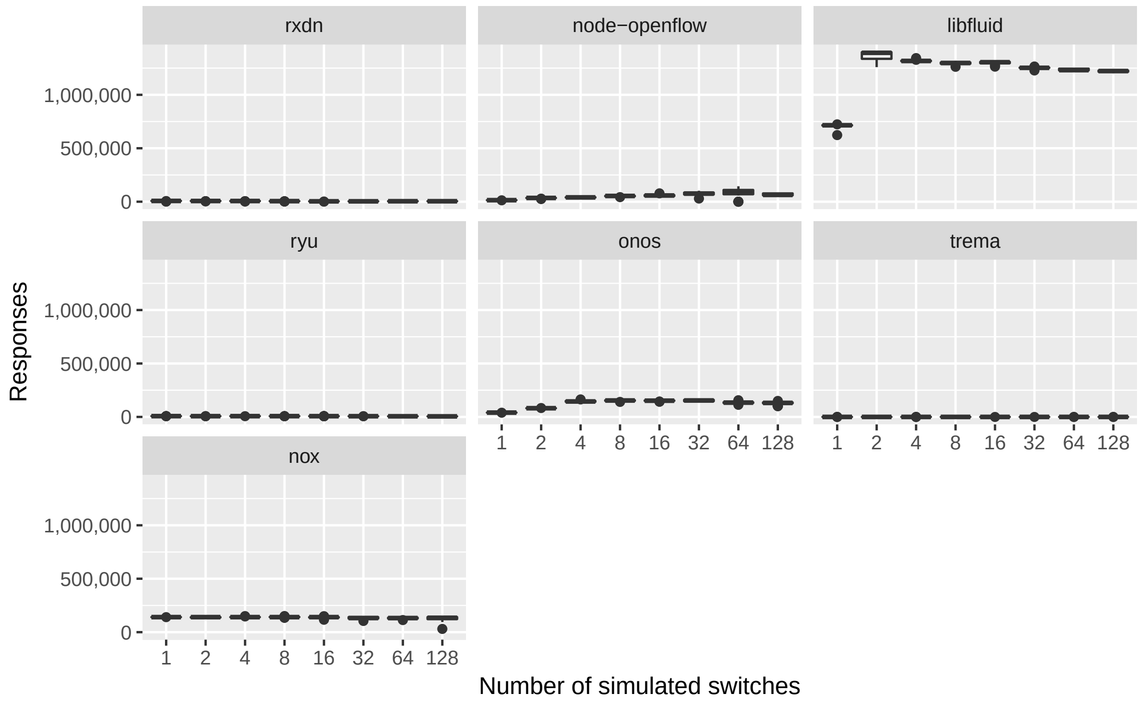


Figure 58. Total number of responses by controller, switches for Cbench throughput mode

individual switch as more switches are added, however the performance of the Libfluid controller still dwarfs the others.

Removing Libfluid and NOX from Figure 59 yields Figure 60. This shows ONOS as the best performer of the subset, and an unexpected per-switch performance improvement at four switches over one and two. Next in this set is node-openflow, which has visibly higher variability at higher numbers of switches.

The three slower controllers are not distinguishable, so it is worth comparing just those three in Figure 61. This figure shows Trema clearly at the bottom, with rxdn and Ryu at similar performance. However, Ryu has visibly lower variability in its performance over rxdn.

Overall, these boxplots demonstrate that for the throughput mode of Cbench, Libfluid is clearly the top performer, followed by NOX, then ONOS, then node-openflow. Ryu appears to narrowly beat rxdn, and at the bottom is Trema.

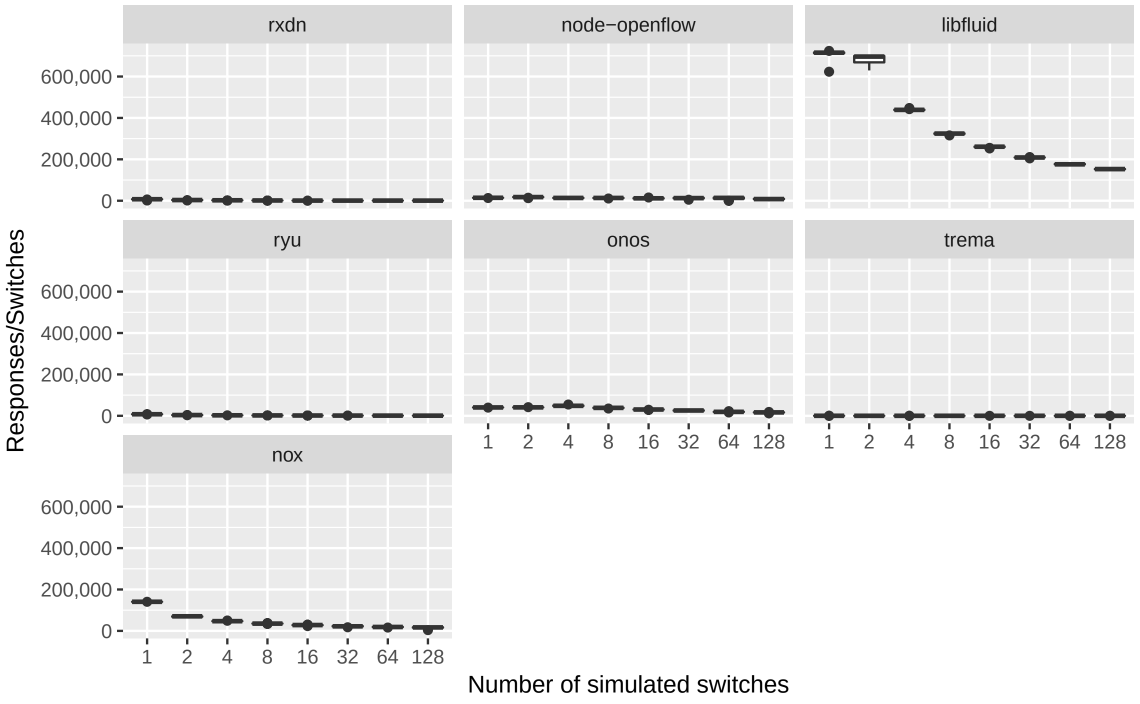


Figure 59. Responses/Switches by controller, switches for Cbench throughput mode

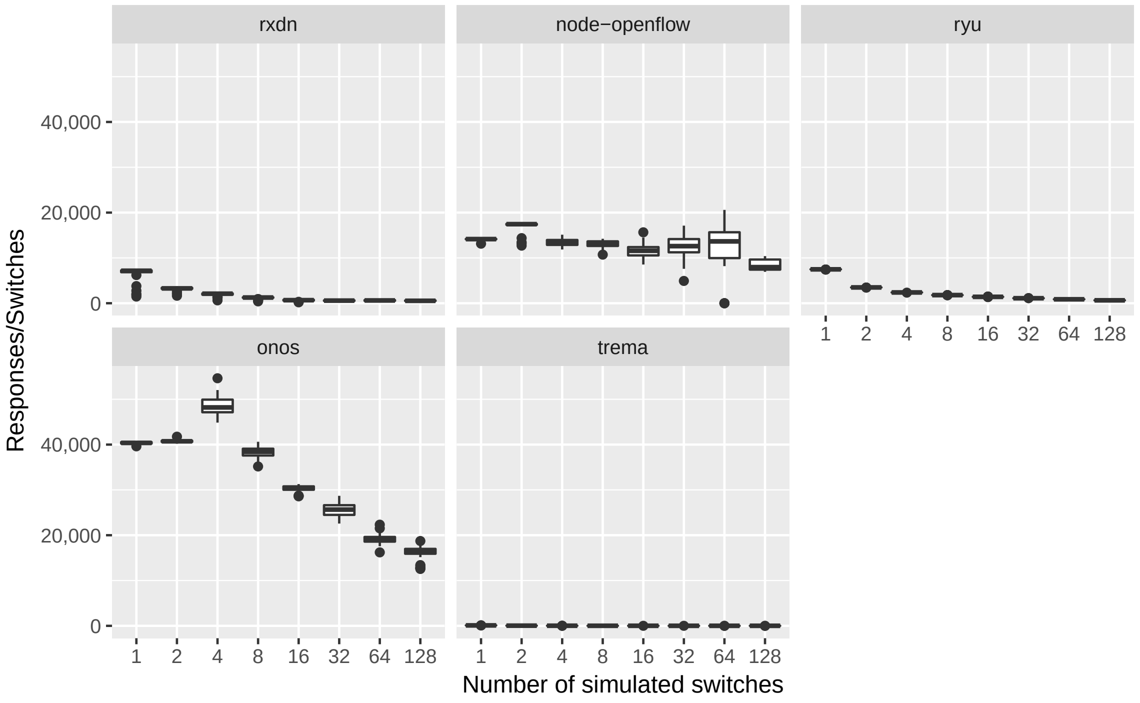


Figure 60. Responses/Switches by controller, switches for Cbench throughput mode, sans Libfluid and NOX

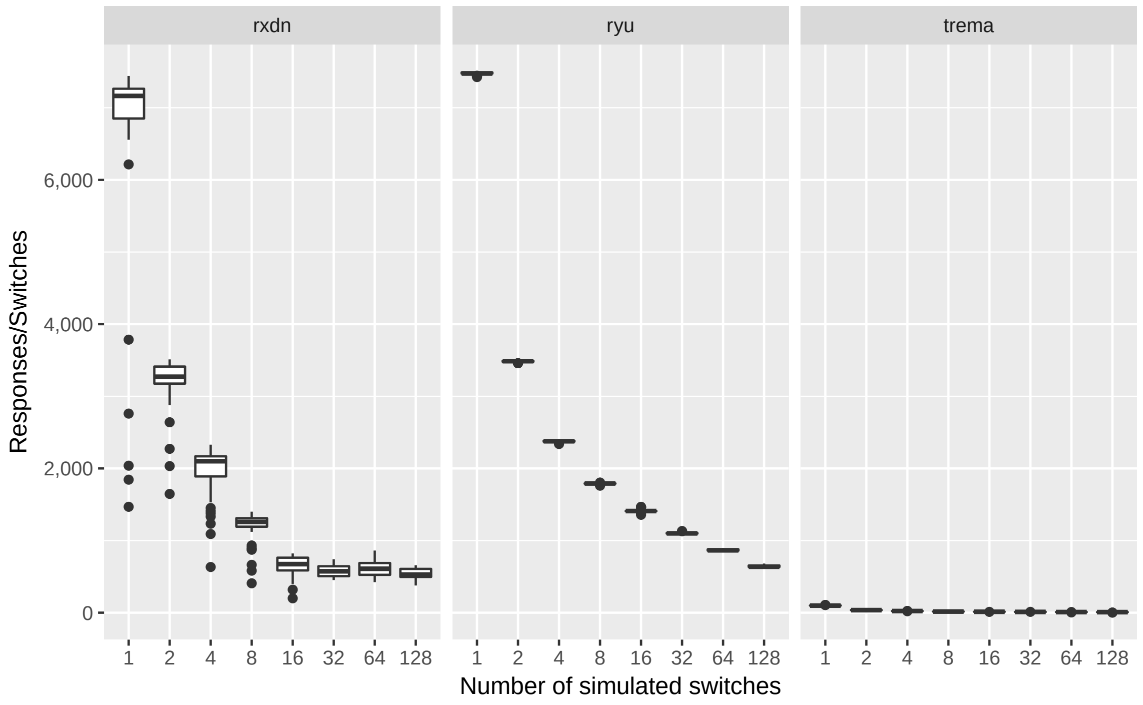


Figure 61. Responses/Switches by controller, switches for Cbench throughput mode, slow 3

7.4.1.2 Latency Mode.

Again, Cbench in latency mode waits between each `PacketIn` for the controller to respond with a `PacketOut` or `FlowMod`, and calculates the number of responses per second it receives from the controller. As opposed to throughput mode, the result of this test is an indication of how quickly the controller responds when it is not overloaded. Figure 62 shows boxplots for each of the controllers where the total number of responses for each run is plotted on the y axis against the number of simulated switches on the x axis.

Again, the total responses is divided by the number of switches in order to show how the controller is able to serve each switch as the total number of switches increases. This is shown for all controllers in Figure 63.

Figure 63 gives an indication that Libfluid again sets the high-water mark, but it is difficult to determine ranking within the other high three (node-openflow, ONOS,

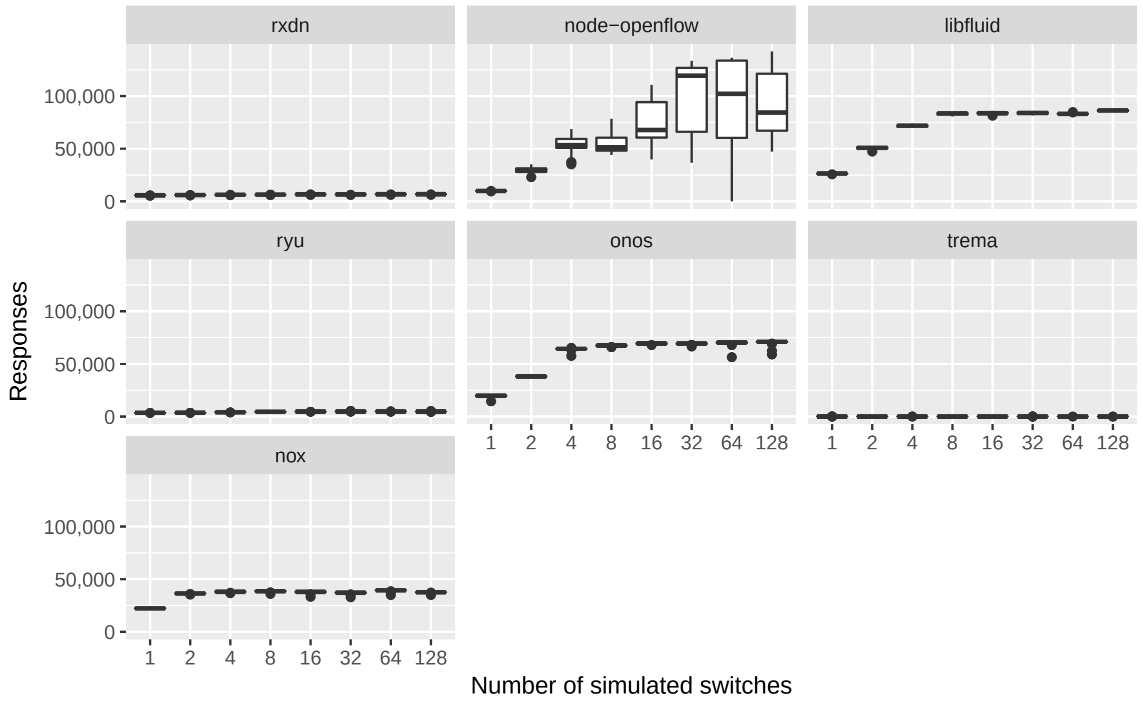


Figure 62. Total number of responses by controller, switches for Cbench latency mode

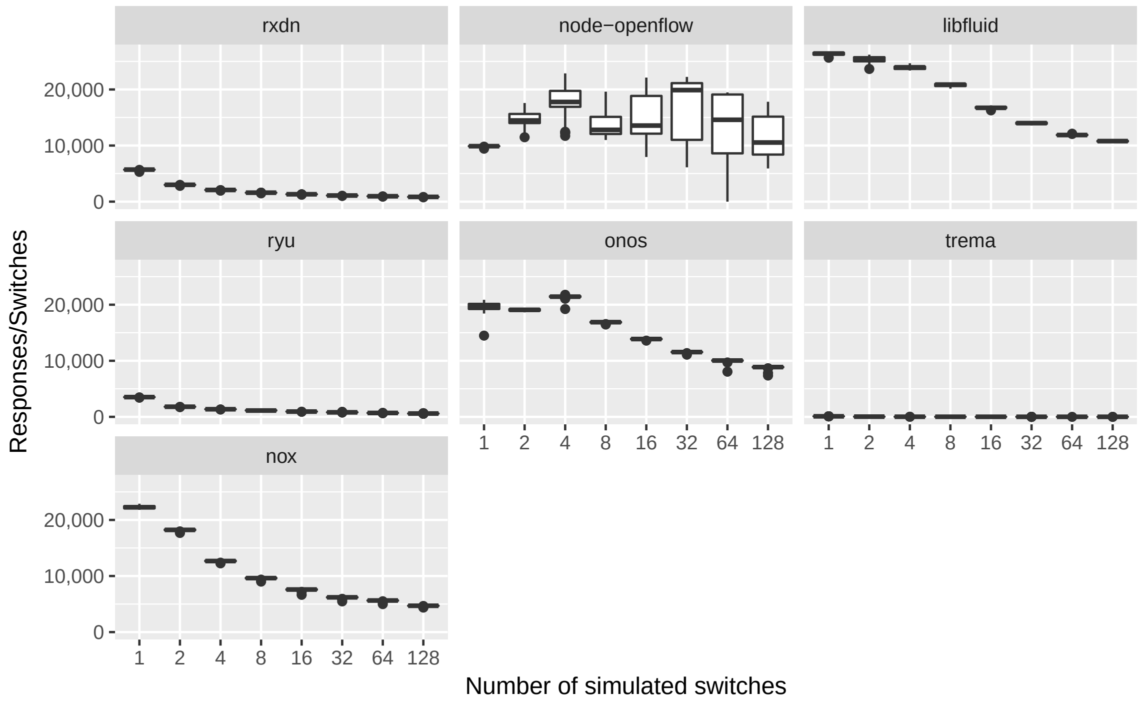


Figure 63. Responses/Switches by controller, switches for Cbench latency mode

NOX) and low three (rxdn, Ryu, Trema), so these are shown separately in Figures 64 and 65, respectively.

