# A Testbed for Software-Defined Networking Applications

Master-Thesis von Sushil Jain aus Darmstadt
Tag der Einreichung:

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Matthias Eichholz, M.Sc.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Software Technology Group

A Testbed for Software-Defined Networking Applications

Vorgelegte Master-Thesis von Sushil Jain aus Darmstadt

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Matthias Eichholz, M.Sc.

Tag der Einreichung:

# Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den April 20, 2017

(Sushil Jain)

## Abstract

Software-Defined Networks (SDNs) is a new approach to manage and control computer networks. In conventional network devices, data plane and control plane were tightly-coupled. These devices are being replaced with special-purpose programmable network devices. In these new programmable devices such as switches, data plane and control plane are separated. These switches can be managed by a centralized controller. The communication between these switches and the controller is realized using well-defined APIs such as OpenFlow. With OpenFlow, network policies are written on top of the controller. These policies are then used by the controller to control switches in the network. However, writing network policies using OpenFlow is cumbersome. Because the features offered by OpenFlow looks like the features offered by the hardware. To solve this problem, many different SDN programming languages have been proposed that offer higher abstractions such querying network state, composition and verification.

Implementing a new SDN language or its feature requires a comparison with already available languges to present how it increases the expressivity of the language while offering at least the same runtime performance. Presently, it is quite difficult to compare SDN languages because there does not exist any automatic mechanism to compare these languages. To overcome this problem, this thesis proposes an extensible testbed for SDN applications which focus on comparing SDN languages. This testbed provides a platform to compare SDN languages for various test cases. It consists of controllers for SDN languages, some sample applications and test cases for comparison purpose. This testbed is extensible and it is easy to accommodate new controllers, applications and test cases.

## Contents


**1 Introduction** **1**
1.1 Motivation . . . 2
1.2 Goals . . . 2
1.3 Objectives . . . 2
1.4 Contribution . . . 3
1.5 Thesis Outline . . . 3

**2 Background** **5**
2.1 Software Defined Network (SDN) . . . 5
2.1.1 SDN Architecture and its Components . . . 5
2.1.2 Packet Handling with SDN . . . 6
2.1.3 SDN Abstractions - Hide Unnecessary Network Details . . . 7
2.1.4 SDN Languages - Network Programming Languages . . . 8
2.1.5 SDN Controllers - Control over Traffic in Network . . . 12
2.1.6 NetIDE . . . 14
2.2 OpenFlow - A Protocol for SDN . . . 15
2.2.1 Communication in Network using OpenFlow . . . 15
2.2.2 Traffic Forwarding using OpenFlow Switch . . . 16
2.3 Mininet - Network Emulator . . . 17
2.4 Wireshark - Capturing Packets in Network . . . 17

**3 Related Work** **20**
3.1 Hardware Based Testbed Solution . . . 20
3.2 Software Based Testbed Solutions . . . 24
3.3 Comparison . . . 27
3.4 Conclusion . . . 27

**4 System Design** **29**
4.1 Use Case Scenarios . . . 29
4.2 Challenges and Requirements for System Design . . . 30
4.3 Assumptions . . . 31
4.4 Extensibility . . . 31
4.5 System Architecture . . . 31
4.5.1 Environment Module - Setup Environment for Test Case . . . 32
4.5.2 Controller Manager Module - To Control and Manage SDN Controller . . . 32
4.5.3 Application Manager - To Control and Manage SDN Application . . . 32
4.5.4 TestSuite Module - TestSuite for Test cases . . . 33
4.5.5 STM - To Control and Manage SDN Test case . . . 33
4.5.6 Data Formatting - Converting Result into specific Data format . . . 33
4.6 Life Cycle . . . 34

**5 Implementation** **37**
5.1 Implementation of Testbed Framework . . . 37
5.1.1 Programming Language . . . 37
5.2 Implementation of Environment Module . . . 37
5.2.1 Controller Class . . . 40
5.2.2 ApplicationRunner Class . . . 40

5.3 Implementation of TestSuite Module . . . 41
5.3.1 TestSuite . . . 41
5.3.2 TestCase . . . 42
5.3.3 DataWriter . . . 43
5.4 Testbed Workflow . . . 43
5.5 SDN Applications . . . 43
5.5.1 Hub . . . 43
5.5.2 Learning Switch . . . 43
5.5.3 Web Firewall . . . 45

**6 Evaluation** **47**
6.1 Evaluation Setup . . . 47
6.2 Network Topologies for Evaluation . . . 47
6.3 Evaluation Metrics . . . 48
6.4 Hub Application . . . 49
6.4.1 Number of rules . . . 49
6.4.2 Number of Packets at Controller . . . 49
6.4.3 Number of Flow rules in switch at a Particular Time . . . 52
6.4.4 Lines of code for Hub Application . . . 55
6.5 Switch Learning Application . . . 55
6.5.1 Number of rules . . . 55
6.5.2 Number of Packets at Controller . . . 55
6.5.3 Number of Flow rules in switch at a Particular Time . . . 56
6.5.4 Lines of code for Switch Learning Application . . . 56
6.6 Web Firewall Application . . . 56
6.6.1 Number of rules . . . 59
6.6.2 Number of Packets at Controller . . . 59
6.6.3 Number of Flow rules in switch at a Particular Time . . . 62
6.6.4 Lines of code for Web Firewall Application . . . 62

**7 Conclusion and Future Work** **72**
7.1 Future Work . . . 72

# 1 Introduction

This chapter briefly describes the problem with conventional network, reason why SDN emerged. It also presents a very short introduction of OpenFlow and its problems, and SDN programming languages. This is followed by the motivation behind this thesis. This chapter ends with the goals and outline of this thesis.

A computer network consists of many network elements such as switches and routers, and these elements are created with help of software and hardware. In conventional networks, these elements are tightly-coupled, closed and proprietary which is an obstacle when there is a need to change the network functionality. This obstacle of changing networking functionality had an effect on innovation that forced the network personnel to write down the network policies using complicated interfaces. A policy is a group of conditions and corresponding actions [40]. Internet is based on the IP network which represents this obstacle at a big level [24] and became a motivation for the network researchers to rethink over the fundamental design of today's network architectures. In older devices, control plane and data plane were tightly coupled. A tight coupling between both was considered vital for the network resilience. In fact, this design has been successful in terms of network performance. Software defined network (SDN) is an emerging network paradigm and hopes to remove the mentioned obstacle. It separates the control plane from the data plane [24]. The control plane can manage a switch remotely using software based controller. In a network, when a packet arrives at the switch interface, it looks for a matching rule in the flow tables of the switch. If a matching rule is found then corresponding action such as drop packet, flood (send packet to all ports) is taken. In case, there is no match, the switch sends the packet to the controller which in turn either generates the packet out event or generate flow rules. This communication between the controller and the switch is carried out using well defined and popular protocols such OpenFlow, NETCONF etc. These protocols offer the functionality to control the network remotely while allowing packets to be processed in the switch hardware. There exist controllers such as Ryu [35], Project FloodLight [16], Frenetic [9], and many more which leverage these protocols to control the network. Now-a-days, many commercial ethernet switch vendors support the OpenFlow and many campuses are using it for their backbone network [11].

With OpenFlow it is possible to develop the new and exciting network applications, but OpenFlow does not make it easy to develop the application because it offers low-level abstraction derived from the hardware. OpenFlow offers prioritized match and action rules; with which it is hard to write the sophisticated network applications. Usually, networks are responsible for multiple tasks like traffic monitoring, firewall, access control, and routing. Unfortunately, decoupling these tasks is very difficult because flow rules written for one responsibility may overlap with flow rules written for another responsibility. Network programmers only get the events for which the switch does not know how to handle. Sometimes, one piece of code may block the event to be triggered for another rule [9]. These things make the network programming difficult using OpenFlow.

To overcome these mentioned difficulties, various SDN programming languages have been proposed. These languages offer the high level abstraction which leverages the OpenFlow API to perform the network related tasks. There exist many SDN languages such as Frenetic, Pyretic, Kinetic, Merlin, and NetKAT. These languages are built on the top of general purpose programming languages such as Python, Java etc. Some of these SDN languages are conceptual while some have their own implementations. However, there exists no mechanism to compare these SDN languages. A comparison can help finding the interesting things such as contributions made by these languages, abstractions offered etc. There exist many testbeds for SDN controllers which offers the ways to measure the controller performance. However, there does not exist any solution which offers the functionality to compare different

SDN languages.

In this thesis, a SDN testbed is proposed to compare the various SDN languages in terms of expressivity, abstraction and performance. This thesis also aims at testbed extensibility so that it becomes easy to compare new SDN languages in future. In the proposed testbed, few SDN applications are there for each controller. These applications can be tested on the desired network topology which is emulated by Mininet [26]. Mininet is an emulator that simulates a network of virtual switches, host, links, and controllers. Virtual hosts or computers in Mininet run the Linux based networking software. Switches in Mininet support OpenFlow which makes it well-suitable for SDN testbed created in this thesis because it fulfils the promise of SDN.

We shall not use NETCONF because it is used for reconfiguring the network devices and is not useful for this thesis. The main goal of this thesis is to create a testbed to compare SDN programming languages. This testbed should also allow for testing the controllers with same and/or different applications in order to find out the interesting results such as number of flow rules generated by controller, number of packets handled by controller etc.

## 1.1 Motivation

Organizations such as Google, IBM, Verizon and Deutsche Telekom are using the SDN to manage their networks. There are numerous SDN switches, controllers, and applications available to make the networking tasks easier. There exist many testbeds for conducting transparent, rigorous, and replicable testing of SDN applications in different environments. These testbeds can be categorized into two categories: hardware based which mainly focus on the infrastructure such as GENI [3], and software based that targets the control plane such as Cbench[1]. Unfortunately, these testbeds do not offer any feature to compare SDN controllers from the programming language perspective.
To tackle this problem, this thesis proposes a testbed solution to compare SDN languages.

## 1.2 Goals

The main goal of this is to implement an extensible testbed framework which allows comparing SDN languages with respect to expressivity and performance. These languages are compared by deploying their corresponding controller. Another goal of this thesis is to use few sample applications for selected SDN language, so that these languages can be compared for abstraction, benchmarking, and statistics data like number of rules generated etc.

## 1.3 Objectives

The available testbeds for SDN do not target the language based features such as abstraction, expressivity etc., and this thesis focuses on this incapability. The goals of this thesis are achieved with following objectives:

- Investigating currently available SDN languages and their corresponding controllers.
- Creating an environment where multiple controllers are installed.

[1] https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343657/Cbench+New

- Exploring mainstream SDN applications and having the same application for every installed controller.
- Exploring relevant test cases for comparing SDN languages.
- Implementing a testbed framework that performs tasks as follows:
  - Running the SDN controller.
  - Deploying the SDN application.
  - Emulating network using Mininet with user-supplied network topology.
  - Running test case.
  - Collecting results.
  - Representing test case result in desired data format.

## 1.4 Contribution

The testbed framework in this thesis is a collection of modules written in Python[2] which provides a mechanism to test the SDN controller from the programming language point of view. Using the testbed created in this thesis, it is easy to accommodate a new controller for comparison purpose. The contributions that have been made in this thesis are as follow:

- An environment to run the experiments with very little manual intervention.
- Test cases such as finding number of flow rules sent by controller, packets handled by controller etc.
- Collecting result into JSON format, it is also easy to extend it for other data formats as well.
- Extensibility to accommodate new controller environment, new test cases and data format.

## 1.5 Thesis Outline

In the chapter 6, we present details about SDN and OpenFlow. This chapter covers the introduction to SDN & its architecture, and working details of OpenFlow protocol. Chapter 3 covers the work which has been done for the SDN testbeds. This chapter includes overview of hardware and software based testbeds along with their incapabilities. Chapter 4 hosts the system design of testbed proposed in this thesis. In the chapter 5, we discuss the implementation details that covers used hardware, a list of softwares, and puesdo code. Chapter 6 is dedicated to evaluation work that shows results for comparison of SDN languages. Finally, chapter 7 elucidates the conclusion and future directions to our work.

[2] https://www.python.org/

# 2 Background

This chapter presents details about computer software-defined networks (SDN) and OpenFlow. A testbed framework realized in this thesis leverages the SDN to run the experiment. A basic understanding of the SDN and related concepts would be helpful in order to get the better understanding of work done in this thesis.

## 2.1 Software Defined Network (SDN)

In computer networks, demands are changing dynamically and rapidly since the days of file sharing, hosting applications on a server, peripheral sharing. Today, companies are using more advanced computer networks such as cloud networks, virtualized servers, remote data storage equipment to meet their needs [23]. These advanced networks require significant computing power, API, labour, resources, power, and proper planning for proper deployment and maintenance. Computing power advanced rapidly but the way of networking virtually never changed. Conventional network approaches have become very complex, closed and proprietary which is a barrier for innovating new services. The main reason behind this limitation is that network devices support ever-increasing number of network protocols and offer proprietary interface only. That is why, it is hard for the network programmers to configure the networks as per their use-case [24].

To solve this serious problem, network researchers came up with idea of software defined networks (SDN). A new architecture was designed that uses the standard API to allow network operator or programmers to configure and control the resources within a network. With the use this standard API, network applications such as cloud computing, email, etc. can easily configure the network and its components as per the requirement [23].

SDN is an approach to manage and control the networks. It decouples the control plane from the data plane. SDN provides a centralized picture of the network where an SDN controller work as the brain. This controller exchanges information with underlying hardware via Southbound APIs and talks to network applications using Northbound APIs. OpenFlow protocol establishes the communication between a controller and switch. It is the most popular protocol used by SDN controllers. It is a standard communication interface [24]. SDN applications uses the API offered by SDN controllers.

### 2.1.1 SDN Architecture and its Components

The layered architecture of the SDN is shown in Figure 1. It shows the distinct data layer, control layer, and infrastructure layer (data layer), with designated APIs (Southbound API and Northbound API) for communication among these components. This is a protocol neutral model for exchanging information across these three layers. Earlier network programmers used provider's way or operations support systems for interfacing the network. However, SDN envisions that network operator's application should have control of network resources with direct access to the SDN controller [12].

#### Data Layer aka Data Plane

One or more network devices such as hardware switch, virtual switch lies within the data plane. Each of this device has a set of traffic forwarding rules or traffic processing resources. Resources in this layer represent the abstraction of underlying physical entities and their capabilities [12]. In the architecture diagram, this layer is shown as infrastructure layer and offers the Southbound API to communicate with the SDN controllers.

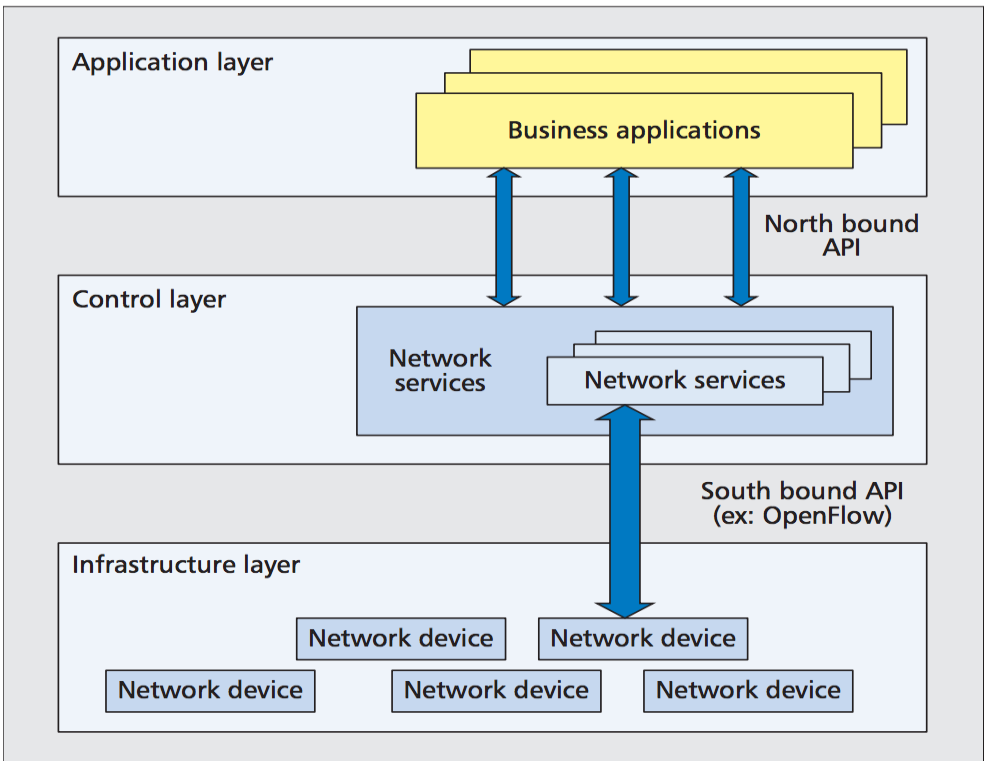



**Figure 1:** SDN Architecture [4]

**Control Layer**

This layer is also known as control plane. The layer is a set of one or more SDN controllers. Each SDN controller in this plane has control over the network resources offered by one or more network elements across data plane. The functionality of the SDN controller is to sincerely execute the requests coming from the application built over it and isolation of these applications. In order to handle requests, an SDN controller may talk to peer SDN controller if necessary. Another function of an SDN controller, which is not essential, is responding to events to recover from failure, and optimized resource allocation [12]. In the architecture diagram, this layer is between the application and infrastructure layer. It leverages the south bound API offered by data layer and offers North bound API to SDN application for communication.

**Application Layer**

This plane may have one or more business applications to control the resources exposed by SDN controller. An application defines that how a network will behave. An application can call or work with another application [12]. This layer leverages the Northbound API offered by the control layer and is shown as the top layer in the architecture diagram.

### 2.1.2 Packet Handling with SDN

Packet handling in a switch is addressed by the flow rules and a flow rule explains the forwarding action which a device should take for all packets matching to the rule. A flow is always unidirectional, for bidirectional there must be two or more flow rules. A flow rule is stored on a device and known as flow entry. A flow table that resides on the network device, contains a set of flow rules and corresponding actions. An action is performed when a packet matches the flow entry [14, 27].

When a packet arrives at the switch interface, this device consults its flow rule table to look for a matching rule. If the device finds a matching rule, then it takes the specified action. In the absence of

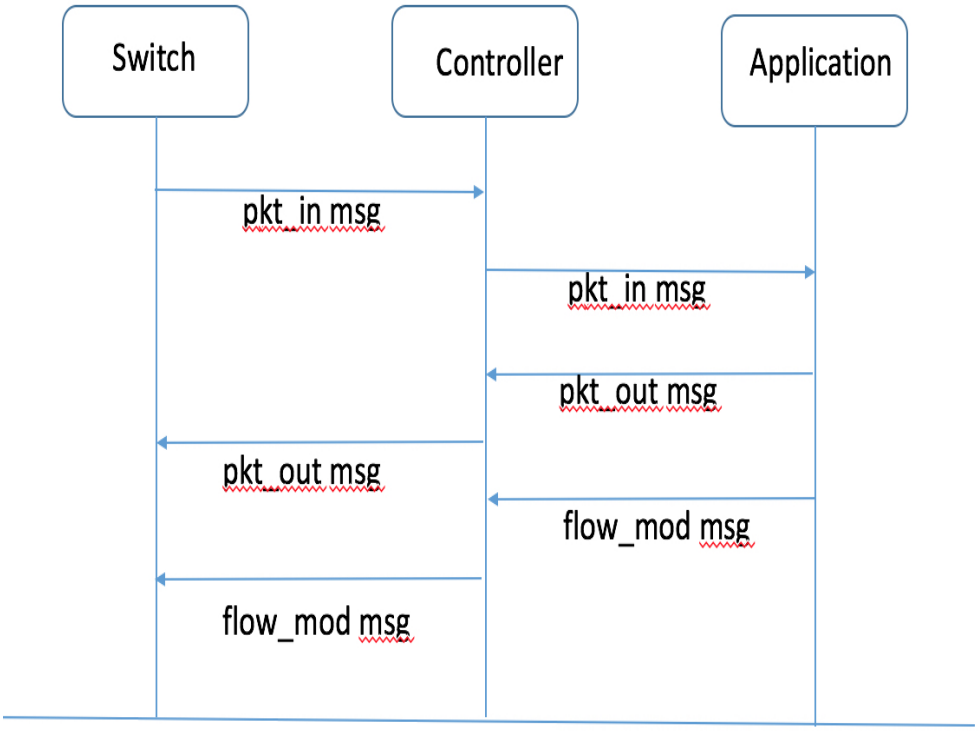



**Figure 2:** SDN Workflow Diagram

matching rule, a request for further instructions is sent to the controller. Figure 2, depicts the typical SDN workflow. When a packet arrives at the switch interface, a request as a pkt_in message is forwarded to the controller, as shown in Figure 2, which in turn forwards it to the network application written on the top of controller. The application receives the request and creates the pkt_out message which is sent to the switch via controller. This network application can also generate flow rule which is a flow_mod message so that packets matching the rule are handled at switch only. Flow rules can be generated in three ways as mentioned below [27]:

1. **Reactive:** In this mode, a controller takes action after receiving a request. First, it inspects the header of packet. After inspection it either decides an action or generates the flow rules.

2. **Proactive:** In this mode, flow rules for all possible traffic matches are populated into the switch. These rules are populated ahead of time.

3. **Hybrid:** This model adapts the flexibility of reactive mode and the low-latency behaviour of proactive mode.

### 2.1.3 SDN Abstractions - Hide Unnecessary Network Details

Abstraction is a way of representing important features while hiding certain details at the same time. In the software engineering domain, abstraction is used to decrease the complexity to come up with a good software design and its implementation. It is used for constructing the large and complex software systems. In computer networks, protocols also have standardized seven layers of model known as ISO OSI model [44]. Network programs are written using proprietary APIs that offers a layer of abstraction to program the network functionality. This section provides the detail about abstractions offered by SDN.

SDN offers three kind of abstractions for the network area: forwarding, configuration, and distributed

state which are explained as follows [5]:

1. **Distributed State Abstraction:** This abstraction offers a way to mitigate the inconsistent distributed state of the network while giving access to the network state. It represents a global view of the network through an API. The control mechanism is also offered as an API which eliminates the need of distributed protocol. A distributed network operating system usually offers implementation of this abstraction.

2. **Specification Abstraction:** It states that control program should express the expected behaviour. This abstraction is not responsible for implementing any behaviour on the physical network. This is also known as network virtualization. An abstract view of the network is given to the control program.

3. **Forwarding Abstraction:** This abstraction offers access to the forwarding model of a network. This abstraction is used to take decisions on the network traffic. OpenFlow protocol is standardize implementation of this abstraction.

With these abstractions, network programmers can build network applications using more general languages where they get an abstract view of the network and reliable communication primitives.

### 2.1.4 SDN Languages - Network Programming Languages

The main goal of this thesis is to compare the existing SDN languages and this section present details about SDN languages and features they provide.

Network programmers often configure network equipment to offer services such as routing, simple packet forwarding, load balancing, access control and monitoring. However, interfaces provided by programmable network devices provide primitive abstractions which make it difficult to perform and manage multiple services [10]. To overcome this limitation, there is a need to raise the abstraction level using existing general purpose languages such as Java, Python, Haskell, oCaml etc. In SDN, OpenFlow provides the forwarding abstractions. However, writing flow-rules using OpenFlow is cumbersome and explained with examples given below in this section.

In the below mentioned examples, network programs are written using NOX platform [15] and OpenFlow to explain applications and difficulties of OpenFlow.

**Listing 1:** Repeater Example

```
def repeater(switch):
p1 = {IN_PORT:1}
p2 = {IN_PORT:2}
a1 = [output(2)]
a2 = [output(1)]
install(switch, p1, a1, DEFAULT)
install(switch, p2, a2, DEFAULT)
```

Let's assume that a network has a topology of a switch connecting two hosts on port 1 and port 2 respectively. In the code listing 1, variables p1 and p2 represent the set of incoming packets. variables a1 and a2 are used for actions, the install function in the code listing 1, installs the flow rule into switch. So when a packet arrives at switch via port 1, it will be forwarded to port 2 and vice-versa.

The first difficulty in implementing programs using NOX is that program composition is not possible [10]. Assume, now we want to extend our code of repeater for counting the web traffic. A typical way to write this monitoring program in NOX is to install another rule which will monitor the web traffic and also count it. A program with such behaviour is represented in code listing 2.

**Listing 2:** Monitor Example

```
def monitor(switch):
p = {IN_PORT:2,TP_SRC:80}
install(switch, p, [], DEFAULT)
query_stats(switch, pattern)
def stats_in(switch, pattern, stats):
print stats['bytes']
sleep(30)
query_stats(switch, pattern)
```

The monitor function in code creates and installs a rule for matching all incoming web traffic from host 2. The stats_in function sends a query to get the counters associated with this installed flow rule and prints it to the console.

Now, let's assume that there is a need to combine the repeater functionality and monitor functionality. On doing that, program would not work as expected due to the overlapping rules. If an http packet arrives, it will be either processed by the repeater function or monitor function. In case of repeater function, there will be no packet statistics on the console. On the other hand, if it is processed by the monitor function, it will break the repeater functionality. Composing two programs is non-trivial and represents the first difficulty of OpenFlow. We have to manually combine these two programs to get the desired behaviour.

The second difficulty is using low-level patterns to describe the packets is error-prone. It makes network programmers to explicitly use many rules with different priorities. It makes reasoning about the program very complicated [10].

**Listing 3:** Repeater-Monitor Example

```
def repeater_monitor(switch):
p1 = {IN_PORT:1}
p2 = {IN_PORT:2}
p2web = {IN_PORT:2,TP_SRC:80}
a1 = [output(2)]
a2 = [output(1)]
install(switch, p1, a1, DEFAULT)
install(switch, p2, a2, DEFAULT)
install(switch, p2web, a2, HIGH)
query_stats(switch, p2web)
```

The code in listing 3 is a combination of repeater and monitor program. This program installs three rules. First two rules represent the repeater functionality and the thdird rule counts and monitor web traffic and also forward it to the port 1. It is necessary to install different rules in a switch with different priorities to solve the overlapping problem.

To overcome these difficulties, it is required to raise the abstraction level so that it can be easy for network programmers to concentrate on network logic only. There exist SDN languages such as Frenetic [9], Pyretic [32], Kinetic [21], Maple [43], Merlin [39], FatTire [39], FlowLog [31], NetKat [1], Nettle [42] and many more to cope up with these difficulties. These languages can be categorized into five

categories. Below, these languages are explained briefly and category-wise.

1. **Imperative:** In this, programs are implemented as a sequence of programming statements. These statements can modify the state [38].

   - *Pyretic* language allows network programmers to concentrate on how to write network policies using high-level abstraction without worrying about low-level OpenFlow perspective. With pyretic, a programmer can write policy for the entire network instead of writing physical rule one by one. Network policy is specified via a function that takes set of located input packets and returns located output packets. Located input packets are packets with incoming location and output located packets are packets that might have modified state with a new location. Pyretic uses this approach to forward the packets and frees the programmer from the worry of OpenFlow rules to move packets. Pyretic offers the modular programming. Using modular programming, a network can easily merge the different policies without worrying about overlapping rules. Table 1 lists the available Pyretic policies. Pyretic also offers composition operators and with these operators, programmers can perform parallel composition or sequential composition [32].

     Parallel composition allows two policy functions to work in parallel. It applies both functions on the located packet and returns the combined result. For example, in code listing 4, two policies are combined using (+) operator. Packets which are destined to host with IP address 10.0.0.1 will be forwarded out to port 1 while those destined to 10.0.0.2 will be forwarded out to port 2. Sequential composition operator (») is used to compose policies sequentially. In listing 4, it is used twice. Pyretic offers other operators as well such as negation (~), conjunction (&) and disjunction.[3]

     **Listing 4:** Pyretic parallel composition

     ```
     R = (match(dstip='10.0.0.1') >> fwd(1)) +
         (match(dstip='10.0.0.2') >> fwd(2))
     ```

     **Table 1:** Pyretic policies

     | Syntax | Summary |
     |---|---|
     | identity | returns original packet |
     | none | returns empty set of packets |
     | match(f=v) | returns none if do not match else identity |
     | modify(f=v) | return packet with value of field f equal to v |
     | fwd(a) | forwards packet to port a |
     | flood() | sends copy of packet to each port |

   - *Kinetic* is s domain specific language that allows developing network control programs for capturing the responses during changing network conditions. It is an extension of Pyretic language and it expresses the network policies as a finite state machine (FSM). In FSM, states represent the distinct forwarding behaviour and events change the state. Kinetic has an event handler which is responsible for listening events and triggering transitions in policy. This change in policy updates the data plane. It is possible with Kinetic to verify that if changes in network behaviour conform to correctness or not [21]. For example, a network operator

[3] https://github.com/frenetic-lang/pyretic/wiki/Language-Basics

wants when intrusion is detected, a specific part of a network should not accessible. Finite state machine for such system may look like as shown in Figure 3. As mentioned previously,

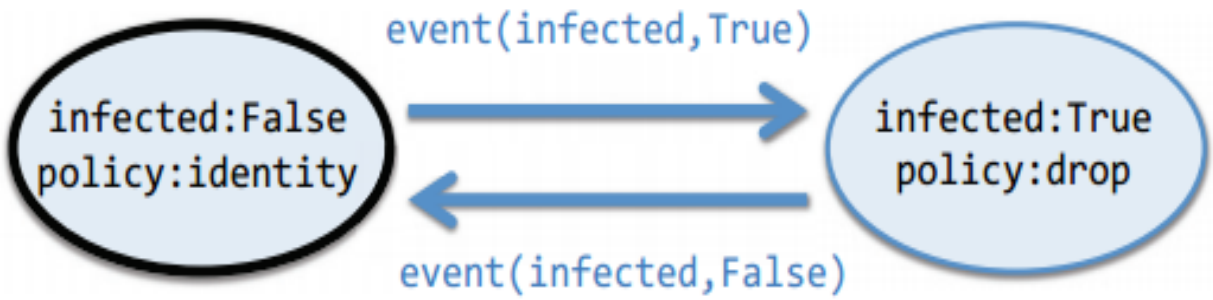



**Figure 3:** FSM for intrusion detection [21]

it is an extension of pyretic language. All operators, policies that are available in pyretic can be used in kinetic program.

