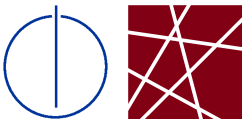
Technische Universität München
Fakultät für Informatik
Lehrstuhl für Netzarchitekturen und Netzdienste



Implementation and Performance Analysis of Firewall on Open vSwitch

Interdisziplinäres Projekt in der Elektrotechnik

durchgeführt am
Lehrstuhl für Netzarchitekturen und Netzdienste
Fakultät für Informatik
Technische Universität München

von

Jay Shah

Aufgabensteller:	Prof. Dr.-Ing. Georg Carle
Betreuer:	M.Sc. Cornelius Diekmann and M.Sc. Florian Wohlfart
Tag der Abgabe:	29. April 2015

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Garching, den 29. April 2015

Abstract:

Software Defined Networking (SDN) is a current research trend that follows the ideology of physical separation of the control and data plane of the forwarding devices. SDN mainly advocates with two types of devices: (1) Controllers, that implement the control plane and (2) Switches, that perform the data plane operations. OpenFlow protocol (OFP) is the current standard through which controllers and switches can communicate with each other. Using OpenFlow, SDN controllers can manage forwarding behaviors of SDN switches by managing Flow Table entries. Switches use these low-level Flow Table entries to forward packets to appropriate hosts.

Firewalls are integral part of today's networks. We can't imagine our network without a Firewall which protects our network from potential threats. As SDN is getting pace in replacing traditional architecture, it would be very interesting to see how much security features can be provided by OpenFlow-enabled switches. Hence, it will be very important to see if SDN, on the top of OpenFlow, can efficiently implement Firewalls and provides support for an advanced feature like connection tracking. The task is straightforward: Controller will add Flow Table entries on switches based upon Firewall rules. Such way, we can enhance packet-processing by providing security.

In this Document, one strategy for implementing Firewall on SDN is presented. We can write some controller applications that work as Firewall and inspect incoming packets against the Firewall rules. These applications are also able to implement connection tracking mechanism. As SDN devices for the experiments, we selected Ryu controller and Open vSwitch. Initially, such applications are tested on local machine with small Firewall ruleset. Later, they are tested with real-world traffic and comparatively large Firewall ruleset. The testing results present that such strategy can be used as a first step in implementing security features (including connection tracking) in SDN environment.

Contents

1. Problem Analysis

This chapter illustrates the technologies that are used in this IDP. Section 1.1 discusses the concept and need for Software Defined Networking along with architectural diagram. In the next Section 1.2, OpenFlow protocol, message exchange types and Open vSwitch are explained. Ryu controller, its architecture & features are described in Section 1.3, whereas Section 1.4 demonstrates Firewall and its implementation strategy on Ryu.

1.1 Software Defined Networking

Software Defined Networking (SDN) is a rising network architecture which splits the functions of networking devices into two groups, namely network control and forwarding; where network control is directly programmable. Earlier, these functions were tightly coupled within a device. But, since they are separated into two individual functions, they make network a logical entity by abstracting the underlying complex architecture for upper layer networking applications and services [1].

SDN has risen from the failure of current networking technologies to meet current market needs [1],[2]. Static nature of a network fails to adapt network traffic and user demands that are tremendously growing. Carriers and enterprises seek to deploy new capabilities and services, however lack of standard and open interfaces between networking devices limit their abilities. Although enterprises are encouraging cloud services, scaling of computing and networking resources has been a painful task yet to be solved. These issues put forward a need of a new standard which should be capable of overcoming them; and hence, Software Defined Networking has been introduced [3],[4].

The architecture of Software Defined Networking, which is illustrated in Figure 1.1, consists of 3 major parts, namely Application Layer, Control Layer and underlying Infrastructure Layer. Unlike traditional network architecture where each device possesses a separate control plane, in SDN architecture it is taken out and made centralized on a remote process (called controller) running at Control Layer. This remote process (controller) provides global view of the network. As a result, the applications and services running at Application Layer appear to be running on a single, logical network switch. Network carriers and companies face significantly less complexity in designing and operating a network since the Infrastructure Layer becomes vendor-independent with the introduction of SDN. Infrastructure Layer devices (for example: switches and routers which are also referred as forwarding devices) should be configured to understand instructions from SDN controllers only. This makes the configuration process easy and fast.

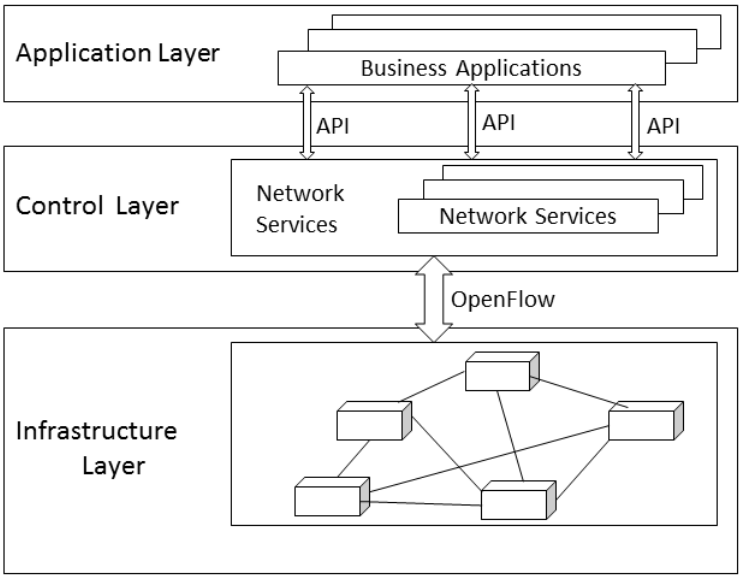


Figure 1.1: Software Defined Networking: Architectural Diagram [1]

Since the Control Layer has become programmable, Network Managers can easily configure the state of the network by writing some SDN programs [5]. These programs break the vendor specific software dependencies of the devices and make them self-contained. In addition to network abstraction, common network services like routing, multicasting, security, load balancing etc. can also be implemented on SDN architecture to achieve business objectives [6]. These advantages and features make SDN right candidate to overcome current network problems and establish a new norm of Networking.

1.2 OpenFlow and Open vSwitch

1.2.1 Introduction to OpenFlow

OpenFlow Protocol(OFP) is the first, industry-specified, standardized SDN protocol which defines a way for the controller to communicate with switches. The purpose behind it is to provide standardized specification for communication interfaces of Control and Infrastructure Layer devices. It allows manipulation of the data stored on forwarding devices to refine policy-based decision for packet forwarding [7].

OpenFlow is implemented on both ends of communication: On SDN controller and on forwarding devices. With OpenFlow, each forwarding device can export their network interfaces to the controller which will manage their forwarding states. The managed states are segregated into Flow Tables on such devices, which are nothing but set of packet header fields and associated actions. Some examples of these set of fields are standard Ethernet, IP and transport protocol fields which are roughly equivalent to the fields used in ASIC match. The actions associated with these fields are basically common packet operations like sending a packet to some ports or modifying protocol fields [8]. OpenFlow has been consistently revising under various versions and in this IDP, OpenFlow version 1.3 is used.

Deploying OFP-enabled SDN on physical (and virtual) networks is thought to be an easy process for an enterprise to gradually introduce SDN on the existing network infrastructure. This is because OFP enabled switches can forward packets based on matching rules as well as in traditional manner. Even in multi-vendor network infrastructure, carriers can easily deploy it ignoring vendor dependencies. According to the opennetworking.org, OpenFlow protocol is currently the only standardized SDN protocol that allows direct manipulation of

the forwarding plane of network devices, which makes it a key enabler for software-defined networks [1].

1.2.2 OpenFlow Events and Messages

The OpenFlow protocol implementation is done between two devices by exchanging series of OpenFlow negotiation messages. It supports three message types, namely controller-to-switch, asynchronous, and symmetric; each with multiple sub-types. Controller-to-switch messages are sent by the controller to directly manage the state of the switch. Asynchronous messages are sent by the switch in order to notify the controller about network events and changes made to the switch state. Symmetric messages are sent by either the switch or the controller to one another without prior request. Messages are summarized in Table 1.1 [9].

Type	Message	Description
Symmetric	Hello	Version Numbers are negotiated
	Echo	Sent to detect liveliness of each other
Controller → Switch	Feature Request	Determine which switch ports are available
	Set Configuration	Controller asks switch to inform about Flow Table entry expirations
	Packet Out	It tells switch to send packets out of specific port. It is also used to forward packets received via Packet In messages
Asynchronous	Packet In	Transfer control of the packet to controller
	Port Status	Inform controller about a change on a port

Table 1.1: OpenFlow protocol: Messages Types

1.2.3 Open vSwitch

According to ovs.org, Open vSwitch (OVS) is an open source software switch designed to be used as a "virtual switch" in virtualized server environments. The goal of OVS is to implement a switching platform that enables standard, vendor-independent management interfaces and opens the forwarding functions of switches to programmatic extension and control [10]. It supports all versions of OpenFlow protocol. Using *ovs-ofctl* tool, any desired OpenFlow version can be implemented on a physical switch.

Simply, Open vSwitch is a "software switch" which implements OpenFlow protocol on switching hardwares [11]. It manages the Flow Tables for the Datapaths which are used for forwarding the incoming traffic according to matched entries. The structure of Flow Table entries is explained in the Table 1.2.

Match Fields	Priority	Counters	Instructions	Timeout	Cookie

Table 1.2: Flow Table Entry structure

- *Match Fields*: will be matched against incoming packets. It includes packet header and input port
- *Priority*: matching priority for this Flow Table entry

- *Counters*: number of received packets matching this rule
- *Instructions*: Used to modify the action to be applied on the packet
- *Timeout*: The number of seconds this Flow Table entry lives in the Table
- *Cookie*: Opaque value chosen by controller. Not used while processing a packet. Used by controller

Figure 1.2 shows the example of entries within a Flow Table.

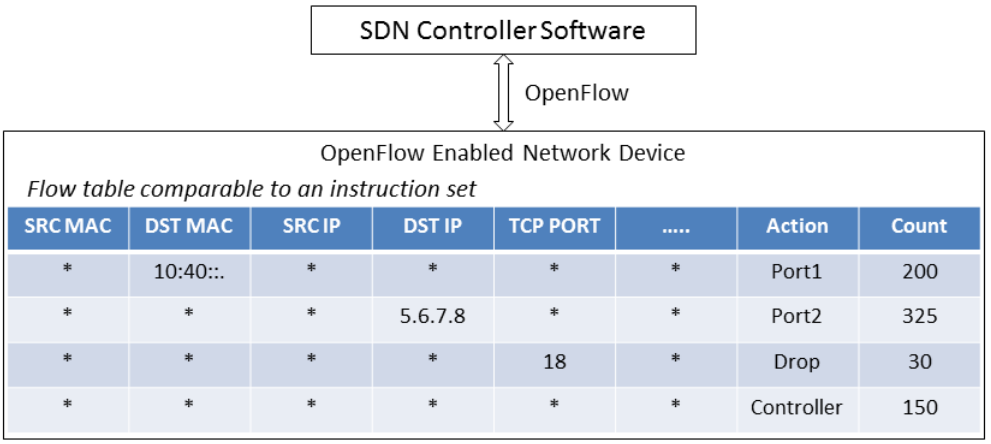


Figure 1.2: Flow Table Entries [9]

1.3 Ryu

1.3.1 Introduction

Ryu [12] is a component-based SDN framework which delivers a suitable platform for SDN applications to run on the top of Ryu controller. It is an open source tool written in Python that provides well-defined APIs & packet libraries and supports all versions of OFP. It is well tested with various OpenFlow switches and suits well on Open vSwitch which is used in this IDP. Ryu has a powerful but complex architecture. However, its active community support, nice documentation and few exemplary applications provide novice developers a room to understand and get adapted to the environment easily. There are numerous controllers that deliver SDN capabilities, but choosing the best of them is confusing for the developers. As a research was carried out with number of controllers, Ryu has been selected as the best controller choice for SDN environment [6].

The following Figure 1.3 depicts the role and features of Ryu SDN framework [13].