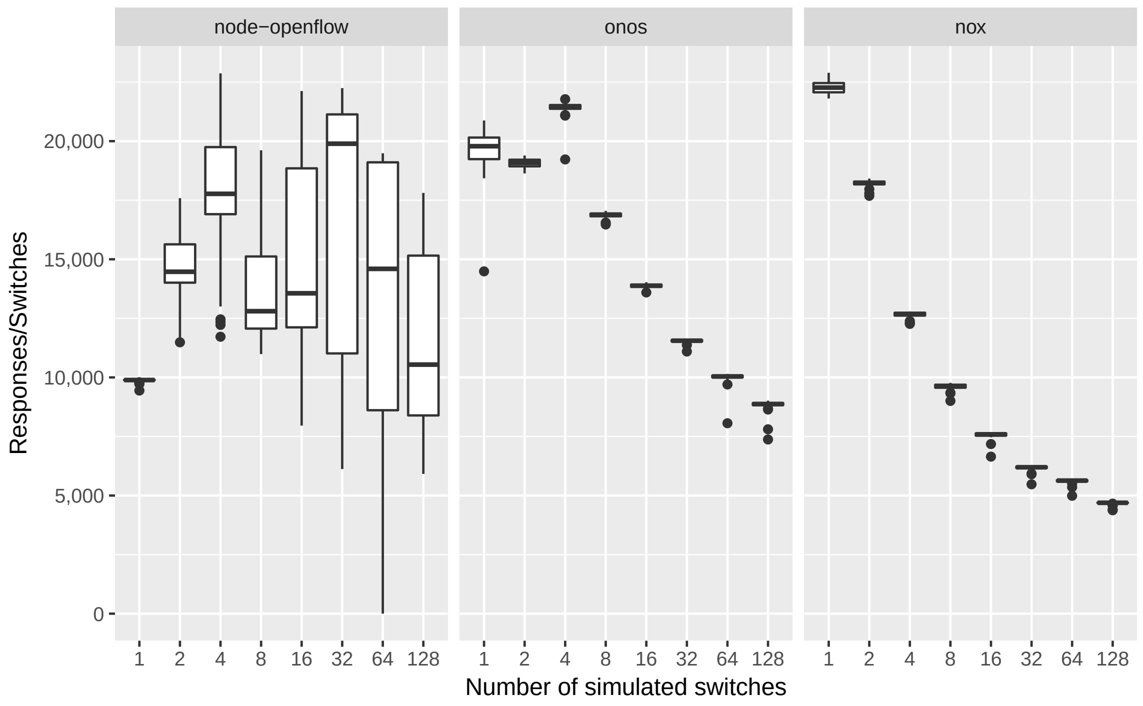


Figure 64. Responses/Switches by controller, switches for Cbench latency mode

Figure 64 shows a very high degree of variance for node-openflow relative to the other two controllers in this range. It also shows an unexpectedly low number of responses for a single switch compared to when more switches are simulated. While the median at more than 16 switches is much higher for node-openflow, the interquartile range is extremely broad at these values, indicating a very inconsistent level of performance from request to request.

Figure 65 shows rxdn with a performance profile very similar to that of Ryu, but visibly higher. Trema is again at the bottom.

More resolution is available on boxplots of individual controller performance. These are available for each controller and Cbench mode in Appendix B.

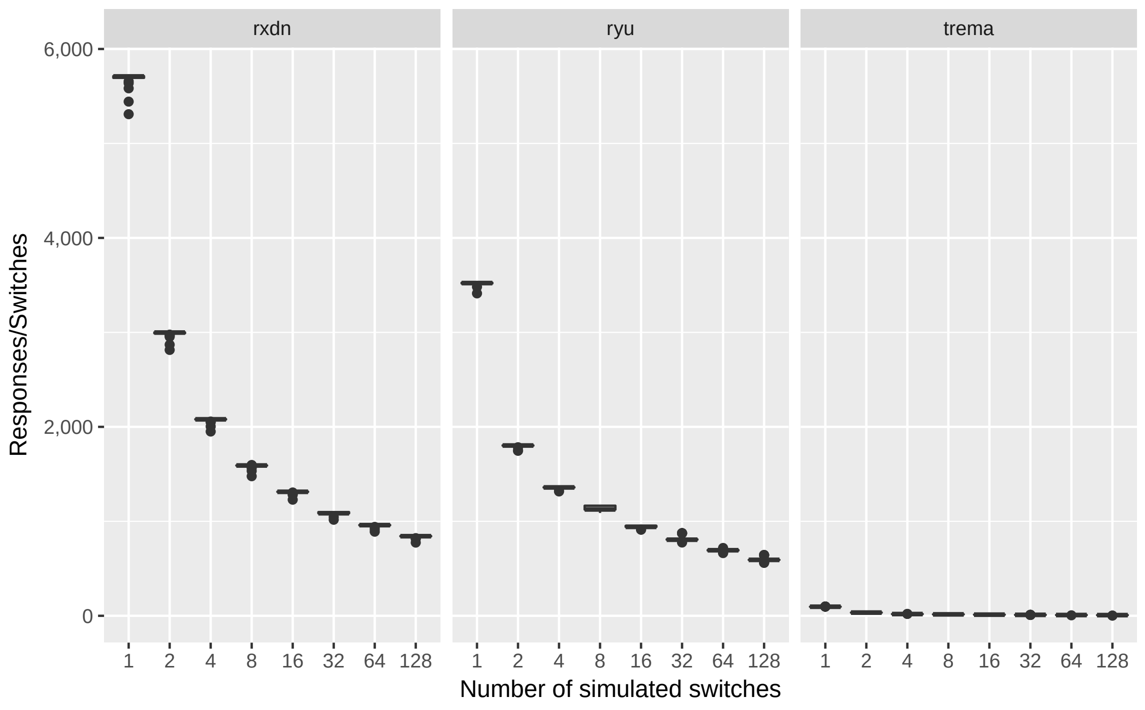


Figure 65. Responses/Switches by controller, switches for Cbench latency mode

7.4.2 Statistical Analysis.

The tests were run with a varying number of switches, as shown above. However for statistical analysis, a single, representative number of switches was chosen, which is the Cbench default, 16. This leaves the controllers as the single factor, and an appropriate tool for such comparison of means is a one-way, balanced ANOVA. All controllers at 16 switches are shown in Figures 66 and 67. The summary of the data for latency mode is given in Table 6 and for throughput in Table 7.

Before using a balanced, one-way ANOVA, the assumptions of the test should be checked. These include the following:

1. the same number of replications between groups (balanced), and
2. like variance between groups (homoscedasticity).

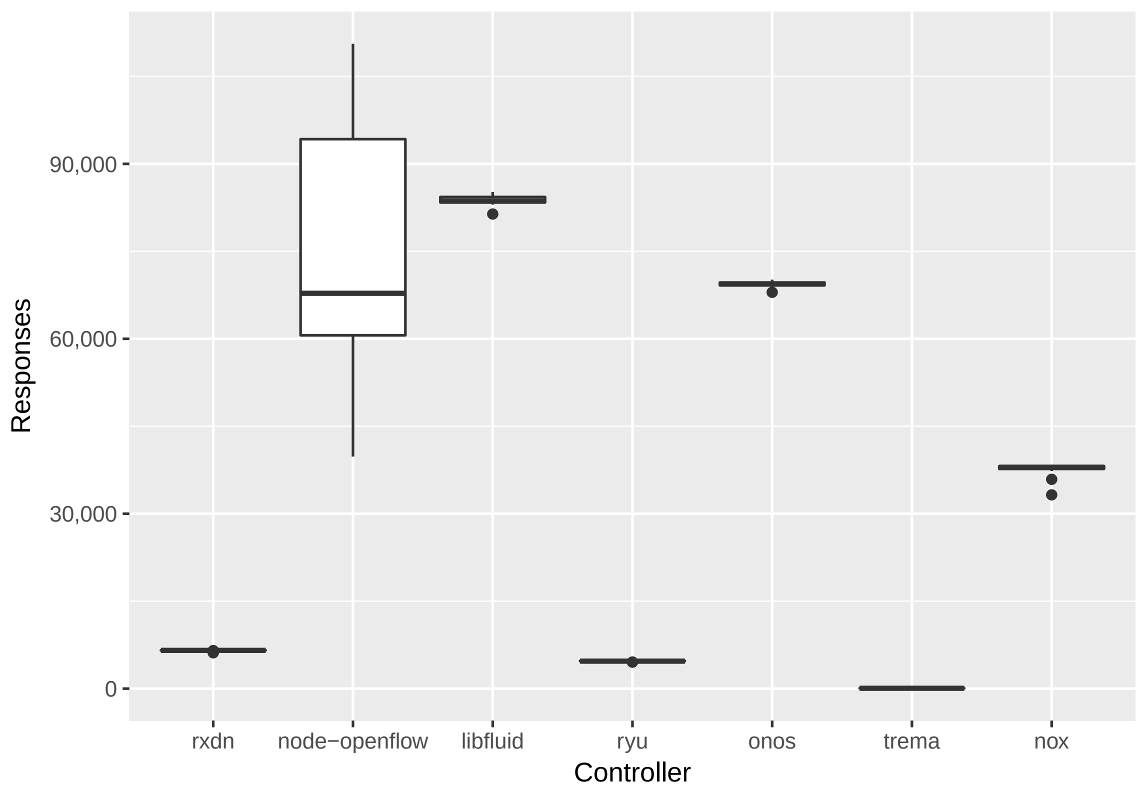


Figure 66. Responses by controller at 16 switches, latency mode

Table 6. Controller responses for 16 switches in latency mode

Controller	**Mean**	**Median**	**Std. dev.**	**Replications**
rxdn	6543.22222	6562.0	77.05920	36
node-openflow	74455.08333	67802.5	20076.46781	36
libfluid	83798.05556	83638.5	745.45039	36
ryu	4730.58333	4707.5	64.20787	36
onos	69365.30556	69394.5	432.76822	36
trema	61.16667	61.0	2.00713	36
nox	37779.19444	37954.5	890.92706	36

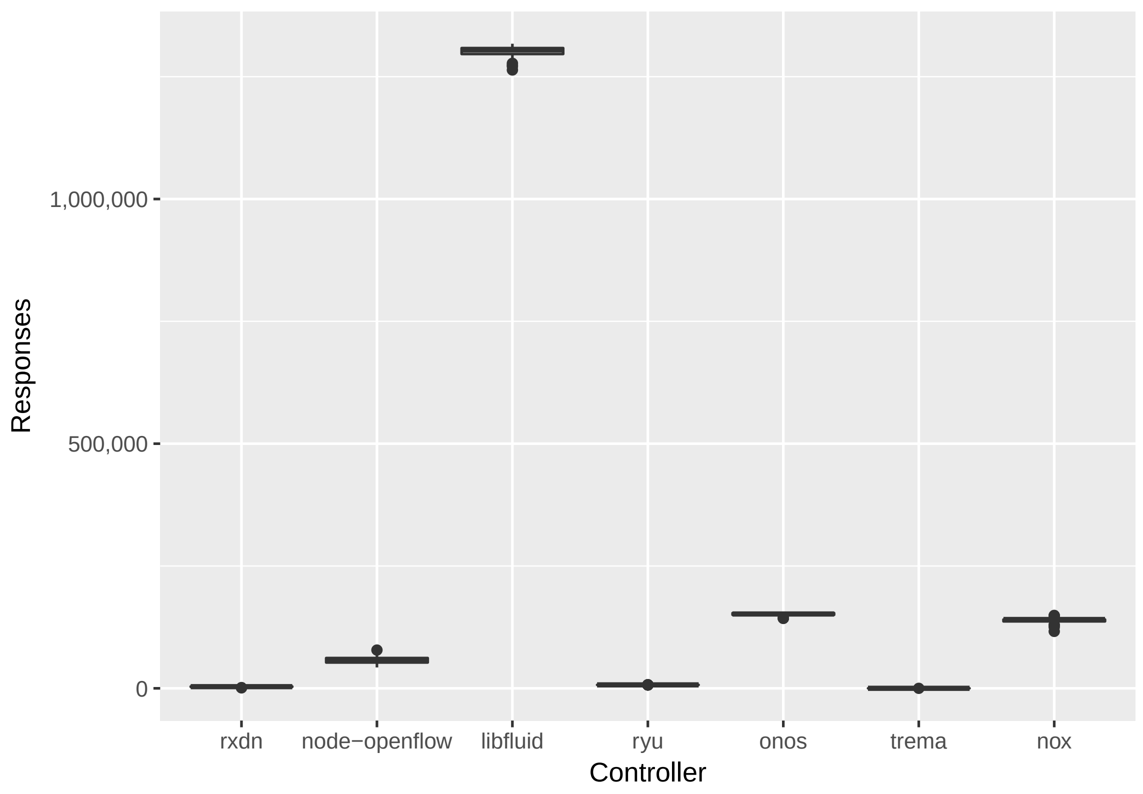


Figure 67. Responses by controller at 16 switches, throughput mode

Table 7. Controller responses for 16 switches in throughput mode

Controller	Mean	Median	Std. dev.	Replications
rxdn	3274.94444	3364.5	695.683300	36
node-openflow	57823.08333	57858.5	8303.721181	36
libfluid	1300059.72222	1304894.0	12907.629707	36
ryu	7052.22222	7043.0	108.091525	36
onos	151178.66667	151932.0	3452.642524	36
trema	63.72222	63.5	2.679138	36
nox	139250.05556	140647.0	6481.411870	36

The test is a balanced test because the number of replications across groups (controllers) is the same (as calculated in Section 7.3.4, $n = 36$). To test the second assumption, Levene's test can be used [37]. If Levene's test is significant (i.e., $p \leq 0.05$) then the assumption of homogeneity of variances is violated. For both latency and throughput results the variances were significantly different, with $F(6, 245) = 60.545, p \ll 0.01$ for latency and $F(6, 245) = 16.926, p \ll 0.01$ for throughput. As a result, robust (nonparametric) methods must be used.

Welch's F is a robust test of multiple means that adjusts for heteroscedasticity. If the test is significant ($p \leq 0.05$), then it can be concluded that means differ significantly between factors [37]. The test was run for the latency mode test where the number of switches was 16, and the result was significant $F(6, 93.384) = 314540, p \ll 0.001$. While this result indicates significant differences between the controllers, it does not give insight into which controllers differ.