2. **Functional:** In this programming paradigm, mathematical functions are used for the computation and often have immutable state [38].

   - *NetKAT* is a domain specific language and belongs to the Frenetic family[4]. The Frenetic family is a suite of network management tools. Kleene algebra with tests (KAT) is the basis of this language. Kleene algebra with tests is a system that is used for reasoning about the partial correctness. KAT is based on Kleene algebra which is an algebra of regular expressions. NetKAT offers primitives for modifying and testing the packet headers. It also offers the functionality of encoding network topologies and reasoning about these constructs. Regular expressions can encode network topologies and depict many reachabilities and offer queries such as security query. These queries can be verified automatically with NetKAT. It is a viable tool for the verification of SDN programming [1].
   - *Merlin:* It is a framework to manage the SDN resources. Merlin allows network administrators to state high-level network policies using a declarative language. This language offers logical predicates that can be used to identify set of packets. These predicates can also be used for regular expressions in order to encode forwarding paths. Arithmetic formulas can also be formed using these predicates to specify constraints on network bandwidth. The merlin compiler portrays these policies into constraint problems. These problems govern bandwidth allocation. It then produces the code that can be used to work on network elements for policy enforcement [39].

3. **Logic:** This programming paradigm is used for writing programs using formal logic [38].

   - *FlowLog:* It is a tierless language for writing programs for SDN. Usually SDN languages provide separate abstraction for each tier i.e. data plane, control plane and controller state. Flowlog offers a single unified abstraction for all these three tiers. FlowLog syntax resembles both SQL and rule-based programming languages [31]. However, this language provides support to program with mutable state. The expressivity of FlowLog is limited to built-in verification and supports proactive compilation. External libraries can be used through callout in order to compensate the limited expressivity. This language compiles forwarding behaviour to

[4] http://frenetic-lang.org/

underlying switch tables proactively. To maintain the controller state using rules, the compiler of FlowLog makes a switch to transmit minimum amount of necessary network traffic to the controller. FlowLog programs can be compiled to Alloy[5]. Alloy is a famous verification tool and can be used to verify program properties and correctness [31].

4. **Data Flow:** In these languages, programs are written as a directed graph of operations [38].

   - *Flow based Management Language:* It is a high-level declarative language which is used to express network policies for many different network management tasks in a single and cohesive framework. FML is a simple programming language that can be used to represent common network configurations. This language is suitable and efficient for large networks. Using FML, policies can be specified for flows. A policy in FML is a set of statements and each statement represents the constraints that apply to the matching flow [18].

5. **Functional Reactive Programming:**

   - *Nettle:* Implementation of this language exists in Haskell language. More information on this can be found in the section 2.1.5.

Some of these languages such as Merlin are conceptual only and do not provide any concrete implementation to try out the programming in SDN network. In our thesis, some of these languages are used for comparison purpose. These languages are chosen on the criteria of concrete implementation availability and development status. SDN languages which were not developed further in last two years were not considered for comparison in this thesis.

### 2.1.5 SDN Controllers - Control over Traffic in Network

As mentioned earlier, the main goal of this thesis is to compare SDN languages. In order to achieve that we have few SDN applications that leverages API offered by SDN controllers. SDN controllers communicate with both switch and application to control the network. This section discusses the details about various SDN controllers and their features.

A controller is responsible for adding, removing or modifying flow-rule entries in the flow table of switches deployed in a network. A static controller simply installs flow-rule entries to inter-connect computers. On the other hand, dynamic controllers have the capabilities to add, remove or modify flow-rules dynamically depending on the requirement [28].

We studied many controllers such as Frenetic, Pyretic, Project FloodLight, Nettle etc. It was not possible to go through all the available controllers due to time constraints. Therefore we shortlisted controllers that are still maintained. Even then, it was not possible to include every popular controller such as OpenDayLight [29] controller, ONOS [2]; these were not chosen because it was cumbersome to add these controllers to the created testbed framework. This testbed offers extensibility and these controllers can be added later. We finally chose five controllers (Frenetic, Pyretic, Kinetic, Project FloodLight and Ryu) out of shortlisted controllers.. A brief introduction of shortlisted controllers is given below.

#### Ryu

It is an on-going open-source SDN platform for writing SDN applications. It offers set of well-defined APIs and libraries that are useful for writing SDN applications. It is completely written using Python language. Applications for this controllers are implemented using Python language only. The goals behind

[5] `http://alloy.mit.edu/alloy/index.html`

developing this framework are as follow [35]:

- To have a standard SDN controller for cloud networks.
- To become a default controller Linux distributions such Ubuntu, Fedora, RHEL.
- Providing high quality for commercial installations.

Ryu offers many features which are as follow [35]:

- Vendor neutrality.
- Supporting standard open interface such as OpenFlow.
- Providing agility while writing SDN applications.
- Readily available components such as Topology viewer that are useful for building SDN applications.

With Ryu, it is also possible to write new components, modify existing component and combine components. Ryu does not only supports OpenFlow protocol but also other protocols such as OF-Config, NETCONF etc [35].

**Frenetic**

It is a high-level SDN programming language which along with frenetic runtime system works as an SDN controller. Applications for this can be written using either oCaml or Python. Frenetic has following features [9]:

- Declarative syntax like SQL.
- Offers various policies such as filter policy.
- Policy composition mechanism.
- Modules can be composed.
- Single tier - every packet in the network is available for analysis purpose.
- It can operate in both proactive and reactive mode. These modes are explained in
- Provides query to reason about network and get network statistics.
- Race free semantic.

**Pyretic and Kinetic**

These are also high-level SDN programming language which along with pyretic runtime system works as SDN controller. Application for both of them are written in python language. Features of these languages have been already explain in the section 2.1.4.

**Project FloodLight**

It is an open-source project published under Apache-license[6]. This Java-based SDN controller has a community of developers that also includes engineers and developers from Big Switch Networks. Big Switch networks is big company which offers automated network solutions. FloodLight applications are written in Java language. Main features offered by Project FloodLight are as follow [16]:

- It has module loading system which makes it simple and extendable
- It can be deployed with minimal number of dependencies
- It supports a wide range of both virtual and physical OpenFlow switches.
- It can work with both OpenFlow and non-OpenFlow networks at same time.
- It leverages the power of multithreading for high-performance.
- It also supports the cloud orchestration platform i.e. OpenStack [36].

**Open Daylight**

ODL is a modular and open-source SDN platform suitable for networks of any size. It offers network services across a range of hardware devices in multivendor environment. OpenDayLight applications are also written in Java like Project FloodLight. It offers following features [29]:

- Micro services can be configured, used and chained together to satisfy the network needs.
- Provides multi-protocol support.
- Security, stability, scalability.
- Policy & intent offers abstraction for network programmability.

**Nettle**

It is also an open-source project which offers domain-specific language to control the network using OpenFlow protocol. It follows the mantra, "Don't configure the network, program it!". It leverages discrete and continuous features offered by functional reactive language i.e. Haskell. Using discrete function, it is easy capture control messages between switch and controller and continuous function can be used to express dynamic behaviour of the network [42]. Applications for this controller are written in Haskell language.

### 2.1.6 NetIDE

During the study of SDN languages and their corresponding controllers, we came across a framework named **NetIDE**[8]. It solves the problem of depending on Northbound interface of a SDN platform which looks like a vendor lock. NetIDE is not an SDN controller platform, it provides an environment where multiple controllers can operate together. In this platform, an SDN application written for one controller can also work for another controller [8]. We found it interesting to study it further because it allows porting one application written to its specific controller to another controller.

It follows the WINE architecture[7]. WINE software allows windows applications to run in Linux environment. There is a penalty of executing one application in an alien environment. However, it is

[6] https://www.apache.org/licenses/LICENSE-2.0.html
[7] https://www.winehq.org/docs/winedev-guide/part-two

tolerable considering the efforts for writing the same application again for another controller. Major benefits of NetIDE are as follows [8, 7]:

- Rapid deployment of already written SDN languages without worrying about the underlying controller.
- Offers applications composition mechanism (Parallel and Sequential)

We did not use this in our testbed framework and reasons are as follows:

- It does not support for every available SDN controller
- And extending it for the set of desired controller requires huge efforts and time.

In our case, deployment of selected controllers and writing applications for them was more feasible, easier, and time-saving. One of the objective of this thesis to compare SDN languages and to achieve that we have few applications for these controllers. The main goal of NetIDE is to save the time of re-writing same application for another controller. However, in this thesis it is needed to have same application for every controller and this goes against the NetIDE main goal. For this reason, instead of extending NetIDE for selected controllers, decision to have same application for each selected controller was taken. This saved us time and efforts as well.

## 2.2 OpenFlow - A Protocol for SDN

In our thesis, OpenFlow protocol is used to establish communication between a controller and the (virtual) switch. This protocol is also used to realize test cases such finding number of flow rules in a switch etc. In this section, a brief detail of OpenFlow protocol and its working is presented.

OpenFlow is a way for network administrators or programmers to manage networks. The goal of OpenFlow is to provide an open and vendor-agnostic interface for innovation in the networking. OpenFlow specification explains a standard collection of features that network device such as switches must offer. These devices must also offer an interface that controllers may use communicate. OpenFlow also includes the instructions which can be used by controllers in order to install, delete or modify the forwarding rules. These instructions are also used to get the information about topology, flows and traffic statistics [28]. It is a protocol which is defined for managing the configuration state. A controller in SDN speaks this protocol to manage and control the switch state and also respond to OpenFlow events.

### 2.2.1 Communication in Network using OpenFlow

A switch that supports OpenFlow maintains one or more forwarding tables and each table stores a list of rules with priority. Each rule in the list represents a pattern that basically explains the set of packets it will match to and a list of actions that will be applied to these matched packets. When a packet arrives at a switch, the switch looks for a rule whose pattern matches the headers of the incoming packet. If a matching rule is found, then corresponding action is taken by the switch. In the case of multiple matching rules, the switch chooses the rule with the highest priority and take the action. In case, no matching rule is found, the switch wraps the packet in an OpenFlow message. After that, this message is sent to a controller. The controller can directly handle and process the packet. It can send an OpenFlow message that instructs the switch to install, delete or modify rules in the flow table.

The capacity of a flow table is hardware dependent. However, usually switches offers sufficient storage for thousands of rules. To keep track of traffic statistics such as number of bytes or number of packets processed with a specified rule, every rule associates this statistics information with it. This information can be read by controllers using OpenFlow message [5].

Rules can also be used to configure the physical ports of a switch. It is also possible to create queries which can rate-limit the traffic or offer minimum bandwidth guarantees. These queries can be useful for writing the traffic engineering network applications [5].

**Table 2:** Example of flow table of a switch

| Pattern | Priority | Action | Statistics |
|---|---|---|---|
| TcpDstPort = 80 | 200 | drop | 2000 |
| IPDstAddr = 10.0.0.1 | 100 | Output:1 | 1987 |
| IPDstAddr = 10.0.0.2 | 100 | Output:2 | 8881 |
| IPDstAddr = 10.0.0.1, TcpDstPort = 80 | 400 | Output:1 | 889 |
| * | 10 | Controller | 0 |

Table 2 represents a flow table of switch, where the first column describes the rule pattern, the second column represents the priority of a rule, the third column shows an action for a rule and the last column gives the statistics information. If we read the table from top to bottom, then it states as follow:

As per first rule, any http traffic will be simply dropped. The second rule matches those packets going to host with IP address 10.0.0.1. These packets will be forwarded to port 1 of the switch. The third rule is similar to the second rule but identifies packets going to host with IP address of 10.0.0.2. The Fourth rule overlaps with the first rule and states that http traffic going to host with IP address 10.0.0.1 will be forward to port 1. Fourth rule has the higher priority, that is why it will have preference over the first rule. The last rule instructs to send all packets to controller. Therefore, when no matching rule is found, then packets will be sent to the controller.

### 2.2.2 Traffic Forwarding using OpenFlow Switch

In SDN network, communication is initiated from the switch. Flow rules generated by a controller are also stored in switch. In our thesis, OpenFlow switches are used for handling packets in network. This section concisely presents the information about OpenFlow switch and way it handles an incoming packet.

An OpenFlow switch stores a group table and one or more flow tables. This switch is used for packet lookup and processing. A switch is connected to an external SDN controller via an OpenFlow channel. A controller controls and manages the switch by leveraging the OpenFlow Protocol. By leveraging the OpenFlow protocol, the controller can add, delete and update flow rule entries stored in the flow tables. A flow rule is a combination of matching fields, priority, action and counters and a flow table holds the flow rules. Flow-rule matching is initiated at the flow table and may continue to other flow tables. Flow entries in the flow table match the packets in the order defined by priority field. If a matching entry is found in a flow table, then associated action is taken on the identified set of packets else packet is sent to other flow tables. If no match is found in any table, then it is sent to the controller as an OpenFlow message [17].

Each flow entry associates an instruction that either modifies pipeline processing or a set of actions.

An Instruction describes the packet forwarding, group table, and packet modification. Pipeline processing instruction instructs packets to be directed to subsequent flow tables for further processing. This pipeline processing is stopped when the next table is not specified in a matching flow. At this stage, packet is either usually forwarded, modified or both. Flow rule entries forward packets to a port which could either be a logical port or a physical port. Actions specified in a flow entry can also send packets to a group table that specifies set of action such as flooding or more complex forwarding semantics. A group table stores group entries and each entry in this table is a list of buckets. A maskable register value known as Metadata is used to carry packet from one table to another table [17].

Figure 4, shows the packet processing in an OpenFlow switch. Packets arrive at ingress port and

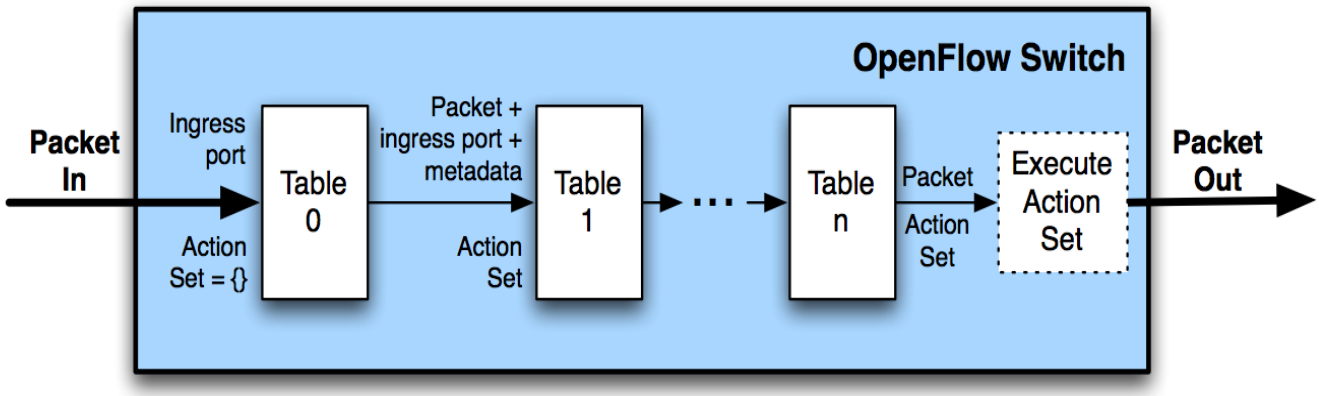



**Figure 4:** OpenFlow Switch - Packet Processing [17]

initially look for instructions in table 0 and then in subsequent tables. And a set of actions are performed at last stage in switch.

## 2.3 Mininet - Network Emulator

Mininet is a network emulator developed with Python. It is a suitable framework to deploy a required network of hosts for experimentation purposes [26]. Mininet can be used to prototype the real network of desired topology on a single machine. It provides Python based interface to developers in order to run experiments on desired node. This emulator is chosen because it is open-source, provides a Python based interface. In this testbed, this interface is leveraged for experimental purposes.

## 2.4 Wireshark - Capturing Packets in Network

Wireshark[8], an open-source tool, is used to analyze the network packets. It tries to capture packets in a network and display the packet information as detailed as possible. This software tool can be used to examine the network. Network administrators can use this tool to identify the problems in network. Protocol developers can use it for debugging purpose and other people can use it to learn internals of network protocols [25]. It offers following key features:

1. It can capture live packets.
2. It can import/export packets from/to a dump file.
3. It displays detailed information about packet.

[8] https://www.wireshark.org/

4. It can filter or search packets on many criteria.

5. Various statistics can be obtained using it.

Wireshark offers a graphical user interface to interact with it. *tshark* is a command line oriented version of Wireshark. It is used in the scenarios when a GUI is not required or available. It provides the same options as Wireshark [25]. In our testbed, it is required to capture packets to collect the results for test cases such as OpenFlow messages handled by a controller etc.

# 3 Related Work

Testbeds are usually used for testing scientific theories, new technologies and computational tools. It also helps to explain the experimental research. Network vendors, operators and the academic institutions all are evaluating and testing the SDN. There are numerous testbeds available to test the SDN in terms of scalability, in the domain of various applications, network performance etc. These testbeds provide a platform to practice the SDN concepts. It becomes quite easy for conducting intensive, transparent and replicable testing of scientific concepts [34].

In this thesis, a testbed for SDN is developed to compare SDN languages. However, there exist many testbeds for SDN to perform various experiments. In this thesis, these testbeds are categorized into two categories: Hardware based testbeds and Software based testbeds. Former testbeds mainly focus on the network infrastructure whereas software based testbeds target the control plane of SDN i.e. SDN controller. The chapter summarizes these testbeds in the sections 3.1 and 3.2. This chapter also mentions details how they are different from the one created in this thesis. This chapter ends with comparison among these testbeds and a conclusion.

## 3.1 Hardware Based Testbed Solution

In this section, we describe the hardware based testbed solutions and represents how they differ from the one created in this thesis.

These testbeds offer the network infrastructure to evaluate the network services implemented using SDN approach. These testbeds are typically used for evaluating the SDN concept in a realistic environment. For brevity, only two testbeds are briefly summarized below and details are also mentioned how these testbeds are not efficient to compare SDN languages. The first testbed GENI OpenFlow [3] which is quite popular for at-scale networking. The second testbed i.e. OF@TEIN [22], this is an OpenFlow testbed that runs over the TEIN4[9] network. These both testbeds use the OpenFlow to realize SDN concept, that's why these are chosen for further explanation.

**GENI OpenFlow**

GENI (Global Environment for Network Innovation) uses emerging network technologies such as SDN, OpenFlow, and network virtualization to offer innovations in the network and at-scale experiment related activities. For implementation, it has used the spiral evolution development model. The End-to-end prototype was created in first two stages. This prototype was developed for continuous experimentation. During the third stage, it extended the support for OpenFlow and installed OpenFlow switches in the experiment environment. OpenFlow in GENI offers a real environment for executing large network experiments to users. It realizes virtualization and deep programmability; these are key concepts of GENI. Deep programmability offers the functionality to centrally define the policies on the network [19, 3]. GENI OpenFlow can be accessed via its portal. [10]

There are three key technologies in GENI: 1) Control, 2) Network Technologies and 3) Slicing technologies, these are defined as follow:

- *Control Framework/Management Technology:* GENI is built using a federated model for large scale and coherent infrastructure. It consists of a range of network resource and each network resource belong to a different provider. Each provider is responsible for managing its own resources and

[9] https://ec.europa.eu/europeaid/regions/asia/tein-3_en
[10] http://www.geni.net/

offers the ability to use resources by experimenters. GENI's Control Framework (GCF) involves four components: GENI Meta-Operations Center (GMOC), Clearinghouse, Tools, and Aggregates. These components communicate with each other in an orchestrated manner.

- *Networking Technologies:* GENI offers many network technologies depending on the existing network and need of experimenter. In Figure 5, an inter-campus connection is used to show a simple network topology to compose an OpenFlow testbed using data plane, and the control & management plane. The control & management plane is responsible for configuring and controlling the experiments performed in the data plane. Users, through SSH connections, can access the slices.

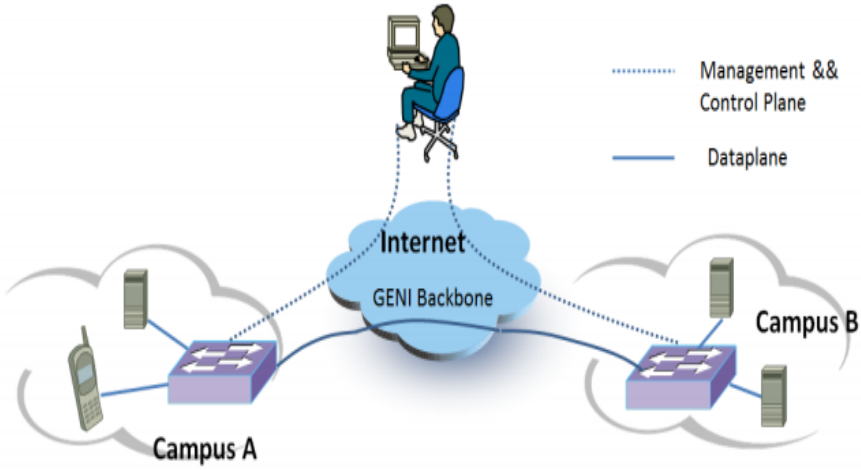



**Figure 5:** Two campus networks interconnected by GENI backbone network [19]

- *Slicing Technologies:* GENI uses the sliceable concept to offer virtualization in order to run multiple experiments simultaneously on shared resources. Traditional networks use the VLAN technology to offer virtualization but GENI uses OpenFlow-enabled network components to provide flexible network virtualization. These components comprise a proxy between switch and OpenFlow controllers (shown in Figure 6) to allow multiple controllers to use the same switch.

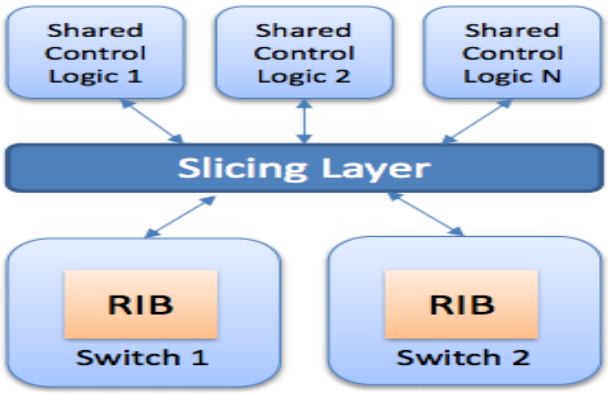



**Figure 6:** Slicing [19]

With GENI OpenFlow testbed, one can run the experiments where a distributed network infrastructure and a specific traffic is required. But it has some limitations which are as follow:

1. An experimenter can not have complete control on network infrastructure.
2. An approval is required to run the experiment
3. For an experimenter, changing network topology is not possible.
4. Network resources are shared in this testbed and it would not be possible to get correct results for some experiments such as flow-rules installed into a switch by the controller.

The testbed created in this thesis does not have such limitations and an experimenter has the full control over it.

**OF@TEIN**

OF@TEIN [22], is a Korean Government funded project that was created to construct and maintain an OpenFlow testbed over TEIN4(Trans-Eurasia Information Network 4) network. It was developed with collaborative efforts of Korean universities. Its main objective was to have a shared OpenFlow-based SDN testbed among Korea, South Asia, South-East Asia. To achieve this, a special hyper-convergent SmartX Box [33] was deployed. This SmartX Box was deployed at each site over a TEIN4 network infrastructure. It has achieved its main three tasks: designing of SmartX Racks, site deployment and interconnecting the sites, and implementation of SDN tools. It is not open for the public to use but may be accessed through National Research and Education Network (NREN) [22, 19]. Key technologies for this testbed are following:

- *Management Technologies:* It is based on the OFELIA Control Framework (OCF) [41] and has a portal which provides an experiment UI to create and monitor the status of various running experiments. This UI, network resources and SmartX Rack can be aggregated to offer resources to experiments. The lifecycle of an experiment in OF@TEIN has three stages: Setup, Execution, and Finish. In the setup stage, experimenters use the UI to setup their experiments. In the execution stage, experimenters execute their experiments, and in the last stage they finish the experiments and release the occupied resources.

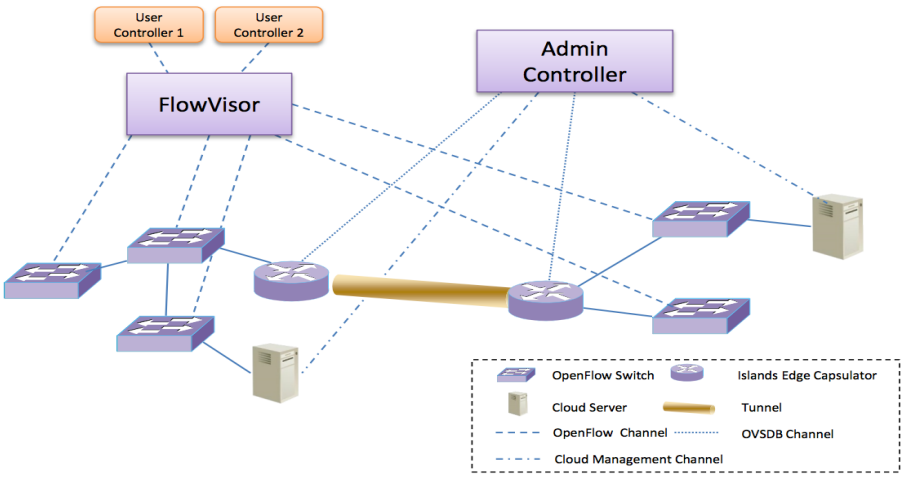



**Figure 7:** OF@TEIN Architecture [19]

- *Networking Technologies:* The architecture diagram of OF@TEIN is represented in Figure 7. Connections among sites are implemented using NVGRE tunneling, as shown in Figure 8. NVGRE is a method for network virtualization to create the large number of VLANs. There are many SmartX Racks deployed in each site of OF@TEIN. In these racks, there are either OpenVSwitch or OpenFlow data plane which is connected to VM servers. In OF@TEIN, the control and management framework is designed as overlay network. Along with OpenFlow channel, network administrators can manage the NVGRE tunnels using OVSDB[11] management protocol.

- *Slicing Technologies:* FlowVisor [37] is used by OF@TEIN to share the network infrastructure among multiple experimenters using the VLAN based slicing scheme. Each slice has DPIDs of switches and range of Port-IDs. All packets in the slice are tagged with same VLAN ID at the ingress switch and VLAN ID is stripped off at the egress switch. FlowVisor identifies the VLAN ID via intermediate switch and forwards the packet to the correct controllers.