1.3.2 Ryu Architecture

RYU has ‘Event-based‘ architecture. As shown in Figure 1.4, it uses ‘Event Dispatcher‘ for generating and handling events from incoming OFP messages [13]. The controller works as an event source. Explanation about the architecture in a stepwise manner can be found below:

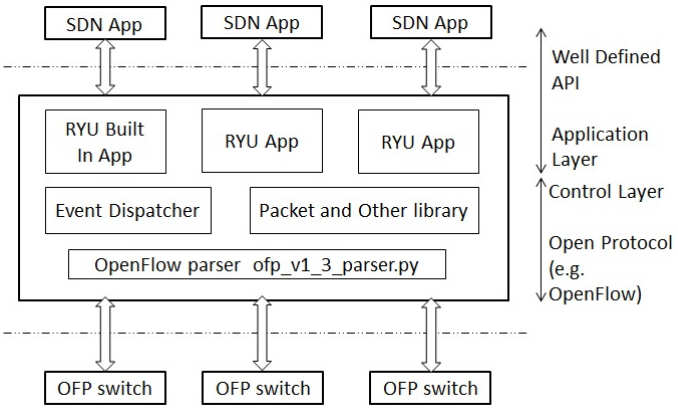


Figure 1.3: Ryu SDN framework

1. Ryu manages two threads for sending and receiving the packets to and from the switch. Upon receiving a packet, controller's Receive Thread stores the packet in a receiving queue. This packet is nothing but only raw bytes and yet to be parsed in order to understand the OFP message that it has carried. To understand the OFP message, controller calls an OFP parser (parser file: ofproto_v1_N_parser.py where N is the protocol version negotiated with the switch upon connection).

2. OFP Parser will generate appropriate OFP event by parsing the packet and labels it as 'OFPxxx'. As for example, a 'Packet In' event becomes 'OFPPacketIn' event [14].

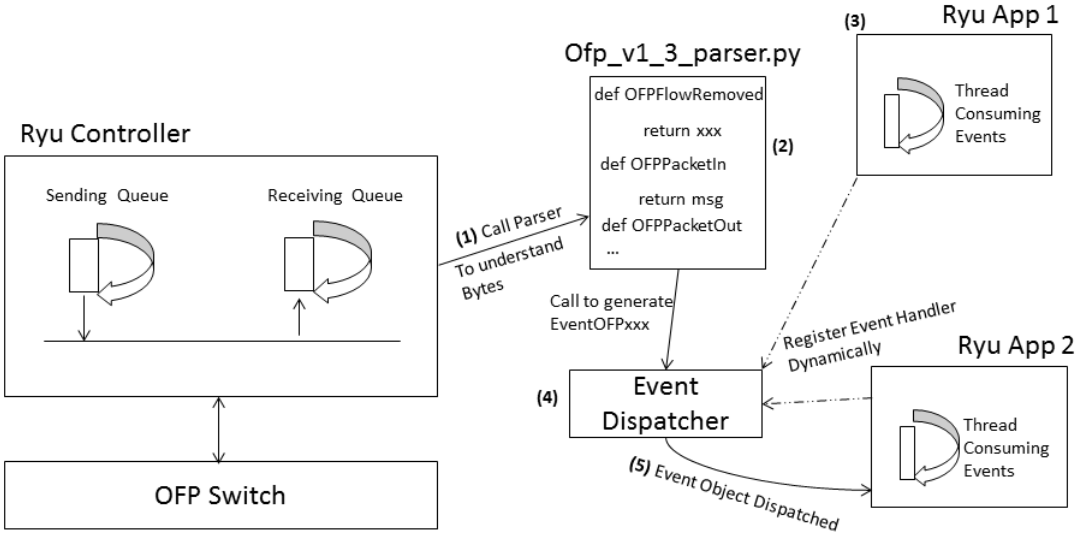


Figure 1.4: Ryu architecture: Structural Diagram

3. On the other hand, RYU applications that are running on the top of RYU framework, maintain queue of events. Applications register event handlers dynamically according to their specification to the Event Dispatcher. It simply means that RYU applications are waiting for necessary events to occur in order to start execution.

4. At this stage, Event Dispatcher plays an important role. It decouples Event source (which is RYU controller) and Event sinks (which is RYU App). Event source will call the methods of Event Dispatcher to construct event objects (from OFPxxx to

EventOFPxxx. [Figure 1.5]). Event dispatcher knows to which RYU app it has to dispatch an event [13].

Figure 1.5: Conversion of an OFPxxx event into EventOFPxxx

5. These events are dispatched to event queues of RYU Apps. As shown in Figure 1.6, each RYU App runs a thread over the event queue to consume the events. Only one instance of RYU app can run at a moment [14].

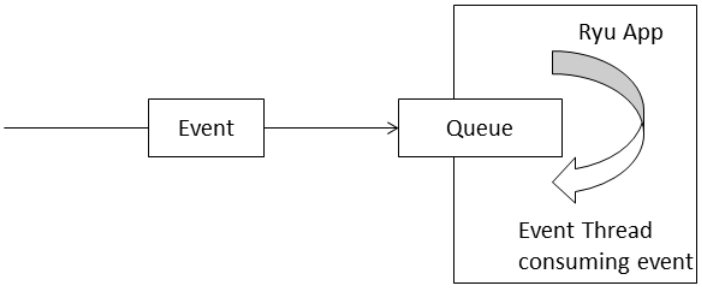


Figure 1.6: Dispatching the event to Event Sink

1.4 Firewall on SDN Controller

Firewalls are the systems which control the incoming and outgoing packets to and from the inner network. They provide security barrier against potential attacks coming from the Internet that can disrupt the services running in the inner network. Firewalls are divided into 2 types: Stateful Firewall and Stateless Firewall. In Stateful Firewall, the connection is tracked by the Firewall and the packets part of the tracked connection, are allowed to pass by. Stateful Firewall uses attributes in order to track the traversing packets. These attributes include the source and destination IP addresses, port numbers and sequence numbers which are also known as state of the connection. In our Ryu application, we will use a Python dictionary to implement connection tracking mechanism of Stateful Firewall.

In contrast, Stateless Firewall checks all incoming packets individually with the ruleset. It does not track connections nor keeps state of the traversed packets, rather simply check each packet with Firewall rules and identifies whether the packet is allowed to pass by or not. It assumes that the information within a packet is trustworthy. Hence, one possible breach could be: a TCP SYN-ACK packet can be passed by the Firewall before the Firewall has seen a TCP SYN packet. With the Stateful Firewall case, such a packet will be discarded by the Firewall itself as there is no state information found for the packet.

The goal of this IDP is to implement both types of Firewall applications with real-world Firewall rules on SDN architecture and analyse the performance of the controller & the applications with incoming traffic.

2. Design and Implementation

In section 2.1, general structural design of a Ryu application is presented by means of a Flow-Chart. It also features security limitation of the applications that are developed using the proposed design. The next section 2.2 covers description about all 4 types of Firewall applications developed for the purpose of the IDP. It also explains design of those applications in a Flow-Chart manner and compares Inefficient and Efficient applications in greater details.

2.1 Ryu Application - General Template Design

Ryu applications can be run via command-line tool called *ryu-manager*. We developed our Ryu applications to run under OFP version 1.3, hence, connecting switch must use same protocol version in order to communicate with the controller. Firewall rules are written down in a text file which will be taken as an input by Ryu application. When the application starts running, it maps these rules into Python dictionary keys, which will be used later for packet inspection. As soon as switch connects with the controller, Ryu application will flush out all existing Flow Table entries from switch and installs a Default-Miss Flow entry. Using Default-Miss Flow entry, switch will forward all incoming host traffic to the controller where packets will be inspected. Figure 2.1 shows the structural diagram of Ryu application.

The applications do not include following security features:

(1) ARP security: Applications are designed to work at Layer 3 or above. Hence, Layer 2 security considerations are not taken into account while designing the applications. This can make ARP spoofing attacks successful.

(2) Link layer security: Applications can learn the MAC address of a host from its association with a switch port. However, applications can not verify the associated MAC address; hence, a spoofed MAC address attack can be successful. More information about the design can be found in E.1.

(3) With Stateful Firewall applications, torn down connection can not automatically remove the 'tracked' state from the application. Hence, A TCP FIN packet does not trigger deletion of the Firewall's state.

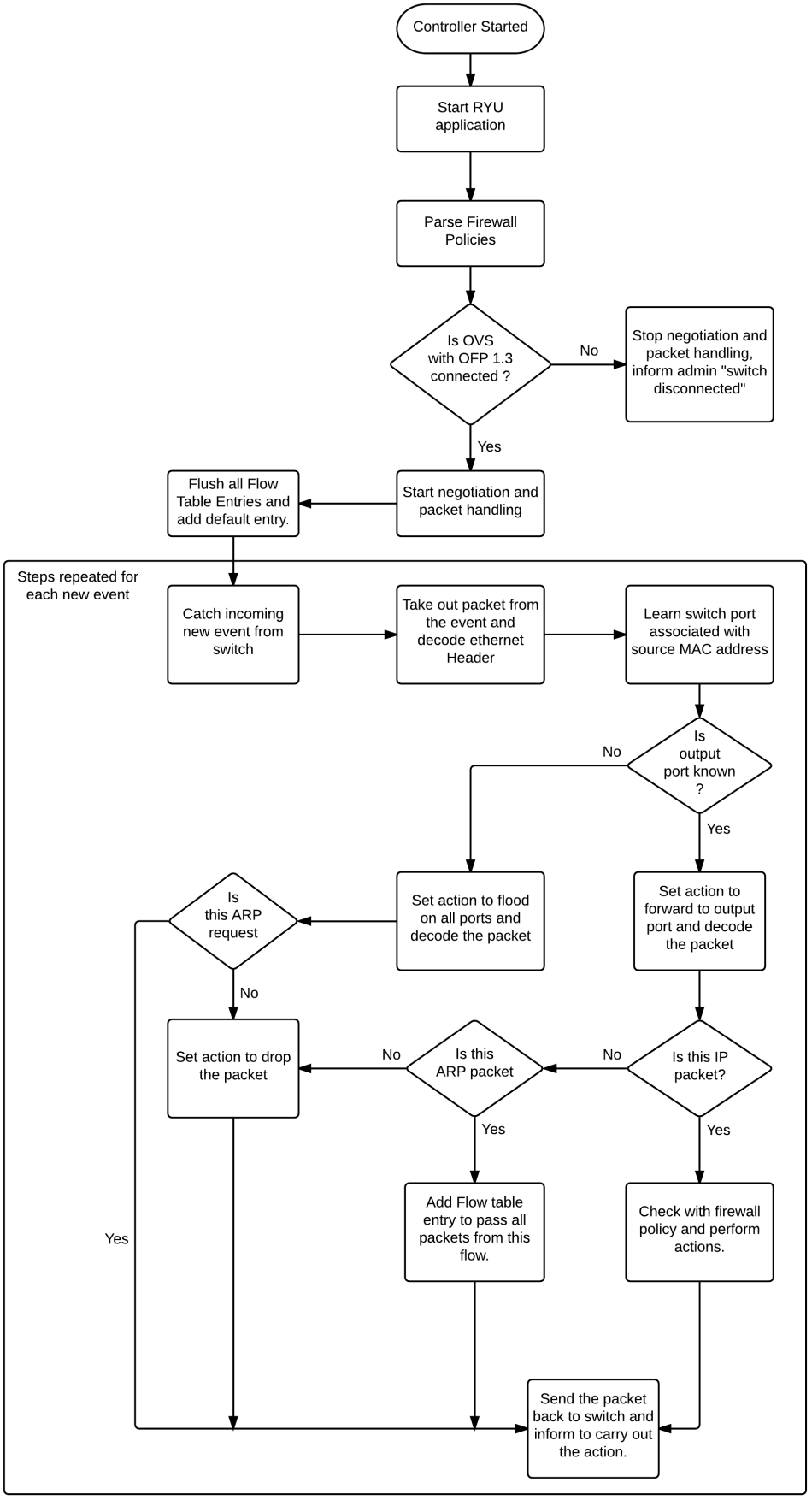


Figure 2.1: Flow-Chart: general template design of a Ryu application

2.2 Firewall Implementation as Ryu Applications

Using a template from previous section, four Ryu applications are developed to measure performance of Stateful and Stateless Firewall in SDN environment. They are listed below.