While it was expected that means would differ between controllers, it was not known which would outperform others. Therefore, specific hypotheses were not made. In such instances, *post hoc* tests can be used to conservatively make conclusions on collected data. Wilcox describes a robust *post hoc* test for comparison of multiple means that is coded in the R programming environment package WRS2.

The results for 16 switches, latency mode are given in Figure 68 and for throughput mode in Figure 69. In the function's output, the p-value is not adjusted to control for familywise error (FWE), therefore significance is instead represented by a confidence interval that does not include zero. With this criteria, all but two of the latency mode comparisons are significant: node-openflow vs. libfluid and node-openflow vs. onos. For throughput mode, all comparisons are significant. These distinctions help clarify interpretations of the boxplots above; for latency, it can be seen that the node-openflow interquartile range encompasses the entire range of both libfluid and onos, making the comparison non-significant. For all others, including throughput, the variance of each is small enough that the differences are significant.

```
> library(WRS2)
> lincon(responses ~ controller, latency)

Call:
lincon(formula = responses ~ controller, data = latency)

                              psihat   ci.lower   ci.upper p.value
rxdn vs. node-openflow     -61919.773 -76525.203 -47314.343 0.00000
rxdn vs. libfluid          -71228.682 -71650.798 -70806.566 0.00000
rxdn vs. ryu                 1720.636   1679.730   1761.543 0.00000
rxdn vs. onos              -57674.909 -57870.465 -57479.353 0.00000
rxdn vs. nox                 6095.773   6088.569   6102.976 0.00000
rxdn vs. trema             -29426.773 -29627.017 -29226.529 0.00000
node-openflow vs. libfluid  -9308.909 -23917.569   5299.751 0.04194
node-openflow vs. ryu       63640.409  49034.950  78245.868 0.00000
node-openflow vs. onos       4244.864 -10361.258  18850.985 0.33436
node-openflow vs. nox       68015.545  53410.116  82620.975 0.00000
node-openflow vs. trema     32493.000  17886.845  47099.155 0.00000
libfluid vs. ryu            72949.318  72526.194  73372.442 0.00000
libfluid vs. onos           13553.773  13103.237  14004.309 0.00000
libfluid vs. nox            77324.455  76902.370  77746.540 0.00000
libfluid vs. trema          41801.909  41349.867  42253.951 0.00000
ryu vs. onos               -59395.545 -59593.337 -59197.754 0.00000
ryu vs. nox                  4375.136   4334.563   4415.709 0.00000
ryu vs. trema              -31147.409 -31349.833 -30944.985 0.00000
onos vs. nox                63770.682  63575.194  63966.170 0.00000
onos vs. trema              28248.136  27983.851  28512.421 0.00000
nox vs. trema              -35522.545 -35722.723 -35322.368 0.00000
```

Figure 68. Wilcox robust comparison of multiple means applied to latency mode tests at 16 switches

A detailed walkthrough of all analysis done in the R programming language is included in Appendix E.

```
> lincon(responses ~ controller, throughput16)

Call:
lincon(formula = responses ~ controller, data = throughput16)

                                  psihat     ci.lower     ci.upper p.value
rxdn vs. node-openflow       -53987.545   -58515.752   -49459.339       0
rxdn vs. libfluid          -1300140.455 -1306513.115 -1293767.794       0
rxdn vs. ryu                  -3636.682    -3991.006    -3282.358       0
rxdn vs. onos               -148339.909  -150260.403  -146419.416       0
rxdn vs. nox                   3342.136     2987.812     3696.461       0
rxdn vs. trema              -136379.318  -138713.567  -134045.069       0
node-openflow vs. libfluid -1246152.909 -1253575.547 -1238730.271       0
node-openflow vs. ryu         50350.864    45830.047    54871.681       0
node-openflow vs. onos       -94352.364   -99120.454   -89584.273       0
node-openflow vs. nox         57329.682    52808.865    61850.499       0
node-openflow vs. trema      -82391.773   -87290.674   -77492.871       0
libfluid vs. ryu            1296503.773  1290136.345  1302871.201       0
libfluid vs. onos           1151800.545  1145269.770  1158331.321       0
libfluid vs. nox            1303482.591  1297115.163  1309850.019       0
libfluid vs. trema          1163761.136  1157139.955  1170382.318       0
ryu vs. onos                -144703.227  -146605.761  -142800.694       0
ryu vs. nox                    6978.818     6976.978     6980.659       0
ryu vs. trema               -132742.636  -135062.282  -130422.991       0
onos vs. nox                 151682.045   149779.512   153584.579       0
onos vs. trema                11960.591     9121.125    14800.057       0
nox vs. trema               -139721.455  -142041.100  -137401.809       0
```

Figure 69. Wilcox robust comparison of multiple means applied to throughput mode tests at 16 switches

7.5 Summary and Future work

Leaving out node-openflow and rxdn, the highest-performing controllers (Libfluid, ONOS, NOX) are those compiled (C++) or JIT-compiled (Java) while the lowest-performing (Trema, Ryu) are those interpreted (Ruby, Python). (The list of tested controllers and their corresponding languages and platforms are shown in Table 4).

The results show that node-openflow, while approaching the performance of compiled controllers, also exhibits much more variance in its performance. This may be a characteristic of the Node.js platform (e.g., due to its garbage collector or the quality of its code) or the quality of the node-openflow library itself. While node-openflow was heavily modified to exhibit correct behavior and high performance, newly released features for performance profiling of Node.js applications may help focus on areas of the code that would benefit from further optimization. Updating the code base may also increase performance, as newly released versions of Node.js purportedly contain large improvements to the performance of the read and write methods of the Buffer class, which node-openflow (and therefore also rxdn) make extensive use of.

Additionally, rxdn shows a large potential for performance improvement. While it exceeds Ryu's performance by at least 33% with statistical significance in latency mode with 16 simulated switches, it is slower in throughput mode and slower in either mode than the library node-openflow or professional controllers (e.g., Libfluid, ONOS, and NOX). Since rxdn is built directly on the node-openflow library, that library's performance should be considered the upper-bound of rxdn's possible future performance. The primary difference in implementation between the two is that the library uses Node.js transform streams while the framework uses RxJS Observables. While some other implementations of Reactive Extensions do contain methods for backpressure handling (e.g., RxJava), RxJS currently does not. This has been identified as a future area for improvement in a developer's discussion in the RxJS source code repository [86, 87].

Besides the library and framework, the test suite itself could be extended to support distributed testing, where the controller resides on a separate physical computer from the Cbench program that is testing it. This would improve the realism of the data and potentially improve its accuracy, as the benchmarking program and controller would no longer be competing for the same system resources.

VIII. Conclusions

8.1 The State of SDN

Despite promises of revolutionary networks sweeping the industry, SDN development has largely plateaued. Network hardware is very expensive, so practitioners are resistant to upgrades unless they can see an objective gain for the money. In most cases, it is not possible to estimate the gains of SDN without committing to all new hardware. The result is that SDN is largely constrained to the laboratory and specialized data center networks.

Once hardware is made available and software installed, programming a network to match with operational objectives and policies is difficult, and in many cases, uncharted territory. While some proprietary solutions exist, these are not easily modified. Open-source solutions lack basic functionality or the performance that would be required for an enterprise network environment.

Barriers across hardware and software have slowed network innovation.

8.2 Research Conclusions

The results and final deductions of this work are broken across the three domains:

1. Hardware: From designing and measuring the performance of the shim, it was found that such a device is feasible, in that it will properly forward traffic according to SDN policy for at least one host at a time.

2. Controller: It was demonstrated that a controller designed with an event loop in an interpreted language can perform at or near the level of a traditional multithreaded, compiled controller.

3. Application: A new framework for creating modular network control applications was developed. The framework can be used to combine functionality of different applications in a way that is intuitive and with reasonable performance.

8.3 Research Contributions

The following contributions were made:

- Hardware: This research has shown that it is possible to create inexpensive, incremental steps to allow older networks to adopt newer SDN protocols.
- Controller: This research has contributed a working OpenFlow library that can be extended and used for research and experimentation. Its high performance makes it suitable for simulations and experiments involving hundreds of switches and thousands of hosts.
- Application: This research has contributed a new framework for network application development that shows it is possible to easily to create modular control applications.

8.4 Limitations of this Research

The limitations are broken down by chapter:

- Shim (Chapter III): The shim created does not implement any of the OpenFlow protocol, but only makes the switch behave as if it were preconfigured by a controller to forward traffic. It proves the concept but does not actually provide the functionality. Adding this functionality, which is called an OpenFlow agent, would require a significant amount of effort.

- Schema (Chapter IV): The schema only implements OpenFlow versions 1.0 and 1.3 (the latest publicly-available version is 1.5). Furthermore, not every message is completely implemented.

- Library (Chapter V): The library also only implements OpenFlow versions 1.0 and 1.3. It is written for Node.js, which is a relatively immature runtime undergoing significant changes from release to release. Also, JavaScript is a language with serious historical baggage, and even when fortified with TypeScript, takes some time to master its idiosyncracies.

- Framework (Chapter VI): The framework also only implements OpenFlow versions 1.0 and 1.3. It relies on Rxjs, which is under heavy development and has not yet been optimized for performance.

- Benchmark (Chapter VII): The benchmarking program, Cbench runner, is highly dependent upon Cbench, which only benchmarks OpenFlow version 1.0. Furthermore, there is not currently a way to have the test run on different computers (e.g., with the controller on one machine and Cbench on another). This reduces the realism of the measurement, as the CPU, RAM, etc. are constrained resources between the benchmarking program and the program being measured. Also, the Docker container platform is constantly changing, with API changes at every minor version. This adds a burden of updating the test program anytime the underlying Docker installation is updated.

8.5 Recommendations for Future Work

Each of the contributions, as stated, could be extended or reworked to provide new conclusions and insights.

- Shim
  - The shim could be extended to support 10 Gbps and use this type port for a trunk in order to lessen the potential bottleneck of traffic.
  - The shim could similarly be extended to implement handoff between output ports on the shim to improve contention.
  - An OpenFlow agent could be implemented to make it a fully-functional shim, able to connect to a controller and make the traditional switch appear as an OpenFlow switch.
- OpenFlow Schema
  - The schema could be extended to support more and newer versions of the OpenFlow protocol.
  - It may be possible to generalize the schema in such a way that it includes encoding and decoding information directly in the schema. This would allow a library to implement new versions just by referencing the updated schema.
- node-openflow Library
  - Like the schema, the node-openflow library could be extended to more OpenFlow versions.
  - The library could be refactored to reduce redundancy in commonly-used structures. This would also help simplify the addition of new protocols.
  - If the schema is updated to include encoding and decoding information, the library could be updated to utilize this to be able to encode and decode messages in a generalized way, greatly reducing the code size and potential for errors.

- rxdn Framework
  - The framework could be analyzed for further performance optimizations that would put it on the level of the library. This may require changing the dependency on Rxjs to another reactive programming library, or making recommendations for changes to the Rxjs project.
  - As the framework exists, it provides an interesting platform to experiment with controller distribution. It already separates logic and state in such a way that a distributed design is possible.
- Cbench runner test suite
  - Beyond highlighting areas for improvement of the library and framework, the test suite itself could be modified to allow for testing across separate physical computers, i.e., to have the Cbench and controller on different systems.

8.6 Concluding Thoughts

The SDN architecture holds great promise for fueling computer network innovation for years to come, however there are still barriers to its use even in research environments. This work has tried to target and mitigate a few of the major hurdles facing adoption, use, and research in SDN. Lowering the barrier of entry to SDN for students, researchers, and practitioners can allow for rapid experimentation and development of new network designs that may be faster, more secure, and more easily maintainable than the systems we have today.

Appendix A. Shim Throughput and Latency Observations

Table 8. Throughput Observations (Mbps)

#	Switch	Switch & Shim
1	957	959
2	944	923
3	946	933
4	939	944
5	943	933
6	947	933
7	939	933
8	938	933
9	940	933
10	947	933
11	957	959
12	946	933
13	944	933
14	941	933
15	941	933
16	940	944
17	939	933
18	940	933
19	944	933
20	942	923

Table 9. Latency Observations (ns)

#	Switch	Switch & Shim
1	3952	7592
2	3880	7512
3	3960	7536
4	3888	7552
5	3840	7600
6	3944	7528
7	3864	7552
8	3872	7528
9	3848	7512
10	3856	7592
11	3920	7624
12	3848	7576
13	3952	7520
14	3904	7584
15	3856	7592
16	3936	7528
17	3872	7552
18	3864	7560
19	3896	7584
20	3856	7616
21	3944	7608
22	3872	7616
23	3952	7608
24	3912	7576
25	3968	7528
26	3936	7640
27	3856	7528
28	3880	7632
29	3936	7536
30	3872	7560
31	3928	7544
32	3840	7552
33	3856	7560
34	3880	7576
35	3840	7632
36	3896	7576
37	3872	7592
38	3872	7536
39	3944	7528
40	3920	7536

Appendix B. Controller Performance Analysis Plots

Figures 70 through 83 are individual boxplots for each controller and Cbench mode (latency and throughput). These are discussed in aggregate in Chapter VII.