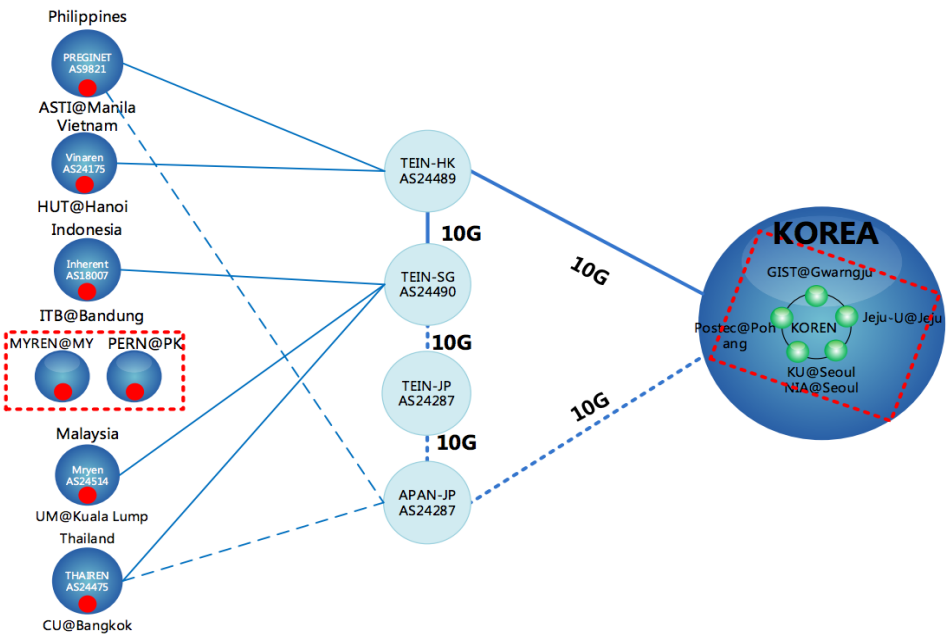



**Figure 8:** OF@TEIN Site Connections [19]

As mentioned above, OF@TEIN is a shared SDN testbed and uses VLAN technology to achieve the virtualization. However, this virtualization is an obstacle as well. For example, an experimenter wants to find out all the rules installed in all the switches. For this experimenter, result will be incorrect because there are chances that another experimenter is also using those switches at the same time. OF@TEIN provides the UI to setup the experiment. However, in order to run the same experiment for multiple controllers and multiple network topologies, an experimenter needs to set up the experiment every time which is a time wasting process. Using testbed created in this thesis, an experimenter can write a python script to specify all these details.

As mentioned earlier, there exist many hardware based testbeds which offer the shared or dedicated network infrastructures for performing various experiments on the network but using a hardware based testbed is expensive. These testbeds mainly focus on the network infrastructure i.e. data plane in SDN and do not provide flexibility to run the controller specific test cases. In the section 3.2, software based

[11] Open vSwitch Database Management Protocol (OVSDB) is used to manage Open vSwitch implementations

solutions are discussed which targets the SDN controller.

## 3.2 Software Based Testbed Solutions

In this section, we explain the software based testbed solution. This section also presents the reasons why these testbeds are not suitable to achieve goals of this thesis. The testbed created in this thesis is also software based and focus on controllers.

There exist software based testbeds which may emulate the network such as Mininet, it is already defined in the section 6. There exist software based solutions for running test cases about SDN controller and Cbench is one of the example. Below, Cbench, NSTAT, and OFNET are summarized briefly.

**Cbench**

Cbench[12](controller benchmarker) is a piece of software for testing the OpenFlow controllers. It emulates OpenFlow switches which talk to SDN controller, forward *packet_in* message to the controller and watch for flow rules to get installed. If one thinks that a minor change may end up in performance impact, then this program can be useful to measure it. Cbench can work in two modes: latency mode and throughput mode. It opens OpenFlow sessions (equivalent to the number of switches) to the controller. In latency mode, for each session, it sends a *packet_in* message to the controller and waits until a matching flow mode comes back. This test can be repeated for multiple times. In throughput mode, for each session *packet_in* messages are queued until the buffer is full and then count flow mod messages.

Cbench is mainly used to benchmark the controller's performance in terms of latency and throughput. It emulates switches and assumes that a controller and its application is running. In a scenario, when an experimenter wants to test the number of flow rules generated by different controllers for different network topologies, manual intervention is required to start/stop the controller and application. These things put an extra burden on the experimenter and waste the time as well. With testbed created in this thesis, a user can specify all these things together for once to save the time and efforts.

**NSTAT**

NSTAT is a testbed environment which is written in Python. It is easy to write and run the stress test cases for SDN controller in highly configurable and modular manner. It also offers the functionality to run its basic components (Southbound emulator, Northbound generator, and controller) on different nodes. Each component can be created, started and monitored on a physical or virtual node. NSTAT itself can also run on a different node. Nodes are assumed to be connected to each other using SSH.[13]

Figure 9, describes the high-level architecture of NSTAT. As explained, this testbed mainly focuses on stress test cases for controllers. However, it provides no information about writing new test cases which make it incapable to achieve goals of this thesis. The testbed created in this thesis has a separate module for writing new test cases. An experimenter can write new test cases easily by using this module.

**OFNET**

This toolset provides an OpenFlow-based network emulation platform like Mininet. It offers exciting features like visual debugging, traffic generation, a temporal view of auto-generated sequence diagrams (Figure 10) of OpenFlow messages exchanged between a switch and controller. SDN controllers are categorized on the basis of their performance. The author of OFNET states that it is an open-source project with Apache license and code will be published soon on Github repository.[14]

[12] `https://github.com/mininet/oflops/tree/master/cbench`
[13] `https://github.com/intracom-telecom-sdn/nstat`
[14] `http://sdninsights.org/`

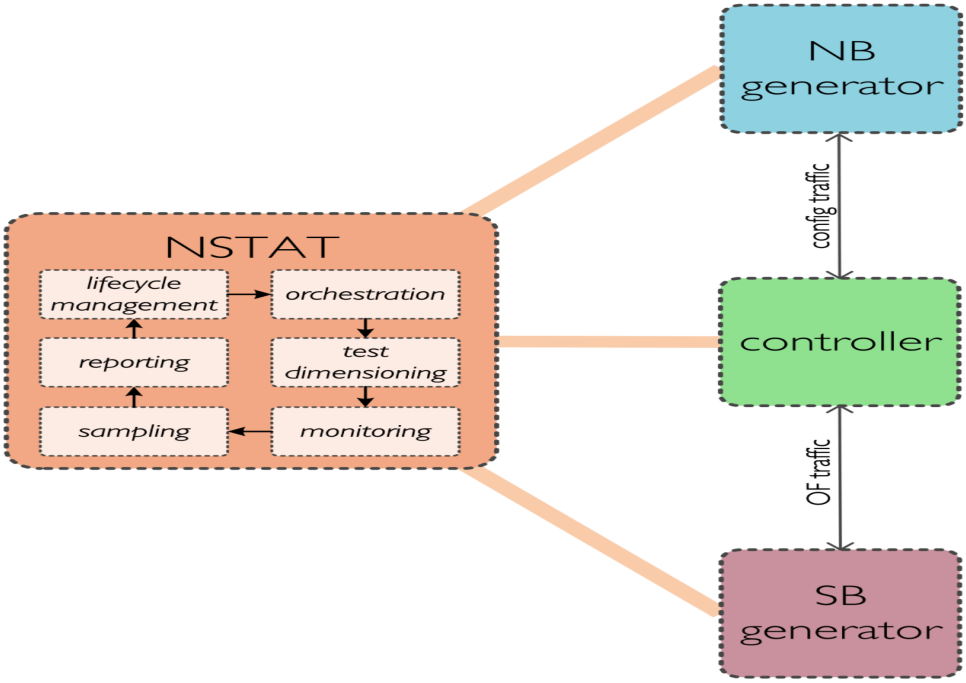



**Figure 9:** NSTAT high-level architecture, each component is running on a different node [13]

This testbed is pretty much similar to our testbed. However, it does not contain any information

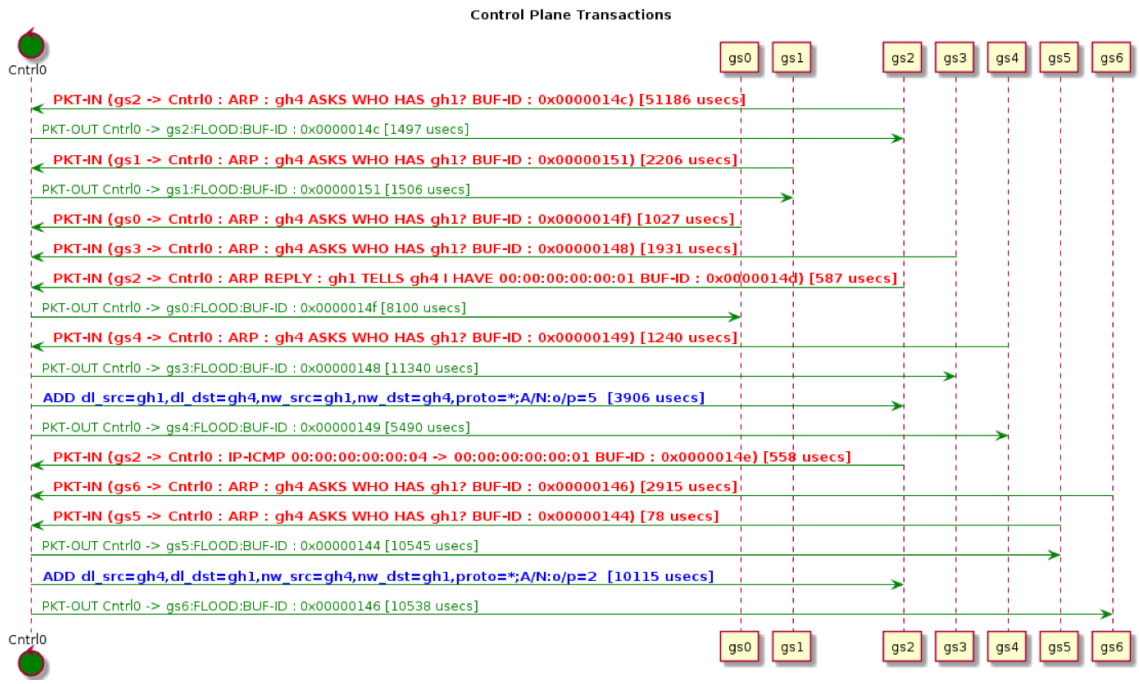



**Figure 10:** Sequence diagram of auto-generated OpenFlow messages between switch and controller [14]

about its extensibility. Also, at the moment, there is no code available for this. This testbed also does not specify that which kind of test cases it can handle. In contrast to this testbed, our testbed has separate and independent modules which can be extended. Using our testbed framework, one can develop any type of test case related to data plane and/or control plane.

**A Testbed for Research and Development of SDN Applications using OpenFlow**

This was a thesis in Universidade de Lisboa. A testbed for SDN application was developed for academic analysis and research. The objectives of this thesis were as follow [13]:

1. Creating the network with OpenFlow support.
2. Configuration and monitoring of the network.
3. Identifying and implementing use cases in a test scenario.
4. Extracting results and analysis.

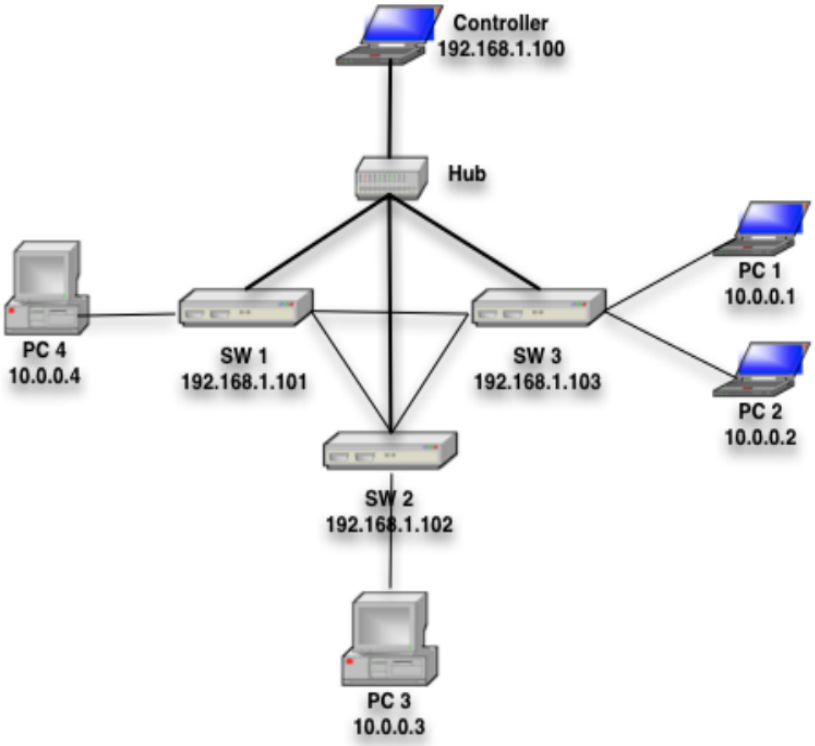



**Figure 11:** Network Topology [13]

Network topology used by this testbed is shown in Figure 11, where there are three OpenFlow switches and computers connected to these switches, and a POX [20] controller running on a laptop. The controller in this topology is connected to all switches using network hub. They compared the traditional and SDN networks with many test cases which were realized by capturing the packets with the help of Wireshark [25, 13]. Using these captured packets, measurements for packet transmission time, latency, response time, and throughput were taken. Measurements were taken for both modes (reactive and proactive) of the deployed POX controller [13].

As explained, the testbed created in the thesis[13] compares the traditional network with SDN network. It captures the packets to realize the test cases like our testbed. However, there is no information about running the testbed with another controller. There is also no information over changing the network topology. In our testbed, a user has the flexibility to run test cases on desired SDN controller and network topology.

## 3.3 Comparison

In table 3, a comparison among discussed related work is given. This comparison represents various information about work done previously. For example, it shows that if work done is software based or hardware based etc.

| | **GENI OpenFlow** | **OF@TEIN** | **Cbench** | **NSTAT** | **OFNET** | **Other** [13] |
|---|---|---|---|---|---|---|
| **Hardware Based** | Y | Y | N | N | N | N |
| **Software Based** | N | N | Y | Y | Y | Y |
| **SDN language abstraction** | N/A | N/A | N | N | N | N |
| **Programming Paradigm** | N/A | N/A | N | N | N | N |
| **Multi-controller support** | Y | Y | N | N | N | N |
| **Controller performance** | N | N | Y | Y | Y | Y |

**Table 3:** Related work comparison

## 3.4 Conclusion

The systems that are explained above, use different approaches for the SDN testbeds. Table 3, summarizes the features of these systems. It can be inferred from the table that which testbeds are hardware based and which are software based. Hardware based solutions target the SDN network infrastructure and do not offer a way to run experiments about controller from the language point of view. On the other hand, software solutions basically measure the controller performance. However, it is not possible to test multiple SDN controllers with these or they target the controller performance. In this thesis, we present an automatic and extensible testbed for multiple SDN controllers which can be used for running various test cases such as flow-rule count, packets handled, etc. It is also very easy to add another SDN controller for testing purpose. A user can also extend it to get the output in various desired data formats such as JSON, XML etc.

# 4 System Design

For developing new SDN language features, a comparison is usually desired with existing languages. This helps in showing that how new features enhance the language expressivity while yielding at least the same execution performance. However, presently it is strenuous to compare SDN languages. These SDN languages offer simple example applications that can not be compared straightforwardly. To address this problem, we present a new modular SDN testbed that can compare SDN languages with respect to expressivity and performance. The presented testbed in this thesis also has a set of sample applications that can be leveraged for comparing the SDN languages for benchmarking and abstractions offered by them.

In our testbed framework, there are two main modules and each module further has one or more submodules. This chapter describes the use case scenarios, challenges, requirements, assumptions, extensibility of the testbed. It also gives a detailed information about system architecture, modules and state chart diagram of create testbed.

## 4.1 Use Case Scenarios

Existing testbeds provide the way to conduct SDN related experiments on a real or virtual infrastructure. However, these do not offer extensibility and flexibility to run experiments for comparing SDN language specific test cases. For example, an experimenter wants to find out number of OpenFlow messages sent to the controller by a switch for a specific network application (e.g. Learning Switch).

For a single controller, it is simple and straightforward to run the testcase but it can be cumbersome in a multi-controller environment. We have identified four different use case scenarions where our testbed can work. These scenarios are explained as follows:

1. **One Controller and One Test Case** - When a user of this testbed wants to run a test case for a single controller. This use case would happen rarely because the user would be interested in comparing the multiple SDN languages. It can be useful in cases such as the user wants to find the packets handled by a controller in reactive mode and pro-active mode.These modes are explained in the section 2.1.2.

2. **One Controller and Multiple Test Cases** - This use case represents the scenario when a user wants to execute multiple test cases for same controller. For example, the user wants to know received *packet_in* messages, generated flow rules and *packet_out* messages.

3. **Multiple Controllers and One Test Case** In this scenario, a user can execute a test case on multiple controllers. This can be useful when the user wants to compare controllers for same test case. For an example, the user wants to know the number of flow rules generated by specified controllers (e.g. Ryu and FloodLight) for switch learning application.

4. **Multiple Controllers and Multiple Test Cases** This use case can be useful when a user wants to execute multiple test cases over multiple controllers. For an example, the user wants to know OpenFlow messages handled and flow rules generated by specified controller (e.g. Ryu and FloodLight).

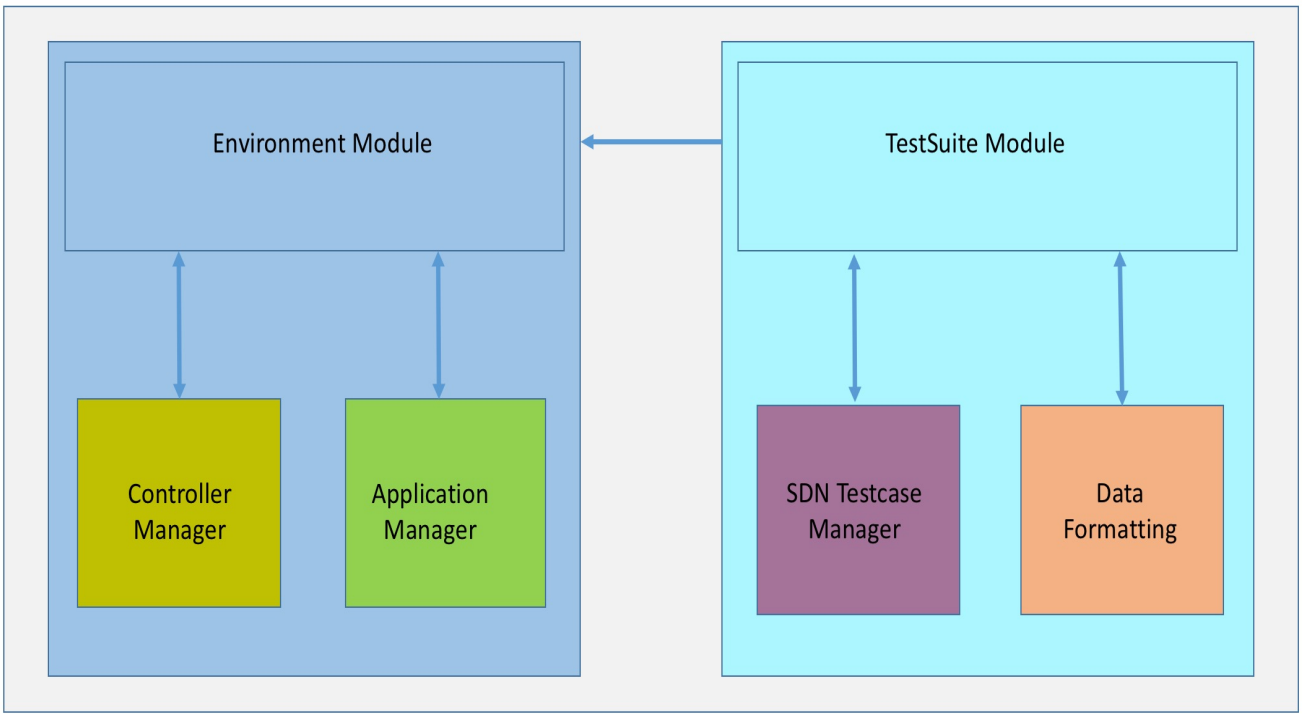



**Figure 12:** Testbed Architecture

## 4.2 Challenges and Requirements for System Design

To achieve the stated goals, we came across few challenges which are as follows:

- Sequence of operations - In order to set up the environment properly for executing the test cases, it is mandatory to execute setup operations in a correct sequence.
- SDN Application Deployment Procedure - There is no standard way to deploy and run the SDN application. For example, to run an application for Frenetic controller, it can be run just like any program placed anywhere. However, for FloodLight, it is necessary to keep application file in its source directory and build the whole project again. For Pyretic and Ryu, it is required to run controller and application together.

These challenges lead us to certain requirements to be considered while designing the testbed strategy:

- For executing the environment setup related operating in a correct sequence, the design of this testbed enforces the template design pattern[15].
- To confront the application deployment challenge, controller related operations such as start and stop are integrated with application specific operations.

[15] http://www.blackwasp.co.uk/TemplateMethod.aspx

## 4.3 Assumptions

Few assumptions were made while outlining the design of the testbed which are as follows:

- It is assumed that the user of this testbed will provide the network topology details in a separate file. This information is necessary to perform test case related tasks. For example, an experiment is interested in knowing number of flow rules installed in the switches. To realize this test case, it is necessary to know number of switches in the network and their interface.
- It is also assumed that ports on which controllers will listen for OpenFlow request are not occupied by other services. For example, if a controller uses port number 6633 for listening OpenFlow message and there exists a process which is already using the same port then either controller will not start or not work as expected.
- It is presumed that all controllers participating for comparison are installed on the local machine only.

## 4.4 Extensibility

It is not possible to consider every existing SDN controller and test case in advance. It is usually required to keep the design open for further changes without modifying the existing code to avoid common software related issues such as tight-coupling.

Our testbed framework can cop up with future requirements. It is a light software that allows for changes in future. The design of our testbed framework follows the separate work elements principle. It disintegrates the components into independent and comprehensible modules in order to provide support for future changes and presents the cohesive design. Each module has single responsibility and can be extended further. This extensible design embraces the changes. With the modular design of our testbed, it is easy to add a new controller, an application or test case. For example, if a user wants to add OpenDayLight controller then it can be achieved effortlessly by extending the module responsible for controllers.

## 4.5 System Architecture

The architecture diagram shown in Figure 12, exhibits the design of our testbed framework. It contains two main modules: **Environment Module** and **TestSuite Module.** The first module is responsible for setting up the environment so that an experimenter can execute the test cases inside it. This module has two submodules i.e., **Controller Manager** and **Application Manager**. **Controller manager** is used to manage the operations related to selected controller such as start and stop. **Application manager** is responsible for deploying, running, and stopping the applications for a specified controller. For example, if an experimenter is interested in running a test case for frenetic controller, then **Environment** module will read the controller specific configurations such as port number and installation directory path from a configuration file. After that, it will start the controller, the application for the controller and Mininet to emulate network environment for a specified test case. Once this environment is in ready state then the test case related tasks are executed by the **TestSuite** module.

The second module is responsible for running a test case and getting result in a specified data format. This module has one submodule namely **SDN Testcase Manager (STM)** and Data Formatting feature. The **STM** is responsible for handling tasks related to test cases such as start, stop, and collecting result operations. The **Data Formatting** feature is used for converting a result into specified data format. For the same example as above, **TestSuite** module will run test cases, over the environment setup by the **Environment** module, and collect result using **STM**. This collected result can be formatted to a specified data format such as JSON using **Data Formatting** feature. These modules and submodules are explained below.

### 4.5.1 Environment Module - Setup Environment for Test Case

In the architecture diagram, this box is placed on the top left position. This module reads the configuration file that is specified by user. This configuration file contains various information such as controller's listening port, installation directory, directory for logs, home directory of testbed etc. This configuration allows for making changes easily. For example, a user changed the settings of Ryu controller to listen on port number 7753 instead of 6653. To convey this information to the testbed framework, a change in configuration file is sufficient. Rest things will be taken care by the testbed itself. This module also reads the network topology file specified in a Python script file. Using Mininet, it emulates the network for user specified network topology.

This module has two submodule that are explained in the sections 4.5.2 and 4.5.3 respectively.

### 4.5.2 Controller Manager Module - To Control and Manage SDN Controller

At this moment, our testbed supports five SDN controllers (Frenetic, Kinetic, Pyretic, Ryu and Project FloodLight). However, it is very easy to extend it for more controllers. A SDN controller is responsible for receiving *packet_in* message and generating the *packet_out*, and *flow_mod* messages for controlling the traffic in a network. To run these controllers, most of the time users are required to execute the command manually. Executing these commands to run and stop the controllers is an overhead and repetitive work. It can also be frustrating to do this multiple times, especially when a user needs to run and get the results one by one. In case of a multi-controller environment, it is time consuming as well. To tackle this problem, we have written this submodule which take cares of mentioned problems and frees the user from this overhead. A user can easily specify the controllers for which test cases are needed to run.

### 4.5.3 Application Manager - To Control and Manage SDN Application

A SDN application communicates with controller via Northbound interface to perform a task in the network. We have three applications (Simple Hub, Switch learning, and Firewall) for each controller in our testbed. Similar to SDN controllers, deploying and running these applications repetitively is boring and can be frustrating. Even, every controller has a different way to run the application. For example, in Project FloodLight, a user must keep the application file in the source directory and specify the fully qualified class name in configuration and module file. To avoid such hassle, this application manager submodule is written. It takes the file name of the program file as an input. After that, it deploys and runs the application by itself. A user can specify the time duration for which application needs to run. After that duration, this module stops the application gracefully.

This module is also extensible since it provides an abstract class with abstract methods. One can extend this class and can easily specify the way in which an application will be deployed, run and stopped

for a controller. It is shown with the green box in the system architecture diagram. This module works in two steps which are as follow:

- *Deployer* - It moves the application to the required directory. This directory is usually controller specific. For example, in Floodlight controller, the application must be in source directory of it. In this case, deployer copies the file from a user defined location to the source directory of FloodLight. Once a program file is in right location then runner module starts its work.
- *Runner* - After deployer finishes the work, this step runs the application. It is not always the case that a controller will be running before the application. There can be a case when a controller starts with an application. For example, to run the Ryu application, it is required to pass application name as an argument.

### 4.5.4 TestSuite Module - TestSuite for Test cases

As its name suggests, this module is used for creating test suites. A test suite in our testbed defines the test cases, an environment for test cases, and data format of the result.

### 4.5.5 STM - To Control and Manage SDN Test case

**STM** is responsible for executing test cases related operations. We have developed few test cases such as flow rules generated by controller, flow rules in switches, packets handled by controller. Usually, test cases are realized as Linux processes. This submodule executes, controls and gathers the data from these Linux processes. For example, to count the OpenFlow messages handled by the controller, tshark [30] command is executed which listens on the loopback interface and captures packets and filters out the OpenFlow messages only. These filtered messages are then stored into a file, and further read for data collection purpose. In configuration file, it can be specified how long a test case will run. STM works in following two steps:

- *Execution* - This step executes the test case, usually as a Linux process and writes the result to a log file.
- *Data collection* - This step reads the log file generated in execution step and provides the result.

New test cases can be written by extending the abstract class offered by this submodule. STM is presented with pink box in the architecture diagram (Figure 12).

### 4.5.6 Data Formatting - Converting Result into specific Data format

This is an additional feature that belongs to TestSuite module. It gets the result as data and converts it to desired data format. At the moment, this testbed has implementation for JSON data format only. However, it is very easy to write data in another format. In order to convert result to a different format, it can be achieved by extending the abstract class and override the method responsible it. This module is shown with orange box in Figure 12.