- Inefficient Stateful Firewall application
- Efficient Stateful Firewall application
- Inefficient Stateless Firewall application
- Efficient Stateless Firewall application

Main difference between these Firewall applications lies within how they inspect incoming packets and how they speed up packet processing by using Flow Tables. Table 2.1 compares Efficient and Inefficient Firewall application in general with various parameters.

	Inefficient Firewall	Efficient Firewall
Terminology	Switch forwards all received packets to controller. No extensive use of OFP Flow tables.	Switch forwards only initial packets of the connection to the controller. Afterwards, it will use Flow table entries to forward/drop packets without asking controller.
Connection Tracking capabilities with Stateful Firewall	Yes	Yes
Which packets of a connection are seen on controller?	ICMP → all PINGs & PONGs TCP → all segments UDP → all datagrams	ICMP → First PING & PONG TCP → SYN & SYN-ACK only UDP → First 2 datagrams
How to handle them?	Simply inspect each packet individually against Firewall policies. If a policy allows the packet then inform switch to forward it, otherwise drop it.	Check connection initiation and response packets against Firewall policies on controller. Add one Flow Table entry per inspection on switch to allow/deny further traffic from the host.
Flow Table rules to be added per flow	None	One
Efficiency	Less	More
Potential Connection Tracking Accuracy	100% accurate	can approximate after hand-shake

Table 2.1: Comparison between Inefficient and Efficient Firewall

2.2.1 Inefficient Stateful Firewall Application

As per terminology, Inefficient Stateful Firewall application makes switch forward all incoming traffic to the controller. It does not store any Flow Table entry on switch that speeds up packet processing. When first packet of an unseen flow is received by the controller, the application checks it against Firewall rules and creates new state information for that flow. This information will be used later when subsequent reply-packet from destination host shows up at the controller. The application tracks down the reply-packet with the stored information and learns about newly established connection. Figure 2.2 explains the mechanism of Inefficient Stateful Firewall application in detail.

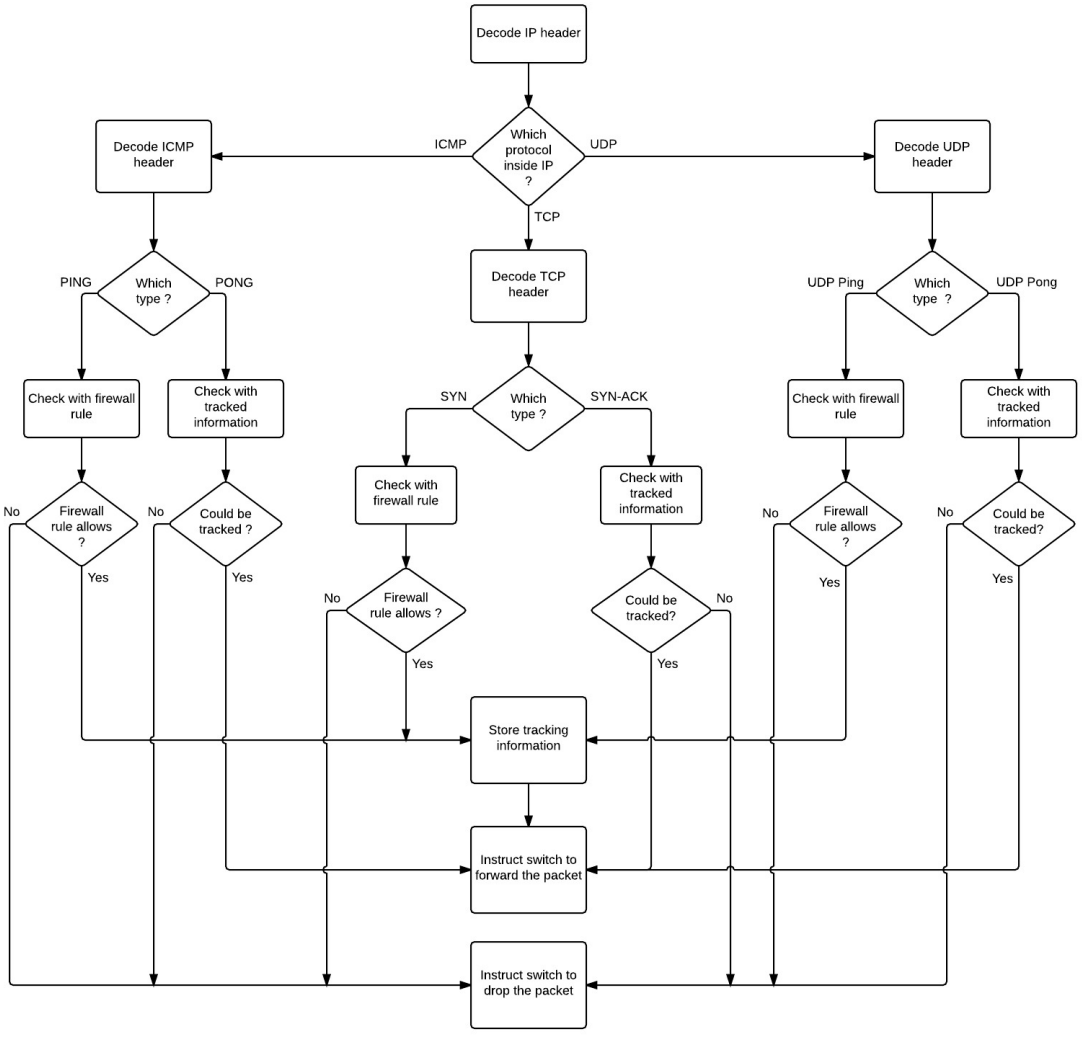


Figure 2.2: Flow-Chart: Infficient Stateful Firewall application

2.2.2 Efficient Stateful Firewall Application

Unlike its Inefficient counterpart, here switch will forward only first few packets of a new flow to the controller. Upon receipt of such packets, Firewall application will decode them, prepare some Flow Table entries according to the Firewall policies and finally add them upon switch's Flow Table. This way, Efficient Stateful Firewall application will take the advantage of OpenFlow's capability to set Flow Table entries for fast packet processing. After that, switch will check incoming packets with the Flow Table entries. Once a matching entry is found, switch will take out necessary action associated with it rather than forwarding the packet to the controller. However, packets which are not matched with any Flow Table entries are simply forwarded to the controller. Flow Table actions can be forwarding or dropping depending upon firewall policies.

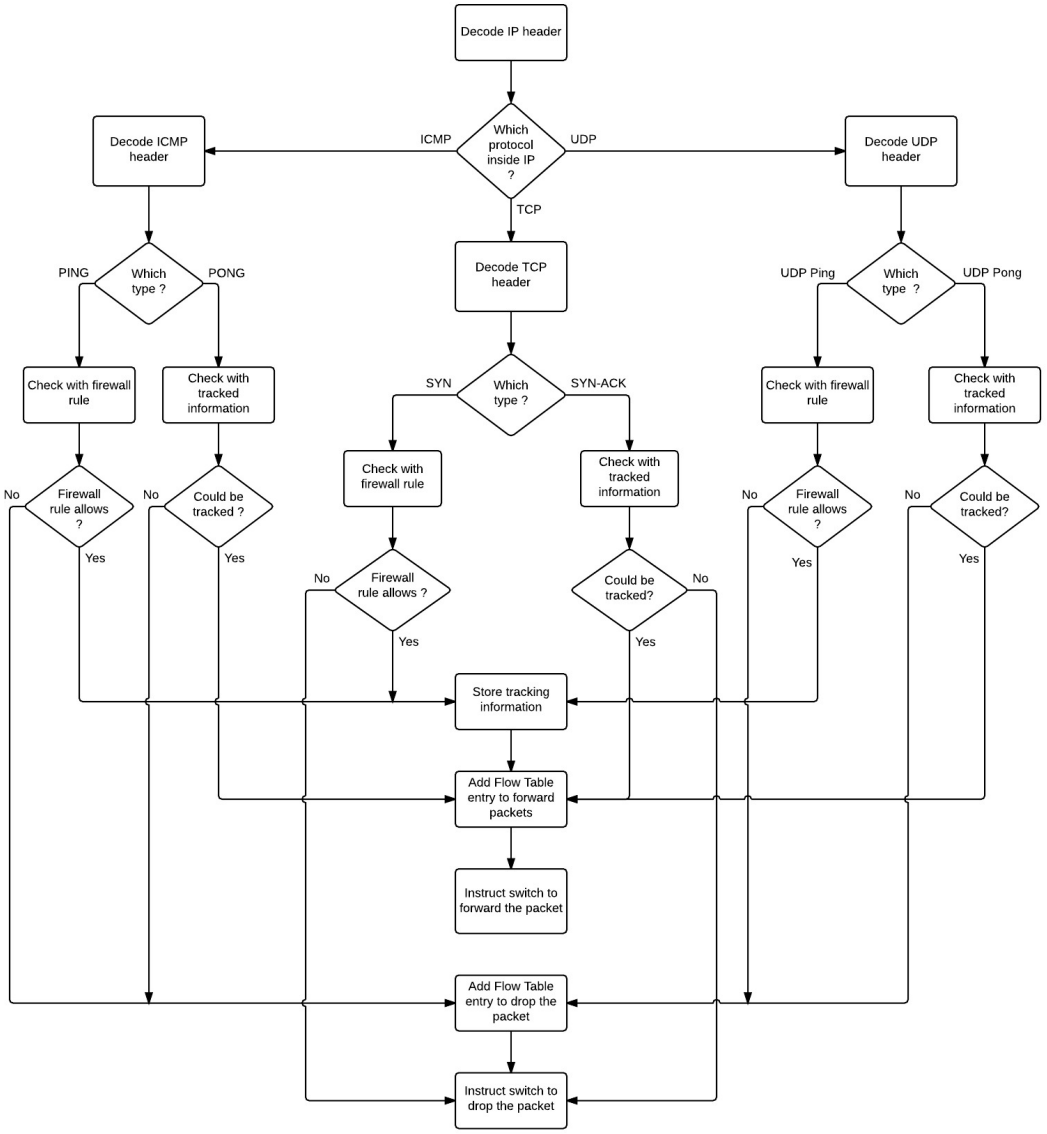
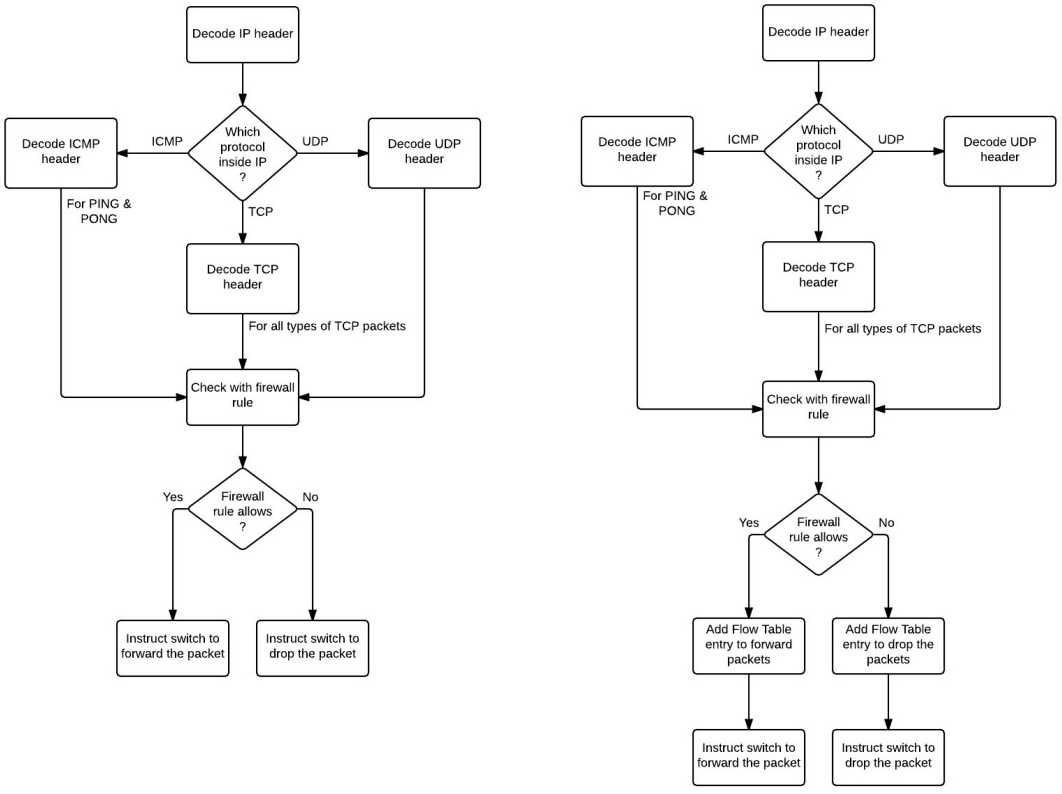


Figure 2.3: Flow-Chart: Stateful Efficient Firewall application

2.2.3 Inefficient Stateless Firewall Application

Compared to 2.2.1 case, Inefficient Stateless Firewall application does not manage any state information for packet flows. It does not use any connection-tracking feature to track down ongoing connections; rather simply checks each packet individually against the Firewall rules and takes out necessary action associated with a rule on the packet. As the name suggests, the application will not store any Flow Table entry on switch either, which makes it inefficient in terms of packet inspection.