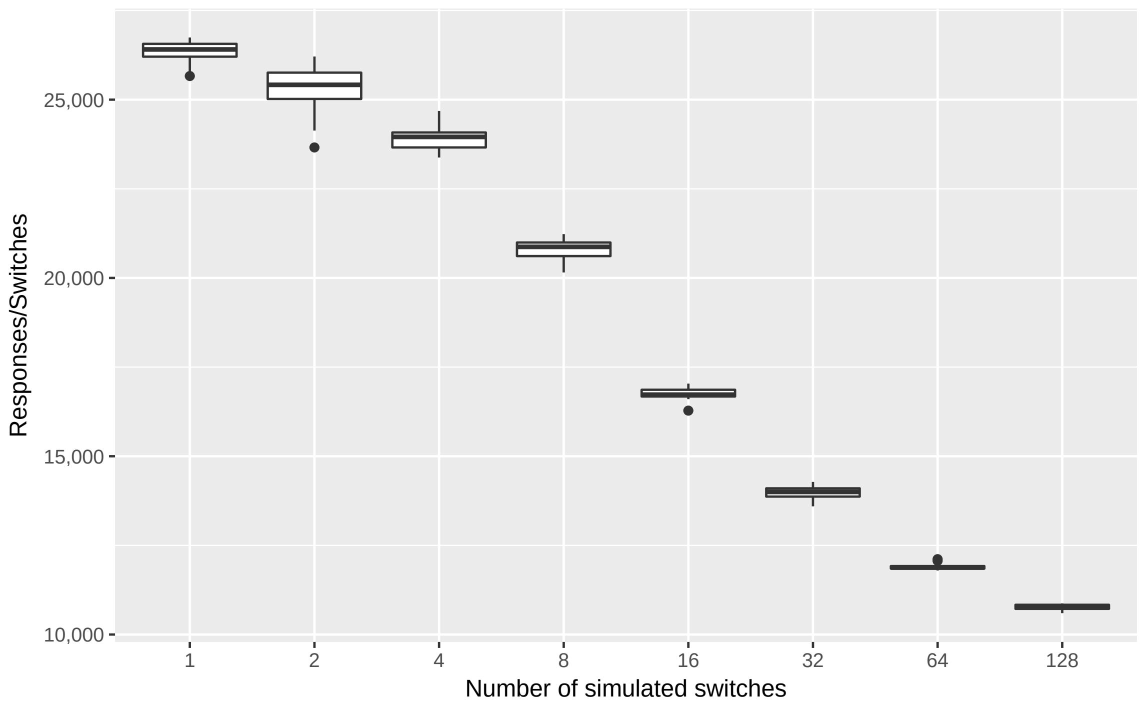


Figure 70. Libfluid Responses/Switches by switches for Cbench latency mode

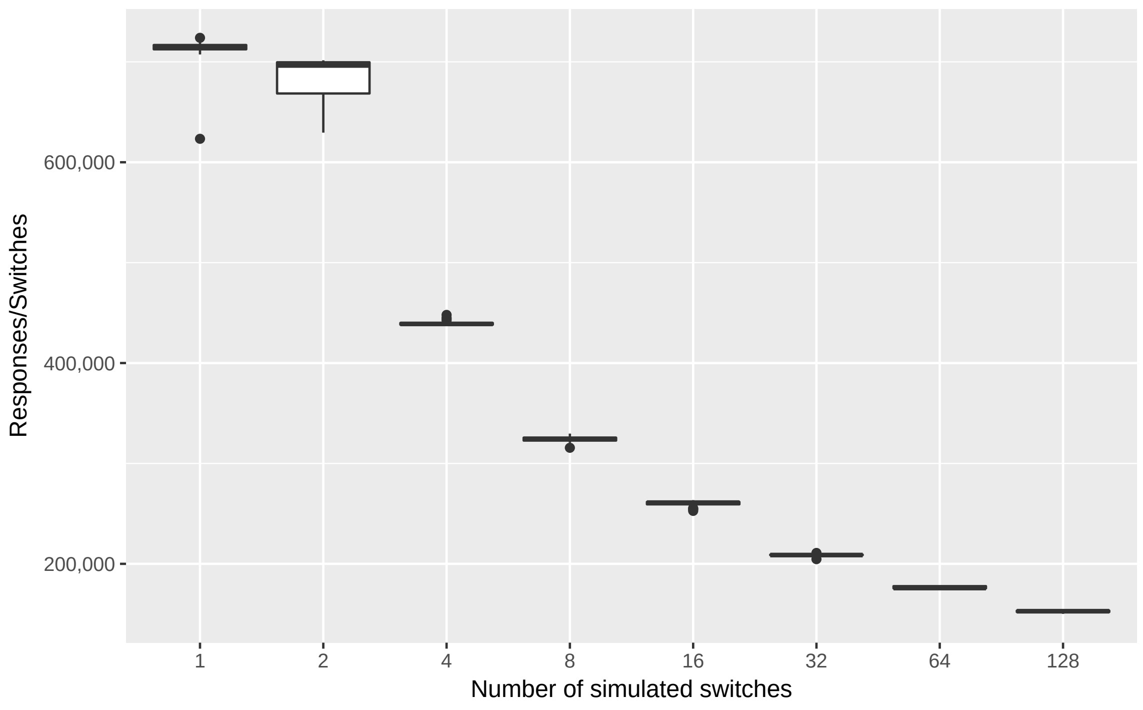


Figure 71. Libfluid Responses/Switches by switches for Cbench throughput mode

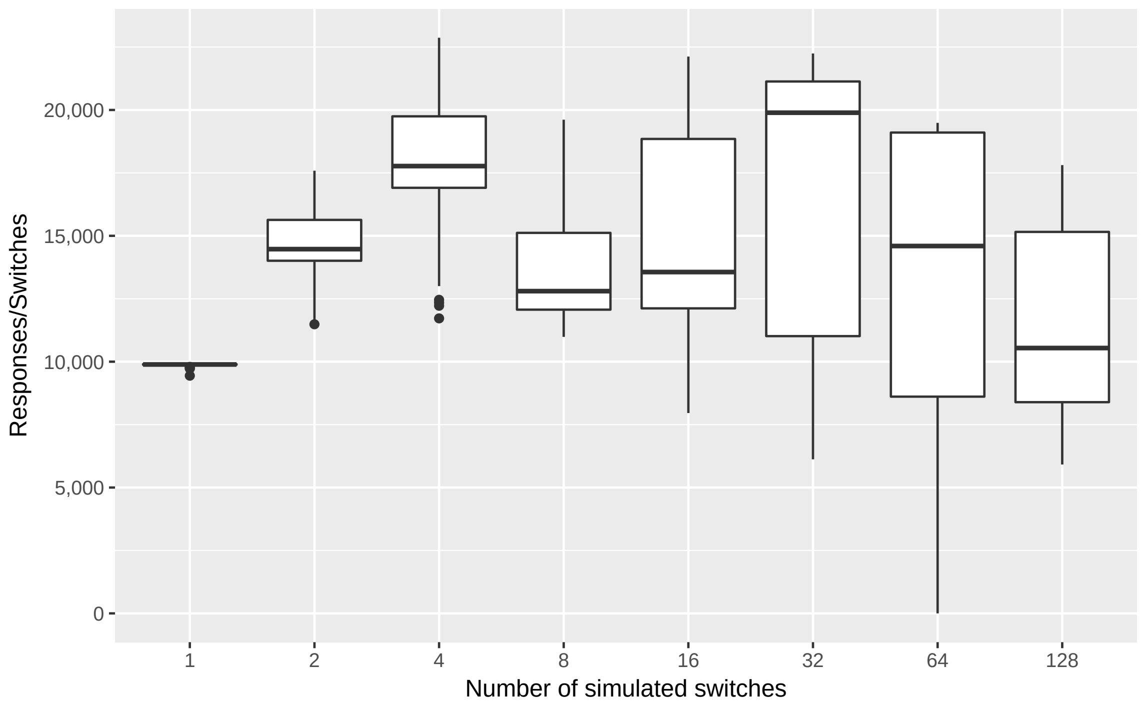


Figure 72. node-openflow Responses/Switches by switches for Cbench latency mode

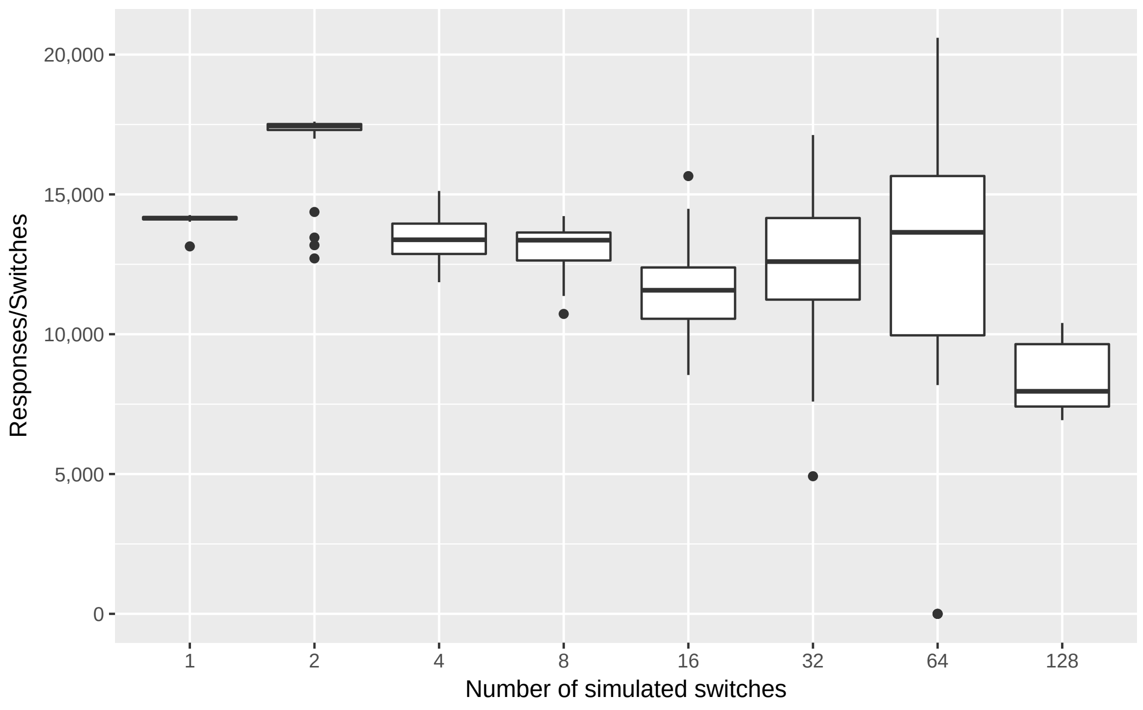


Figure 73. node-openflow Responses/Switches by switches for Cbench throughput mode