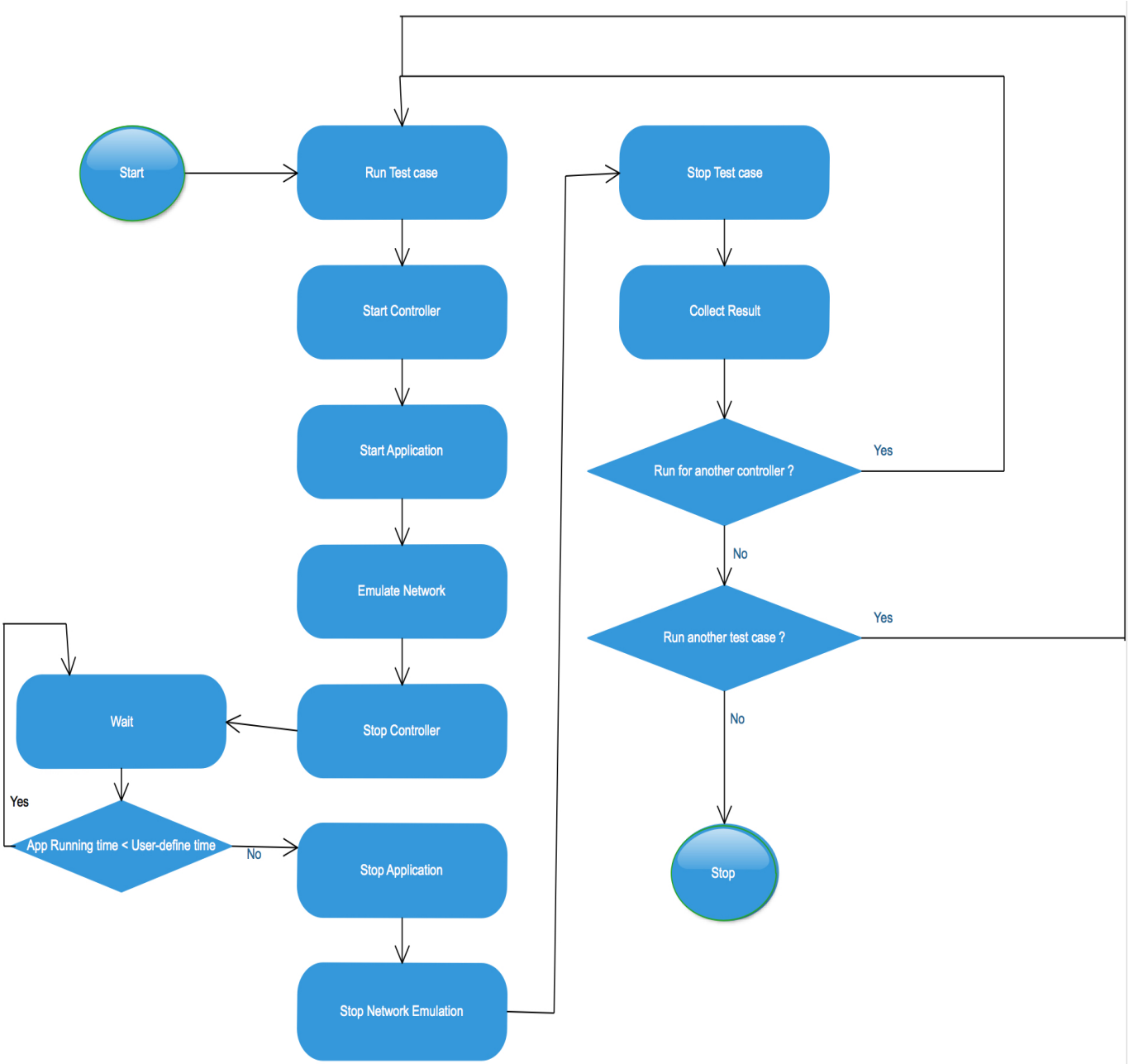



**Figure 13:** Life cycle of the testbed

## 4.6 Life Cycle

This section holds the details about life cycle of our testbed. This life cycle is represented with the state chart diagram in Figure 13.

An example will provide better understanding of the life cycle of our testbed. Assume, that a user wants to find out the flow rules generated by the controllers (Ryu and Frenetic) for switch learning application over a single switch topology. In order to achieve it, initially (Run Test case ) test case related tasks will be executed such as capturing OpenFlow packets. Then, testbed will set up the environment so that OpenFlow packets are generated for the test case. Then a controller, switch learning application for the controller and network will be emulated. There will be exchange of OpenFlow messages once environment setup is done. An application will run for the user-define duration. After that application, controller and network emulation will be stopped. These stopping steps for the controller, application

and network emulator are necessary to run the testbed properly for next experiment. After that test case will be stopped and result will be collected. The same process will be repeated for second controller. Then, it checks if there are more test cases, if yes then whole process will be repeated else testbed will halt.

This chapter presented high-level details of our testbed framework. In this chapter, we explained the use case scenarios, assumptions, challenges and requirements, extensibility, comparison metrics, modular architecture, modules and the life cycle of our testbed.

# 5 Implementation

Up to this point, the objectives and the system design to achieve the goals of this thesis were discussed. This section presents the implementation details of the system developed. It is divided into three parts. Firstly, programming language requirement of the developed system is explained along with the reasons for choosing it. Secondly, the implementation details of each of the module mentioned in section 4 are elucidated followed by the workflow of the testbed. Finally, SDN applications are briefly explained.

## 5.1 Implementation of Testbed Framework

This section focusses on the implementation of the system design described in the section 4. It covers tools and technologies used in order to achieve the desired results. This section also gives the short description of software and hardware configurations of the system.

### 5.1.1 Programming Language

Python 2.7 is used to implement the testbed framework. It is a scripting language and powerful for rapid prototyping [6]. It offers good flexibility to control for executing OS-specific commands. The object-oriented concepts of this language also help to achieve extensibility. Python, it is also a requirement for the Pyretic and Kinetic controllers.

## 5.2 Implementation of Environment Module

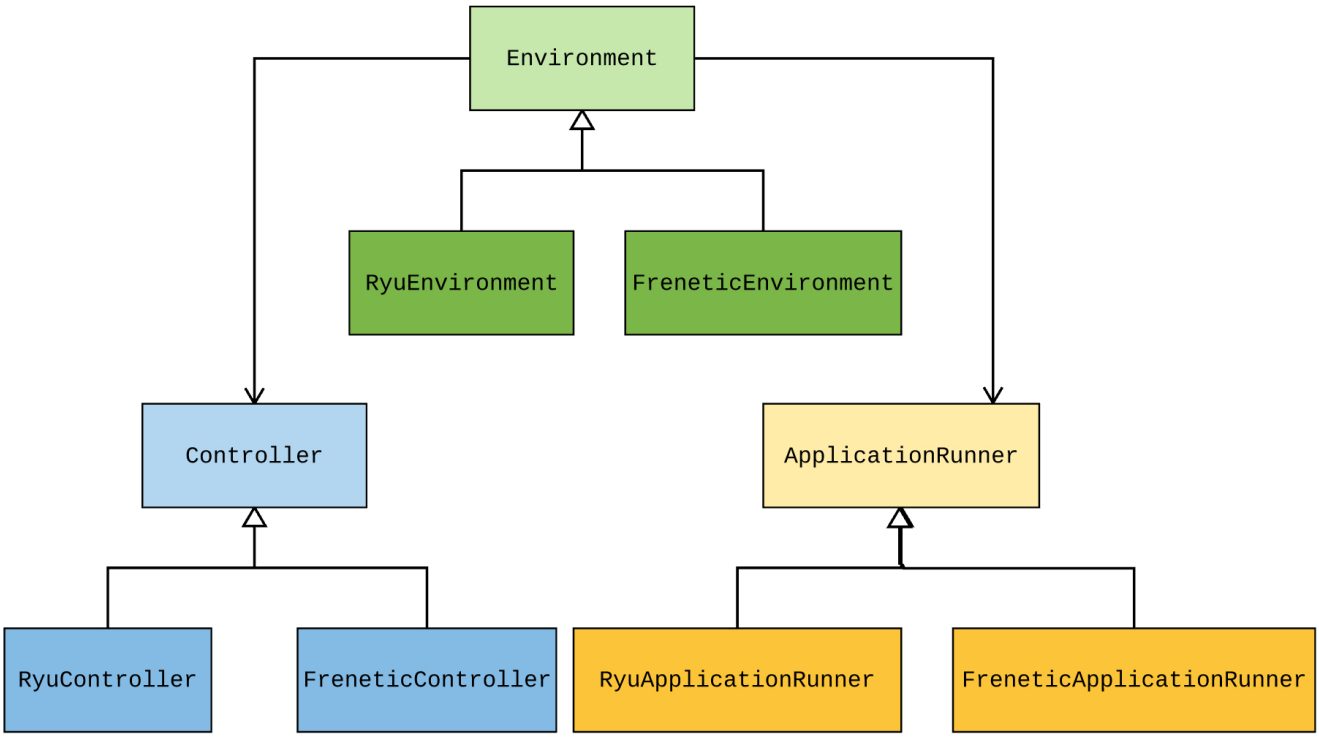



**Figure 14:** Class diagram of Environment Module

The class diagram for this module is shown in Figure 14. This diagram presents three classes and their corresponding subclasses. The Environment is a main and abstract class of this module and this class

offers few *abstract* and concrete methods. Each function has its own responsibility which may, in turn call another method. This class can be distinguished by the green color in Figure 14. Functions of this class are explained as follows:

**readConfigFile()**

This is the first method that is invoked in order to extract environment specific configurations. It reads the contents of the user-specified configuration file.

**additionalConfigFile()**

This is an abstract method and its implementation should be specified by the subclass. Few controllers require additional configurations files and this method meet their needs. It maintains the list of configuration file with their path on the operating system. For example, in order to deploy and run an application for FloodLight controller, it is required to specify two files namely *floodlightdefault.properties* and *net.floodlightcontroller.core.module.IFloodlightModule*. The first file specifies the configurations such as port number and list of applications to be executed on FloodLight controller. The second file helps the module loader for loading the modules. An applications is called as a module in FloodLight and before starting up the controller it is required to tell the FloodLight to load the module. To load a module, it is required to specify about the module location and it is done by specifying the fully qualified module name. Module entries in this file are written like this:

```
floodlight.modules = <default list of modules>,
net.floodlightcontroller.<fully qualified module name>
```

This method is not required to be implemented by every subclass.

**setTopology()**

This testbed framework allows the experiments to be executed over any network topology. A user can specify the topology information in an external Python file. This function accepts two parameters; the first parameter is the location of the Python file and the second parameter is the name given to topology. This information can be used to emulate network with specified topology or also can be used to realize some test case such as finding the number of flow rules stored in the switches.

**runMininet()**

This function is useful to execute operations regarding network setup. This testbed uses the Mininet to emulate the network and details according to Mininet is specified in this section. This function can be overridden to make this testbed work with other network emulators or hardware devices.

This function accepts three arguments. The first argument is the absolute name of topology file and the second argument is topology name. The third argument holds the additional details for the network such as IP address and port on which a controller is running. This information is required for the network so that switches can communicate with a controller.

**stopMininet()**

This function is useful to execute to stop the network emulation. As mentioned above, this testbed uses the Mininet to emulate the network. In order to achieve the network emulation with Mininet, it is

required that Mininet should not be running already. Therefore, to run the test cases it is required to tear off the network and re-initialize it. Similar to **`startMininet()`**, this function can be overridden to specify the stopping instructions for the network.

**`startController()`**

This is an abstract method and its implementation must be provided by the subclass. It takes one optional parameter i.e., additional configurations. It is not really responsible for running the controller but it does the pre-work such as finding the installation directory of the controller and calls the method which starts the controller.

**`stopController()`**

This is also an abstract method and its implementation must be provided by the subclass. It takes one optional parameter i.e., additional configurations. This is useful for performing operations before and after stopping the controller. It invokes the method accountable for stopping the controller.

**`startApplication()`**

It is an abstract method and must be implemented by the subclass else an implementation error will be thrown. It accepts one parameter which holds the application name of the file that will be running. It calls the method responsible for starting the application. It is similar to **`startController()`** but for applications.

**`stopApplication()`**

This is also an abstract method and its implementation must be provided by the subclass. It invokes the method which really stops the application. This method is helpful for executing operations before and after stopping the application.

**`generateTraffic()`**

This method is responsible for generating the traffic in the network such as HTTP. This function is necessary for some applications such as Web Firewall (section 5.5) that blocks the HTTP traffic.

**`setTestbedHome()`**

This function is used to store the directory path where our testbed framework is placed.

**`setupEnvironment()`**

A requirement about executing the operations in a correct sequence is discussed in section 4.2. This function meets the specified requirement and invokes the methods in a specified sequence for the proper working of the testbed. This function calls the **`startController()`** and **`startApplication()`** methods followed by **`runMininet()`** function. Then, it invokes **`stopApplication()`**, **`stopMininet()`** and **`stopController()`**.

This section illustrated the implementation details of **Environment** class with the explanation of all

abstract and concrete methods. It also described the details about meeting the requirement of executing operations in a particular sequence (section 4.2). The next sections elucidate the implementation details of **Controller** and **ApplicationRunner** classes. These classes factually execute the start/stop operations for controller and applications.

### 5.2.1 Controller Class

This class can be identified with blue color in Figure 14 and contains the methods for starting and stopping the controller. These functions are explained as follows:

**`runController()`**

This is an abstract method and its implementation must be provided by the subclass. This function takes one optional parameter i.e., additional configurations. This function is made abstract intentionally, because every controller provides a specific way to start it. For example, to run the Pyretic controller it is required to change current directory to the Pyretic home directory and execute the Python file responsible for it. However, in order to start the FloodLight controller, it is required to run the jar file. To solve this problem, we left the implementation of this method for the subclass. This method must be implemented by the subclass else an implementation error will be thrown.

This function starts a Linux process that executes the command to start the controller.

**`stopController()`**

This is also an abstract method and its implementation must be provided by the subclass. This function is made abstract intentionally, because there is not a standard way to stop the controller. For instance, let's assume that the Pyretic controller is running on port 6633 now it should be stopped. In order to that, processes id of the program is identified and then kill command is executed. Kill command takes the process id as parameter and kills it. However, in the case of the Frenetic controller, killing a process that uses the port 6633 is not sufficient. It uses the port number 9000 as well. Therefore, it is required to release this port (9000) as well so that any another process can use it.

**`checkPort()`**

This method returns either true or false and is responsible for checking out if a given port in its arguments is opened or not. This function finds if a controller is in ready state or not. Therefore, to achieve it, this function continuously checks for the opened port which the controller uses. Once a port is opened, it returns true and the further processing in testbed takes place.

### 5.2.2 ApplicationRunner Class

This class can be differentiated with yellow color in Figure 14 and contains the methods for starting and stopping the application. These functions are explained as following:

**`runApp()`**

This is an abstract method and its implementation must be provided by the subclass. This method is made abstract intentionally because every application has a specific way to start it. For example, to run the Pyretic application it is required to change current directory to the Pyretic home directory and execute the application with its runtime. However, in order to start the FloodLight application, it is

required to copy the application to the source directory of FloodLight. To solve this problem, we left the implementation of this function for the subclass. This method must be implemented by the subclass else an implementation error will be thrown. It accepts one parameter which holds the name of the application.

This method starts a Linux process that executes the command to start the application. It is capable to meet the requirement, specified in the section 4.2, of starting application and controller together.

**`stopApp()`**

This is also an abstract method and its implementation must be provided by the subclass. This function is responsible for stopping the application. As mentioned above, sometimes it is also necessary to copy the file to controller's directory. To make the directory look like to its previous position or to make other cleaning stuff, this function removes the deployed application file.

## 5.3 Implementation of TestSuite Module

This module is mainly responsible for defining the test cases and data format of the result. The class case diagram for this module is shown in Figure 15. It represents three main classes namely: TestSuite, TestCase and DataWriter. Following sections elaborate these classes.

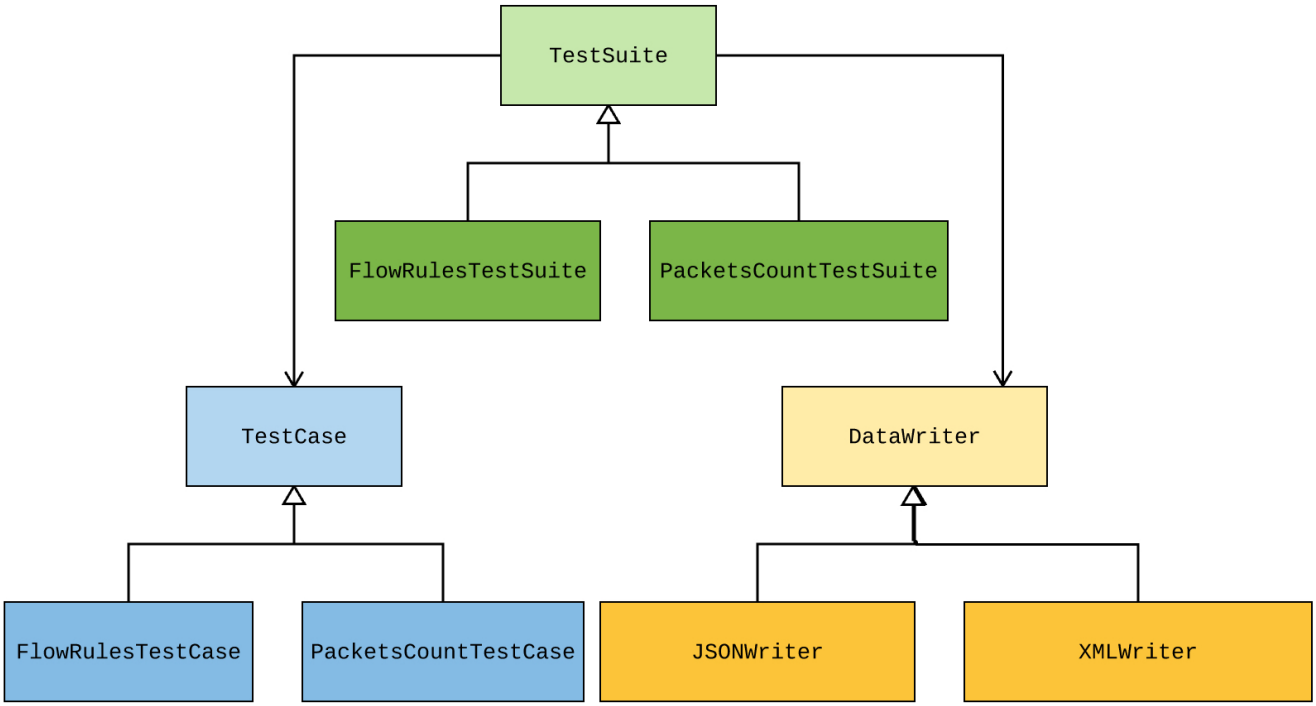



**Figure 15:** Class diagram of TestSuite Module

### 5.3.1 TestSuite

TestSuite is a main and abstract class and this class offers one *abstract* and three concrete methods. This class can be distinguished by the green color in Figure 15 and the methods of this class are described as follows:

**`addEnvironment()`**

This method accepts the object of **Environment** class an input. It is responsible for maintaining a list of controllers for which the test cases will run.

**`addTestCase()`**

This method takes the object of **TestCase** class an input. It keeps a list of test cases to be executed.

**`setDataFormat()`**

This function receives the object of **DataWriter** class an input. This received object is later used to call a function that writes result of a test case in specified format.

**`run()`**

It is required to implement this function in the subclass else an error will be thrown. This function uses the object of **Environment** class to setup the environment for running the test case. Once the environment is ready then test cases are executed. However, it is not always the case the environment needs to be setup before running test cases. There could be scenarios where test case will run before environment is ready. For an example, to count the flow rules generated by a controller, it is required to run the test case before the environment. On the other hand, in order to find the number of flow rules in a switch , it is required to setup the environment before test case. To alleviate this problem, this function is made abstract so that a user can handle the scenario according to the test case requirement.

### 5.3.2 TestCase

**TestCase** is a class and it is differentiated with blue color in Figure 15. This class is responsible for realizing the test case and collecting result for same. This class contains one concrete and two *abstract* functions and these functions are explained as follows:

**`setTopology()`**

This function stores the network topology information. To realize some test cases, it is necessary to have a knowledge about the network. For an example, a user wants to count the number of flow rules stored in a particular switch. In order to achieve this, the switch is queried to get this information. In a case of a single switch, it maybe simple but for multiple switches, it is necessary to know the interface provided by the switch. This function provides the topology information to get this information about switch interfaces.

**`execute()`**

This is an abstract function and must be implemented in the subclass in order to execute the test case. Test case us such as counting OpenFlow messages, flow rules etc., are implemented by executing Linux commands that in turn capture OpenFlow messages using Tshark. However, it depends on the test case requirement if Linux command is required or not. To make it flexible to handle various test case, this function is made abstract.

**`result()`**

This function collects the result for a test case and returns it. The mechanism of collecting the results can be different for every test case. To make it flexible, this function is also declared as abstract like **`execute()`**.

### 5.3.3 DataWriter

**DataWriter** is an abstract class and it can be identified with yellow color in Figure 15. This class contains one abstract method named **`write()`** which must be implemented in the subclass. This method takes the data object and converts the data to another user defined data format. It can be implemented to support any data format.

## 5.4 Testbed Workflow

This section highlights the interaction among modules and submodules developed for our testbed. The sequence diagram of the testbed is shown in Figure 16 and explained with an example.

Assume, that a user wants to count the number of flow rules generated by the Frenetic controller and needs this information in JSON format. In order to achieve that **`execute()`** is invoked. This function starts capturing the OpenFlow packets and these packets get stored into a file. Then, **`setupEnvironment()`** is executed which in turns calls the **`startController()`** and **`startApplication()`**. These functions result into starting the Frenetic controller and its application and results into to exchange of OpenFlow messages. Then, **`stopController()`** and **`stopApplication()`** functions are executed which stop the Frenetic controller and its application. To collect the results of a test case, **`result()`** method is invoked which reads the file generated by **`execute()`** method of TestCase class. This function parses those packets which hold the flow rules information and the total number of such packets is returned by this function. Finally, **`write()`** method is called which converts the result into JSON format.

## 5.5 SDN Applications

This section briefly summarizes the SDN applications which are used for evaluation purpose. These applications are explained as follows:

### 5.5.1 Hub

This application makes the switch to act like a network hub where every incoming packet is forwarded to all ports except the incoming port. When a packet arrives at the switch, then the switch creates the duplicate copies of the packet for each port. An implementation of this application was already available for all chosen controllers for the testbed.

### 5.5.2 Learning Switch

In contrast to Hub, this application makes the switch to work more intelligently. It creates a table where MAC address of each host is mapped to the port they are connected to. When a packet arrives at the switch, it looks for the destination MAC address in the table. If a matching entry is found, then the packet is forwarded to the mapped port else the switch behaves like a Hub. This application was also available for all the controllers. However, for Frenetic controller, we extended this application for multi switch topologies.

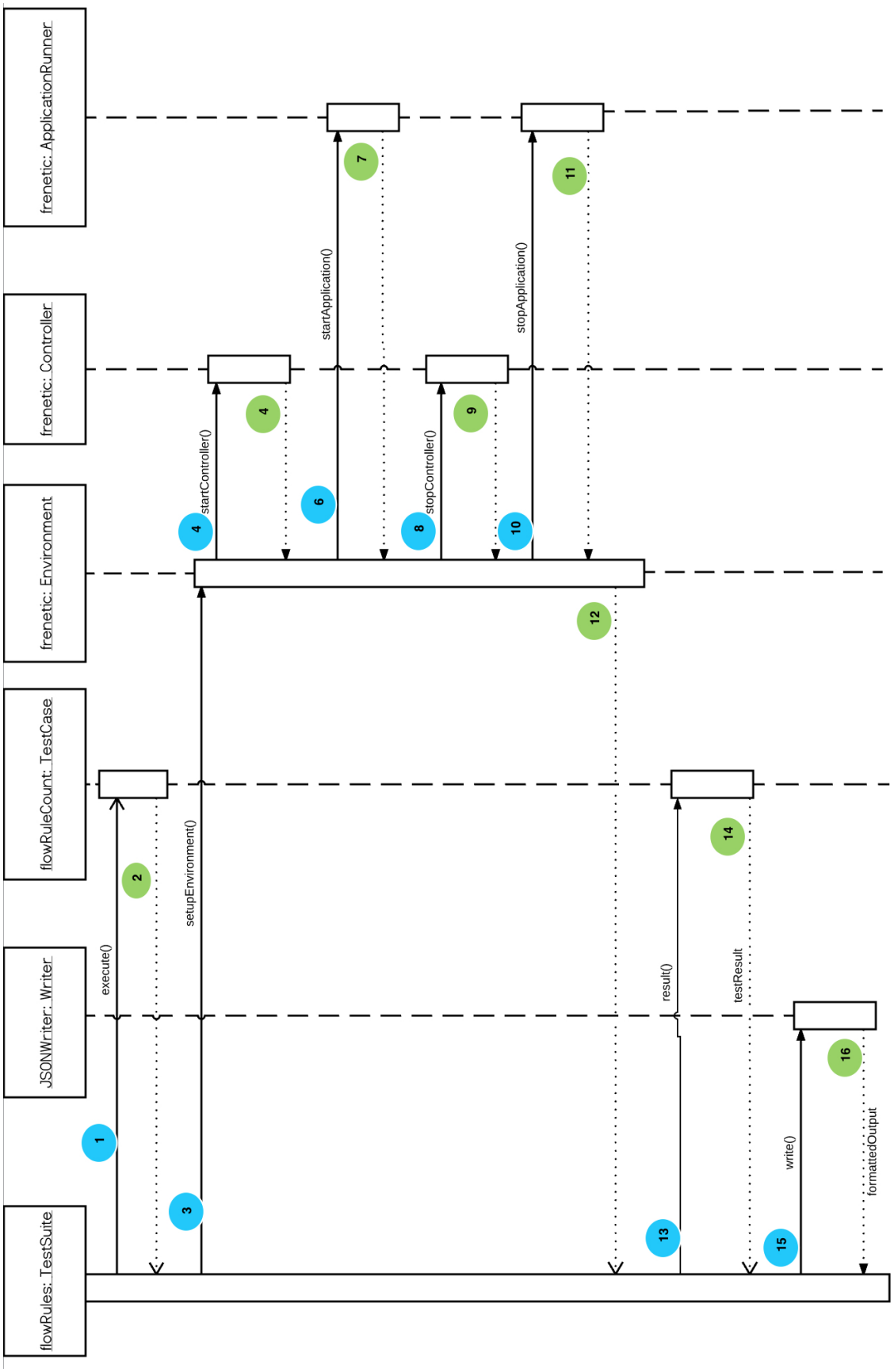



**Figure 16:** Sequenece diagram of Testbed

### 5.5.3 Web Firewall

This application allows all kind of traffic between two hosts except the HTTP. It is implemented by identifying the ethernet protocol (value 0x800) and network layer protocol (value 6). If these values are matched in packet header, then OpenFlow flow rules are generated to block such packets. This application is realized by the modification of learning switch application, available for all controllers.

# 6 Evaluation

In this section, we present the evaluation of the proposed testbed for SDN languages. The evaluation is divided into four sections. First section describes the setup used for evaluation. Second section explains topologies used for evaluation. Third section highlights the evaluation metrics. Fourth section evaluates SDN apps for multiple network topologies. This evaluation is carried out to compare SDN languages in order to find out their behaviour for same application. This comparison also presents the details about language expressivity and abstraction.

## 6.1 Evaluation Setup

The testbed framework created in this thesis is evaluated on Ubuntu 14.04 LTS Trusty as the underlying operating system. This operating system is open-source and offers several packages to support the software development. The operating system is installed on Intel x86 architecture with 1 CPU core with 2.5GHz CPU speed and 4 GB memory.

## 6.2 Network Topologies for Evaluation

The evaluation is done on six different network topologies which are as follows:

1. This topology is represented with Figure 17. It has two hosts which are connected to the same switch.
2. This topology is similar to the previous topology but it has five hosts.
3. This topology is shown in Figure 18. This topology has three switches and every switch is connected to two nodes.
4. This topology is similar to the third topology but it has five hosts.
5. This topology has four switches and each leaf switch is connected to two hosts. It is illustrated with Figure 19.
6. This topology is similar to the fifth topology but it has five hosts.