(a) Inefficient Stateless Firewall application (b) Efficient Stateless Firewall application

Figure 2.4: Flow-Charts of Stateless Firewall applications

2.2.4 Efficient Stateless Firewall Application

Similar to 2.2.2, Efficient Stateless Firewall application also stores Flow Table entries on switches to reduce bandwidth consumption of controller-switch link and to speed up packet-processing. It also does not keep any state information to track down ongoing connections. It can be seen from details of Flowchart 2.4b that Packet inspection of the application is very similar to the one with 2.4a except the Flow Table rule addition for each inspected packet.

Application code are uploaded to Github.org and can be found via [15].

3. Results and Evaluations

This section explains different tests that are conducted with the designed Firewall applications on local machine and Memphis Testbed. General network tools like Ping and Netcat are used for the tests. Section 3.1 enhances the testing on local system in more details. Next section 3.2 encompasses details about Memphis Testbed, its topology designs and testing results.

3.1 Testing on Local Machine

In order to determine correct behavior, the applications are tested with Mininet [16] on local machine. Mininet is a command-line tool that creates a virtual network with controllers, switches & hosts and runs applications on the top of controllers to simulate Software Defined Networking environment. Virtual image file consisting Ryu controller with Mininet tool can be downloaded from [17]. Figure 3.1 shows network topology used for local testing.

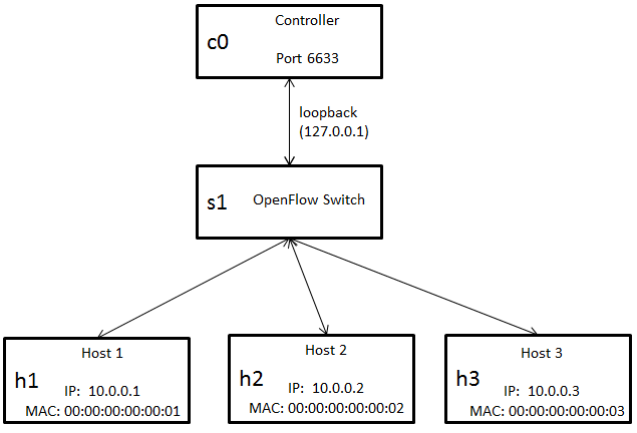


Figure 3.1: Mininet Topology for Local Testing

Following Fig. 3.2 depicts how switch can be connected with a controller in Mininet environment. On the controller, Ryu application code is given as argument to *ryu-manager* command. Correct OpenFlow version can be provided by *ovs-vsctl* command whereas, *ovs-ofctl* command is used to list out Flow Table entries of the switch. More details about the commands can be found in Appendix E.4.

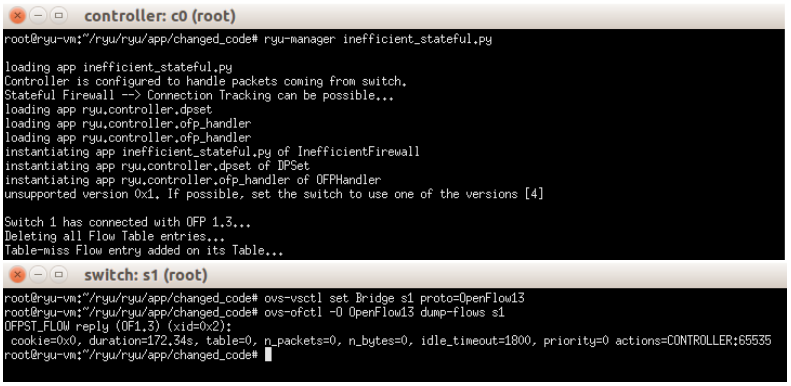


Figure 3.2: Console outputs after starting application

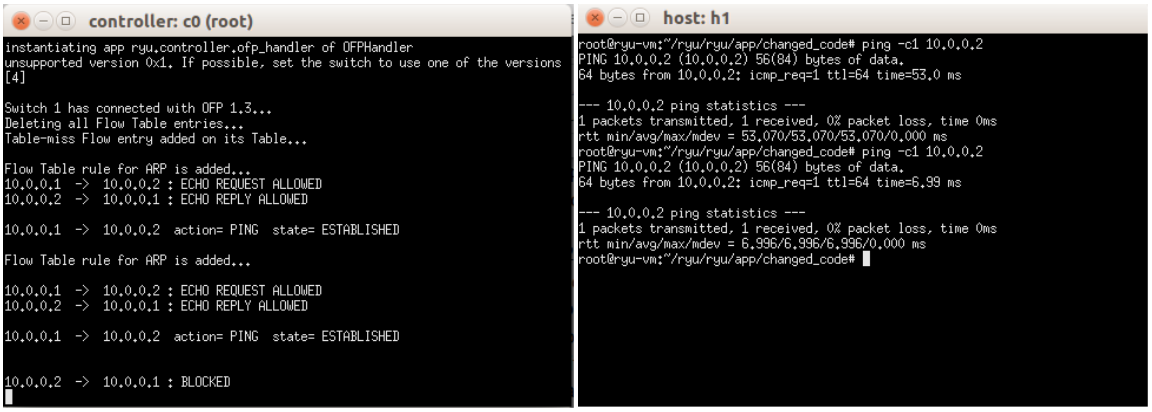
In this IDP, default policy for the Firewall is set to *deny-all*. This means all packets not matching Firewall rules, will be dropped by the Firewall. For local tests, following Firewall policies are considered. Motivation behind such design of policies is to cover all functionalities of Firewall applications; e.g., pinging policy covers ICMP functionality, TCP policy covers Stateful & Stateless functionality and datagram policy covers UDP functionality of the applications. In this way, these policies will be used to test correctness of the four Firewall applications.

1. Host 1 can ping Host 2 & Host 3 but they can not ping it back.
2. Only Host 1 can initiate TCP connection to port 8080 on Host 2 & Host 3 using 1000 as source port.
3. Host 2 & Host 3 can establish TCP connection with each other only on port 1000.
4. Host 1 can send datagrams to port 8080 of Host 2 & Host 3 through port 1000.

Following subsections represent local tests that are conducted with each of the applications.

3.1.1 Inefficient Stateful Firewall

Ping Test: Ping Test is carried out by sending ICMP Echo Request packets from Host 1 to Host 2. Upon receipt of such packets, switch forwards them directly to the controller. It can be seen from controller's screen that application checks the request with Firewall Policy to take out necessary actions. Since Policy 1 permits this, application creates new states corresponding to the packet and forwards it to Host 2 via switch. When Host 2 sends the response message, it should also be allowed by the controller. Here, the application tracks down the response via managed states and allows it to pass through Firewall. Figure 3.3 shows the results of successful ping test with Host 1. The application also behaved correctly when Host 2 tried to ping Host 1. (which is not allowed by the Firewall.)



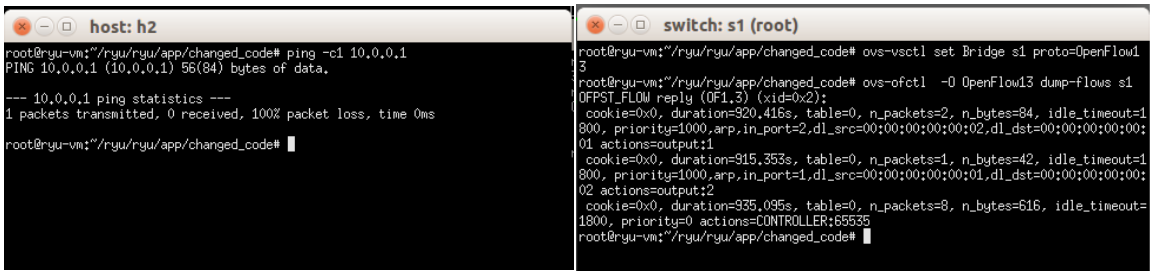


Figure 3.3: Console outputs for Ping Test

TCP Test: When Host 1 first sends TCP SYN packet, Firewall application checks this new connection initiation request with the Firewall Policy. Since Policy 2 allows the port combination, it would send out the packet to Host 2 and create a new TCP state to track down other packets of the flow coming from this direction. Host 2, upon getting SYN packet, prepares SYN ACK packet and sends to Host 1 which will be forwarded to controller by the switch. Application will track down this SYN ACK response as it is part of the previously created connection state and also creates a new state to track all packets of the flow coming from this direction to get passed through. This way, packets coming from both directions will be tracked down further and hence, the TCP handshake will get completed successfully. Other tests were also carried out in order to see correctness of the application with different port combinations and with different connection initiators (both of which should be denied). The result of such tests are shown in Figure 3.4.

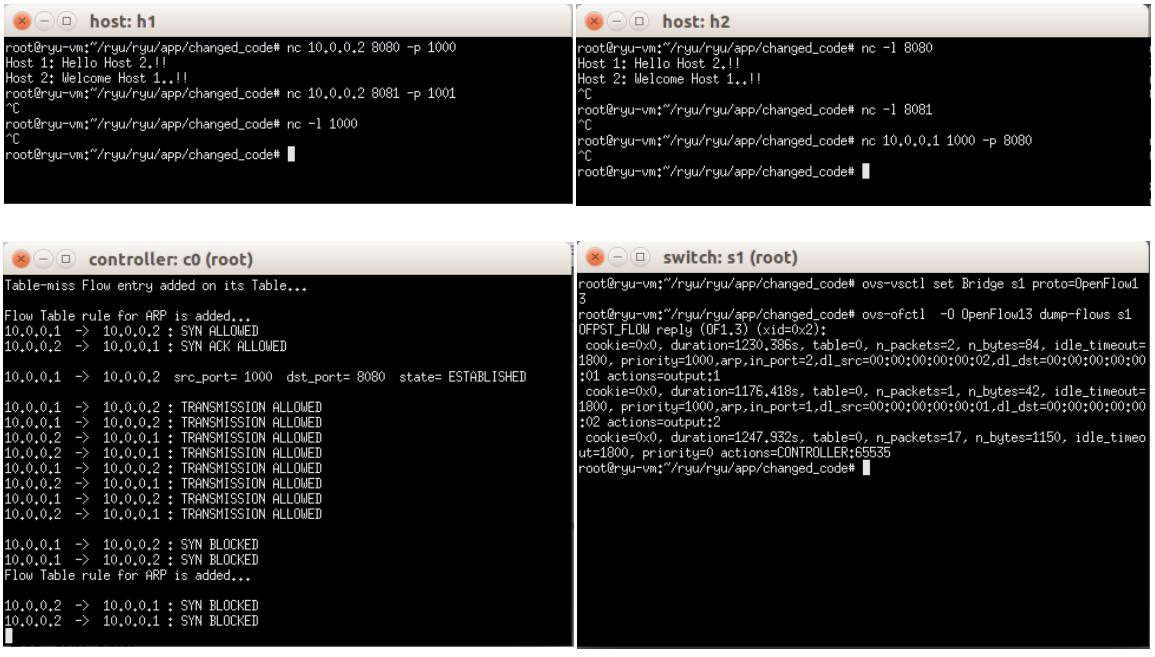


Figure 3.4: Console outputs for TCP Test

UDP Test: Packet inspection in UDP Test is exactly same as TCP Test. Application will create new state information when datagram with appropriate port combination is received from Host 1 and forwards it to Host 2 via switch. Subsequent datagram from Host 2 will be tracked down using state information and sent to Host 1. Main difference of UDP handling compared to TCP handling is: UDP is a connection-less protocol. Unlike TCP, tracking down one entire session is not possible because it lacks handshake mechanism. Hence, application will keep track of one Round Trip of UDP datagram exchange. So,

when Host 1 sent datagram, application will keep state as 'UNREPLIED' and change it to 'ASSURED' once Host 2 replies back. Results of UDP Test are shown in Figure 3.5.