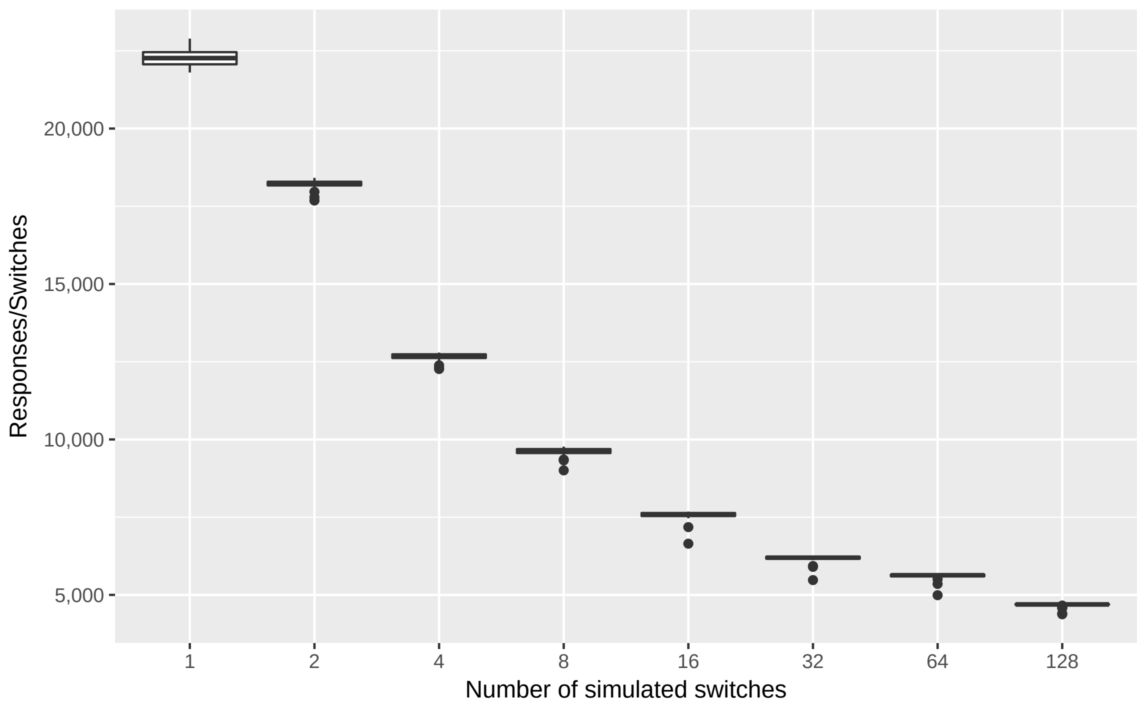


Figure 74. NOX Responses/Switches by switches for Cbench latency mode

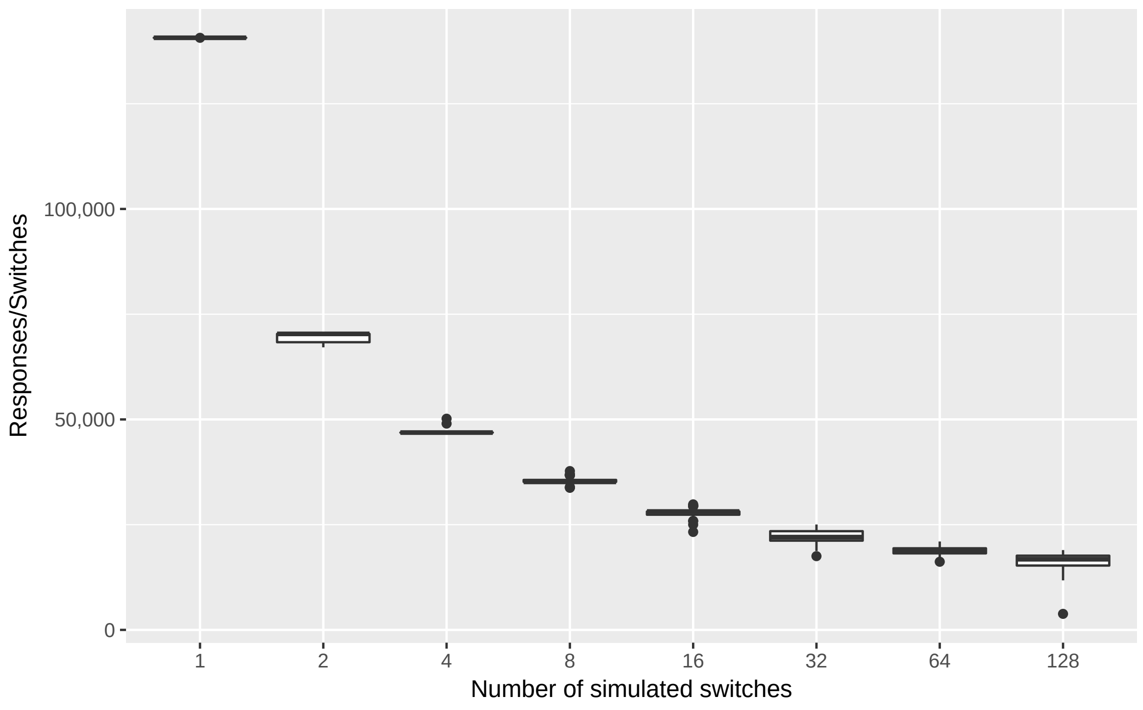


Figure 75. NOX Responses/Switches by switches for Cbench throughput mode

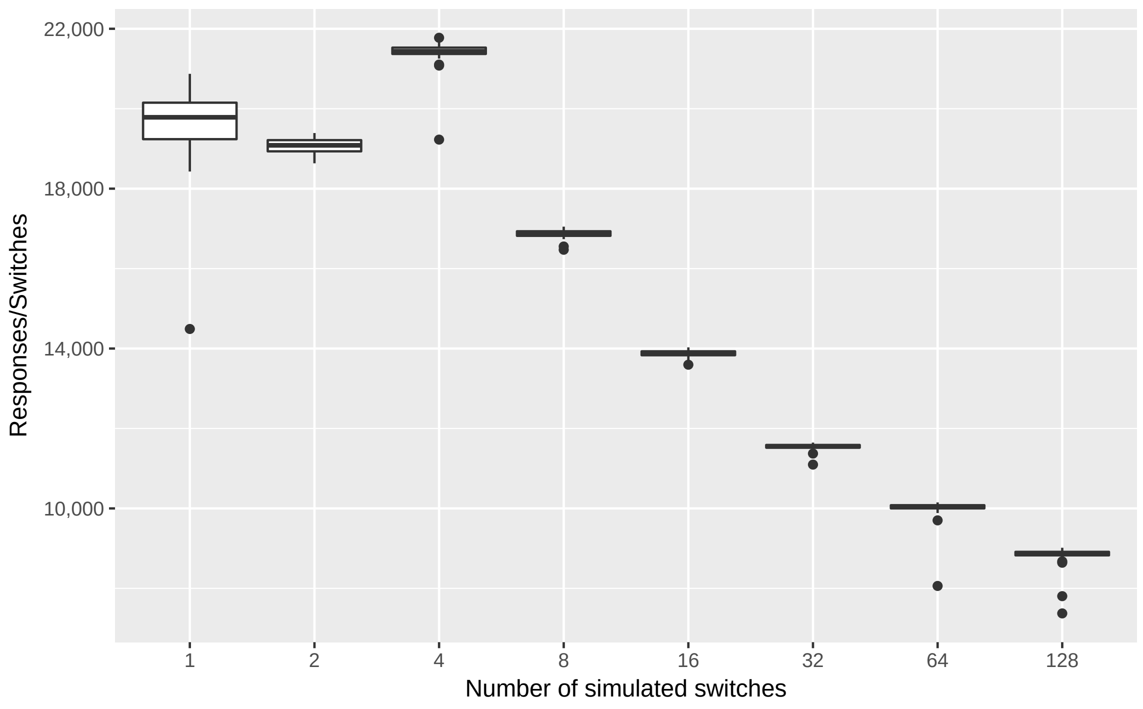


Figure 76. ONOS Responses/Switches by switches for Cbench latency mode

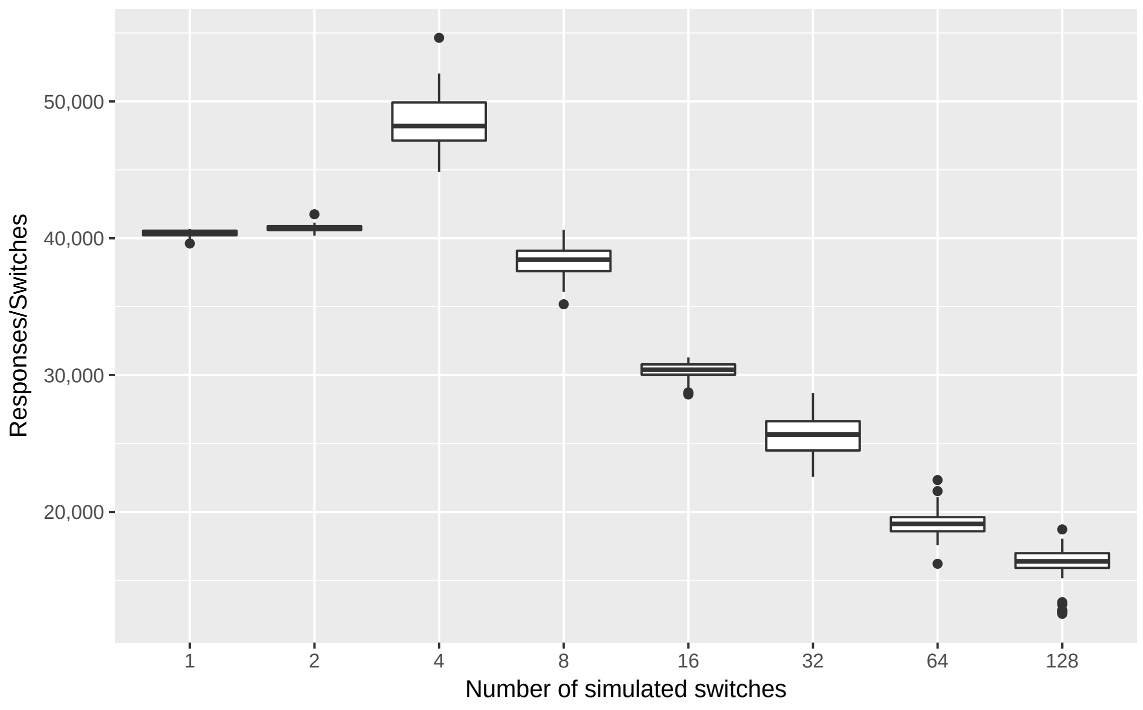


Figure 77. ONOS Responses/Switches by switches for Cbench throughput mode

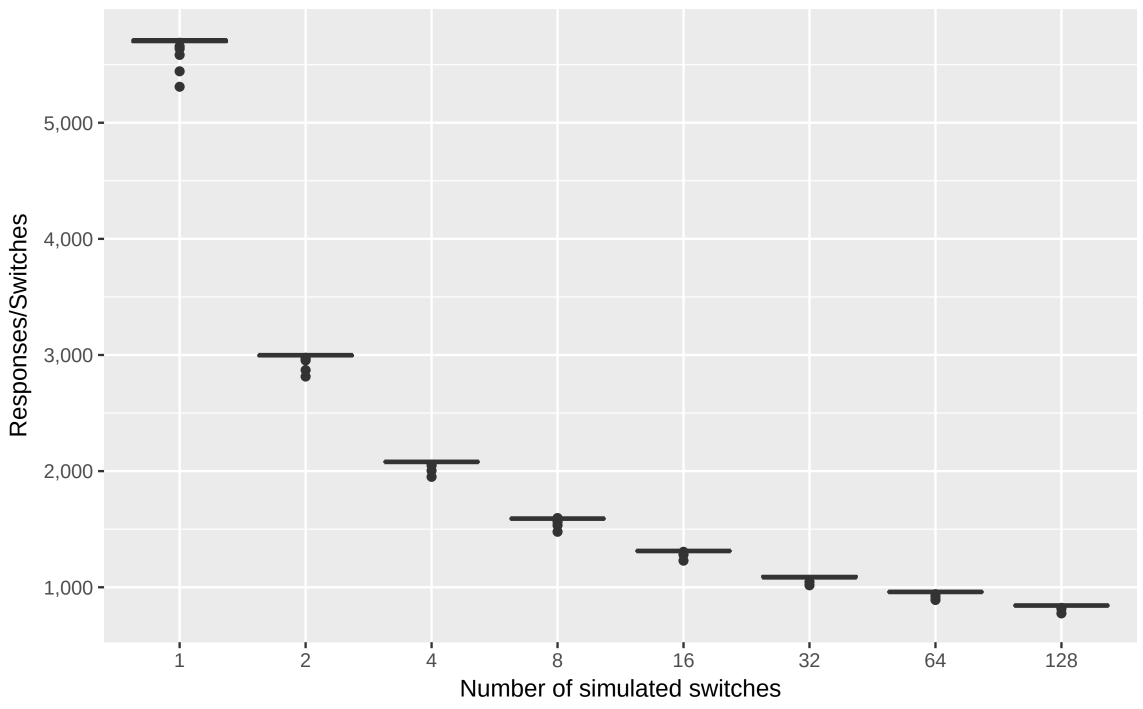


Figure 78. rxdn Responses/Switches by switches for Cbench latency mode

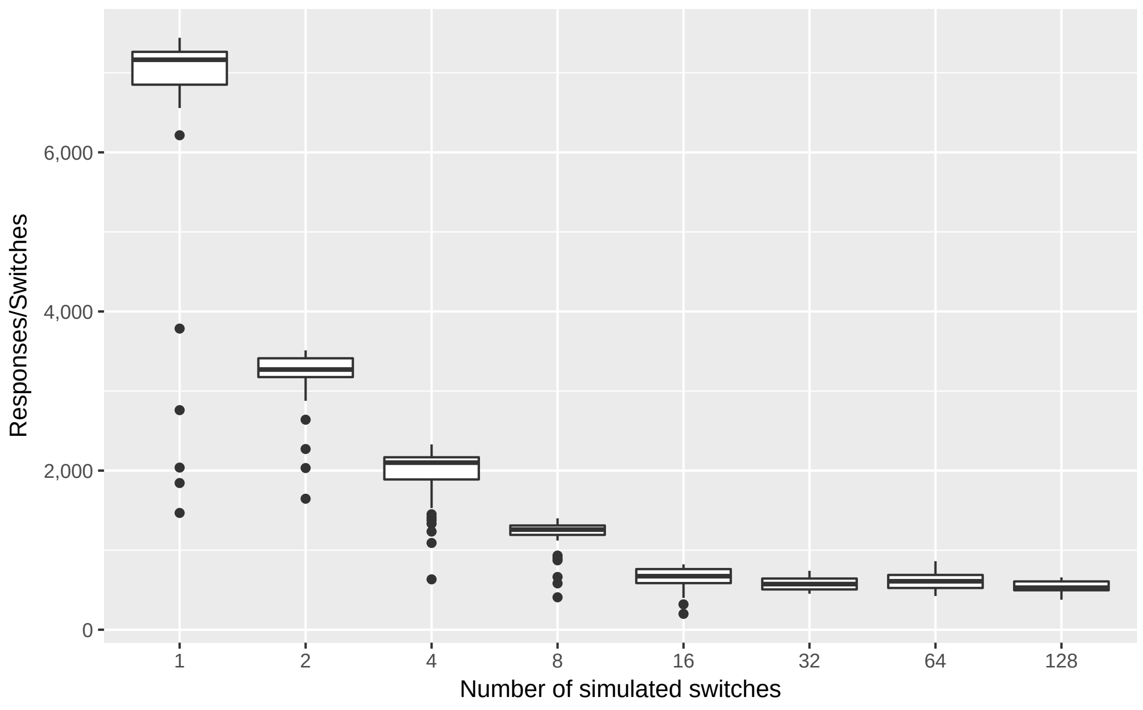


Figure 79. rxdn Responses/Switches by switches for Cbench throughput mode

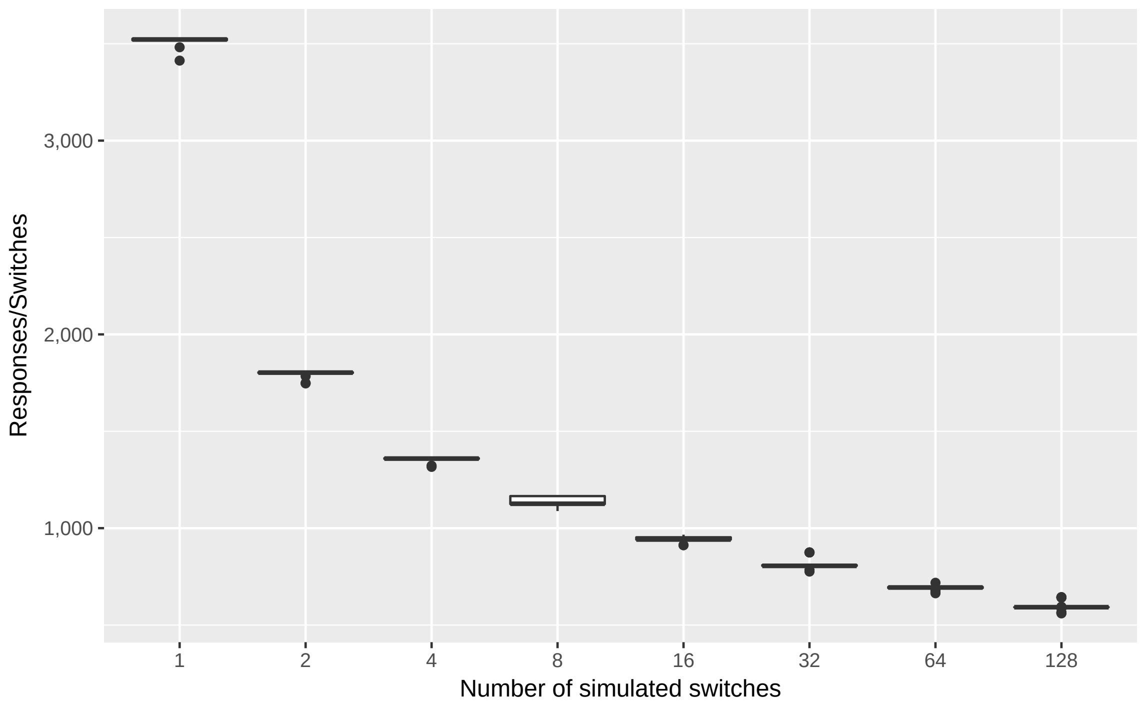


Figure 80. Ryu Responses/Switches by switches for Cbench latency mode

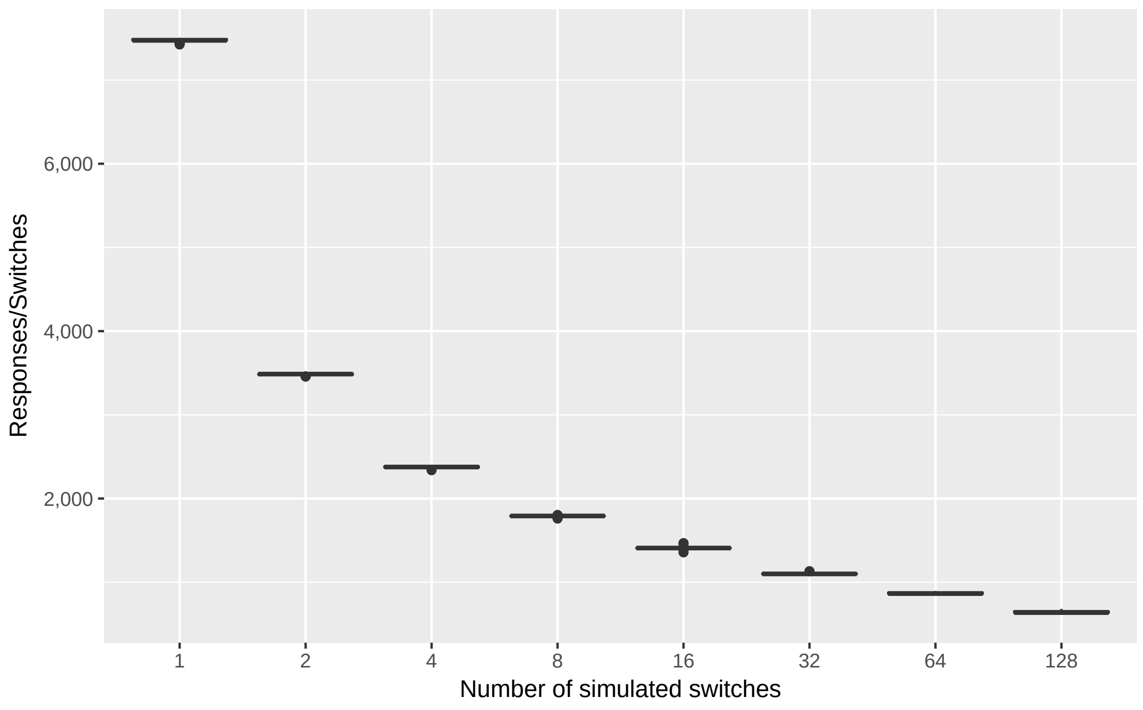


Figure 81. Ryu Responses/Switches by switches for Cbench throughput mode

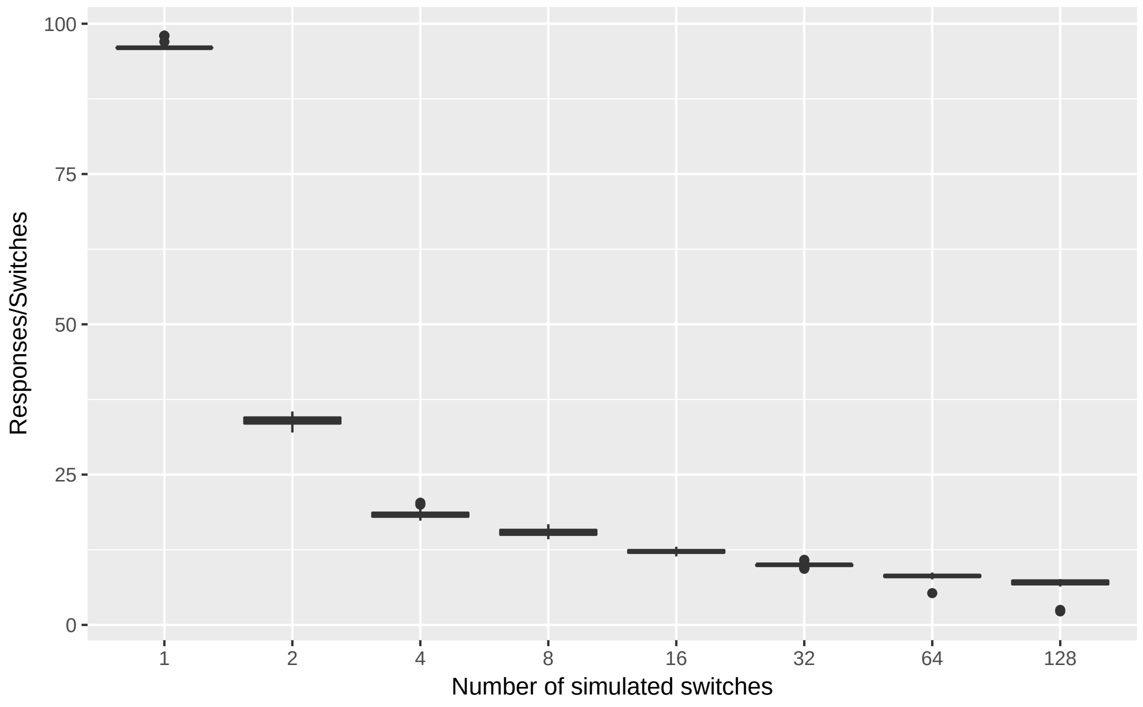


Figure 82. Trema Responses/Switches by switches for Cbench latency mode

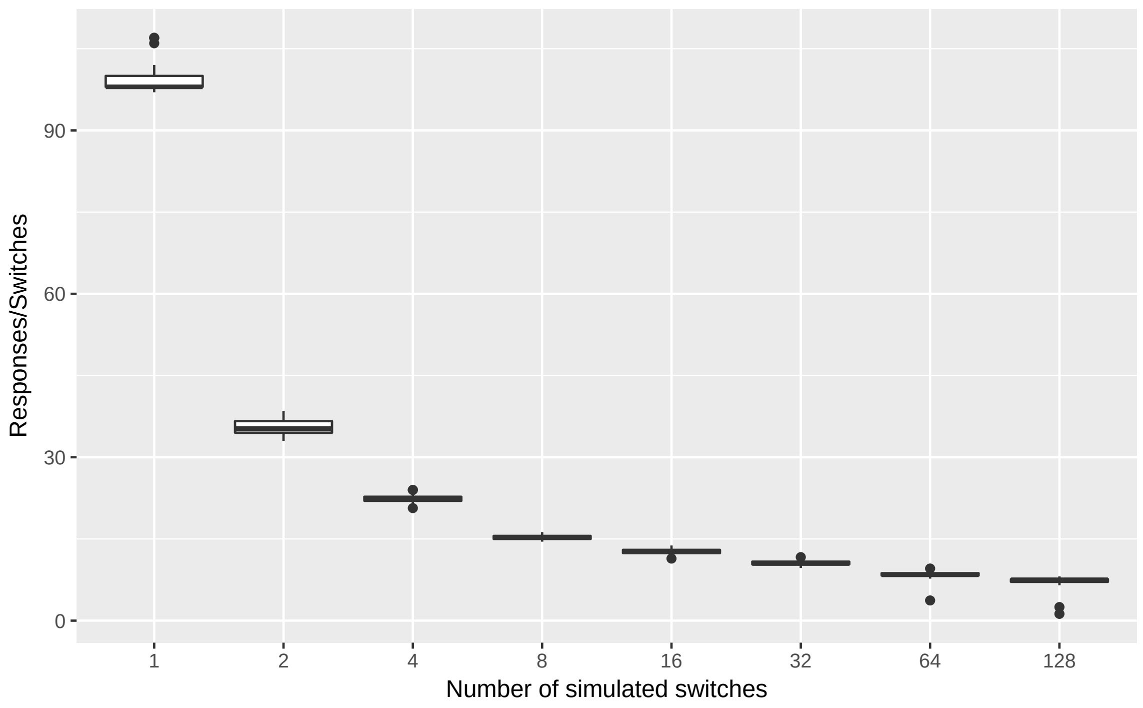


Figure 83. Trema Responses/Switches by switches for Cbench throughput mode

Appendix C. Cbench runner

The following is the source code for the `start.ts` file of Cbench runner, which executes the tests with the specified containers and collects the results from the Cbench program. This is discussed in Section 7.3.5.

```
import * as fs from "fs";
import * as yargs from "yargs";
import columnify = require("columnify");
import {safeLoad} from "js-yaml";
import {join} from "path";
import {create, start, wait, logs, remove, stop} from "./docker";

const argv = yargs
  .usage("Usage: node $0 [options]")
  .strict()
  .help("h")
  .array("switches")
  .number([
    "just",
    "loops",
    "mac-addresses",
    "ms-per-test",
    "switches",
  ])
  .alias("a", "append")
  .alias("f", "file")
  .alias("h", "help")
  .alias("j", "just")
  .alias("l", "loops")
  .alias("M", "mac-addresses")
  .alias("m", "ms-per-test")
  .alias("s", "switches")
  .alias("t", "targets")
  .boolean("append")
  .describe("append", "Append results")
  .describe("file", "File for results")
  .describe("just", "Specify a single test to run")
  .describe("mac-addresses", "Unique source MAC addresses")
  .describe("ms-per-test", "Test length in ms")
  .describe("loops", "Number of loops")
  .describe("switches", "Numbers of switches")
  .describe("targets", "YAML test specification")
  .default("file", "results.csv")
  .default("loops", 37)
  .default("mac-addresses", 100000)
  .default("ms-per-test", 1000)
  .default("switches", [1, 2, 4, 8, 16, 32, 64, 128])
  .default("targets", "targets.yaml")
  .argv;

const switches: number[] = Array.isArray(argv.switches) ? argv.
    switches : [argv.switches];

let targets_file: string;
try {
  targets_file = fs.readFileSync(join(__dirname, "..", argv.targets)
      , "utf8");
} catch (error) {
  console.error(`Error opening targets file ${argv.targets}: ${error
     }`);
  process.exit(-1);
}

const {targets} = safeLoad(targets_file);
const pause = 5;
const cbench_image = "dancasey/cbench-csv";

// If appending, need to read; else, just write
const header_mode = argv.append ? "r" : "wx";

// Run all, or just one? (Set number on command line)
let tests = targets;
if (argv.just !== undefined) tests = [targets[argv.just]];

// If appending, read header; else, write header
let results: number;
try {
  results = fs.openSync(argv.file, header_mode);
} catch (error) {
  console.error(`Error opening file ${argv.file}: ${error}`);
  process.exit(-1);
}

// If append mode, read header to find number of NAs; else write
    header
let switchMax: number;
if (argv.append) {
  let b = Buffer.alloc(2000);
  fs.readSync(results, b, 0, 2000, 0);
  let s = b.toString().split("\n")[0];
  switchMax = parseInt(s.slice(s.lastIndexOf("s") + 1), 10);
  if (switchMax < Math.max(...switches)) {
    console.error(`Error: ${argv.file} header max switches ${
        switchMax} is less than s ${Math.max(...switches)}`);
    console.error(`'NA' columns will not align; use a new results
        file`);
    process.exit(-1);
  }
} else {
  switchMax = Math.max(...switches);
  let header = "controller, mode, switches, warmup";
  for (let x = 1; x < switchMax + 1; x++) header += `, s${x}`;
  header += "\n";
  fs.writeSync(results, header);
}

// Re-open results as a writable stream
fs.closeSync(results);
const resultStream = fs.createWriteStream(argv.file, {flags: "a"});

// Show settings
console.log(`\nRunning cbench with -s {${switches}} -l ${argv.loops} -m ${argv.m} -M ${argv.M}`);
if (argv.append) {
  console.log(`Appending results to ${argv.file} with max switches ${switchMax}\n`);
} else {
  console.log(`Writing results to ${argv.file}\n`);
}
console.log(columnify(tests, {columnSplitter: " | "}));
console.log(`\nContinuing in ${pause}... (ctrl-c to cancel)`);

// promisify timeout
const delay = (sec = 1) => new Promise<void>(res => setTimeout(res, sec * 1000));

async function runTests() {
  await delay(pause);

  // Run once for each mode
  for (let mode = 0; mode < 2; mode++) {
    const modeName = mode === 1 ? "throughput" : "latency";

    for (let {name, image, port} of tests) {
      for (let s of switches) {
        console.log(`\nStarting test ${name}, s = ${s}, ${modeName}`);

        // start controller container
        let controller: string;
        if (image) {
          try {
            controller = await create(image, false, null, null, name);
            await start(controller);
          } catch (error) {
            console.error(`Creating/starting controller image ${image} failed:\n${error}`);
            process.exit(-1);
          }
          // delay for controller set-up
          await delay(2);
        }

        // cbench arguments
        let cbargs: string[] = [
          "-c", name,
          "-p", port.toString(),
          "-s", s.toString(),
          "-l", argv.loops.toString(),
          "-m", argv.m.toString(),
          "-M", argv.M.toString(),
        ];
        if (mode === 1) {
          cbargs.push("-t");
        }

        // run cbench, collect results
        let cbench: string;
        try {
          cbench = await create(cbench_image, false, [name], cbargs)
             ;
          await start(cbench);
        } catch (error) {
          console.error(`Creating/starting cbench image ${image}
             failed:\n${error}`);
          process.exit(-1);
        }

        // wait for cbench to finish
        await wait(cbench);

        // get cbench logs
        let stdout: string;
        try {
          stdout = await logs(cbench);
        } catch (error) {
          console.error(`Failed to get cbench output:\n${error}`);
          process.exit(-1);
        }

        // if no output, error
        if (!stdout) {
          // wait and try again
          console.error("Failed to get output: retry");
          try {
            await stop(cbench);
            await delay(4);
            await start(cbench);
            await wait(cbench);
            stdout = await logs(cbench);