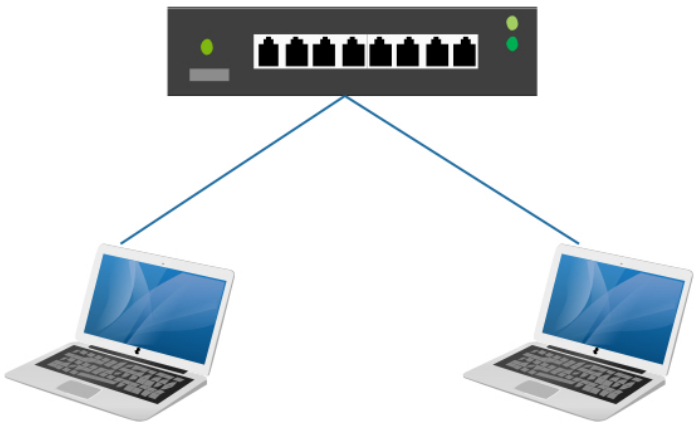


**Figure 17:** Single Switch Topology

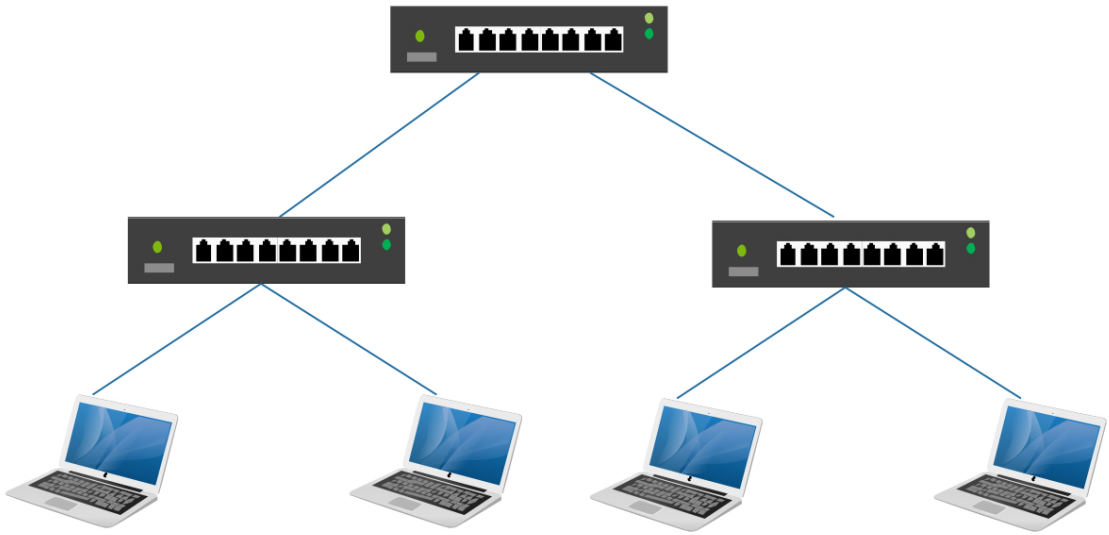


**Figure 18:** Two Switches Topology

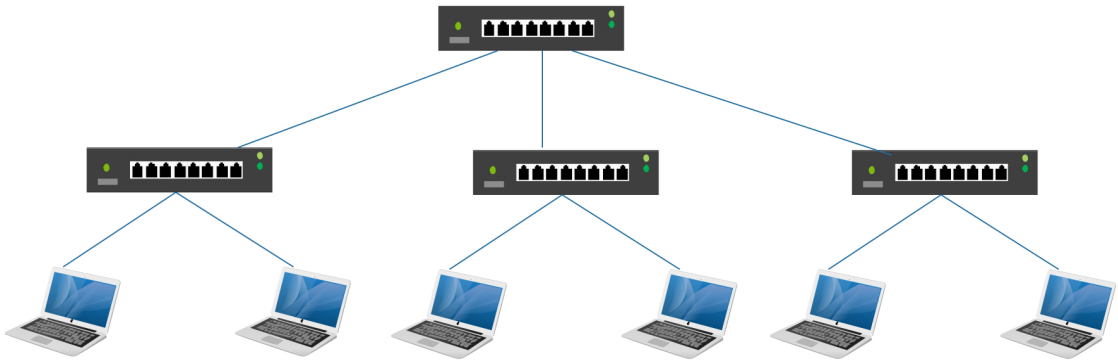


**Figure 19:** Three Switches Topology

## 6.3 Evaluation Metrics

For evaluation metrics, we have used some test cases for comparing SDN languages. These test cases are evaluated on aforementioned topologies with different SDN applications. The results obtained are then plotted onto the graph. These evaluation criteria are defined as follows.

- **Packets sent/received at Controller:** It presents the number of packet in and packet out messages at a controller. It can be used to find out how efficient a language is in terms of traffic generated for the controller. If a language generates less traffic for the controller, it is more efficient than the one which generates more traffic.

- **Number of Flow-rules generated:** It describes the number of flow mod messages sent by a controller. It is used for showing effective flow-rules generated by the language. Less number of flow-rule messages by a language implies better efficient for the switch.

- **Number of Flow-rules in switch at a particular time:** It represents the number of flow-rule in a switch at a particular time. This metric helps in identifying a total number of installed flow rules in switches and saturation time when all the flow-rules are installed. A language installing less number of total installed flow-rules is efficient than others installing more rules. A lower value of the saturation time for a language implies better efficiency.

- **Number of Lines of code:** It represents the total number of lines written for different SDN applications. Fewer lines of code for writing an application implies less human efforts.

SDN languages offer abstractions to write applications. These evaluation criteria will explain how a SDN langauge behaves differently than other SDN languages used for writing the same application. Using these mentioned evaluation criteria, it can be found how a language is efficient than others.

## 6.4 Hub Application

Hub application has already been discussed in section 5.5.1. This section represents the comparison of SDN languages using test cases discussed in section 6.3. This section is divided into multiple sections and each section represents a test case and results collected for that testcase. We are using identical Hub application for all evaluation metrics.

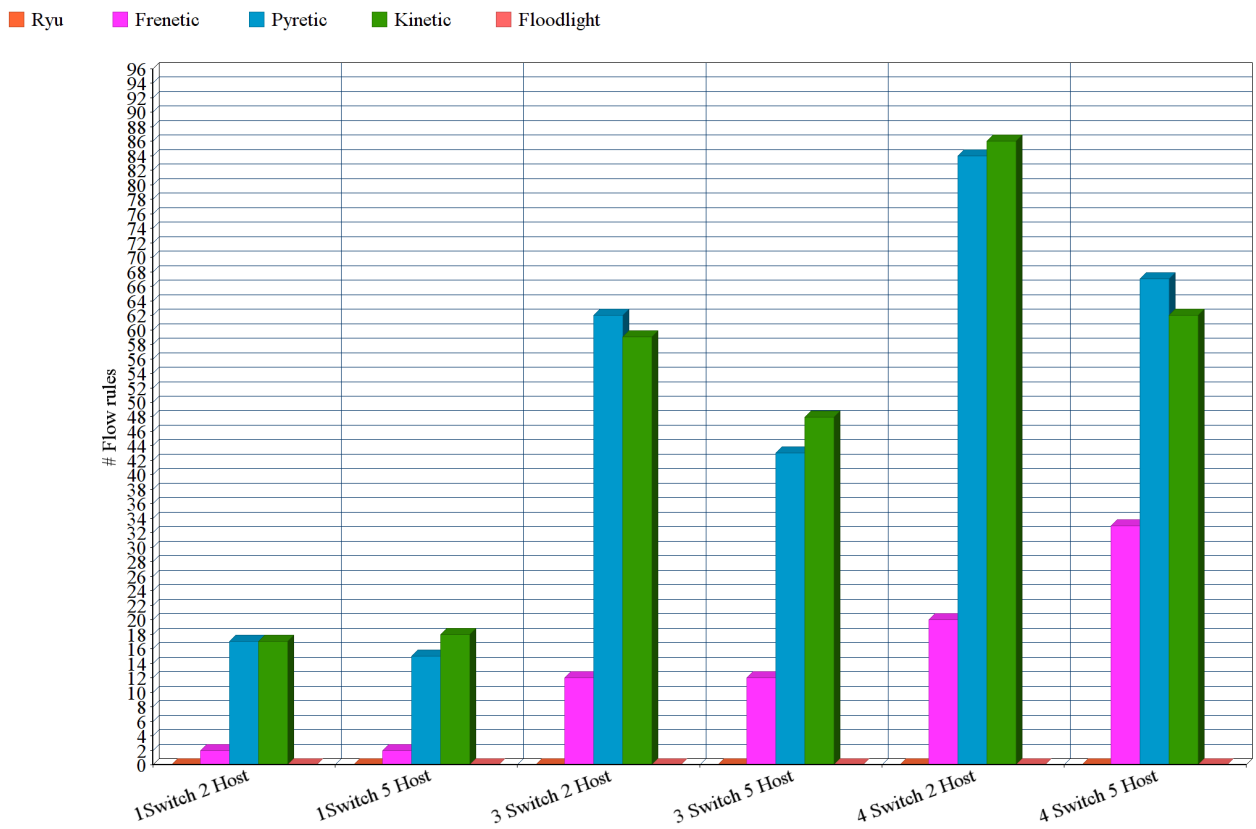


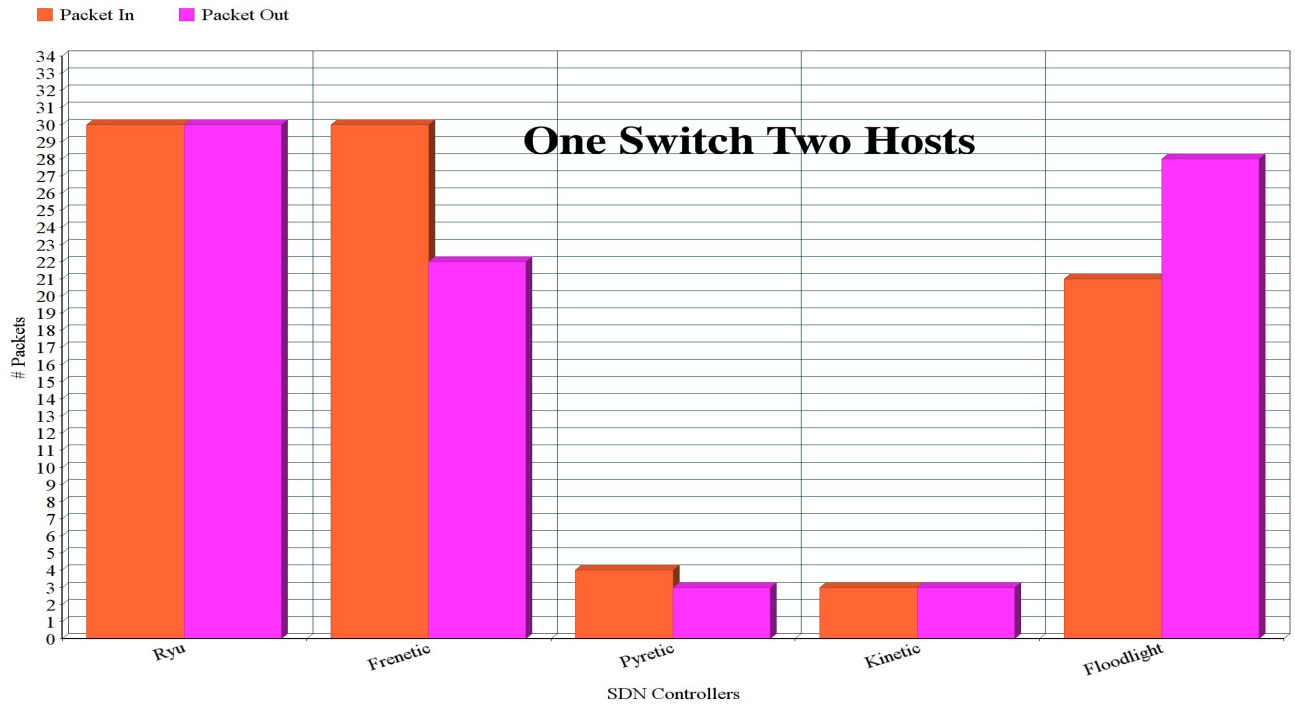

**Figure 20:** Number of Flow rules generated by controllers (Hub App)

### 6.4.1 Number of rules

Figure 20 depicts the number of flow rules message generated by the controllers on various network topologies. It can be inferred from the figure that Ryu and Floodlight generated the less number of flow rules messages whereas Pyretic and Kinetic generated higher than others. This figure also shows that Pyretic and Kinetic generated almost same number of rules. Pyretic and Kinetic translates the flooding instruction into a set of OpenFlow rules, that is why they generate flow-rules.

### 6.4.2 Number of Packets at Controller

Figure 21 and 22 represent the number of packets generated and encountered by the controllers on various network topologies. It can be inferred from the figures that Pyretic and Kinetic generated and



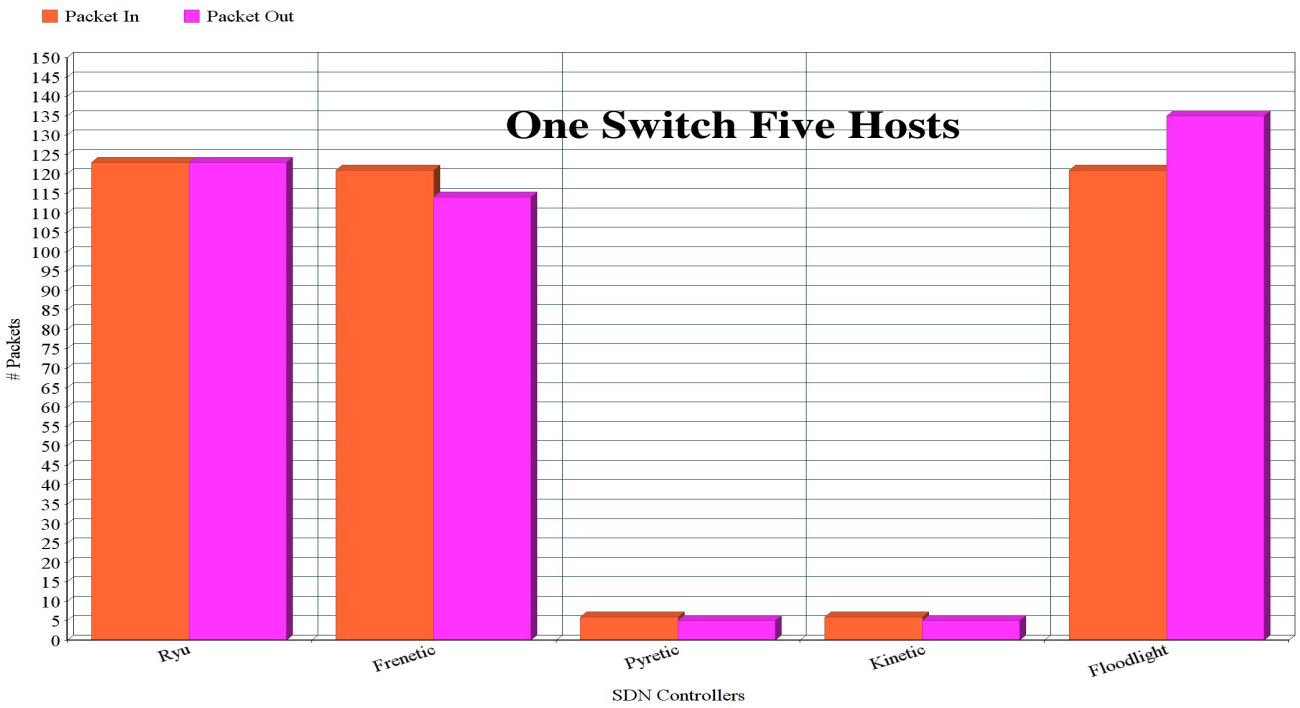
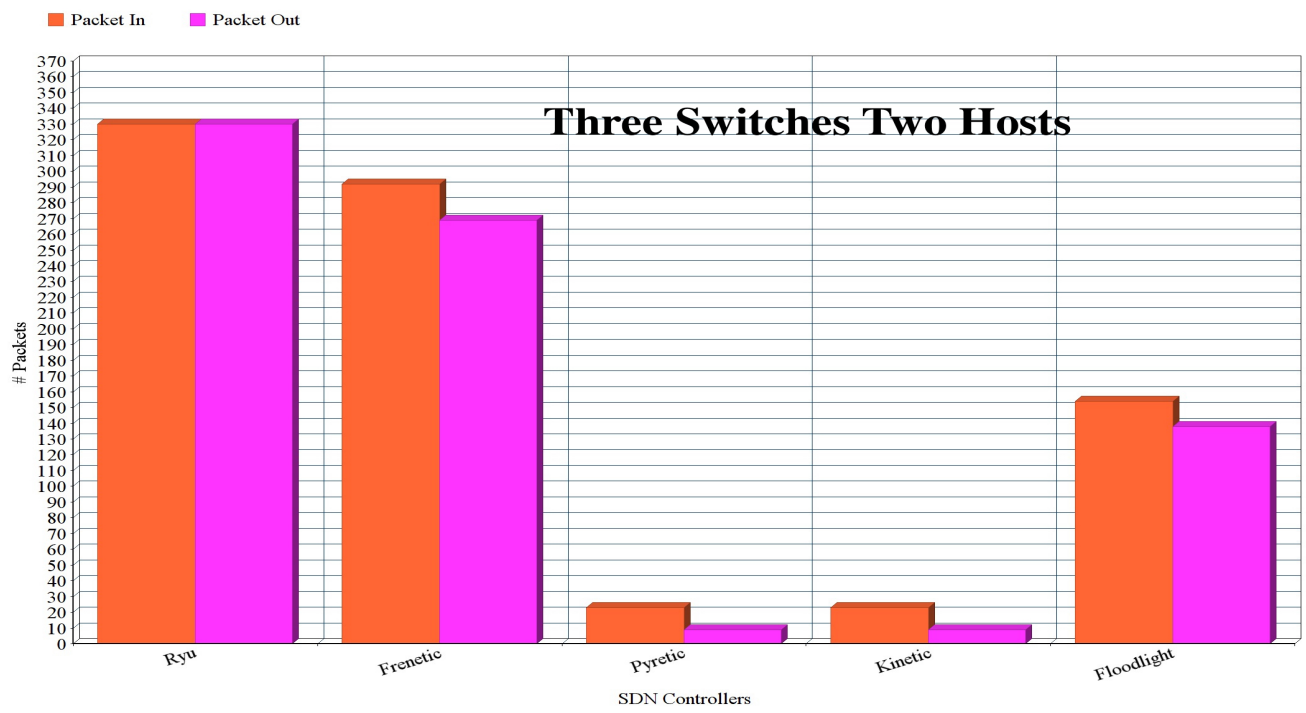




**Figure 21:** Number of packets generated/encountered by controllers (Hub App)

received the lesser number of packets than others whereas Ryu and Floodlight generated and encountered higher packets than others. It is clear from the figures that Pyretic and Kinetic have almost similar



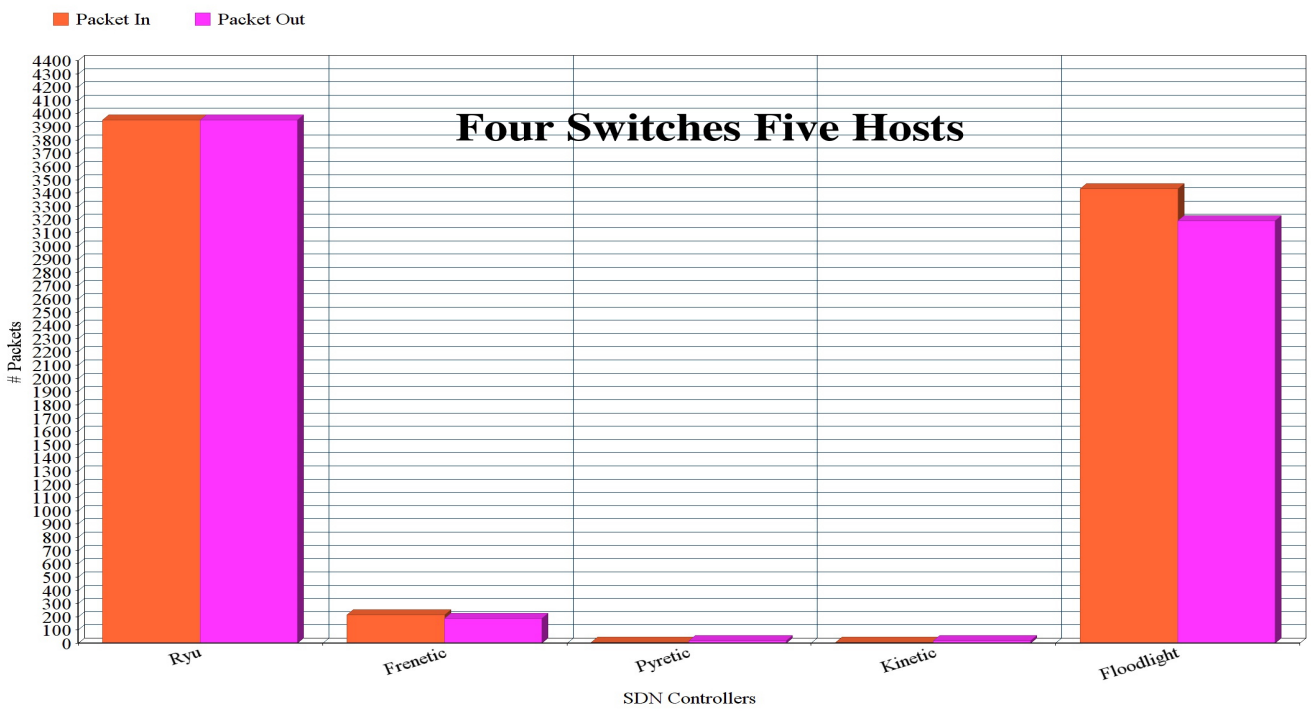




**Figure 22:** Number of packets generated/encountered by controllers (Hub App)

behaviour. Pyretic and Kinetic generates some rules that allows for less traffic at the controller where as

every packet goes to the controller for Ryu and Floodlight controllers.

### 6.4.3 Number of Flow rules in switch at a Particular Time

Figure 23 shows the number of flow rules at a time in switch for one switch topology (2 host and 5 hosts) for all controllers. Figure 24, 25 an 26 represent the installed flow-rules in multi switch topology. It is clear from the figures that Frenetic installed one rule whereas Pyretic and Kinetic installed more rules. It happens because of the same reason explained in the previous sections.