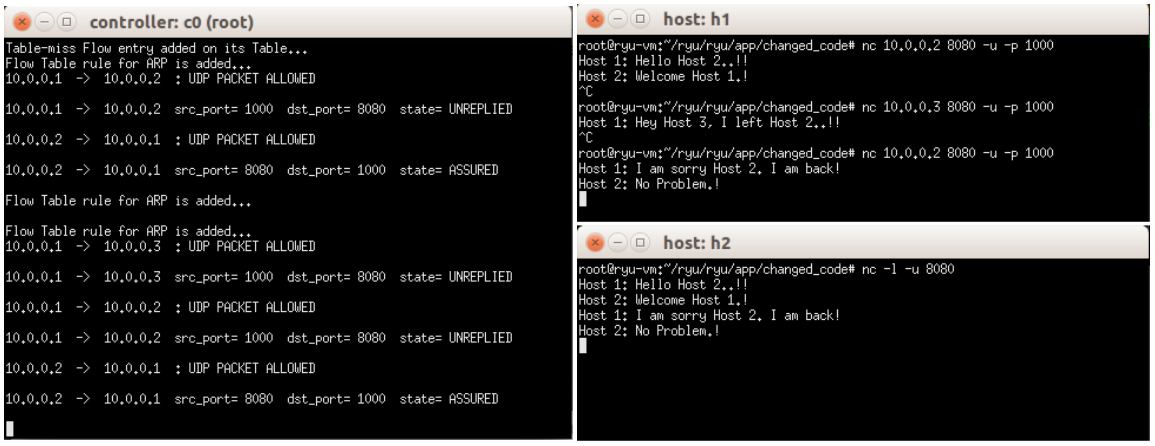


Figure 3.5: Console outputs for UDP Test

3.1.2 Inefficient Stateless Firewall

Ping Test: Unlike previous Test, application simply checks all incoming ICMP packets against Firewall rules rather than tracking down subsequent replies. Hence, Ping packet from Host 1 and Pong packet from Host 3 are treated equally for packet inspection. Since connection tracking mechanism is not adapted, a Pong packet from Host 3 without preceding Ping packet is also allowed by Firewall. However, upon receipt of such unsolicited Pong packet, Host 1 will simply discard it. Figure 3.6 shows application behavior for Policy 1.

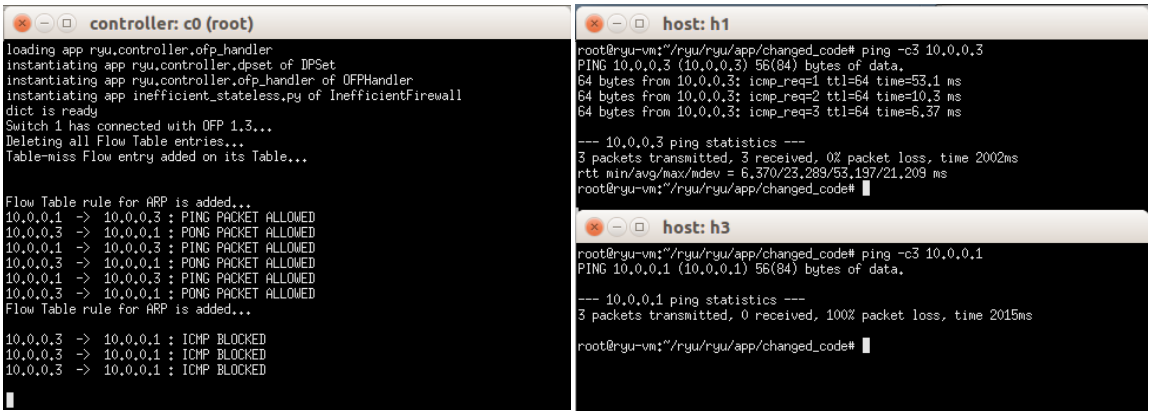


Figure 3.6: Console outputs for Ping Test

TCP Test: The Test was carried out by initiating connection request from Host 1 to Host 3. As Policy 2 allows such connection, application will allow all packets to pass through Firewall. Here, application does not differentiate between any of the packets while doing packet inspection. Each incoming packets are tested only against Firewall rules. So, if only response packets are allowed from a Host, then Firewall designers should keep that in mind while creating rules rather than leaving it upto the application to handle them.

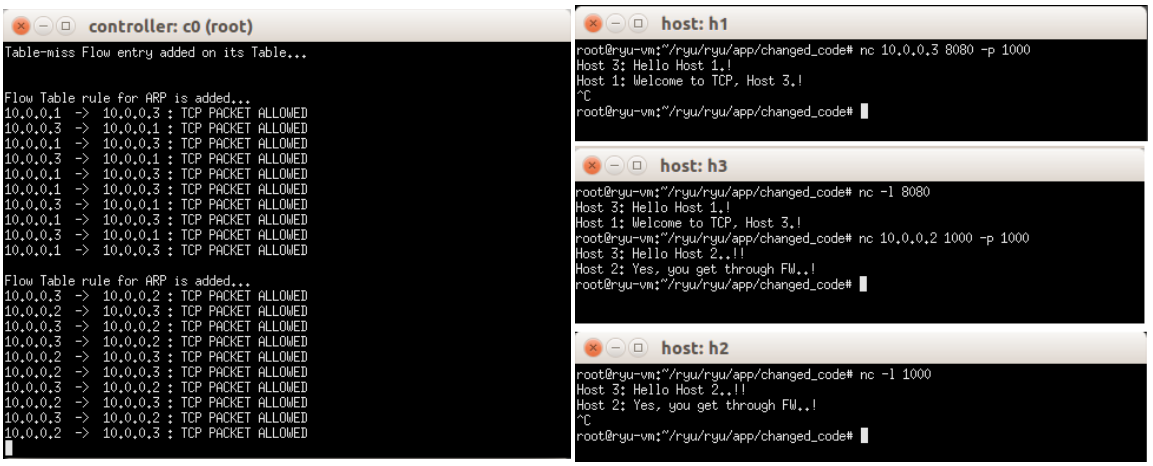


Figure 3.7: Console outputs for TCP Test

UDP Test: It can be seen from Fig. 3.8 that UDP Test also succeeds with the application when Host 2 and Host 3 communicated with each other on port 1000. As opposed to 3.1.1 UDP Test, here application does not keep track of 'UNREPLIED' or 'ASSURED' datagrams. It simply forwards datagrams if Firewall Policy allows, otherwise rejects.

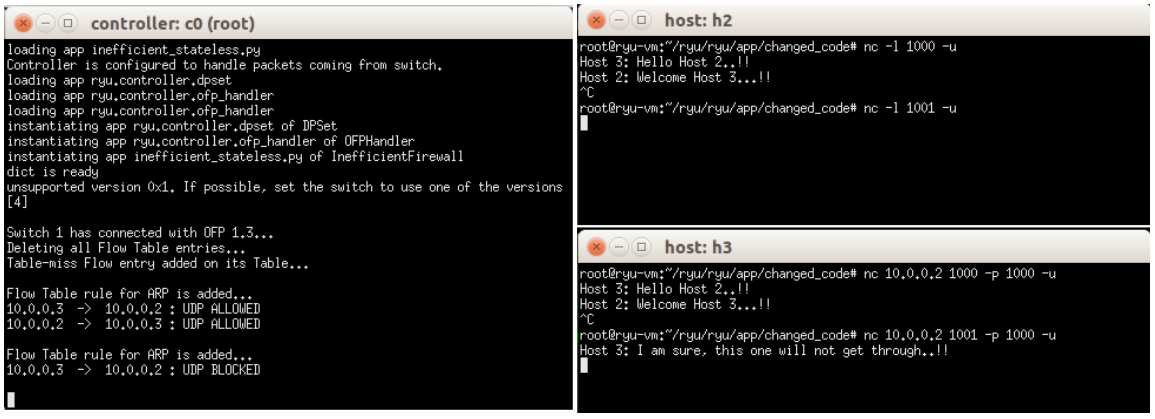


Figure 3.8: Console outputs for UDP Test

3.1.3 Efficient Stateful Firewall

Ping Test: Figure 3.9 shows behavior of Efficient Stateful Firewall with Ping Test. Upon receipt of first Ping packet at switch, it is forwarded to the controller. Since Policy 1 allows it, application creates new state information to track down Pong packet and also adds Flow Table entry on switch to allow all Ping traffic from Host 1 to Host 2 in future. When subsequent reply from Host 2 is received by application, it finally learns about established Ping connection and adds another Flow Table entry to allow Pong traffic from Host 2 to Host 1 in future. As a result, next 4 Ping request/reply packets are handled by switch itself which take less RTT time than first packet exchange.

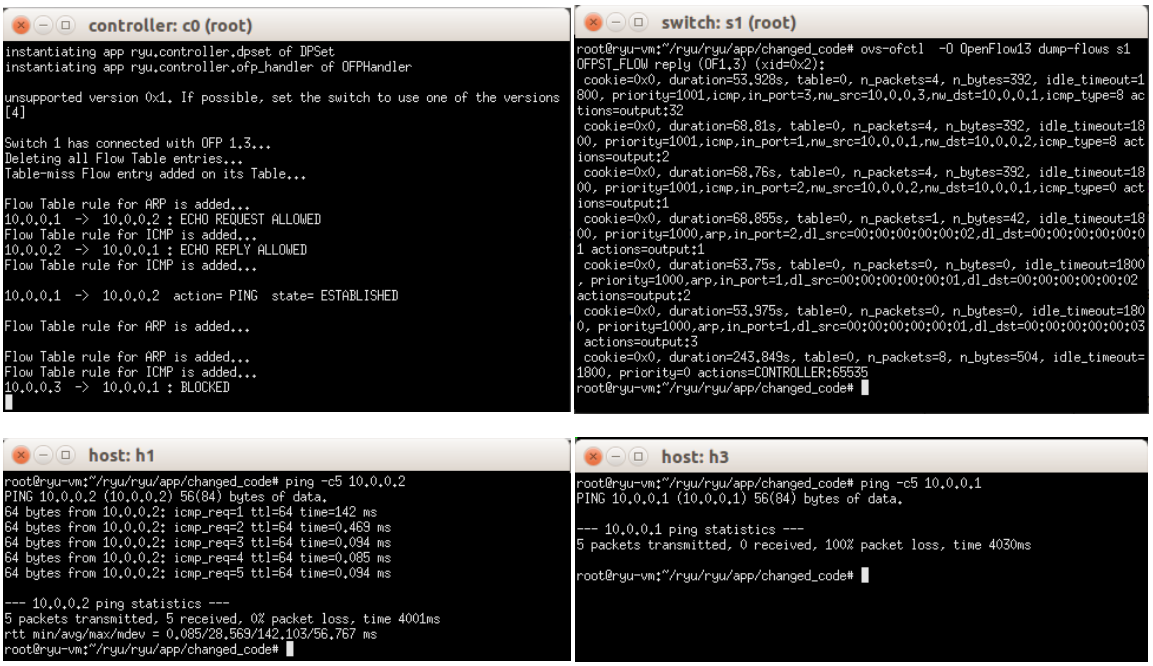


Figure 3.9: Console outputs for Ping Test

TCP Test: TCP Test is carried out with Host 3 initiating connection to Host 2. The connection request is allowed by application and it creates state information for SYN ACK packet to track down the Handshake. Here, application can only track down TCP Handshake rather than entire session of the connection. This is because, upon receipt of SYN and SYN-ACK packet, application adds two Flow Table entries to allow future traffic from both hosts. Hence, data transmission and connection termination packets will not be seen by the controller. However, since TCP flag-based matching is not supported with OFP 1.3, Flow Table entries for TCP packets will only have allowing action rather than rejecting. E.2