          } catch (error) {
            console.error("Failed twice to capture cbench output");
            process.exit(-1);
          }
        }

        // stop controller container, remove it and cbench
        if (image) {
          await stop(controller);
          await remove(controller);
        }
        await remove(cbench);

        // process output chunks
        console.log(`Writing results for ${name}, s = ${s}, ${
           modeName}`);
        let lines = stdout.split("\n");
        for (let line of lines) {
          // sanitize line
          line = line.replace(/[^ -~]+/g, "");
          if (line.length > 0) {
            let str = `${name}, ${modeName}, ${line}`;
            // add in required number of "NA" columns and then
               newline
            const NAcols = switchMax - s;
            for (let x = 0; x < NAcols; x++) str += ", NA";
            str += "\n";
            resultStream.write(str);
          }
        }

        await delay();
      }
    }
  }
  // Close results file
  resultStream.end();
}

runTests();
```

Appendix D. Cbench Patch

This patch for `cbench.c` changes `stdout` to a format consistent with CSV. The patch can be applied with `patch cbench.c < cbench.patch` and then Cbench may be compiled normally. This is discussed in Section 7.3.2.1.

```
52c52
< double run_test(int n_fakeswitches, struct fakeswitch * fakeswitches, int mstestlen, int delay)
---
> double run_test(int n_fakeswitches, struct fakeswitch * fakeswitches, int mstestlen, int delay, int warmup)
83c83
<     printf("%02d:%02d:%02d.%03d %-3d switches: flows/sec:  ", tmNow->tm_hour, tmNow->tm_min, tmNow->tm_sec, (int)(now.tv_usec/1000), n_fakeswitches);
---
>     printf("%d, %d", n_fakeswitches, warmup);
88c88
<         printf("%d  ", count);
---
>         printf(", %d", count);
90a91
>     printf("\n");
94d94
<     printf(" total = %lf per ms \n", sum);
402c402
<             v = 1000.0 * run_test(i+1, fakeswitches, mstestlen, delay);
---
>             v = 1000.0 * run_test(i+1, fakeswitches, mstestlen, delay, j < warmup);
423,427c423,427
<         printf("RESULT: %d switches %d tests "
<             "min/max/avg/stdev = %.2lf/%.2lf/%.2lf/%.2lf responses/s\n",
<                 i+1,
<                 counted_tests,
<                 min, max, avg, std_dev);
---
>         // printf("RESULT: %d switches %d tests "
>         //      "min/max/avg/stdev = %.2lf/%.2lf/%.2lf/%.2lf responses/s\n",
>         //          i+1,
>         //          counted_tests,
>         //          min, max, avg, std_dev);
```

Appendix E. Controller Performance Analysis with R

Pre-test calculations

Calculate effect size, sample size

```
library(pwr)
cohen.ES(test = "anov", size = "medium")
```

```
##
##      Conventional effect size from Cohen (1982)
##
##            test = anov
##            size = medium
##     effect.size = 0.25
```

```
pwr.anova.test(k = 7, f = 0.25, sig.level = 0.05, power = 0.8)
```

```
##
##      Balanced one-way analysis of variance power calculation
##
##               k = 7
##               n = 32.05196
##               f = 0.25
##       sig.level = 0.05
##           power = 0.8
##
## NOTE: n is number in each group
```

Import and arrange data

Import raw data

```
library(tidyverse)
raw_data <- read_csv("other_results.csv", col_types = cols(
  controller = readr::col_factor(levels = c("rxdn","node-openflow","libfluid","ryu",
                                            "onos","trema","nox")),
  mode = readr::col_factor(levels = c("latency", "throughput")),
  switches = readr::col_factor(levels = c(1, 2, 4, 8, 16, 32, 64, 128)),
  warmup = readr::col_factor(levels = c(0, 1)
  ))) %>%
  # exclude warmup observations and its column
  filter(warmup == 0) %>%
  select(-warmup)
```

Arrange data for analysis

- Sum up all switch responses per row: `responses`
- Divide 'responses' by 'switches': `responses/switches`

```
sums_raw <- rowSums(raw_data[, -1:-3], na.rm = TRUE)
summed_raw <- raw_data %>%
  mutate(responses = sums_raw) %>%
  select(-matches("s\\d{1,3}")) %>%
  # add column that divides responses by switches
  mutate('responses/switches' = responses / as.numeric(switches))
```

Results which include all controllers and numbers of switches

```
latency_all <- filter(summed_raw, mode == "latency") %>% select(-mode)
throughput_all <- filter(summed_raw, mode == "throughput") %>% select(-mode)
```

Results for just `switches == 16`

```
latency16 <- latency_all %>%
  filter(switches == 16) %>%
  select(-switches)
throughput16 <- throughput_all %>%
  filter(switches == 16) %>%
  select(-switches)
```

Boxplots

Full boxplot generation omitted; two examples given below

Generate boxplot for `node-openflow`, latency mode

```
ggplot(filter(latency_all, controller=="node-openflow"),
       aes(switches, `responses/switches`)) +
  labs(x = "Number of simulated switches", y = "Responses/Switches") +
  scale_y_continuous(labels = scales::comma) +
  geom_boxplot()
ggsave("node-openflow_boxes_lat_div.pdf")
```

Generate boxplot latency mode, faceted by controller, with x = switches and y = responses

```
ggplot(latency_all, aes(switches, responses)) +
  facet_wrap(~ controller) +
  labs(x = "Number of simulated switches", y = "Responses") +
  scale_y_continuous(labels = scales::comma) +
  geom_boxplot()
ggsave("all_boxes_lat.pdf")
```

Analysis for `switches == 16`

Summarise

```
latency16 %>%
  group_by(controller) %>%
  summarise(
    mean = mean(responses),
    median = median(responses),
    sd = sd(responses),
    n=n())
```

```
## # A tibble: 7 x 5
##      controller        mean  median          sd     n
##          <fctr>       <dbl>   <dbl>       <dbl> <int>
## 1          rxdn  6543.22222  6562.0    77.05920    36
## 2 node-openflow 74455.08333 67802.5 20076.46781    36
## 3      libfluid 83798.05556 83638.5   745.45039    36
## 4           ryu  4730.58333  4707.5    64.20787    36
## 5          onos 69365.30556 69394.5   432.76822    36
## 6         trema    61.16667    61.0     2.00713    36
## 7           nox 37779.19444 37954.5   890.92706    36
```

```
throughput16 %>%
  group_by(controller) %>%
  summarise(
    mean = mean(responses),
    median = median(responses),
    sd = sd(responses),
    n=n())
```

```
## # A tibble: 7 x 5
##      controller          mean    median            sd     n
##          <fctr>         <dbl>     <dbl>         <dbl> <int>
## 1          rxdn    3274.94444    3364.5    695.683300    36
## 2 node-openflow   57823.08333   57858.5   8303.721181    36
## 3      libfluid 1300059.72222 1304894.0  12907.629707    36
## 4           ryu    7052.22222    7043.0    108.091525    36
## 5          onos  151178.66667  151932.0   3452.642524    36
## 6         trema      63.72222      63.5      2.679138    36
## 7           nox  139250.05556  140647.0   6481.411870    36
```

ANOVA Assumptions Checks

Check whether variance varies across groups - Levene's test is *non-significant* if $p > 0.05$, which means normal ANOVA is okay to use - If $p < 0.05$, must use Welch's F or a robust version of ANOVA