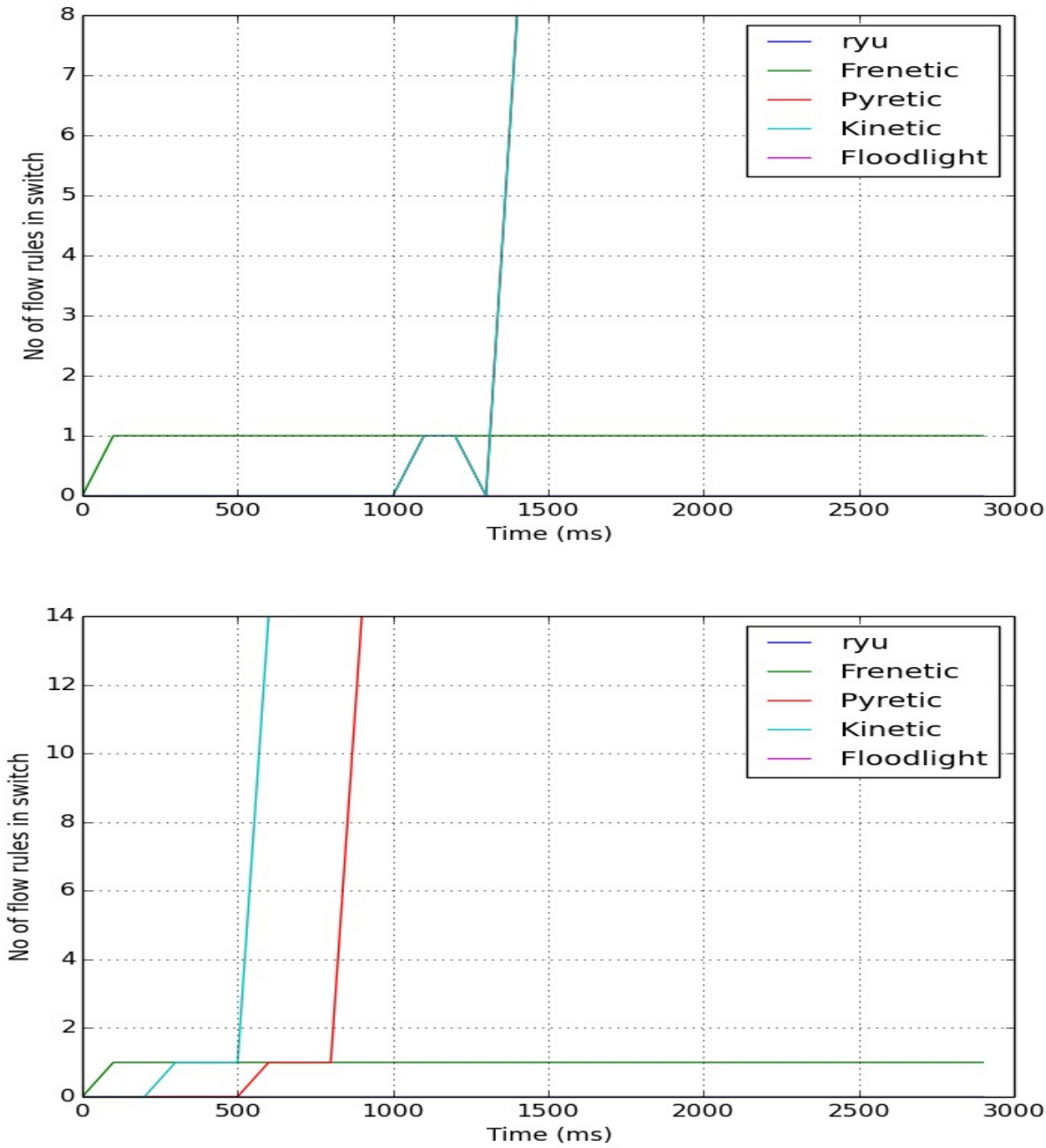



**Figure 23:** Number of rules in Switches at a particular time (Hub App) - Single Switch Topology

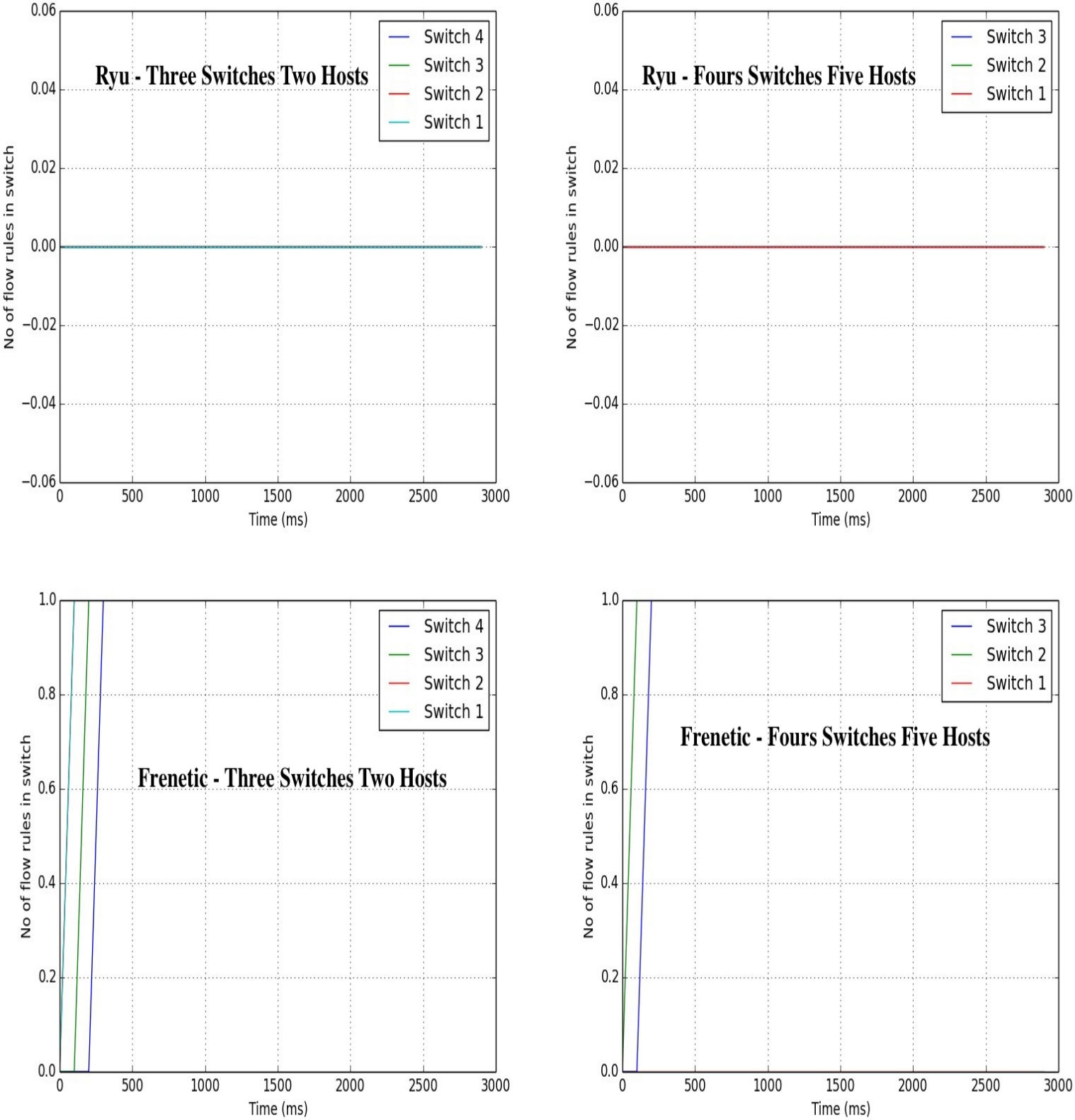



**Figure 24:** Number of packets generated/encountered by controllers (Hub App) - Multiswitch Topology

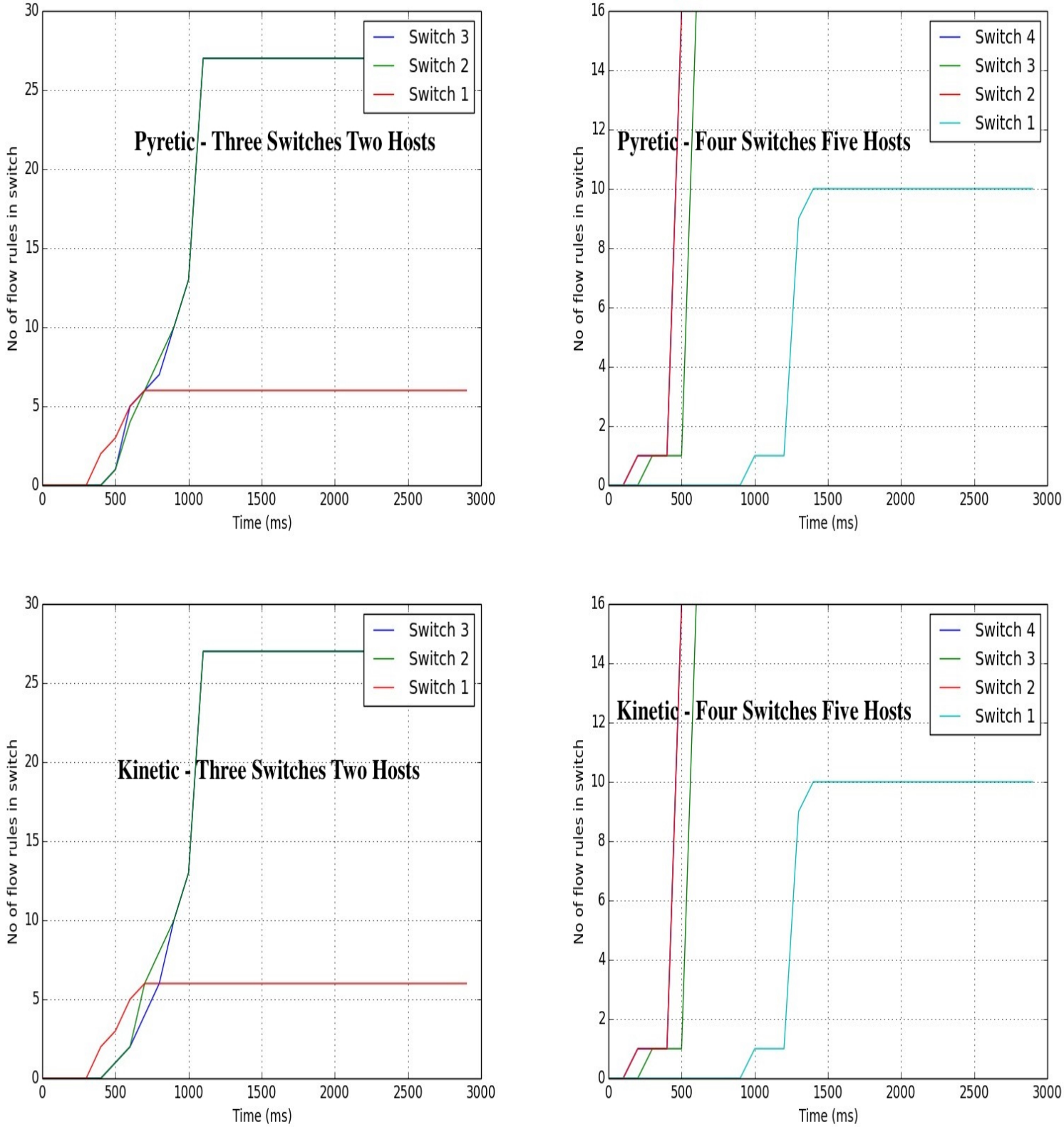
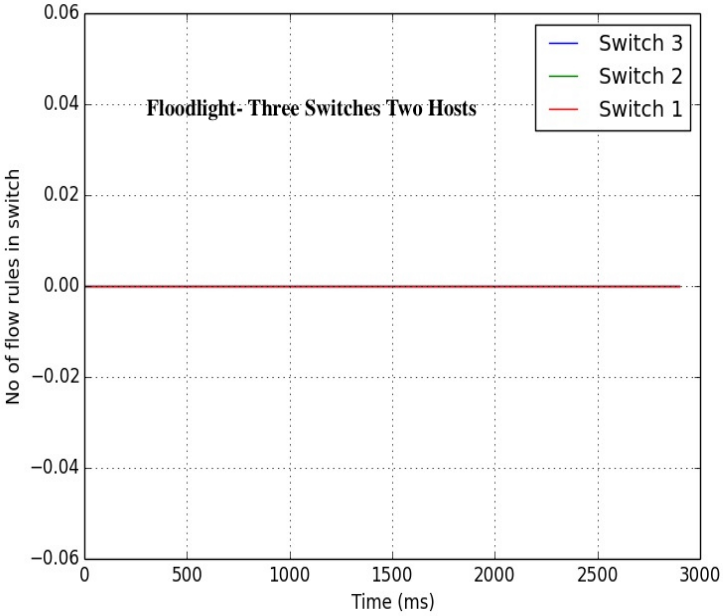



**Figure 25:** Number of packets generated/encountered by controllers (Hub App) - Multiswitch Topology



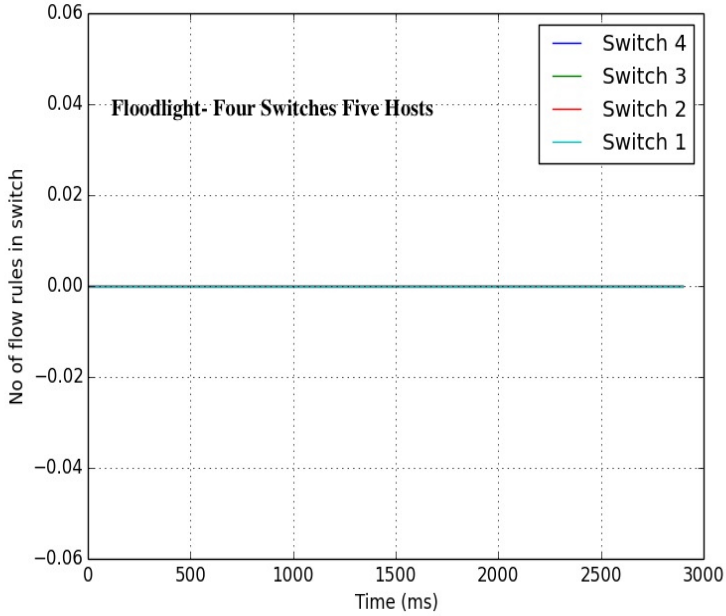




**Figure 26:** Number of packets generated/encountered by controllers (Hub App) - Multiswitch Topology

### 6.4.4 Lines of code for Hub Application

Figure 41 represents the number of lines of code written. It can be deduced from that image that hub application is written in less number of lines in Pyretic and Kinetic languages which implies that these languages offer higher level abstraction than others. This holds true for other applications as well. Figure also represents that writing application using Floodlight requires more number of lines than others. For Frenetic and Ryu number of lines of code is almost same. There was no need to write another hub application to deal with multi switch topology.

## 6.5 Switch Learning Application

Switch Learning application has already been discussed in section 5.5.2 This section represents the comparison of SDN languages using this application. For all evaluation metrics, we are using identical switch learning application.

### 6.5.1 Number of rules

Figure 27 depicts the number of flow rules message generated by the controllers on various network topologies. It can be deduced from the figure that Floodlight and Ryu created the less number of flow rules messages whereas Pyretic and Kinetic generated higher than others. This figure also describes that Pyretic and Kinetic generated almost equal number of rules.

### 6.5.2 Number of Packets at Controller

Figures 28 and 29 represents the number of packets generated and encountered by the controllers on various network topologies. It is clear from the figures that Pyretic and Kinetic generated and received the lesser number of packets than others. On the other hand, Ryu and Floodlight encountered higher number of packets than others. It is clear from the figures that Pyretic and Kinetic have almost similar behaviour.

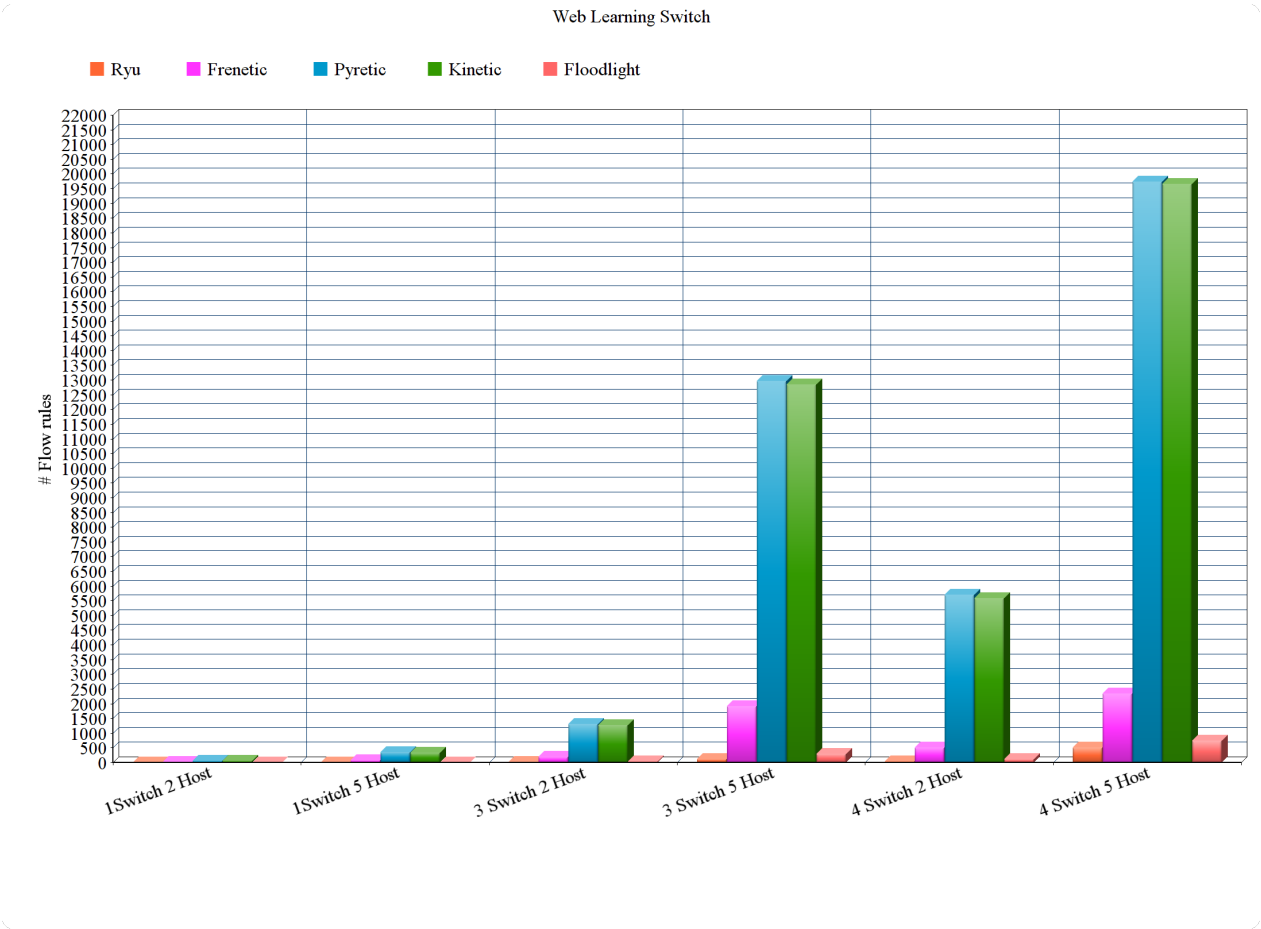



**Figure 27:** Number of Flow rules generated by controllers (Learning Switch App)

### 6.5.3 Number of Flow rules in switch at a Particular Time

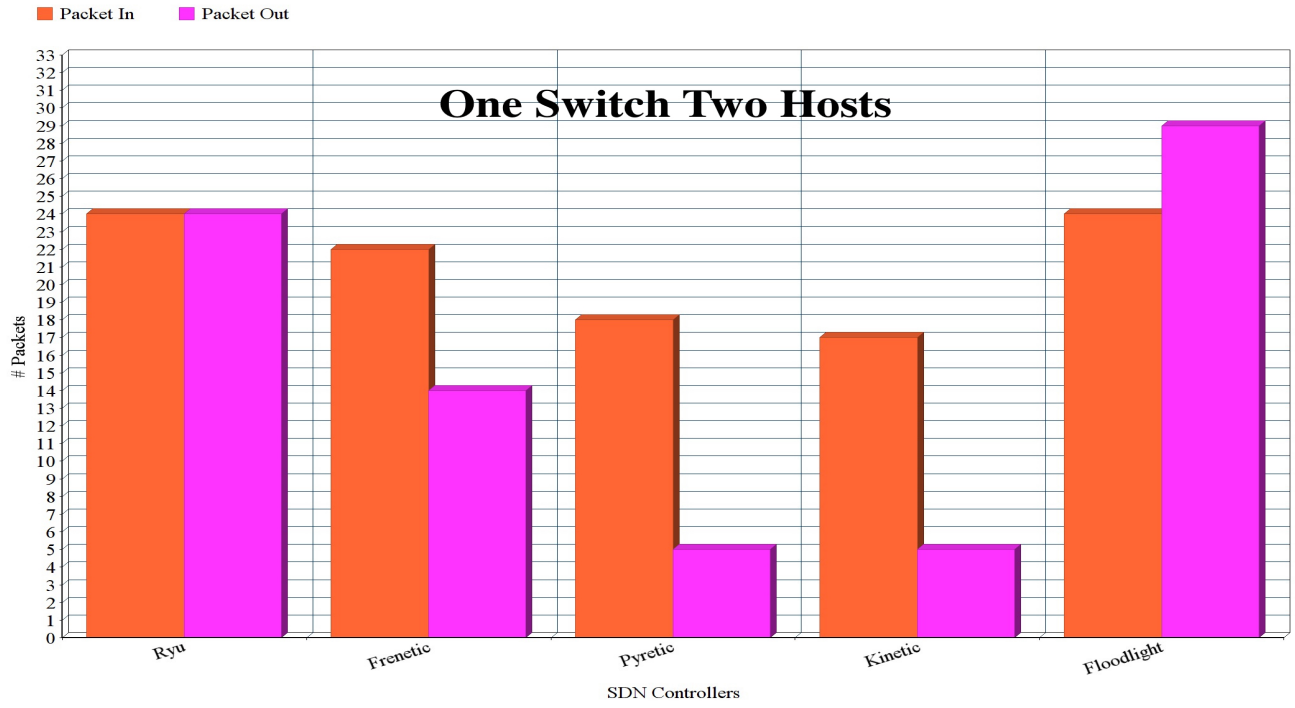
Figure 30 represents the number of flow rules in switch at a particular time for one switch topology (2 host and 5 hosts) for all controllers and figures 31, 32 and 33 represent the same for multi switch topology. It can be deduced that for Frenetic, Pyretic and Kinetic rules are compartively there more number of rules in switches than the others.

### 6.5.4 Lines of code for Switch Learning Application

Figure 41 depicts the number of lines of code written. It can be inferred from that image that switch learning application is written in relatively less number of lines in Pyretic and Kinetic languages. It can also be deduced that implementing application using Floodlight requires higher number of lines of code than others. For switch learning application is written with almost same number of lines in Ryu and Frenetic languages. Frenetic requires writing another application for multi switch topology. Writing another program to deal with multiple switches is not required for rest languages. This Figure also depicts that switch learning application and web firewall is implemented with same number of lines in Frenetic language.

## 6.6 Web Firewall Application

This application blocks the HTTP traffic going between two specified hosts. This application works on the top of switch learning application. This application is an extension of switch learning application and we have identical implementation of this application for evaluation.



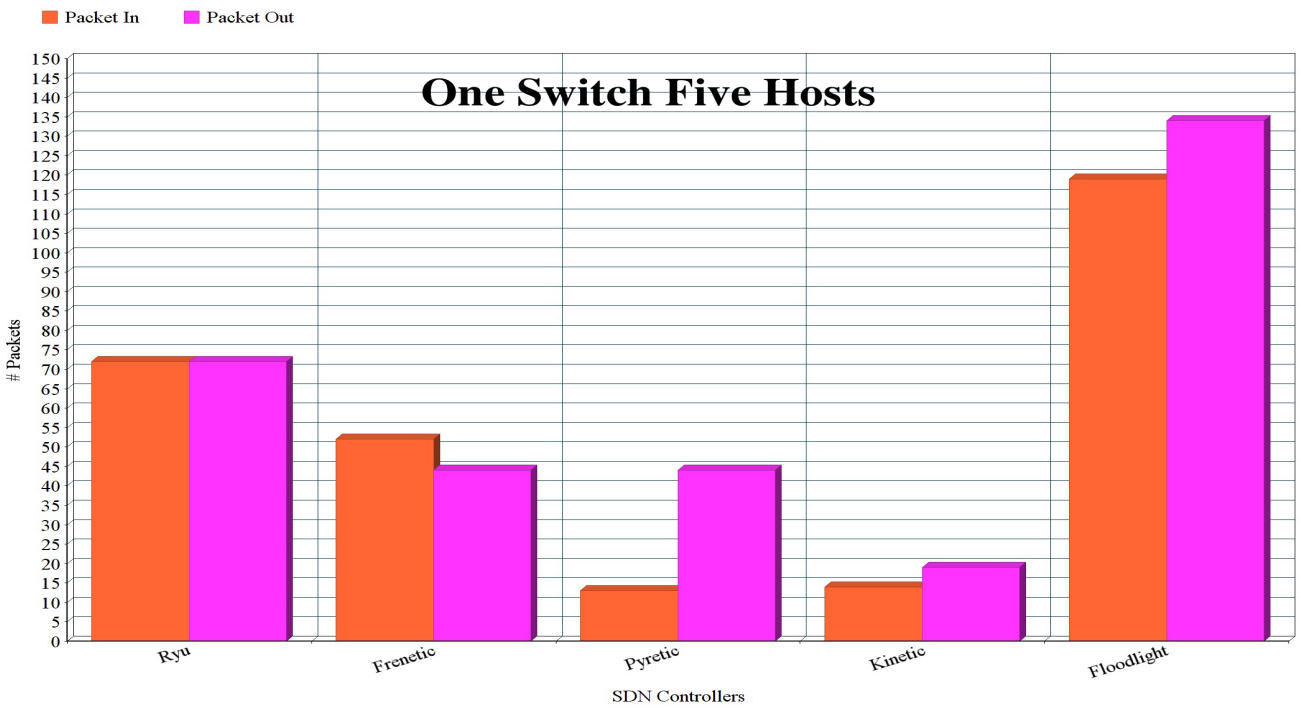
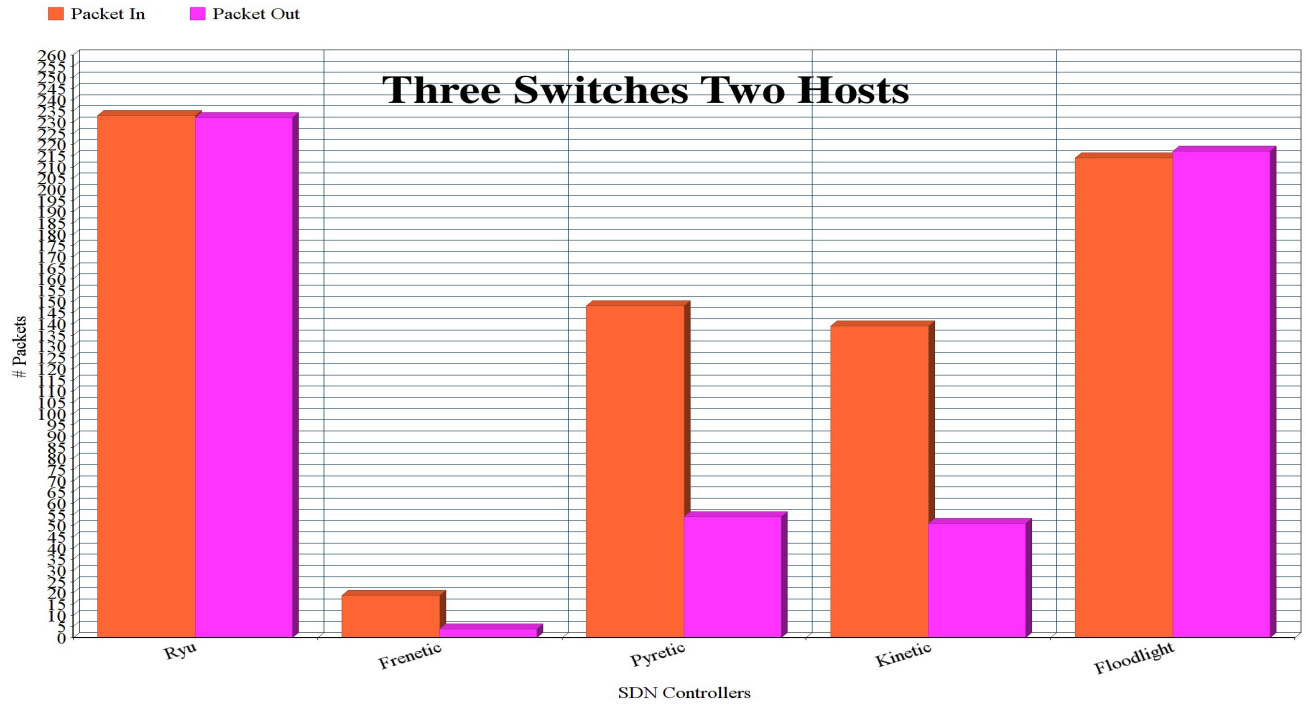




**Figure 28:** Number of packets generated/encountered by controllers (Switch Learning App)



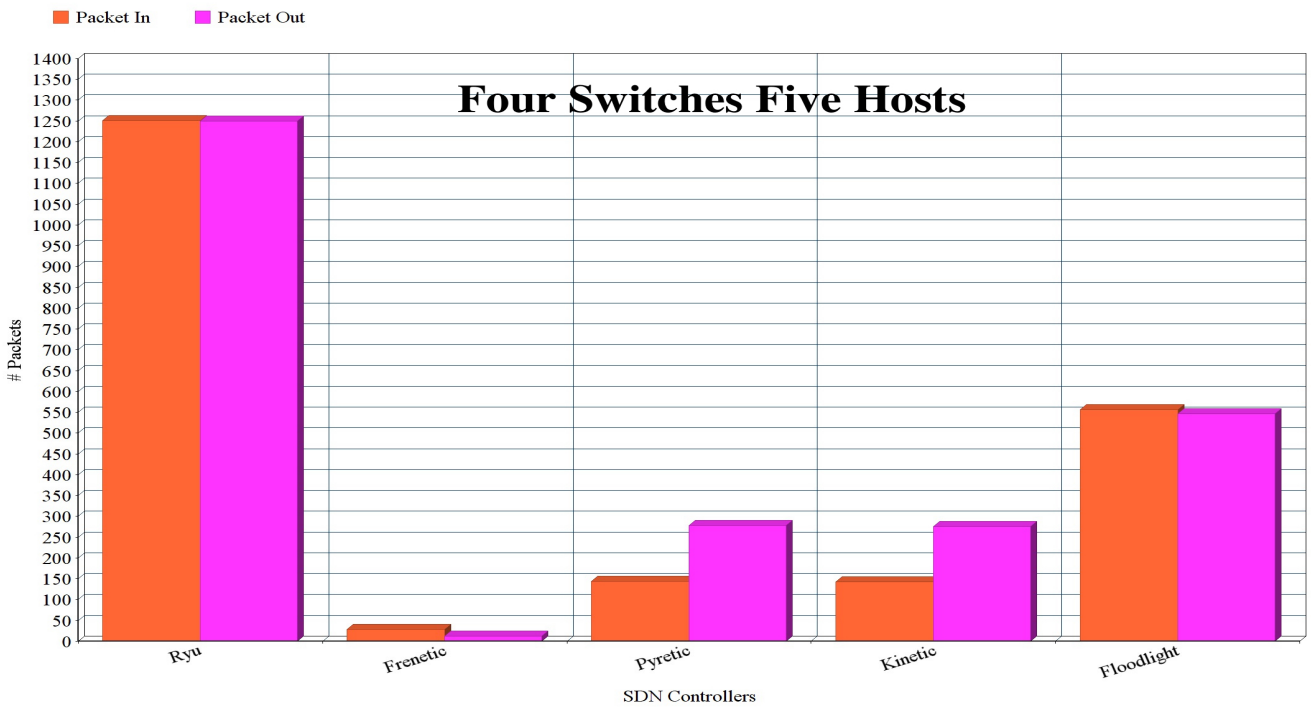




**Figure 29:** Number of packets generated/encountered by controllers (Switch Learning App)

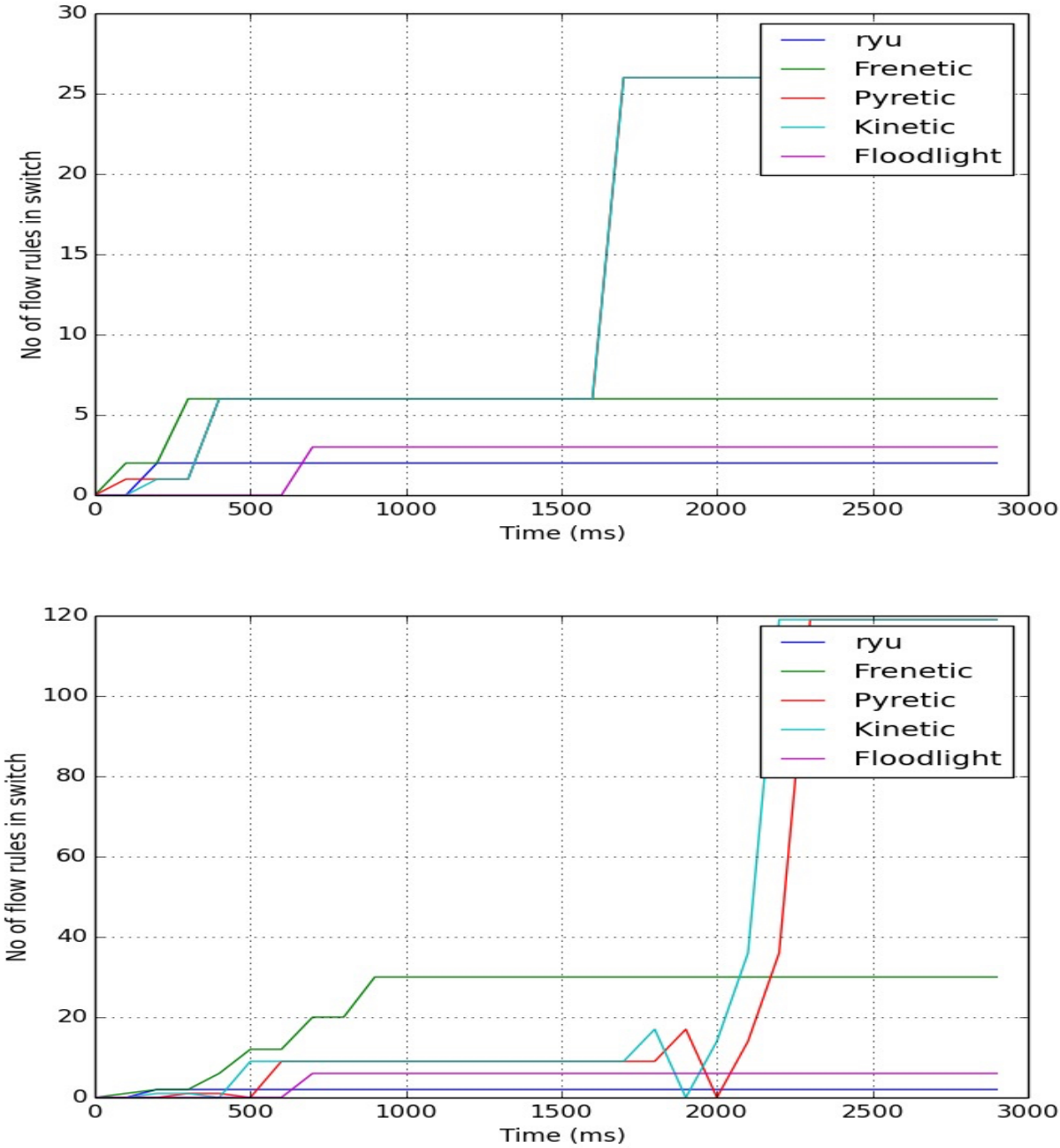



**Figure 30:** Number of rules in Switches at a particular time (Switch Learning App) - Single Switch Topology

### 6.6.1 Number of rules

Figure 34 provides information on the number of flow rules message generated by the controllers on various network topologies. In comparison to switch learning, this application generated more rules.