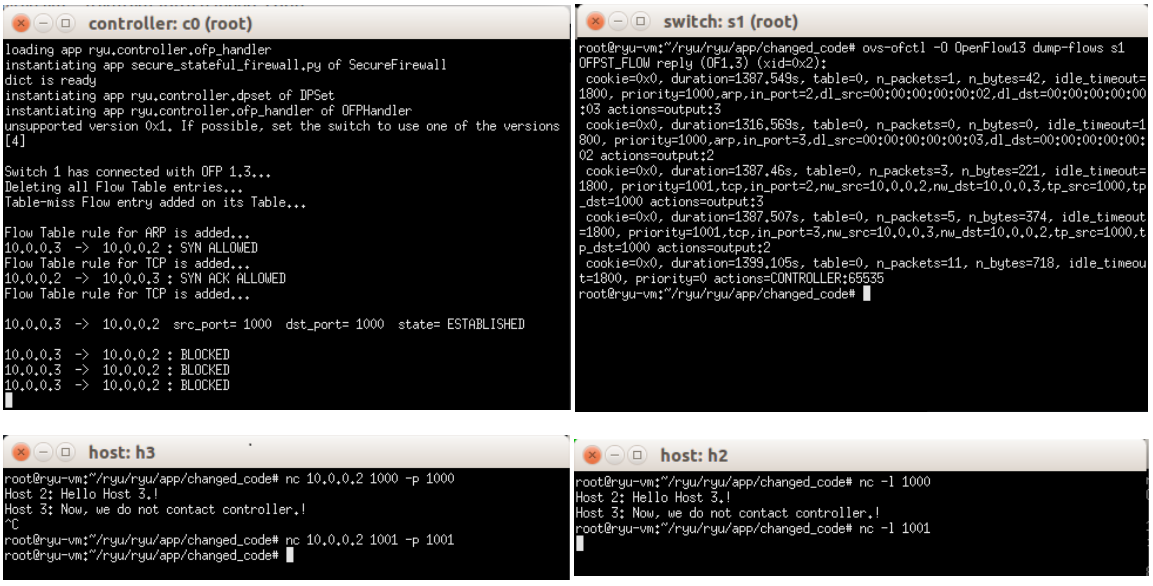


Figure 3.10: Console outputs for TCP Test

UDP Test: As shown in Figure 3.11, application carried out connection tracking with UDP datagrams and also added two Flow Table entries (1 per incoming packet at con-

troller) to transfer flow-handling to the switch. However, since UDP is connectionless protocol, it is possible to add Flow Table entry to reject a UDP flow if Firewall Policy denies. This will stop unnecessary consumption of controller-switch link and fastens packet processing.

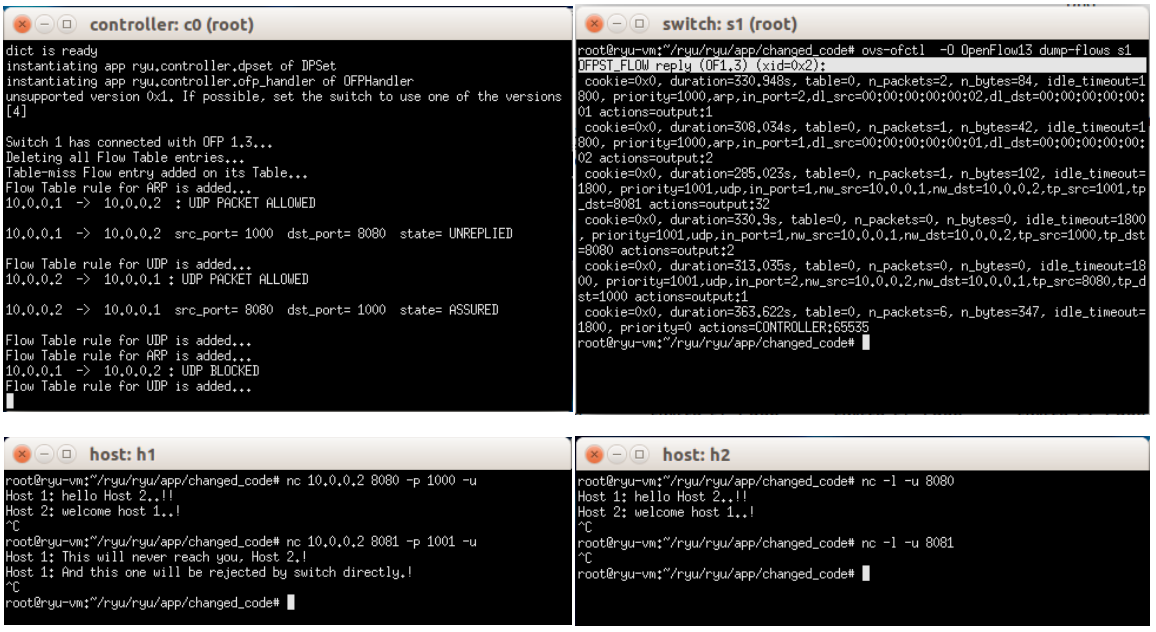


Figure 3.11: Console outputs for UDP Test

3.1.4 Efficient Stateless Firewall

Ping Test: Compared to 3.1.2, only first pair of Ping request-reply packets is seen at controller. As it is intended, application does not create any state information to differentiate between Ping and Pong inspections. It allows the packets to pass through Firewall and also adds Flow Table entries on switch to make switch handle this Ping-Pong flow on its own.

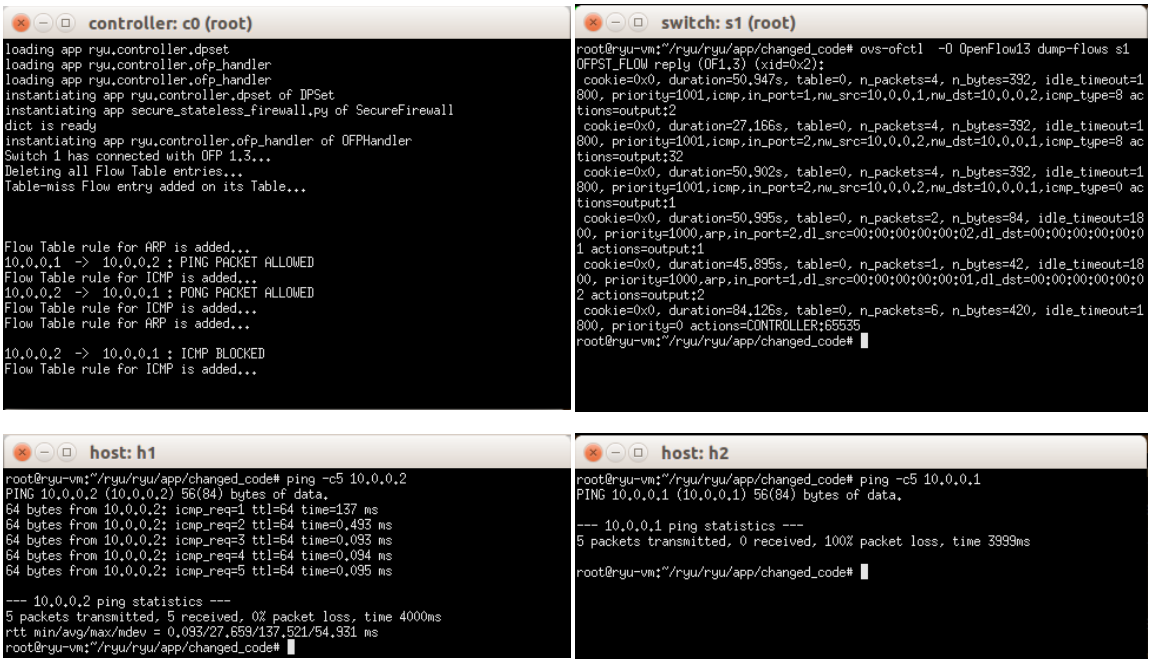


Figure 3.12: Console outputs for Ping Test

TCP Test: TCP test is carried out with Host 1 and Host 3. First packet from each host is checked against Firewall rules by the application. This inspection adds two-way Flow Table entries upon switch, hence, rest of the packet exchange will not be seen by the controller. However, as explained in TCP Test of 3.1.3, decision of packet rejection was performed only at the controller. Figure 3.13 shows console output of controller, switch, Host 1 and Host 3 with TCP test.

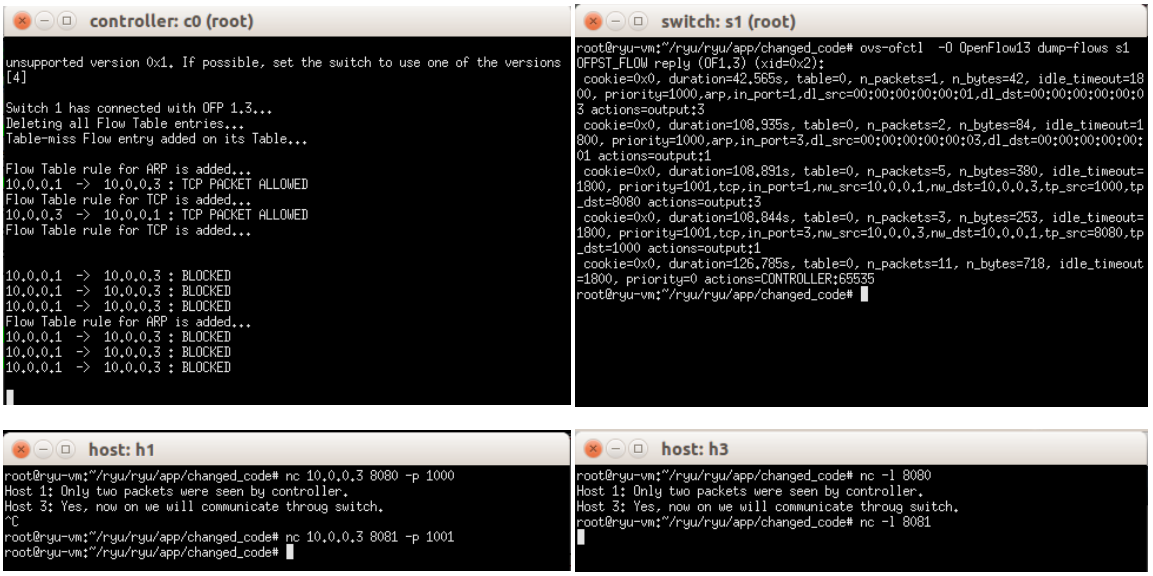


Figure 3.13: Console outputs for TCP Test

UDP Test: UDP datagram handling was inherited from 3.1.2 and 3.1.3 applications. E.g., application does not keep state information for incoming datagrams, as well as, it also adds Flow Table entries upon switch to fasten packet processing. If the inspection allows/rejects the packet, Flow Table entries with respective actions will be added on switch. Similar to TCP Test, only first two packets from communicating parties are seen by controller. Rest of them will be handled by the switch.