For latency

```
library(car)
```

```
##
## Attaching package: 'car'

## The following object is masked from 'package:dplyr':
##
##     recode

## The following object is masked from 'package:purrr':
##
##     some
```

```
leveneTest(latency16$responses, latency16$controller, center = median)
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##        Df F value    Pr(>F)
## group   6  60.492 < 2.2e-16 ***
##       245
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

For throughput

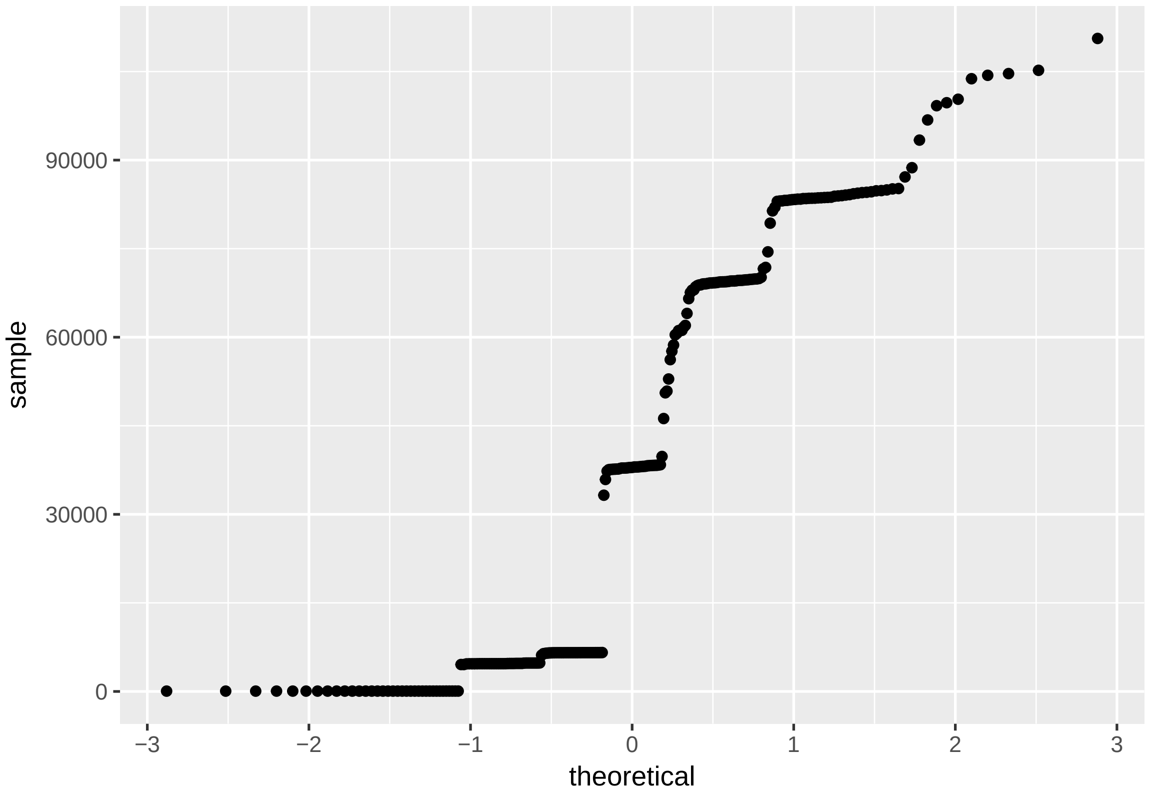
```
leveneTest(throughput16$responses, throughput16$controller, center = median)
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##        Df F value    Pr(>F)
## group   6  17.943 < 2.2e-16 ***
##       245
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

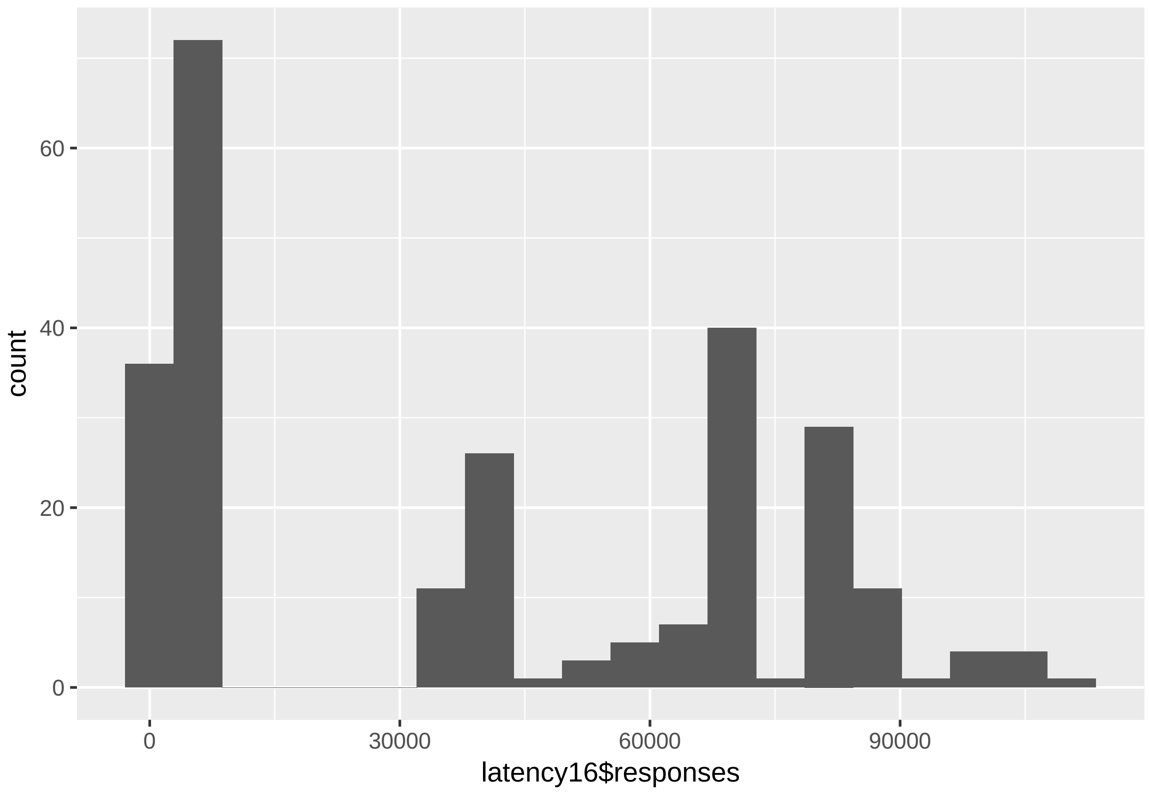
However, *Discovering Statistics Using R* note on page 440: > ...in large samples even small differences in variances might be deemed significant. As such, don't place too much weight on Levene's test if it's non-significant in a small sample, or significant in a large sample.

So plot to visualize:

```
qplot(sample = latency16$responses, geom="qq")
```



```
qplot(latency16$responses, geom = "histogram", bins = 20)
```



From above it is clearly not normal. Must use robust methods.

Welch's F

Welch's F makes adjustments for differences in group variances.

```
oneway.test(responses ~ controller, latency16)
```

```
##
##  One-way analysis of means (not assuming equal variances)
##
## data:  responses and controller
## F = 303580, num df = 6.000, denom df = 93.386, p-value < 2.2e-16
```

```
oneway.test(responses ~ controller, throughput16)
```

```
##
##  One-way analysis of means (not assuming equal variances)
##
## data:  responses and controller
## F = 97135, num df = 6.000, denom df = 93.353, p-value < 2.2e-16
```

Results from above can be summarized (with adjusted degrees of freedom):

- Latency: Welch's F(6, 93.384) = 314540, p << .0001
- Throughput: Welch's F(6, 93.354) = 95704, p << .0001

Therefore, the mean response times differ significantly across different controllers.

Robust pairwise *post hoc* tests

Again, use functions that do not assume heterscedastic data.

Source: Wilcox, R. (2012). Introduction to Robust Estimation and Hypothesis Testing (3rd ed.). Elsevier.

- `lincon()` is based on trimmed means
- `mcppb20()` uses a percentile bootstrap to compute p-values as well as trimming the group means (this is especially good at controlling Type I error rate)

```
library(WRS2)
lincon(responses ~ controller, latency16)
```

```
## Call:
## lincon(formula = responses ~ controller, data = latency16)
##
##                                 psihat   ci.lower   ci.upper p.value
## rxdn vs. node-openflow      -66050.500 -81628.031 -50472.969 0.00000
## rxdn vs. libfluid           -77185.727 -77624.467 -76746.987 0.00000
## rxdn vs. ryu                  1835.000   1791.364   1878.636 0.00000
## rxdn vs. onos               -62844.682 -63085.076 -62604.288 0.00000
## rxdn vs. nox                  6502.045   6494.176   6509.915 0.00000
## rxdn vs. trema              -31389.091 -31603.740 -31174.441 0.00000
## node-openflow vs. libfluid -11135.227 -26716.030   4445.575 0.02417
## node-openflow vs. ryu       67885.500  52307.938  83463.062 0.00000
## node-openflow vs. onos       3205.818 -12372.693  18784.330 0.49183
## node-openflow vs. nox       72552.545  56975.016  88130.075 0.00000
## node-openflow vs. trema     34661.409  19083.097  50239.721 0.00000
## libfluid vs. ryu            79020.727  78580.886  79460.569 0.00000
## libfluid vs. onos           14341.045  13860.254  14821.837 0.00000
## libfluid vs. nox            83687.773  83249.068  84126.477 0.00000
## libfluid vs. trema          45796.636  45324.733  46268.539 0.00000
## ryu vs. onos               -64679.682 -64922.124 -64437.240 0.00000
## ryu vs. nox                  4667.045   4623.779   4710.312 0.00000
## ryu vs. trema              -33224.091 -33441.049 -33007.133 0.00000
## onos vs. nox                69346.727  69106.398  69587.056 0.00000
## onos vs. trema              31455.591  31151.074  31760.108 0.00000
## nox vs. trema              -37891.136 -38105.713 -37676.559 0.00000
```

```
lincon(responses ~ controller, throughput16)
```

```
## Call:
## lincon(formula = responses ~ controller, data = throughput16)
##
##                                  psihat     ci.lower     ci.upper p.value
## rxdn vs. node-openflow       -53987.545   -58515.752   -49459.339       0
## rxdn vs. libfluid          -1300140.455 -1306513.115 -1293767.794       0
## rxdn vs. ryu                  -3636.682    -3991.006    -3282.358       0
## rxdn vs. onos               -148339.909  -150260.403  -146419.416       0
## rxdn vs. nox                   3342.136     2987.812     3696.461       0
## rxdn vs. trema              -136379.318  -138713.567  -134045.069       0
## node-openflow vs. libfluid -1246152.909 -1253575.547 -1238730.271       0
## node-openflow vs. ryu         50350.864    45830.047    54871.681       0
## node-openflow vs. onos       -94352.364   -99120.454   -89584.273       0
## node-openflow vs. nox         57329.682    52808.865    61850.499       0
## node-openflow vs. trema      -82391.773   -87290.674   -77492.871       0
## libfluid vs. ryu           1296503.773  1290136.345  1302871.201       0
## libfluid vs. onos          1151800.545  1145269.770  1158331.321       0
## libfluid vs. nox           1303482.591  1297115.163  1309850.019       0
## libfluid vs. trema         1163761.136  1157139.955  1170382.318       0
## ryu vs. onos               -144703.227  -146605.761  -142800.694       0
## ryu vs. nox                   6978.818     6976.978     6980.659       0
## ryu vs. trema              -132742.636  -135062.282  -130422.991       0
## onos vs. nox                151682.045   149779.512   153584.579       0
## onos vs. trema               11960.591     9121.125    14800.057       0
## nox vs. trema              -139721.455  -142041.100  -137401.809       0
```

```
library(WRS2)
mcppb20(responses ~ controller, latency16, nboot = 5000)
```

```
## Call:
## mcppb20(formula = responses ~ controller, data = latency16, nboot = 5000)
##
##                                psihat   ci.lower   ci.upper p-value
## rxdn vs. node-openflow     -66050.500 -82063.591 -54125.091  0.0000
## rxdn vs. libfluid          -77185.727 -77628.909 -76859.227  0.0000
## rxdn vs. ryu                 1835.000   1791.591   1862.409  0.0000
## rxdn vs. onos              -62844.682 -63043.409 -62616.500  0.0000
## rxdn vs. nox                 6502.045   6481.818   6509.364  0.0000
## rxdn vs. trema             -31389.091 -31562.045 -31201.682  0.0000
## node-openflow vs. libfluid -11135.227 -23041.545   4857.000  0.0152
## node-openflow vs. ryu       67885.500  55975.500  83903.955  0.0000
## node-openflow vs. onos       3205.818  -8741.545  19266.591  0.4640
## node-openflow vs. nox       72552.545  60626.227  88563.545  0.0000
## node-openflow vs. trema     34661.409  22710.727  50759.909  0.0000
## libfluid vs. ryu            79020.727  78690.136  79459.273  0.0000
## libfluid vs. onos           14341.045  13964.273  14854.409  0.0000
## libfluid vs. nox            83687.773  83360.636  84128.636  0.0000
## libfluid vs. trema          45796.636  45418.727  46258.955  0.0000
## ryu vs. onos               -64679.682 -64882.864 -64447.727  0.0000
## ryu vs. nox                  4667.045   4641.727   4704.273  0.0000
## ryu vs. trema              -33224.091 -33397.955 -33030.045  0.0000
## onos vs. nox                69346.727  69120.273  69545.500  0.0000
## onos vs. trema              31455.591  31165.591  31744.636  0.0000
## nox vs. trema              -37891.136 -38064.364 -37703.909  0.0000
```

```
mcppb20(responses ~ controller, throughput16, nboot = 5000)
```

```
## Call:
## mcppb20(formula = responses ~ controller, data = throughput16,
##     nboot = 5000)
##
##                                  psihat     ci.lower     ci.upper p-value
## rxdn vs. node-openflow       -53987.545   -58203.864   -49597.955       0
## rxdn vs. libfluid          -1300140.455 -1304072.136 -1292349.182       0
## rxdn vs. ryu                  -3636.682    -4033.455    -3344.682       0
## rxdn vs. onos               -148339.909  -149974.045  -146105.773       0
## rxdn vs. nox                   3342.136     2939.682     3633.864       0
## rxdn vs. trema              -136379.318  -138905.545  -133903.227       0
## node-openflow vs. libfluid -1246152.909 -1252210.182 -1236716.273       0
## node-openflow vs. ryu         50350.864    45981.955    54527.636       0
## node-openflow vs. onos       -94352.364   -98835.500   -89563.682       0
## node-openflow vs. nox         57329.682    52958.136    61517.045       0
## node-openflow vs. trema      -82391.773   -87786.364   -77724.455       0
## libfluid vs. ryu            1296503.773  1288731.455  1300382.864       0
## libfluid vs. onos           1151800.545  1143869.591  1156240.136       0
## libfluid vs. nox            1303482.591  1295696.000  1307361.318       0
## libfluid vs. trema          1163761.136  1155458.091  1168494.773       0
## ryu vs. onos                -144703.227  -146282.455  -142328.500       0
## ryu vs. nox                    6978.818     6956.182     7014.409       0
## ryu vs. trema               -132742.636  -135354.909  -130228.864       0
## onos vs. nox                 151682.045   149306.273   153261.318       0
## onos vs. trema                11960.591     8626.273    14606.045       0
## nox vs. trema               -139721.455  -142333.500  -137207.227       0
```

Bibliography

[1] Paul Goransson, Chuck Black, and Timothy Culver. *Software Defined Networks: A Comprehensive Approach*. 2nd Ed. Waltham, MA: Morgan Kaufmann, 2016.

[2] Open Networking Foundation. “Software-Defined Networking: The New Norm for Networks”. *ONF White Paper* (2012), pp. 1–12.

[3] Theophilus Benson, Aditya Akella, and David Maltz. “Unraveling the complexity of network management”. *6th USENIX Symposium on Networked Systems Design and Implementation*. Boston, MA: USENIX Association, 2009, pp. 335–348.

[4] Nick Feamster, Jennifer Rexford, and Ellen Zegura. “The Road to SDN”. *ACM Queue* 11.12 (Dec. 2013), pp. 20–40.

[5] Jonathan M Smith and Scott M Nettles. “Active networking: One view of the past, present and future”. *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews* 34.1 (Feb. 2004), pp. 4–18.