### 6.6.2 Number of Packets at Controller

Figures 35 and 36 represent the number of packets generated and encountered by the controllers. It can be inferred from the figures that Ryu and Floodlight received and sent higher number of packets than others. On the other hand, it is clear from the figures that Pyretic and Kinetic generated and received

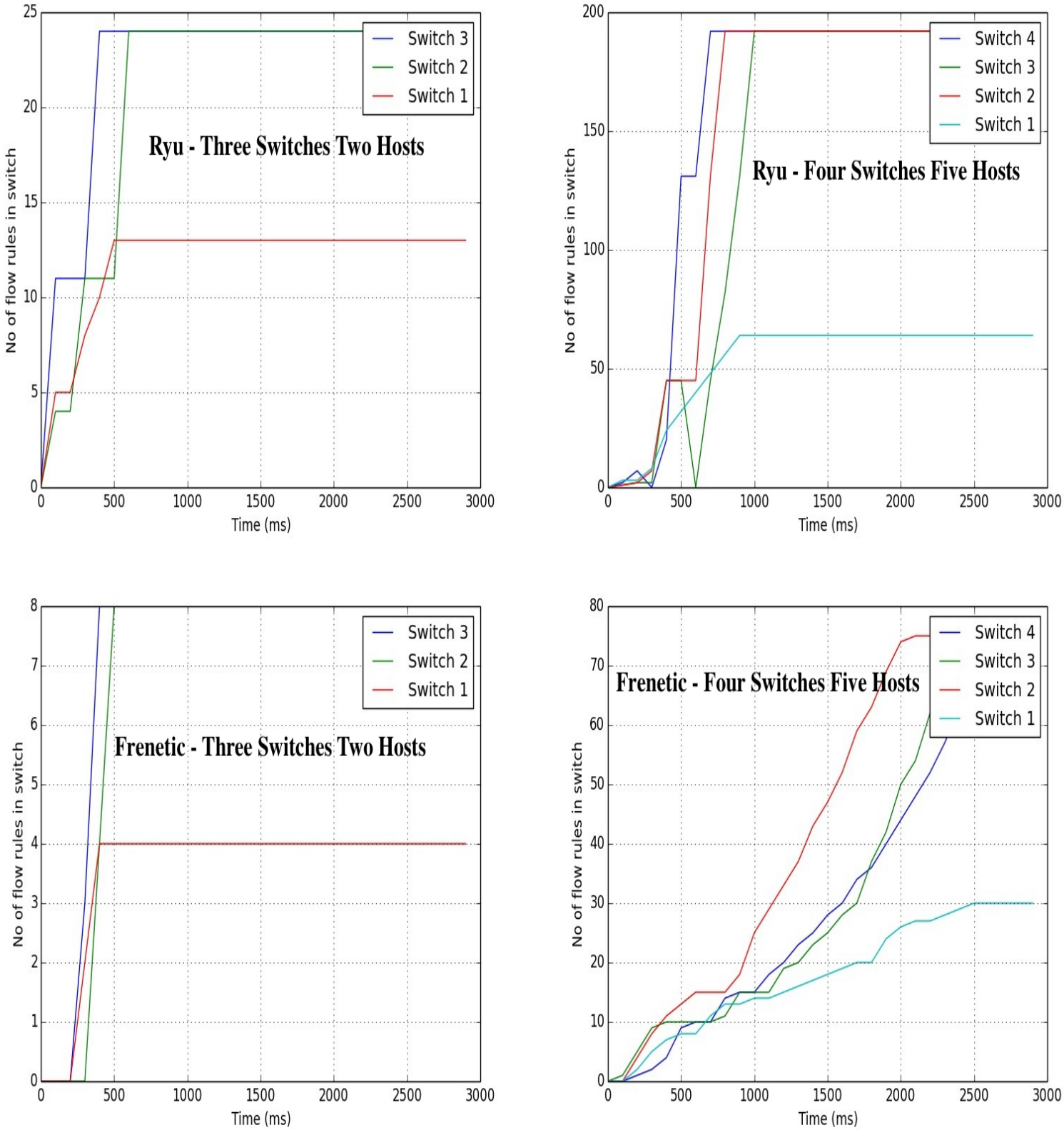



**Figure 31:** Number of packets generated/encountered by controllers (Switch Learning App) - Multiswitch Topology

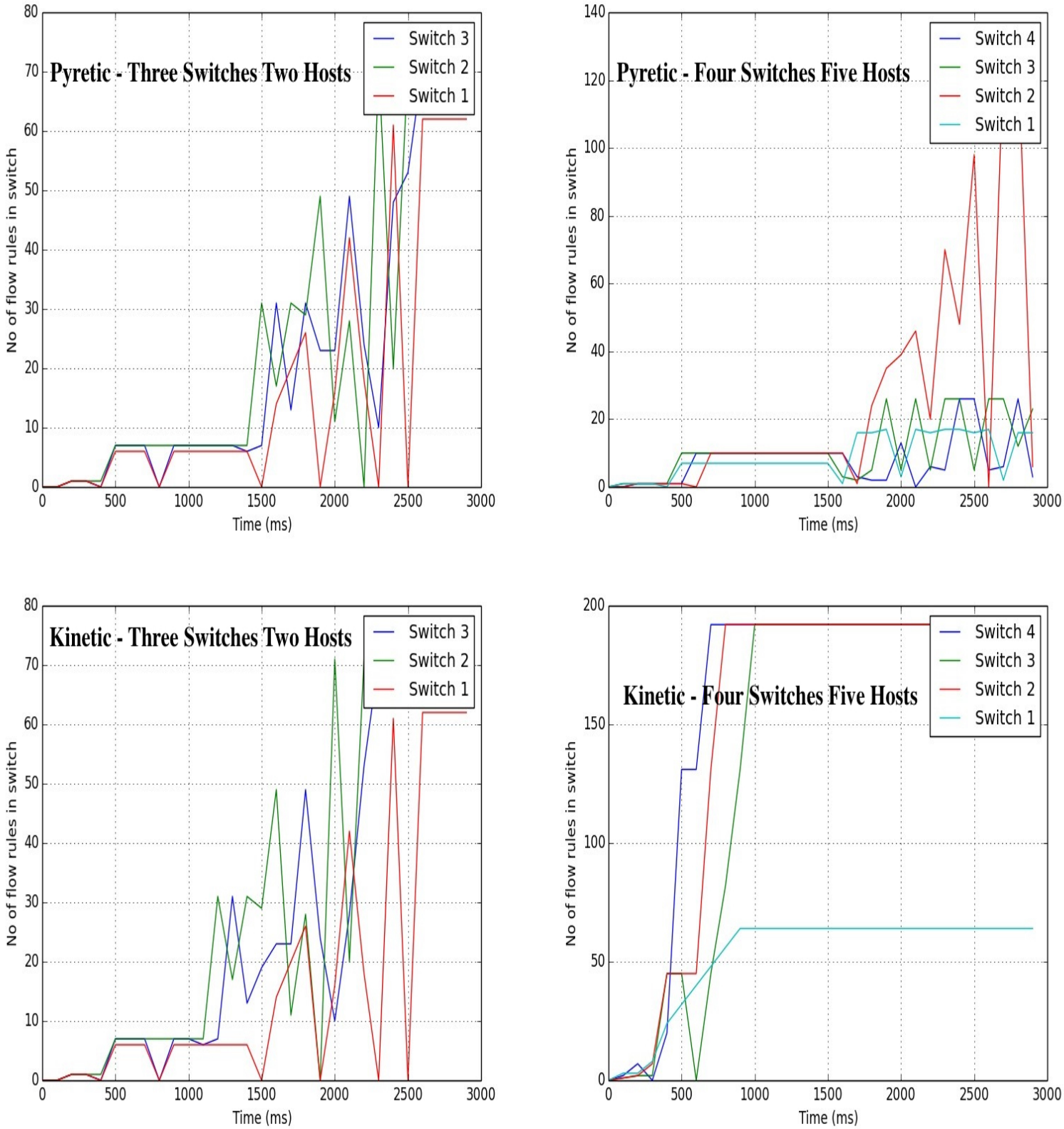
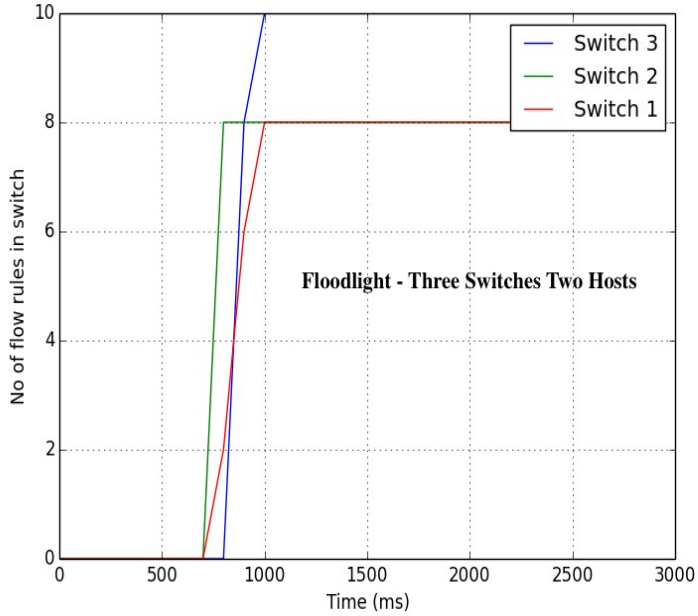



**Figure 32:** Number of packets generated/encountered by controllers (Switch Learning App) - Multiswitch Topology



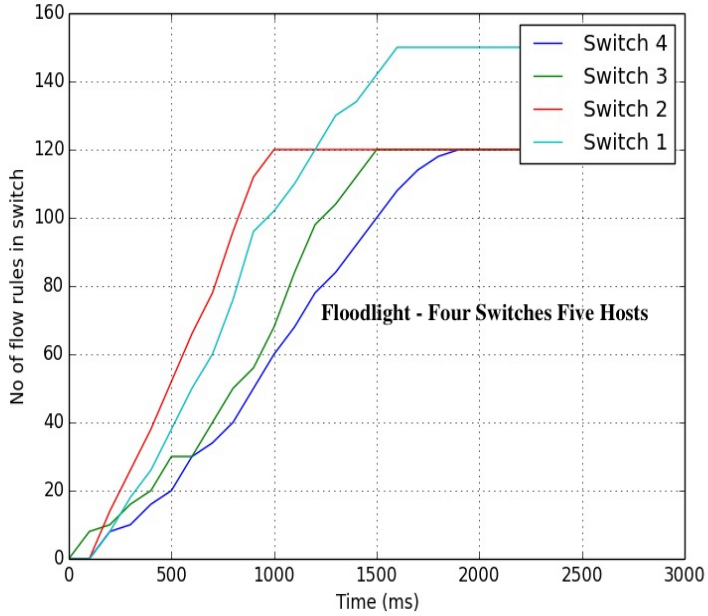




**Figure 33:** Number of packets generated/encountered by controllers (Switch Learning App) - Multiswitch Topology

relatively less packets than others.

### 6.6.3 Number of Flow rules in switch at a Particular Time

Figure 37 illustrates the number of flow rules in switch at a particular time for one switch topology (2 host and 5 hosts) for all controllers. The results for multi-switch topologies are shown in figures 38, 39 and 40. It can be deduced that for Frenetic, Pyretic and Kinetic rules are comparatively have more number of rules in switches than the others during the whole interval.

### 6.6.4 Lines of code for Web Firewall Application

Figure 41 represents the number of lines of code written. It can be deduced from that image that web firewall application is written in less number of lines in Pyretic and Kinetic languages. It is also clear from Figure that writing application using Floodlight requires higher human efforts than others. A total number of lines of code written in Ryu and Frenetic is almost same. In Frenetic, it was required to write another application for multi switch topology. Writing another program to handle multiple switches is not required for rest languages. It is clear from the image that switch learning application and web firewall is implemented with same number of lines in Frenetic.

The sections above explained the evaluation of testbed. For hub application, as shown in Figure 20, Ryu and Floodlight generated no rules and Frenetic generated only one rule. But, Pyretic and Kinetic installed more rules. However, Figures 21 and 22 depict the total number of packet in and packet out messages encountered/generated by contoller. It is clear from Figures that Pyretic and Kinetic generated less number of packets than the others. Figures 23, 24, 25 and 26 represents the number of rules installed into switches at a particular time. It is clear from figures that saturation time to install flow-rules for Ryu and Floodlight is zero because these languages did not install any rule. For Frenetic, it was

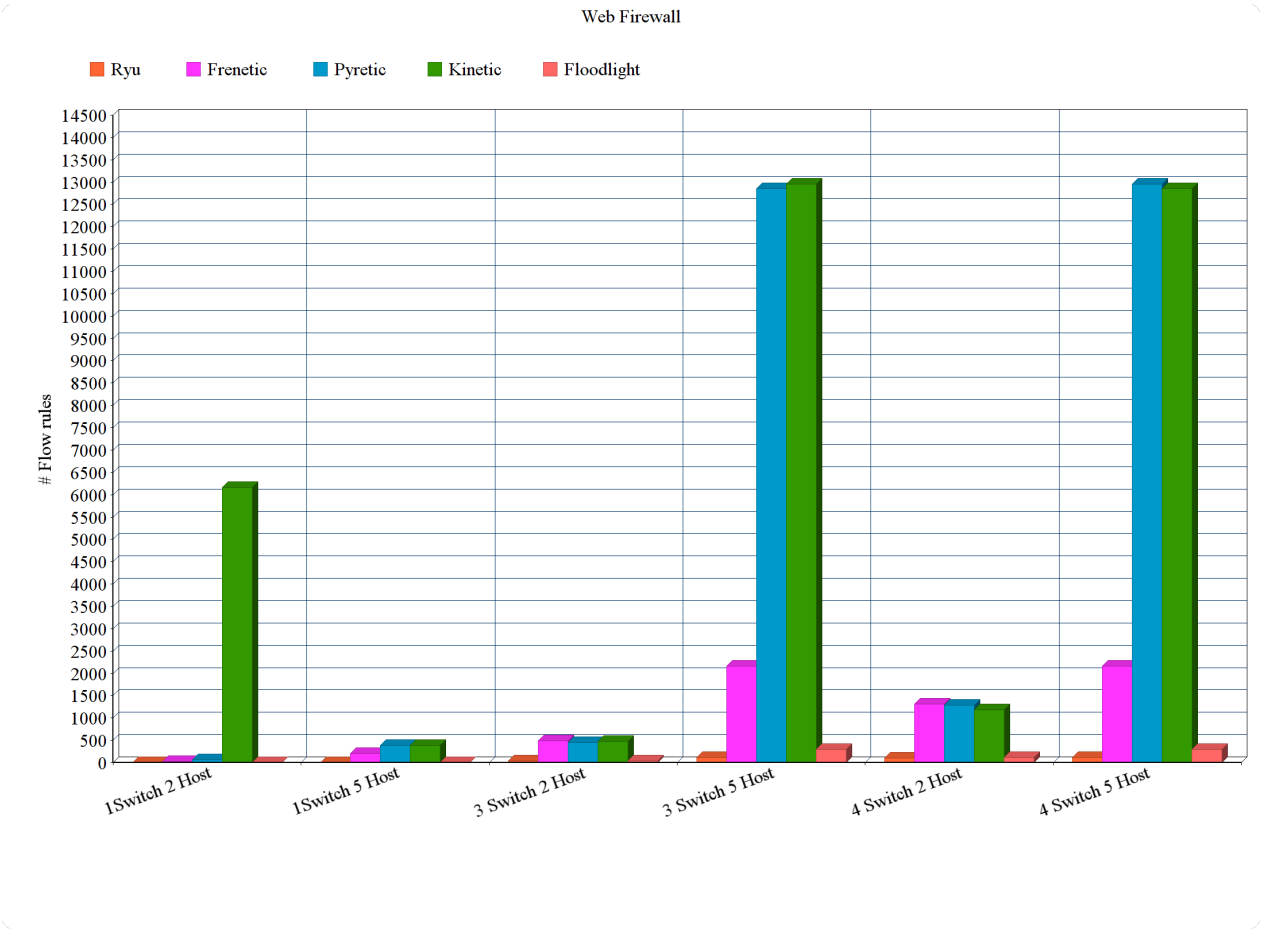
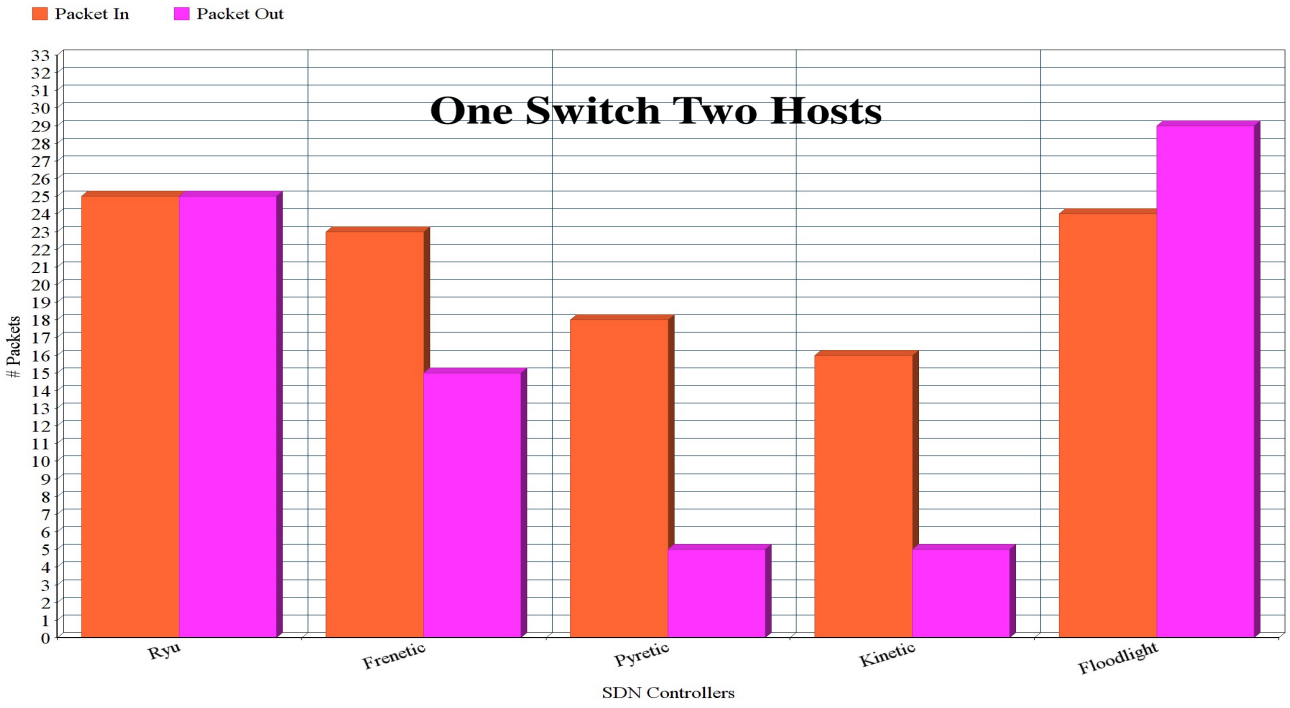



**Figure 34:** Number of Flow rules generated by controllers (Web Firewall App)

always around 100 milliseconds whereas it took more time for Pyretic and Kinetic than others.

For switch learning application, as shown in Figure 27, Ryu and Floodlight generated lesser rules than others. Pyretic and Kinetic installed more rules. Frenetic installed more rules than Ryu and Floodlight but lesser than Pyretic and Kinetic. Figures 28 and 29 depict the total number of packet in and packet out messages encountered/generated by controller. It is clear from Figures that Pyretic and Kinetic generated less number of packets than the others. Figures 30, 31, 32 and 33 displays the number of rules installed into switches at a particular time. It is clear from figures that saturation time to install flow-rules is always lesser for Ryu and Frenetic than the others. It can also be deduced from Figures that saturation time increases with increase in a number of switches and hosts. It can also be identified that there is not much difference in results for learning switch and firewall application.

When comparing languages in terms of generating traffic towards a controller, it is clear that Kinetic and Pyretic are more efficient languages than others. For generating effective rules, Ryu and Floodlight are more efficient than others. In the case of flow-rules installed into switches, Pyretic, Kinetic and Frenetic are less efficient than others.



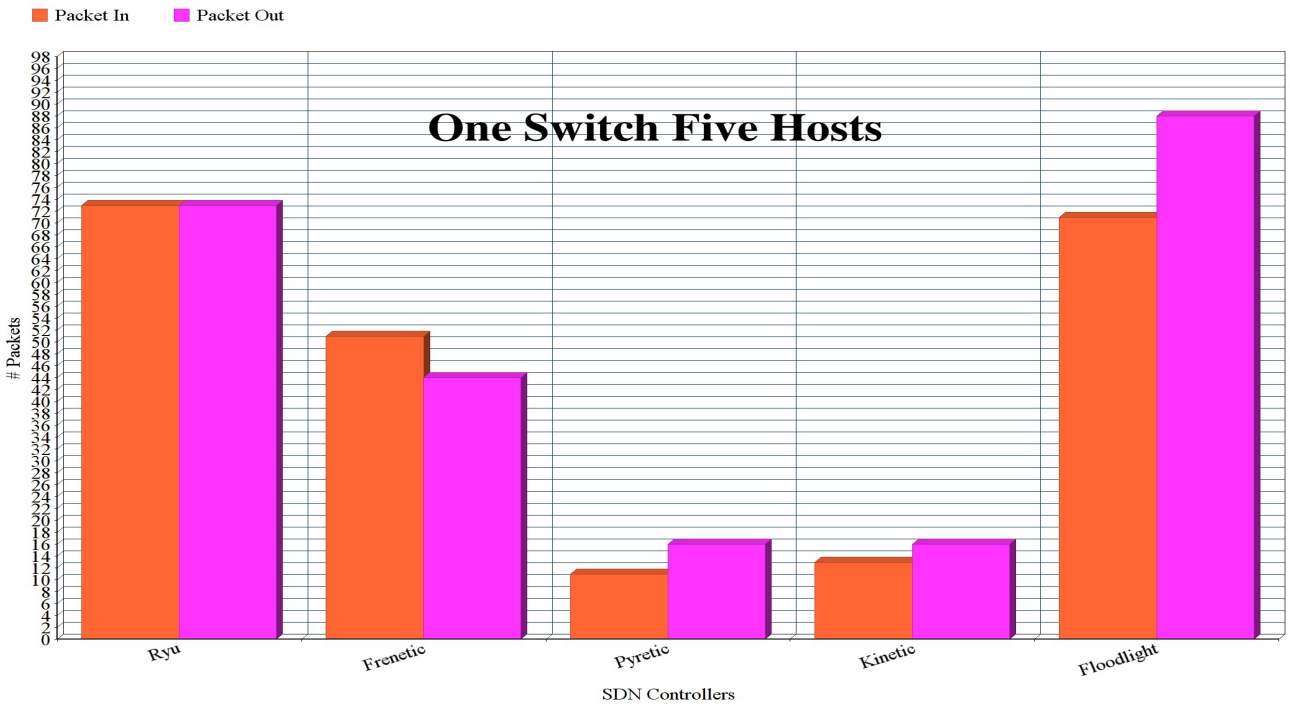
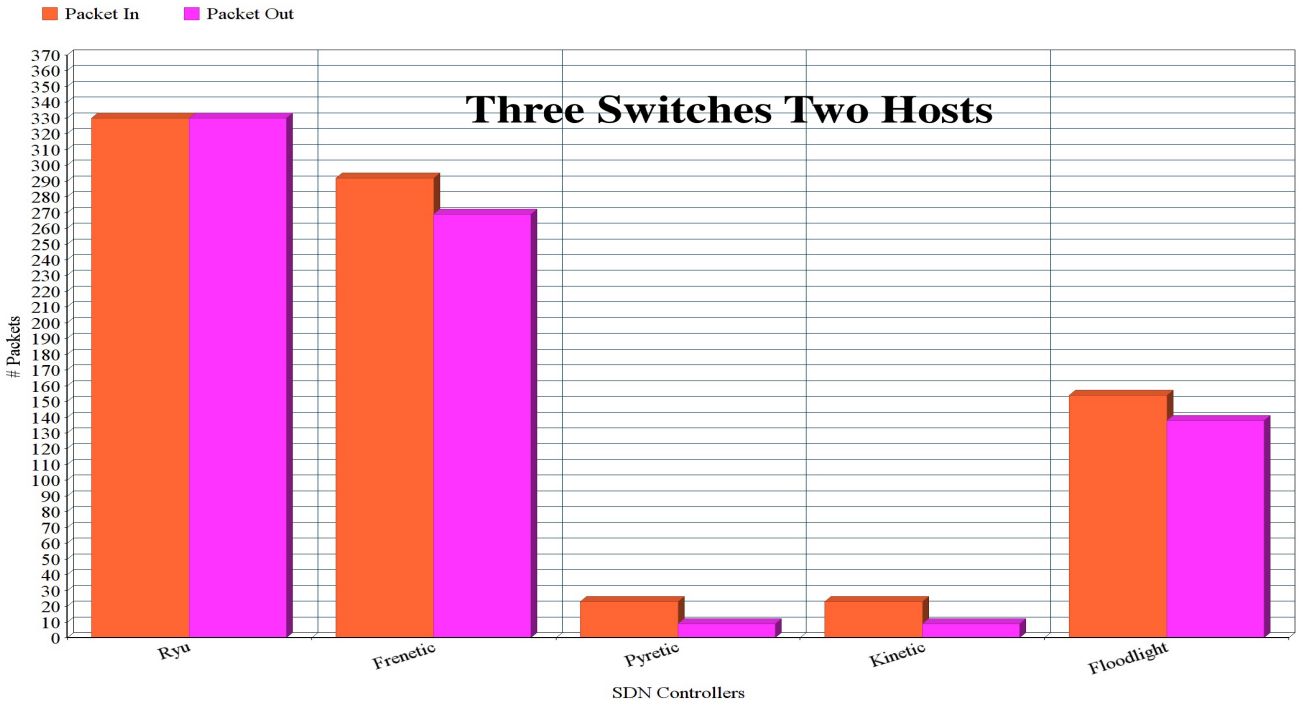