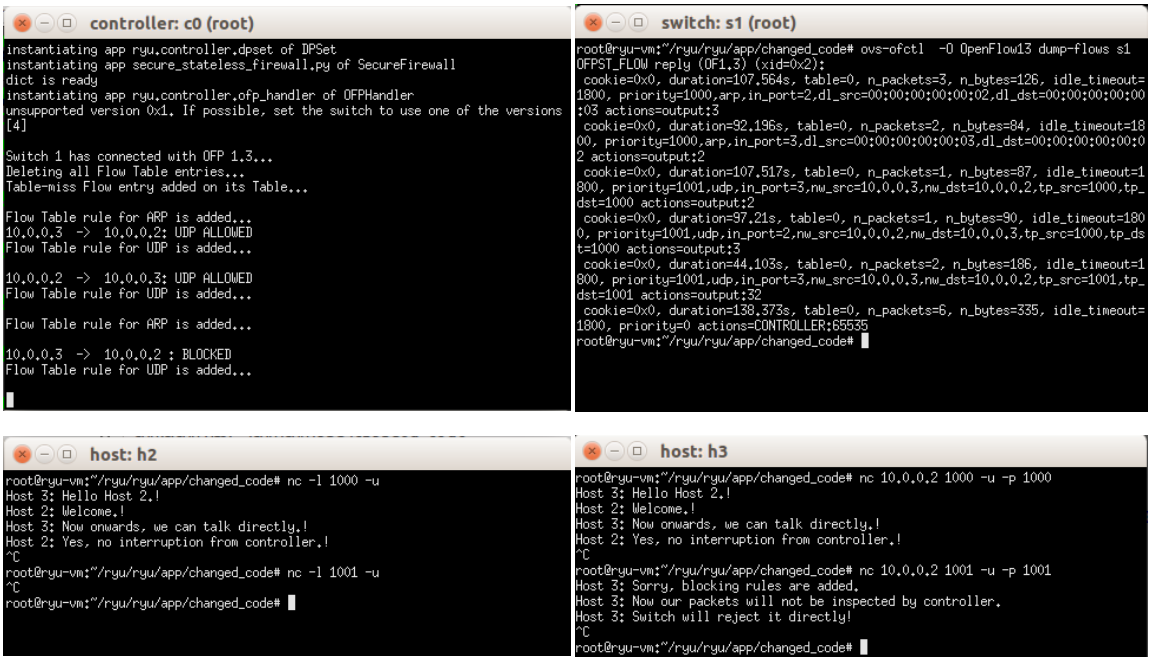


Figure 3.14: Console outputs for UDP Test

These test-results prove correct behavior of the designed Firewall applications. Applications allow only those packets which are intended to pass through Firewall and reject everything else. Ping test, TCP test and UDP test succeed with each application which determines credibility of applications' conduct.

3.2 Deployment on Memphis Testbed

These applications were deployed on Memphis Testbed to test their robustness against real world traffic. Memphis Testbed (part of Memphis Project) is developed at the Chair for Network Architectures and Services, TUM [18]. The goal of this testing environment is to use commodity hardwares for high-speed data transmission and to evaluate the performance for improving packet processing on current computer systems. Real world traffic was provided by means of pcap (packet capture) files consisting of random sized packets captured via Internet. For this IDP, four such pcap files with variable number of packets (e.g. 50000, 100000, 500000, 1000000 packets) were used for testing.

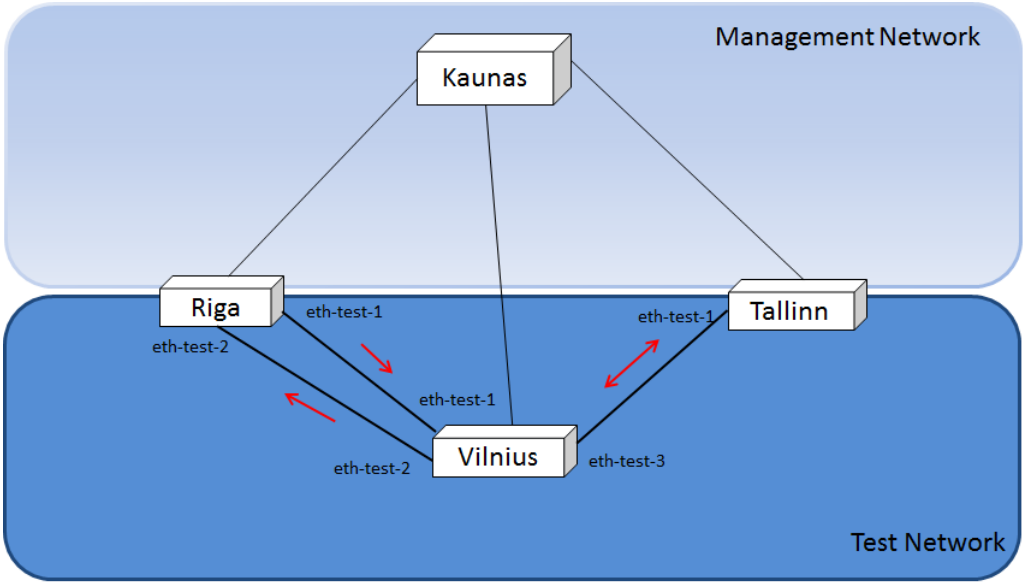


Figure 3.15: Memphis Testbed

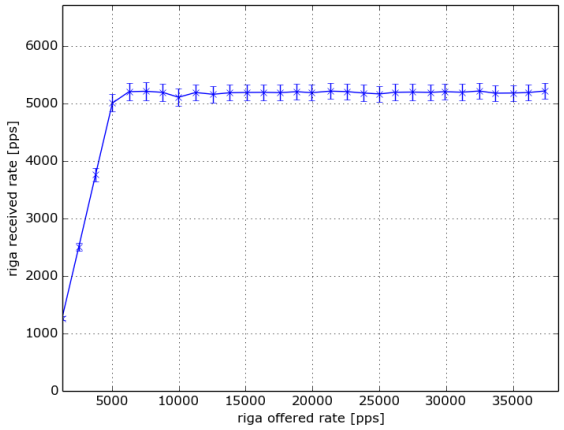
The topology of Memphis Testbed used for the tests is shown in Figure 3.15. Three hosts, namely Riga, Vilnius and Tallinn, are used for the test setup where their respective roles are predefined. All of the hosts are connected with management host, Kaunas, which will define their respective roles prior to start of the test and in the end, collects data from each host for analysis purpose. Riga acts as an end-host which sends out packets from the pcap files using *eth-test-1* interface. These packets are, however, forwarded to Tallinn (controller) via Vilnius (Open vSwitch). Tallinn runs one of the Firewall applications on top of it where packet inspection will be carried out. Based upon Firewall Policies, Tallinn will guide Vilnius either to forward these packets to Riga or to drop them off. Riga will receive the forwarded packets from Vilnius on its interface *eth-test-2*.

The following subsections present results for running Firewall applications with different bandwidth and different ruleset.

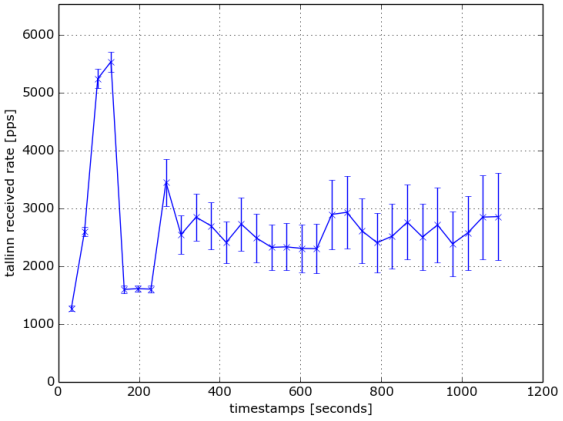
3.2.1 Tests with single TCP packet

To measure behavior of the system, test is conducted with single, 69 Byte TCP packet. The packet is sent out repeatedly from Riga at variable packet rates. PF_RING [19], a high

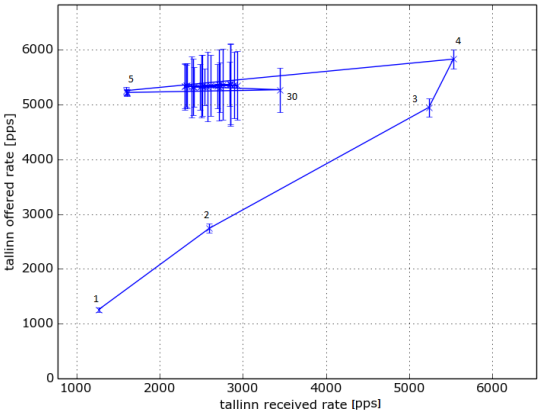
speed Linux network library, is used on each host to count number of captured packets for analysis purposes. Here, the test is performed at packet rates ranging from 1 Mbps to 30 Mbps with increment of 1 Mbps. Pfsend, a PF_RING application, is used on Riga to send packets at the specified rates. Each testcase is set to run for duration of 30 seconds. Prior to start of the test, Vilnius is configured to act as Open vSwitch. On Tallinn, Inefficient Stateful Firewall application is deployed with a single Firewall rule that allows this TCP packet to pass through Firewall. Graphical analysis of the test is presented in Figure 3.16. Figure 3.16a shows that relation between offered packets and received packets



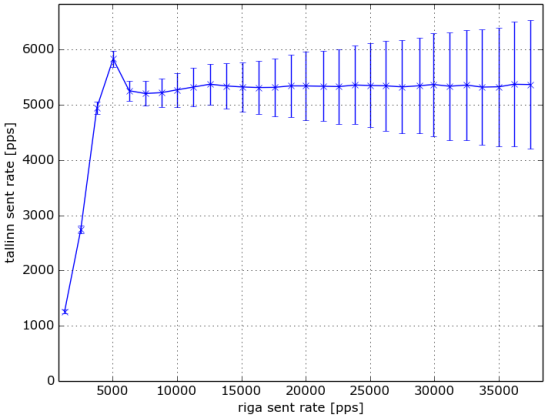
(a) Packet capture on Riga



(b) Packet capture on Tallinn



(c) Packet capture: Tallinn vs Vilnius



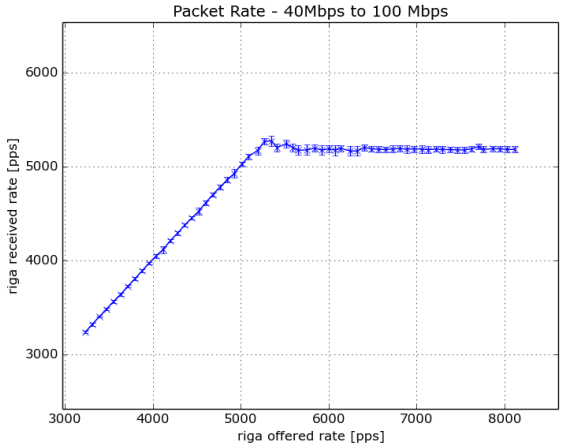
(d) Packet capture: Tallinn vs Riga

Figure 3.16: Test: Single TCP packet [69 Bytes] with Inefficient Stateful Firewall

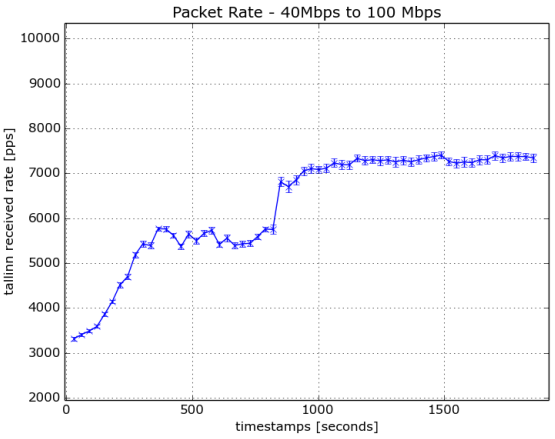
on Riga grows almost linear until we reach equilibrium state at roughly 5000 PPS. After this point, the relation becomes steady with increasing offered rate. Similarly it can be seen from Figure 3.16b that after some timestamp value ($\sim$ 100 seconds since start of the test), switch reduces down the number of packet exchange with controller to 2500 PPS. The actual packet exchange between controller and switch is illustrated in Figure 3.16c. Here, a point represents a testcase. The packet exchange varies over testcases. Initially, it grows linearly and later it moves back and forth. After reaching point 4, controller's offering rate remains constant ($\sim$ 5000 PPS) with varying receiving rate. This behavior reflects in Figure 3.16d where controller handles only 5000 PPS even though Riga's sending rate increases significantly.