[6] D.L. Tennenhouse et al. “A survey of active network research”. *IEEE Communications Magazine* 35.1 (Jan. 1997), pp. 80–86.

[7] J.D. Case et al. *Simple Network Management Protocol*. Internet Requests for Comment. Aug. 1988. URL: https://www.rfc-editor.org/info/rfc1067.

[8] Douglas Mauro and Kevin Schmidt. *Essential SNMP*. 2nd Ed. Sebastopol, CA: O’Reilly Media, Inc., 2005.

[9] *NETCONF Configuration Protocol*. Internet Requests for Comment. Dec. 2006. URL: https://www.rfc-editor.org/info/rfc4741.

[10] Open Networking Foundation. *OpenFlow Configuration and Management Protocol OF-CONFIG 1.0*. 2011. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config1dot0-final.pdf (last accessed 12/19/2017).

[11] Ben Pfaff et al. “The Design and Implementation of Open vSwitch”. *12th USENIX Symposium on Networked Systems Design and Implementation*. Oakland, CA, 2015, pp. 117–130.

[12] B. Pfaff. *The Open vSwitch Database Management Protocol*. Internet Requests for Comment. Dec. 2013. URL: https://www.rfc-editor.org/info/rfc7047.

[13] S. Denazis et al. *Software-Defined Networking (SDN): Layers and Architecture Terminology*. Internet Requests for Comment. Jan. 2015. URL: https://www.rfc-editor.org/info/rfc7426.

[14] Lily Yang et al. *Forwarding and Control Element Separation (ForCES) Framework*. Internet Requests for Comment. Apr. 2004. URL: https://www.rfc-editor.org/info/rfc3746.

[15] Nick McKeown et al. “OpenFlow: Enabling Innovation in Campus Networks”. *ACM SIGCOMM Computer Communication Review* 38.2 (Mar. 2008), p. 69.

[16] Open Networking Foundation. *OpenFlow Protocol Specifications*. 2015. URL: https://www.opennetworking.org/sdn-resources/technical-library (last accessed 07/09/2015).

[17] Teemu Koponen et al. "Onix: A distributed control platform for large-scale production networks". *9th USENIX conference on Operating Systems Design and Implementation*. Vol. 10. Vancouver, BC: USENIX Association, 2010, pp. 1–6.

[18] Andrew D. Ferguson et al. "Participatory networking". *ACM SIGCOMM Computer Communication Review* 43.4 (Aug. 2013), pp. 327–338.

[19] Martin Casado and Teemu Koponen. *What Might an SDN Controller API Look Like?* 2011. URL: http://networkheresy.com/2011/08/09/what-might-an-sdn-controller-api-look-like-and-should-we-standardize-it/ (last accessed 07/09/2015).

[20] Natasha Gude et al. "NOX: Towards an Operating System for Networks". *ACM SIGCOMM Computer Communication Review* 38.3 (July 2008), p. 105.

[21] *NOX Repository*. 2013. URL: http://www.noxrepo.org (last accessed 12/19/2017).

[22] Amin Tootoonchian et al. "On Controller Performance in Software-Defined Networks". *2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*. San Jose, CA: USENIX Association, 2012, p. 10.

[23] Nate Foster et al. "Languages for software-defined networks". *IEEE Communications Magazine* 51.2 (Feb. 2013), pp. 128–134.

[24] Joshua Reich et al. "Modular SDN Programming with Pyretic". *USENIX Magazine* 38.5 (2013), pp. 128–134.

[25] Christopher Monsanto et al. "Composing Software Defined Networks". *10th USENIX Symposium on Networked Systems Design and Implementation*. Lombard, IL: USENIX, 2013, pp. 1–13.

[26] David Erickson. "The beacon openflow controller". *2nd ACM SIGCOMM workshop on Hot topics in software defined networking*. New York, NY: ACM Press, 2013, p. 13.

[27] Big Switch Networks. *Floodlight Controller*. 2012. URL: http://www.projectfloodlight.org/floodlight (last accessed 12/19/2017).

[28] Linux Foundation. *OpenDaylight*. 2015. URL: http://www.opendaylight.org (last accessed 07/09/2015).

[29] Fei Hu. *Network Innovation Through OpenFlow and SDN: Principles and Design*. Boca Raton, FL: CRC Press, 2014.

[30] Jan Medved et al. "OpenDaylight: Towards a Model-Driven SDN Controller architecture". *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*. Sydney: IEEE, June 2014, pp. 1–6.

[31] Fernando Farias et al. “Integrating Legacy Forwarding Environment to OpenFlow/SDN Control Plane”. *15th Asia-Pacific Network Operations and Management Symposium*. Hiroshima, Japan, Sept. 2013, pp. 1–3.

[32] Ryan Hand and Eric Keller. “ClosedFlow: OpenFlow-like Control over Proprietary Devices”. *3rd workshop on Hot topics in software defined networking*. New York, NY: ACM Press, Aug. 2014, pp. 7–12.

[33] Dan Levin et al. “Panopticon: Reaping the Benefits of Incremental SDN Deployment in Enterprise Networks”. *USENIX Annual Technical Conference*. Philadelphia, PA: USENIX Association, June 2014, pp. 333–345.

[34] Stanford University and National Science Foundation. *NetFPGA Project*. 2013. URL: http://netfpga.org (last accessed 07/09/2015).

[35] ESnet & Lawrence Berkeley National Laboratory. *iperf3*. 2015. URL: http://software.es.net/iperf/ (last accessed 07/09/2015).

[36] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. New York, NY: John Wiley & Sons, Inc., 1991.

[37] Andy Field, Jeremy Miles, and Zoe Field. *Discovering Statistics Using R*. London: SAGE Publications Ltd, 2012.

[38] Jad Naous et al. “Implementing an OpenFlow switch on the NetFPGA platform”. *4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS ’08. New York, NY: ACM, 2008, pp. 1–9.

[39] Diego Kreutz et al. “Software-Defined Networking: A Comprehensive Survey”. *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76.

[40] Douglas Crockford. *JavaScript Object Notation (JSON)*. 2006. URL: http://www.json.org (last accessed 07/09/2015).

[41] Francis Galiegue and Kris Zyp. *JSON Schema*. 2015. URL: http://json-schema.org (last accessed 07/09/2015).

[42] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. *YAML Ain’t Markup Language (YAML) Version 1.2*. 2009. URL: http://www.yaml.org/spec/1.2/spec.html (last accessed 07/09/2015).

[43] Isaac Z. Schlueter, Laurie Voss, and Rod Boothby. *npm: The Package Manager for Node.js*. 2011. URL: https://www.npmjs.com (last accessed 07/09/2015).

[44] Evgeny Poberezkin. *Ajv: Another JSON Schema Validator*. 2015. URL: http://epoberezkin.github.io/ajv/ (last accessed 12/19/2017).

[45] Dan Talayco et al. *Loxigen: OpenFlow protocol bindings for multiple languages*. 2013. URL: https://github.com/floodlight/loxigen (last accessed 07/09/2015).

[46] Allan Vidal et al. *libfluid: The ONF OpenFlow driver*. 2013. URL: http://opennetworkingfoundation.github.io/libfluid/ (last accessed 07/15/2015).

[47] Isaku Yamahata and Fujita Tomonori. *The Ryu Network Operating System*. 2013. URL: http://osrg.github.io/ryu/ (last accessed 07/15/2015).

[48] Sergey Shepelev, Bob Ippolito, and Donovan Preston. *Python Eventlet Asynchronous I/O Library*. 2009. URL: http://eventlet.net (last accessed 12/19/2017).

[49] Armin Rigo and Christian Tismer. *Python Greenlet Micro-threads*. 2010. URL: https://greenlet.readthedocs.io/en/latest/ (last accessed 12/19/2017).

[50] Kai Lei, Yining Ma, and Zhi Tan. "Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js". *17th International Conference on Computational Science and Engineering*. Chengdu, China: IEEE, Dec. 2014, pp. 661–668.

[51] Node.js Foundation, Joyent Inc., and The Linux Foundation. *Node.js*. 2010. URL: https://nodejs.org/en/ (last accessed 07/09/2015).

[52] Google Inc. *V8 JavaScript Engine*. 2008. URL: https://developers.google.com/v8 (last accessed 07/09/2015).

[53] Ryan Dahl et al. *Libuv: Cross-platform asynchronous I/O*. 2015. URL: http://libuv.org (last accessed 12/19/2017).

[54] Stefan Tilkov and Steve Vinoski. "Node.js: Using JavaScript to Build High-Performance Network Programs". *IEEE Internet Computing* 14.6 (Nov. 2010), pp. 80–83.

[55] Ecma International. *ECMA-262: ECMAScript Language Specification 5.1*. 2015. URL: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (last accessed 07/09/2015).

[56] Douglas Crockford. *JavaScript: The Good Parts*. Sebastopol, CA: O'Reilly Media, Inc., 2008.

[57] Anders Hejlsberg. *Introducing TypeScript*. 2012. URL: https://channel9.msdn.com/posts/Anders-Hejlsberg-Introducing-TypeScript (last accessed 12/19/2017).

[58] Jeremy Ashkenas. *CoffeeScript*. 2015. URL: http://coffeescript.org (last accessed 07/09/2015).

[59] Ian Hickson and World Wide Web Consortium. *Web Workers Draft API*. 2015. URL: https://www.w3.org/TR/workers/ (last accessed 12/19/2017).

[60] Gerald Jay Sussman and Guy L. Steele Jr. "Scheme: A Interpreter for Extended Lambda Calculus". *Higher Order Symbolic Computation* 11.4 (1998), pp. 405–439.

[61] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. "An evaluation of reactive programming and promises for structuring collaborative web applications". *7th Workshop on Dynamic Languages and Applications*. New York, NY: ACM Press, 2013, pp. 1–9.

[62] Daniel P. Friedman and David S. Wise. “Aspects of Applicative Programming for Parallel Processing”. *IEEE Transactions on Computers* C-27.4 (Apr. 1978), pp. 289–296.

[63] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. “Don’t Call Us, We’ll Call You: Characterizing Callbacks in Javascript”. *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Beijing, China: IEEE, Oct. 2015, pp. 1–10.

[64] Petka Antonov. *Bluebird.js: Why Use Promises?* 2016. URL: http://bluebirdjs.com/docs/why-promises.html (last accessed 12/19/2017).

[65] Semih Okur et al. “A study and toolkit for asynchronous programming in C#”. *36th International Conference on Software Engineering*. New York, NY: ACM Press, 2014, pp. 1117–1127.

[66] Microsoft Corp. *Typescript*. 2013. URL: http://www.typescriptlang.org (last accessed 12/19/2017).

[67] Microsoft Corp. *ReactiveX*. 2010. URL: http://reactivex.io (last accessed 12/19/2017).

[68] Erik Meijer. “Your mouse is a database”. *Communications of the ACM* 55.5 (May 2012), p. 66.

[69] *ReactiveX / RxClojure: RxJava bindings for Clojure*. 2017. URL: https://github.com/ReactiveX/RxClojure (last accessed 11/28/2017).

[70] Netflix Inc. & Microsoft Corporation. *ReactiveX / rxjs: A reactive programming library for JavaScript*. 2015. URL: https://github.com/ReactiveX/rxjs (last accessed 11/28/2017).

[71] *ReactiveX / RxScala: Reactive Extensions for Scala*. 2015. URL: http://reactivex.io/rxscala/ (last accessed 12/19/2017).

[72] *ReactiveX / RxSwift: Reactive Programming in Swift*. 2016. URL: https://github.com/ReactiveX/RxSwift (last accessed 12/19/2017).

[73] Andre Staltz. *Cycle.js*. 2014. URL: https://cycle.js.org (last accessed 12/19/2017).

[74] Erik Meijer. “The world according to LINQ”. *Communications of the ACM* 54.10 (Oct. 2011), p. 45.

[75] Amin Tootoonchian and Yashar Ganjali. “HyperFlow: A distributed control plane for OpenFlow”. *Internet Network Management Conference on Research on Enterprise Networking*. San Jose, CA: USENIX Association, 2010, p. 3.

[76] Pankaj Berde et al. “ONOS: Towards an Open, Distributed SDN OS”. *3rd workshop on Hot topics in software defined networking*. New York, NY: ACM Press, 2014, pp. 1–6.

[77] Michael Jarschel et al. “A Flexible OpenFlow-Controller Benchmark”. *European Workshop on Software Defined Networking*. Darmstadt, Germany: IEEE, Oct. 2012, pp. 48–53.

[78] Alexander Shalimov et al. “Advanced study of SDN/OpenFlow controllers”. *9th Central & Eastern European Software Engineering Conference in Russia*. Moscow: ACM, 2013, pp. 1–6.

[79] Michael Jarschel et al. “OFCProbe: A platform-independent tool for OpenFlow controller analysis”. *5th International Conference on Communications and Electronics*. Danang, Vietnam: IEEE, July 2014, pp. 182–187.

[80] C Laissaoui et al. “A measurement of the response times of various OpenFlow/SDN controllers with CBench”. *12th ACS/IEEE International Conference of Computer Systems and Applications*. Vol. 2016-July. Marrakech, Morocco: IEEE, Nov. 2015, pp. 1–2.

[81] Ola Salman et al. “SDN controllers: A comparative study”. *18th Mediterranean Electrotechnical Conference*. Limassol, Cyprus: IEEE, Apr. 2016.

[82] Clay Ford. *Getting started with the pwr package for the R programming language*. 2017. URL: https://cran.r-project.org/web/packages/pwr/vignettes/pwr-vignette.html (last accessed 12/12/2017).

[83] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. 2nd Ed. Hillsdale, NJ: L. Erlbaum Associates, 1988.

[84] Linux Torvalds. *Git version control system*. 2007. URL: https://git-scm.com (last accessed 12/19/2017).

[85] Docker Inc. *Docker software containerization platform*. 2013. URL: https://www.docker.com (last accessed 12/19/2017).

[86] RxJS Contributors. *Issue: Backpressure story*. 2015. URL: https://github.com/ReactiveX/rxjs/issues/71 (last accessed 12/19/2017).

[87] RxJava Contributors. *Backpressure in RxJava*. 2013. URL: https://github.com/ReactiveX/RxJava/wiki/Backpressure (last accessed 12/19/2017).

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

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD–MM–YYYY)* 23–03–2018	**2. REPORT TYPE** Doctoral Dissertation	**3. DATES COVERED** *(From — To)* Sept 2013 — Mar 2018
4. TITLE AND SUBTITLE Progressive Network Deployment, Performance, and Control with Software-Defined Networking		**5a. CONTRACT NUMBER**
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Casey, Daniel J., Major, USAF		**5d. PROJECT NUMBER** 17G139
		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-DS-18-M-017
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Undisclosed		**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

This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

The inflexible nature of traditional computer networks has led to tightly-integrated systems that are inherently difficult to manage and secure. New designs move low-level network control into software creating software-defined networks (SDN). Augmenting an existing network with these enhancements can be expensive and complex. This research investigates solutions to these problems. It is hypothesized that an add-on device, or "shim" could be used to make a traditional switch behave as an OpenFlow SDN switch while maintaining reasonable performance. A design prototype is found to cause approximately 1.5% reduction in throughput for one flow and less than double increase in latency, showing that such a solution may be feasible. It is hypothesized that a new design built on event-loop and reactive programming may yield a controller that is higher-performing and easier to program. The library node-openflow is found to have performance approaching that of professional controllers, however it exhibits higher variability in response rate. The framework rxdn is found to exceed performance of two comparable controllers by at least 33% with statistical significance in latency mode with 16 simulated switches, but is slower than the library node-openflow or professional controllers (e.g., Libfluid, ONOS, and NOX). Collectively, this work enhances the tools available to researchers, enabling experimentation and development toward more sustainable and secure infrastructure.

15. SUBJECT TERMS

Software-Defined Networking, FPGA, VHDL, network controller, schema, JSON Schema, event loop, Reactive Extensions, Rxjs, JavaScript, Node.js, TypeScript, Cbench

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dr. Barry E. Mullins, AFIT/ENG
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
U	U	U	UU	196	(937) 255–3636 x7979; barry.mullins@afit.edu