**Figure 35:** Number of packets generated/encountered by controllers (Web Firewall App)



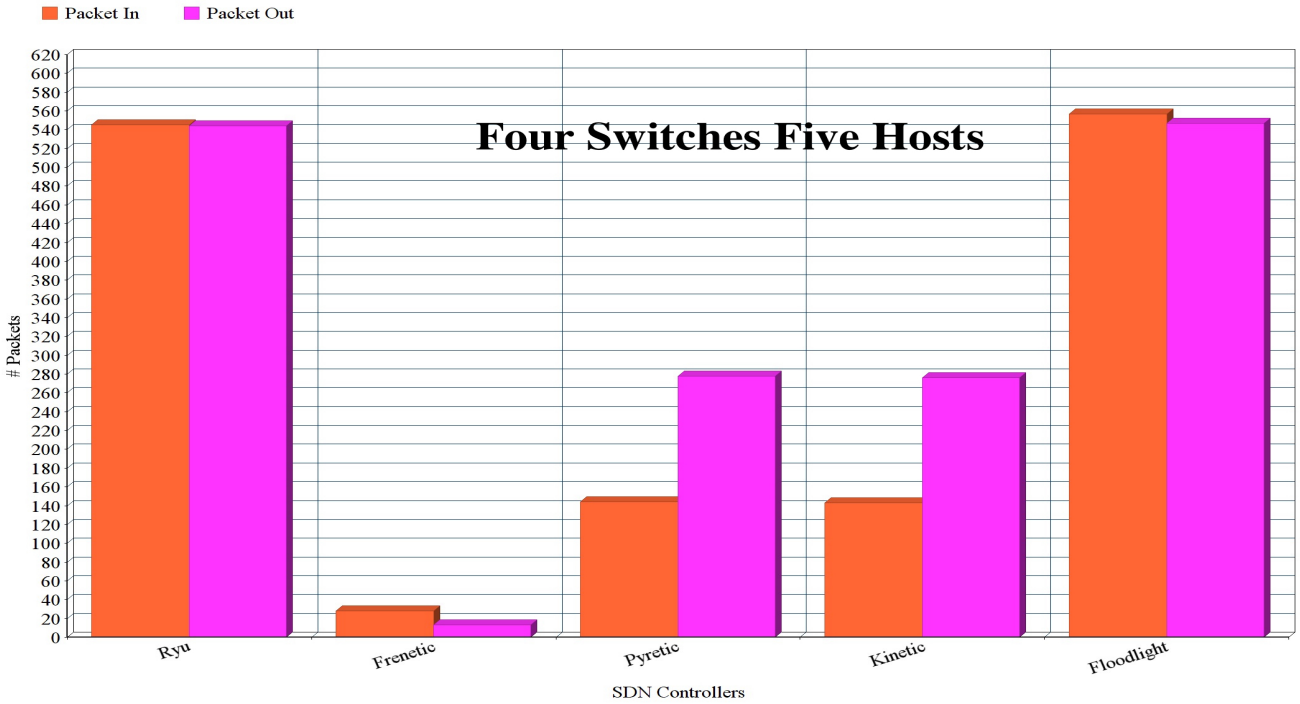




**Figure 36:** Number of packets generated/encountered by controllers (Web Firewall App)

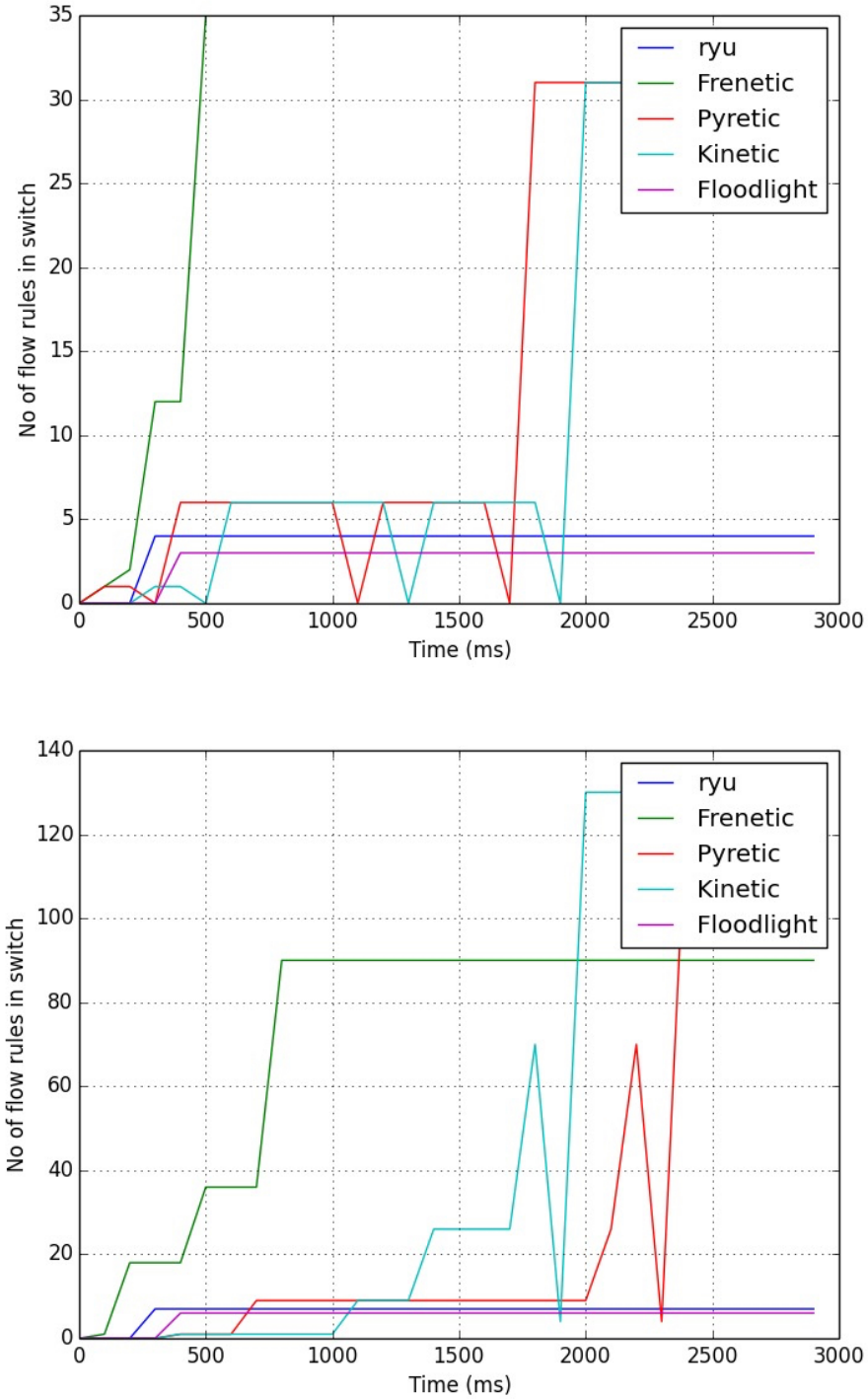



**Figure 37:** Number of rules in Switches at a particular time (Web Firewall App) - Single Switch Topology

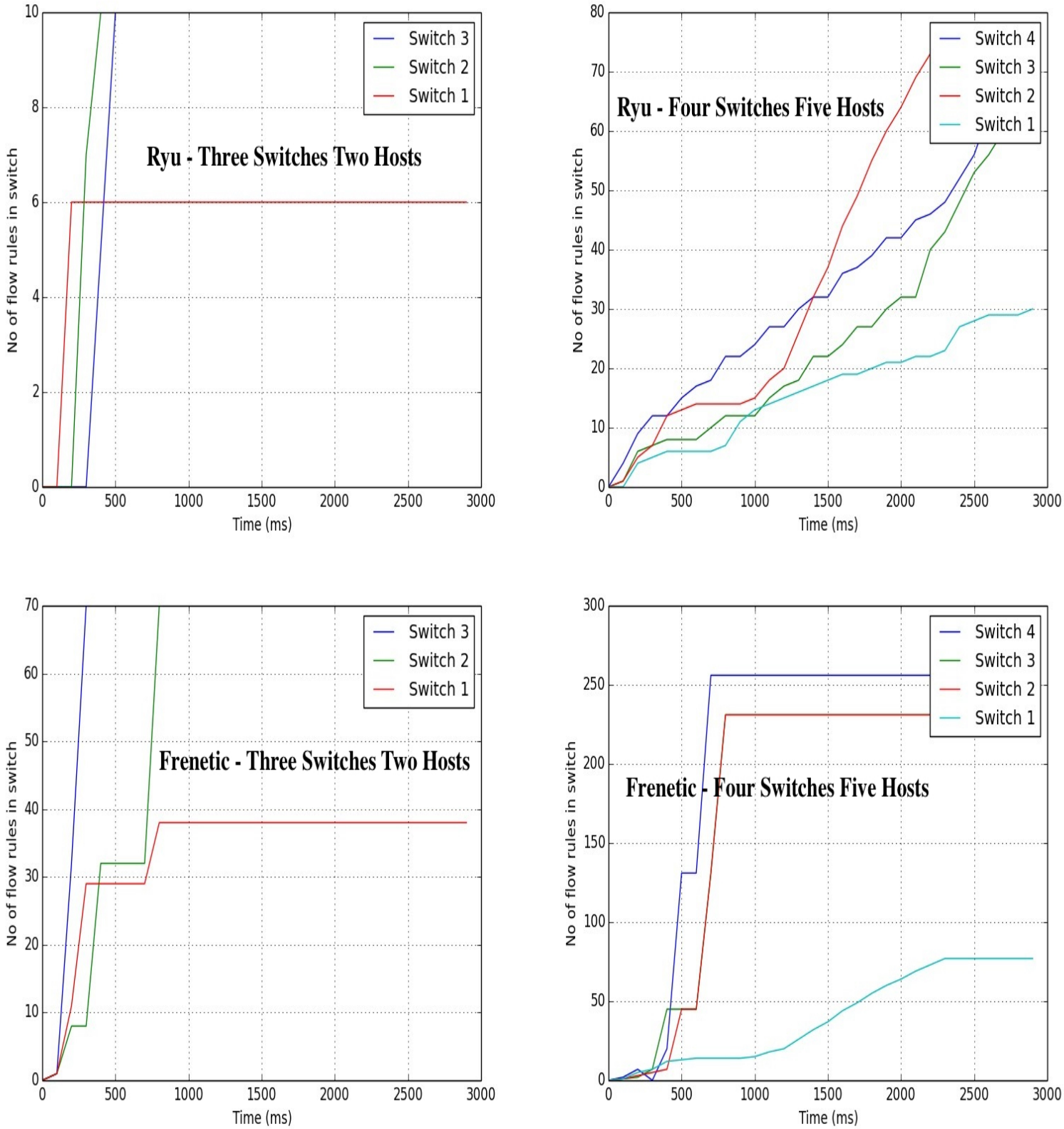



**Figure 38:** Number of packets generated/encountered by controllers (Switch Learning App) - Multiswitch Topology

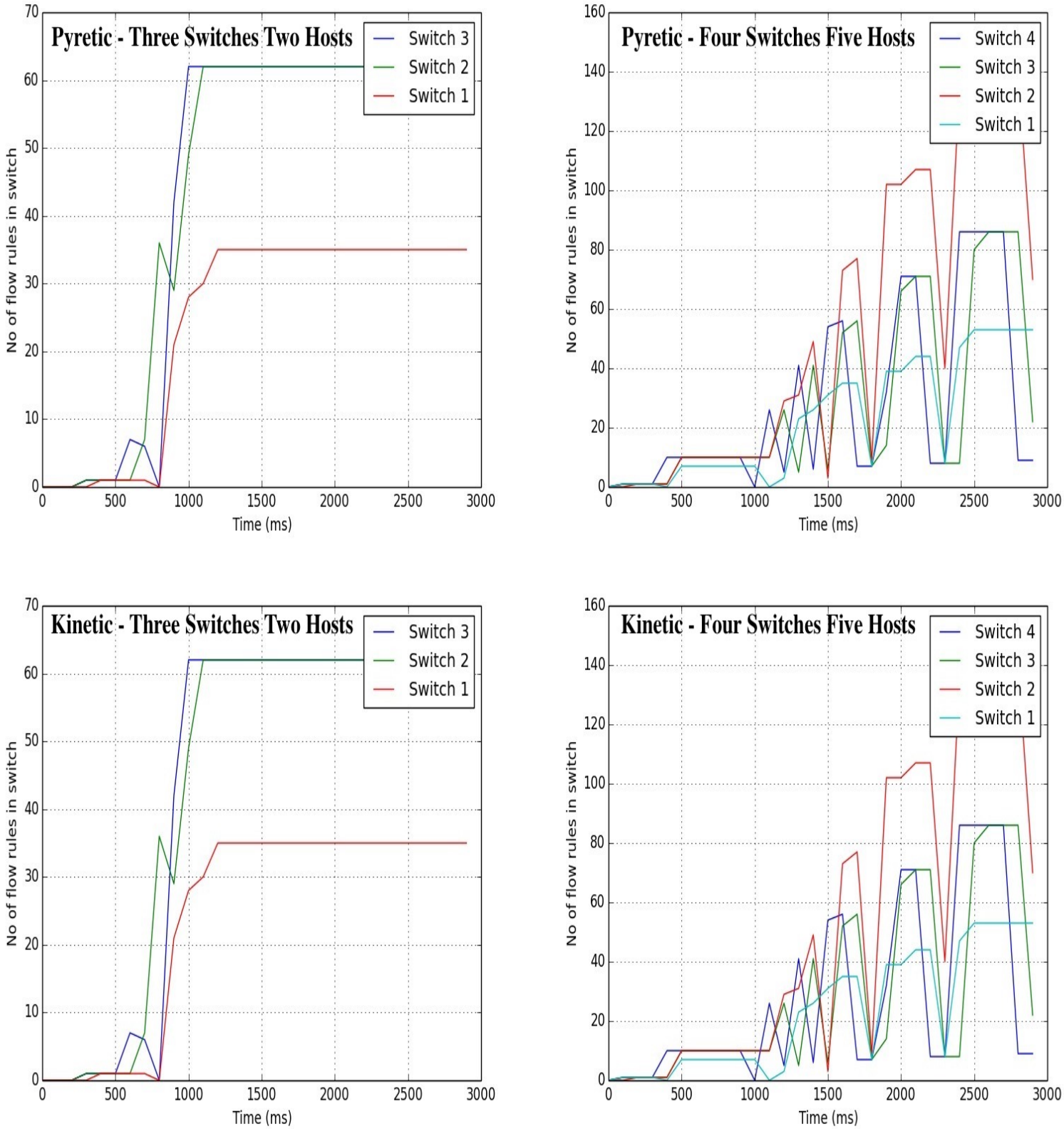
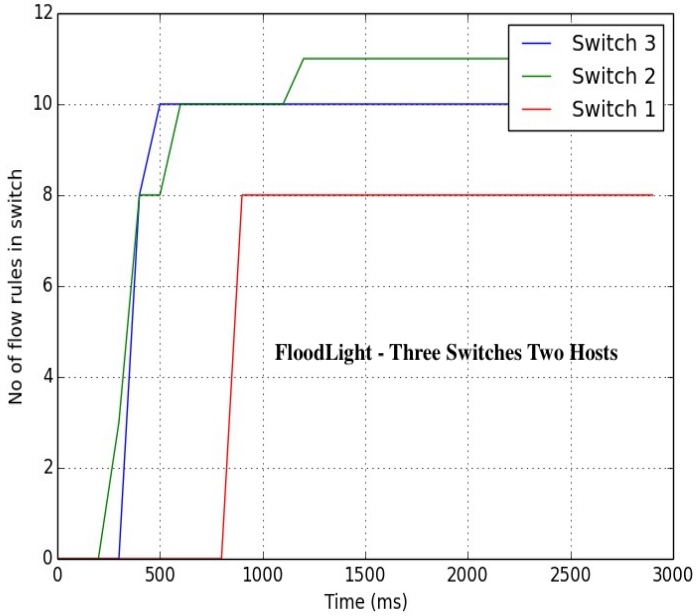



**Figure 39:** Number of packets generated/encountered by controllers (Switch Learning App) - Multiswitch Topology



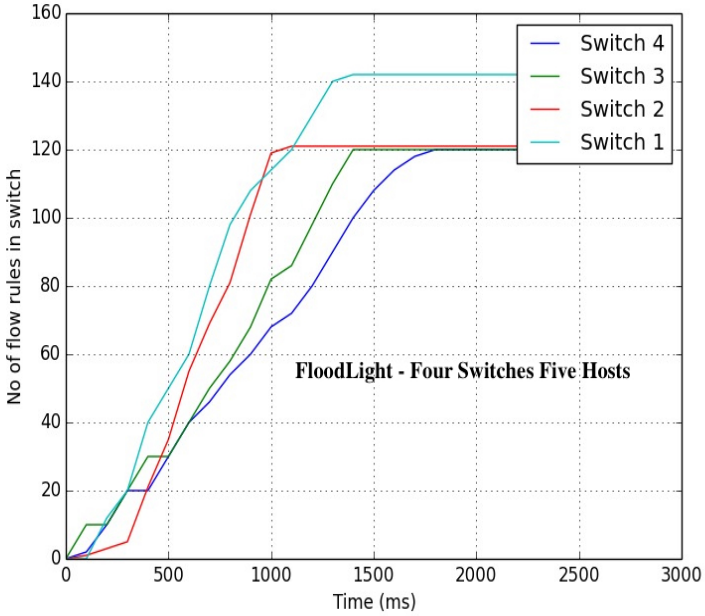




**Figure 40:** Number of packets generated/encountered by controllers (Switch Learning App) - Multiswitch Topology

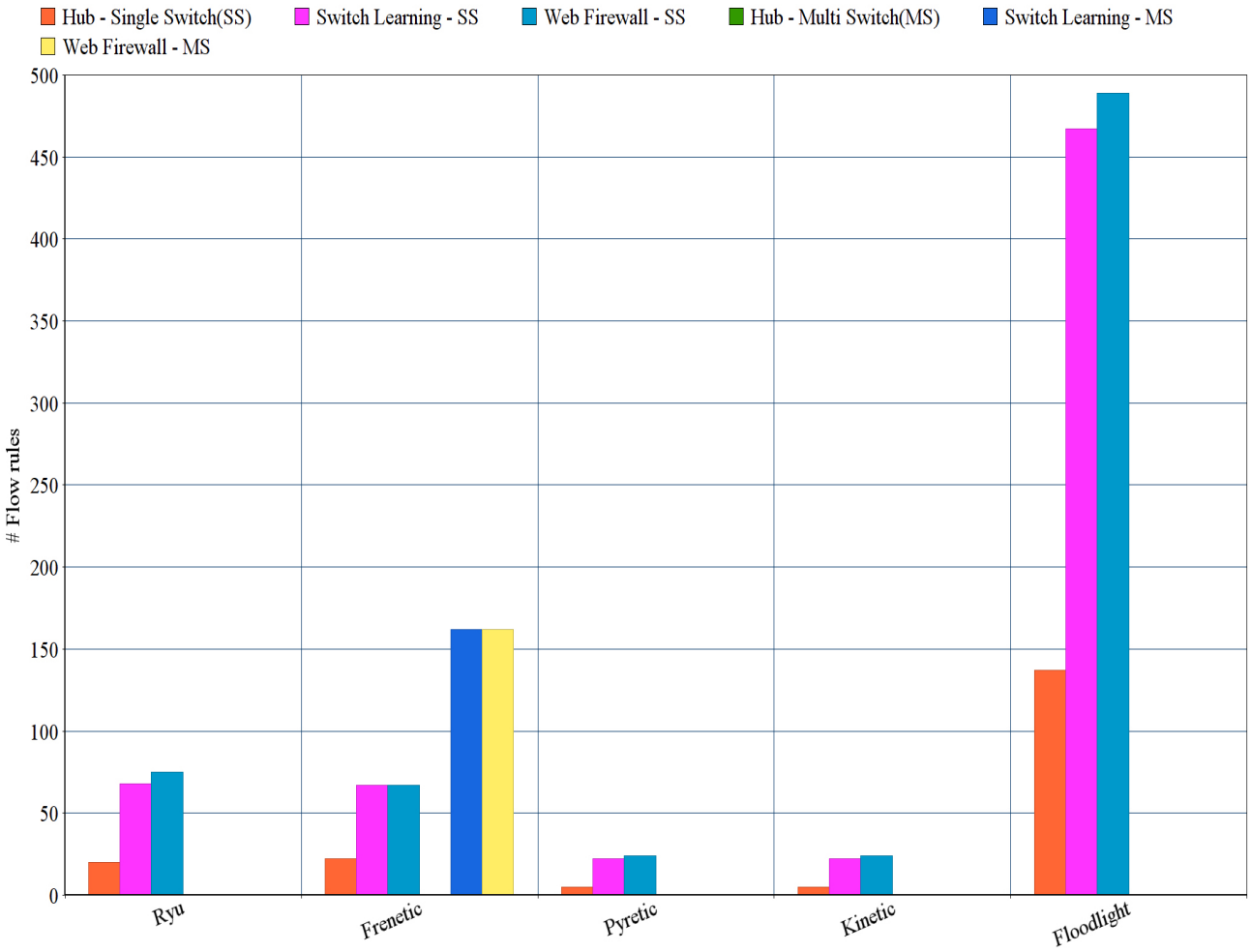



**Figure 41:** Number of Lines of code written

# 7 Conclusion and Future Work

SDN is an emerging networking topology that decouples the data plane from the control plane. It allows network programmers to control and manage a network. OpenFlow is the standard protocol to manage and control the SDN network. However, writing programs using OpenFlow is difficult. To solve this problem, there exist many SDN programming languages. There exits many testbeds for SDN as explained in section 3. However, these testbeds do not provide any special mechanism to compare SDN languages. However, there does not exist any testbed which can compare SDN languages in terms of expressivity, abstraction and performance. So, our thesis proposes an extensible testbed to compare SDN languages. This comparison could be helpful while developing a new SDN language features.

This thesis contributes a study of SDN languages and their corresponding controllers, test cases for comparison, and few SDN applications. In order to show the feasibility, this thesis presents a prototypical implementation of the testbed and test cases. Each test case was evaluated for three applications (section 5.5) on various network topologies. The observations of the evaluation are as follows.

- Flow rules generated by Frenetic, Pyretic and Kinetic languages increase exponentially with increase of hosts and/or switches in the network.
- Frenetic, Pyretic and Kinetic languages also generate some rules which are never used.
- Pyretic and Kinetic showed almost the same behaviour for each application and test case on different topologies.
- Ryu and Floodlight controllers handle/generate more packets than Frenetic, Pyretic and Kinetic.
- For Hub application, Floodlight and Ryu did not install any rule into switches. Frenetic installed only one rule. It implies that these languages were more efficient in case of generating and installing flow-rules. However, these are less efficient than Pyretic and Kinetic for sending traffic towards controller.
- For same application and network topology, the total number of rules installed by Pyretic and Kinetic was greater than others.

Generally, the evaluation results suggest that the approach works and can contribute as a solution to the previously stated problems. From the above observations, we conclude that our system was successfully able to compare SDN languages.

## 7.1 Future Work

While implementing the testbed, we came across many SDN languages. Few languages were chosen to compare in this thesis. We did not consider all the languages for comparison in this testbed such as etc. Adding those languages for comparison may provide some more insights and interesting results.

As mentioned in section 4.3, this thesis assumes that all controllers are installed on the local machine. However, there are possibilities of having a different controller installed on a different machine. To solve this problem, this testbed needs improvement so that it can communicate with remote machines. This communication can also be realized using an API that offers a mechanism to communicate in order to run experiments and collect results. This can also be achieved by leveraging the SSH protocol to connect to the remote machine.

The testbed created in this thesis offers the Python programming interface to specify Testsuites. Therefore, it is could be an obstacle for non-Python users. This is another scope of improvement. The Graphical User Interface (GUI) could be a good approach to solve this. A web-based interface is also a good candidate to provide that can be used by programmers or non-programmers. This can be achieved with writing the front end in HTML. A server-side language such as PHP is also required to handle user requests and communicate with testbed.

At present, this testbed has three different SDN applications. Implementing more applications such as VLAN may present more interesting results.

We compared SDN languages on four evaluation criteria. Having more criteria can tell more about SDN languages and their characteristics.

## References


[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic Foundations for Networks. In *ACM SIGPLAN Notices*, volume 49, pages 113–126. ACM, 2014.

[2] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, pages 1–6. ACM, 2014.

[3] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. Geni: A Federated for Innovative Network Experiments. *Computer Networks*, 61:5–23, 2014.

[4] Lionel Bertaux, Samir Medjiah, Pascal Berthou, Slim Abdellatif, Akram Hakiri, Patrick Gelard, Fabrice Planchou, and Marc Bruyere. Software Defined Networking and Virtualization for Broadband satellite networks. *IEEE Communications Magazine*, 53(3):54–60, 2015.

[5] Martin Casado, Nate Foster, and Arjun Guha. Abstractions for Software-defined Networks. *Commun. ACM*, 57(10):86–95, September 2014.

[6] Matthew Conway, Randy Pausch, Rich Gossweiler, and Tommy Burnette. Alice: A Rapid Prototyping System for Building Virtual Environments. In *Conference companion on Human factors in computing systems*, pages 295–296. ACM, 1994.

[7] Roberto Doriguzzi Corin, Pedro A Aranda Gutiérrez TID, Alec Leckey, and Carmen Guerrero. Netide App Engine 1st Release. `http://www.netide.eu/sites/www.netide.eu/files/documents/NetIDE_D4_1_revised.pdf`, 2015. [Online; last accessed 11-April-2017].

[8] Federico M Facca, Elio Salvadori, Holger Karl, Diego R López, Pedro Andrés Aranda Gutiérrez, Dejan Kostic, and Roberto Riggio. Netide: First Steps Towards an Integrated Development Environment for Portable Network apps. In *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pages 105–110. IEEE, 2013.

[9] Nate Foster, Michael J Freedman, Rob Harrison, Jennifer Rexford, Matthew L Meola, and David Walker. Frenetic: A High-level Language for Openflow Networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, page 6. ACM, 2010.

[10] Nate Foster, Michael J Freedman, Rob Harrison, Jennifer Rexford, Matthew L Meola, and David Walker. Frenetic: A high-level Language for Openflow Networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, page 6. ACM, 2010.

[11] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *ACM Sigplan Notices*, volume 46, pages 279–291. ACM, 2011.

[12] Open Networking Foundation. SDN Architecture. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf`, 2014. [Online; last accessed 30-March-2017].

[13] Nádia Pires Gonçalves. A Testbed for Research and Development of SDN Applications using Openflow. `https://fenix.tecnico.ulisboa.pt/downloadFile/395146459058/dissertacao.pdf`. [Online; last accessed 11-April-2017].

[14] Paul Goransson, Chuck Black, and Timothy Culver. Software-defined networks: A Comprehensive Approach. https://books.google.de/books?isbn=0128045795, 2016. [Online; last accessed 11-April-2017].

[15] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[16] Vidya B. Harkal and A.A.deshmukh. Article: Software-defined Networking with Floodlight Controller. *IJCA Proceedings on International Conference on Internet of Things, Next Generation Networks and Cloud Computing*, ICINC 2016(3):23–27, July 2016.

[17] Brandon Heller. Openflow Switch Specification. http://yuba.stanford.edu/~derickso/openflow-spec-v1.0.0-cookieenhancements.pdf, 2009. [Online; last accessed 11-April-2017].

[18] Timothy L Hinrichs, Natasha S Gude, Martin Casado, John C Mitchell, and Scott Shenker. Practical Declarative Network Management. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 1–10. ACM, 2009.

[19] Tao Huang, F Richard Yu, Chen Zhang, Jiang Liu, Jiao Zhang, and Junjie Liu. A Survey on Large-scale Software defined Networking (SDN) Testbeds: Approaches and Challenges. *IEEE Communications Surveys & Tutorials*, 2016.

[20] Sukhveer Kaur, Japinder Singh, and Navtej Singh Ghumman. Network Programmability using POX Controller. In *ICCCS International Conference on Communication, Computing & Systems, IEEE*, number s 134, page 138, 2014.

[21] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russell J Clark. Kinetic: Verifiable Dynamic Network Control. In *NSDI*, pages 59–72, 2015.

[22] J Kim, B Cha, Jongryool Kim, Namgon Lucas Kim, Gyeongsoo Noh, Youngwan Jang, Hyeong Geun An, Hongsik Park, JiHoon Hong, DongSeok Jang, et al. Of@ tein: An Openflow-enabled SDN Testbed over International Smartx Rack sites. *Proceedings of the Asia-Pacific Advanced Network*, 36:17–22, 2013.

[23] Keith Kirkpatrick. Software-defined Networking. *Communications of the ACM*, 56(9):16–19, 2013.

[24] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[25] Ulf Lamping and Ed Warnicke. Wireshark User's guide. https://www.wireshark.org/download/docs/user-guide-a4.pdf, 2004. [Online; last accessed 01-April-2017].

[26] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[27] Nick McKeown. Software-defined Networking. https://www.cs.rutgers.edu/~badri/552dir/papers/intro/nick09.pdf. [Online; last accessed 11-April-2017].

[28] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[29] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a Model-driven SDN Controller Architecture. In *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on*, pages 1–6. IEEE, 2014.

[30] Borja Merino. *Instant Traffic Analysis with Tshark How-to*. Packt Publishing Ltd, 2013.

[31] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. Tierless Programming and Reasoning for Software-defined Networks. In *NSDI*, volume 14, pages 519–531, 2014.

[32] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular SDN Programming with Pyretic. *Technical Report of USENIX*, 2013.

[33] Aris Cahyadi Risdianto, Junsik Shin, and JongWon Kim. Building and Operating Distributed SDN-cloud Testbed with Hyper-convergent Smartx boxes. In *International Conference on Cloud Computing*, pages 224–233. Springer, 2015.

[34] Guillermo Romero de Tejada Muntaner. Evaluation of OpenFlow Controllers. `http://kth.diva-portal.org/smash/get/diva2:563469/FULLTEXT01.pdf`. [Online; last accessed 11-April-2017].

[35] Ryu. Component-based Software Defined Networking Framework. `https://osrg.github.io/ryu/`. [Online; last accessed 30-March-2017].

[36] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. Openstack: Toward an Open-source Solution for Cloud Computing. *International Journal of Computer Applications*, 55(3), 2012.

[37] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Flowvisor: A Network Virtualization Layer. *OpenFlow Switch Consortium, Tech. Rep*, pages 1–13, 2009.

[38] R. Soulé. Network Programming Languages. `http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/netpl/lecture01-introduction.pdf`. [Online; last accessed 30-March-2017].

[39] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A Language for Provisioning Network Resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 213–226. ACM, 2014.

[40] Gary N Stone, Bert Lundy, and Geoffrey G Xie. Network Policy Languages: A Survey and a New Approach. *IEEE network*, 15(1):10–21, 2001.

[41] Marc Suñé, Leonardo Bergesio, Hagen Woesner, Tom Rothe, Andreas Köpsel, Didier Colle, Bart Puype, Dimitra Simeonidou, Reza Nejabati, Mayur Channegowda, et al. Design and Implementation of the OFELIA FP7 Facility: The European Openflow Testbed. *Computer Networks*, 61:132–150, 2014.

[42] Andreas Voellmy and Paul Hudak. Nettle: Taking the Sting out of Programming Network Routers. In *International Symposium on Practical Aspects of Declarative Languages*, pages 235–249. Springer, 2011.

[43] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN Programming using Algorithmic Policies. *ACM SIGCOMM Computer Communication Review*, 43(4):87–98, 2013.

[44] Hubert Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on communications*, 28(4):425–432, 1980.