Another test is carried out by changing the packet size to 1400 Byte. The sending packet rate is now ranged between 40 Mbps to 100 Mbps. It can be seen from Figure 3.17a that

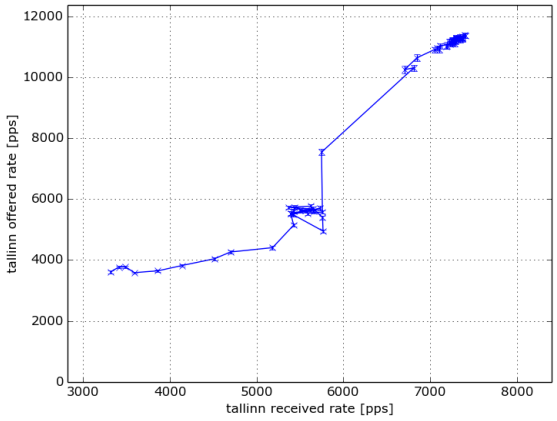
the relation between offered and received packets on Riga grows exactly the same way as with the previous test. However, Figure 3.17b shows different mechanism of the switch that is divided into 3 regions. In the first region (till ∼ 400 seconds since start of the test), switch's offered rate increases linearly with the time. After that, it remains in equilibrium state (till ∼ 800 seconds) by offering roughly 5500 PPS. At 800th second (third region), there is a sudden increase in the offered rate (7000 PPS) and it remains constant for the rest of the testcases. On the other hand, Figure 3.17c depicts packet exchange between controller and switch. Initially, controller's offering rate grows linearly (like first region of Fig. 3.17b), then remains constant at 5500 PPS of receiving rate (like second region of Fig. 3.17b) and after a quick increase, it remains constant again (like third region of Fig. 3.17b). This behavior reflects in Figure 3.17d in which Tallinn has offered more than 11000 PPS compared to Riga's offering rate of 6000 PPS. Such results make us debug more about controller-switch connection.



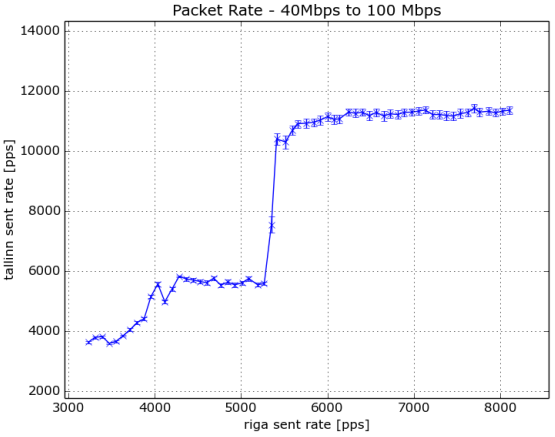
(a) Packet capture on Riga



(b) Packet capture on Tallinn



(c) Packet capture: Tallinn vs Vilnius

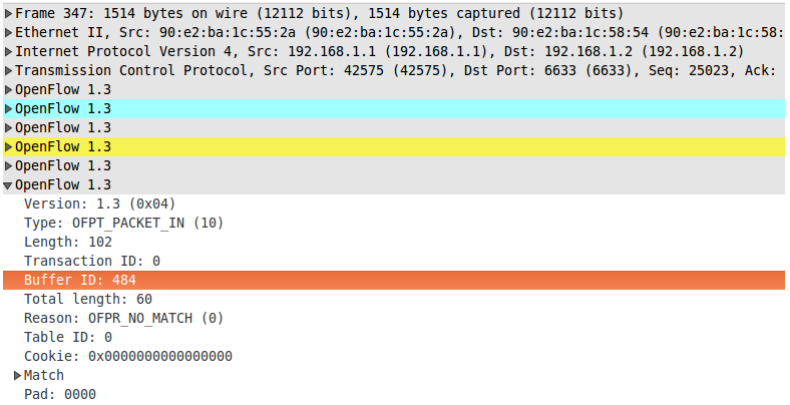


(d) Packet capture: Tallinn vs Riga

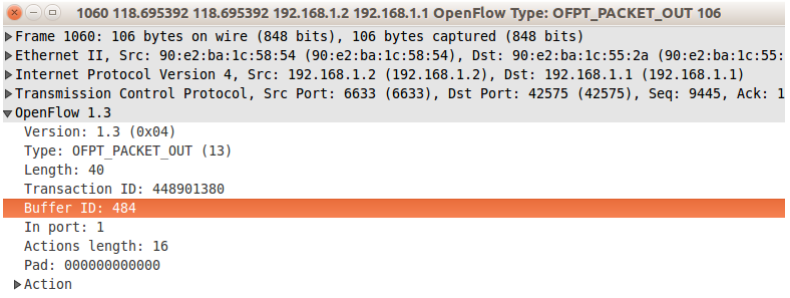
Figure 3.17: Test: Single TCP packet [1440 Bytes] with Inefficient Stateful Firewall

To debug the connection, packets are captured on controller-switch link using tcpdump and later analyzed using Wireshark. This analysis explains interesting TCP mechanism. As mentioned in OpenFlow 1.3 specification book [9], OpenFlow channel between switch and controller runs over TCP. Upon connection establishment, TCP parameters like Congestion Window, MSS etc. are negotiated to control congestion. Using such parameters, TCP acts as a rescuer when there is a sense of congestion between two communicating nodes. This is what we are seeing in our test-results. Initially, when there is no congestion, switch could

forward all the packets to the controller without dropping them (for eg. first region in Fig 3.17b). However, after sometime, switch compresses lot of host packets within few giant-sized OFP Packets and sends them to the controller. This can be illustrated from Figure 3.18a. On the other side, controller will decompress such giant OFP Packets, takes out all compressed host packets and checks each of them individually with the Firewall policies. Instead of compressing all these host packets to bulky OFP packets, controller returns each packet as a separate OFP Packet Out message associated with OFP action to the switch.[see Figure 3.18b] Hence the output of 3.16c and 3.17c, where controller replied more number of packets than it actually received, is well justified. We also see in the results that TCP governs the OpenFlow channel by dropping some packets at the switch when incoming packet rate exceeds certain limits [20]. It is advised in [9] that switch should define the forwarding policies based upon QoS or rate limiting for the packets destined for the controller to prevent denial-of-service attack to the controller connection. Since TCP congestion control is out of the scope of the IDP, we can say with such results and explanation that OFP's performance is highly influenced by the underlying TCP congestion control.



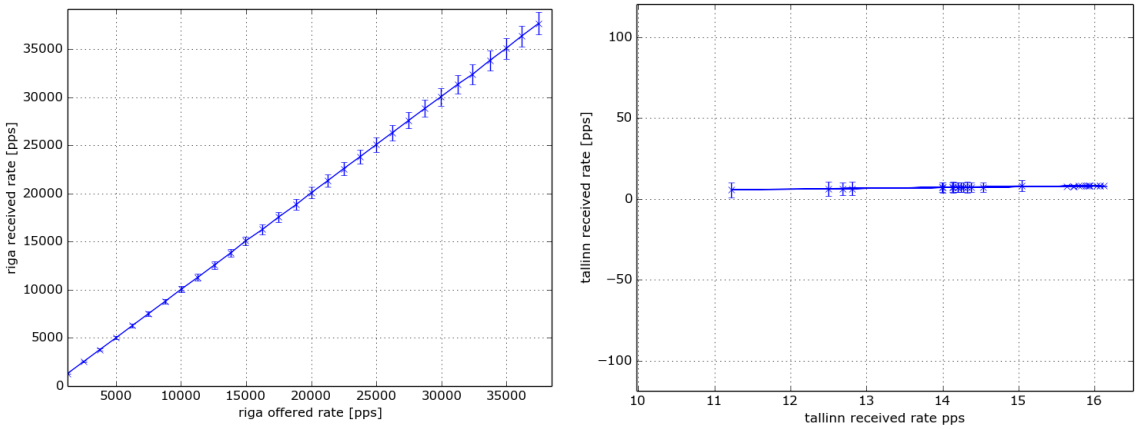
(a) Giant Sized Packet In message



(b) Single Packet Out message

Figure 3.18: Wireshark Output

The same test is also carried out with Efficient Stateful Firewall to determine the behavior of the system against fast packet-processing. The result of the test are presented in Figure 3.19. Obviously, the number of packet exchange with Tallinn should be reduced down drastically. Upon seeing this flow for the first time, Tallinn stores Flow Table entry on Vilnius for handling future traffic-flow. Hence, receiving rate in Figure 3.19a remains in linear proportion throughout with the sending rate. With each incoming packet from Riga, Vilnius is still able to handle it on its own by looking into Flow Table and finding suitable match. This reduces down traffic on controller-switch link which reflects in Figure 3.19b. Now the packets exchanged on this link are basically 'Keep Alive' messages and unwanted LLDP packets generated by the system.



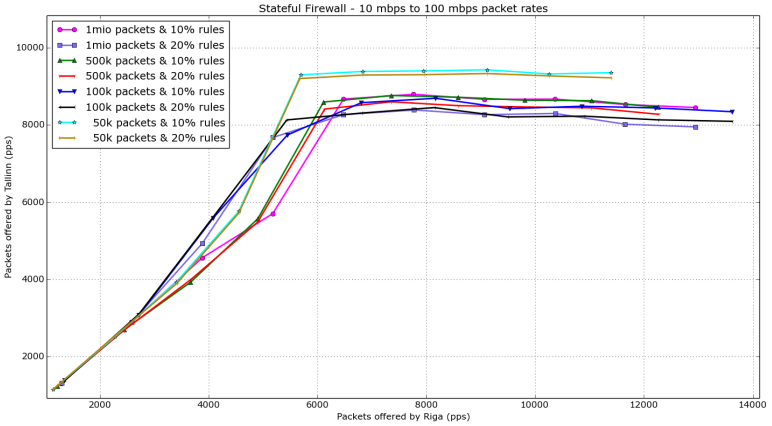
(a) Packet capture on Riga

(b) Packet capture on Tallinn

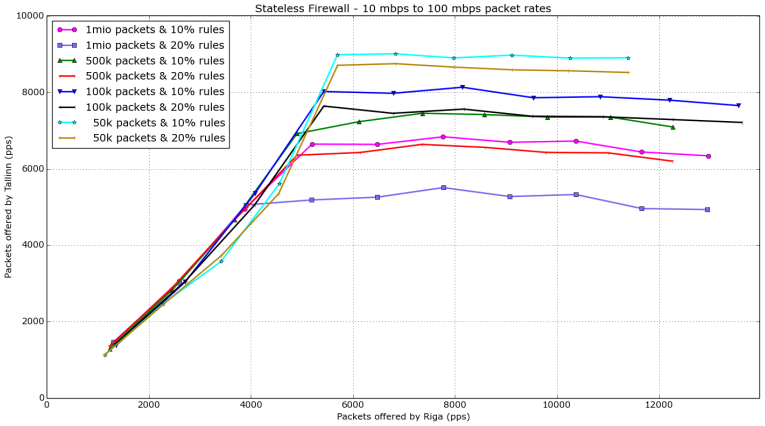
Figure 3.19: Test: Single TCP packet with Efficient Stateful Firewall

3.2.2 Tests with Real world traffic

After testing with single TCP packet, tests are performed with previously mentioned 4 pcap files. The goal of these tests is to determine packet handling capacities of the controller



(a) Test: Inefficient Stateful Firewall application



(b) Test: Inefficient Stateless Firewall application

Figure 3.20: Test: Real-world traffic with Inefficient Firewall applications

with respect to both types of Firewalls. First, tests are performed with Inefficient Stateful Firewall application. Later, same parameters are used to perform tests with Inefficient Stateless Firewall application. Packets are captured on Tallinn and Riga using PF_RING. Riga sends packets from a pcap file at 10 Mbps to 100 Mbps packet rates. In addition to packet rates, size of Firewall ruleset is also considered as a testing parameter. For each pcap file, two types of Firewall rules are created in a purely random fashion: one allows 10% flows and second allows 20% flows from the pcap. Graphical representation of such tests is shown in Figure 3.20.

Graph 3.20a shows that with Stateful Firewall, Tallinn's handling capacity overlaps for all testcases. It handles roughly 8000 PPS for every testcase once it reaches equilibrium state which is highly influenced by TCP congestion mechanism as explained earlier. However, its behavior differs largely with Stateless Firewall. Graph 3.20b shows the result of the tests with Stateless Firewall application where Tallinn's handling capacity scatters for all testcases. For each pcap file, Tallinn's handling capacity is noted higher with 10% matching rules compared to 20% matching rules. This observation makes us find the relation between Tallinn's handling capacity and size of Firewall ruleset.

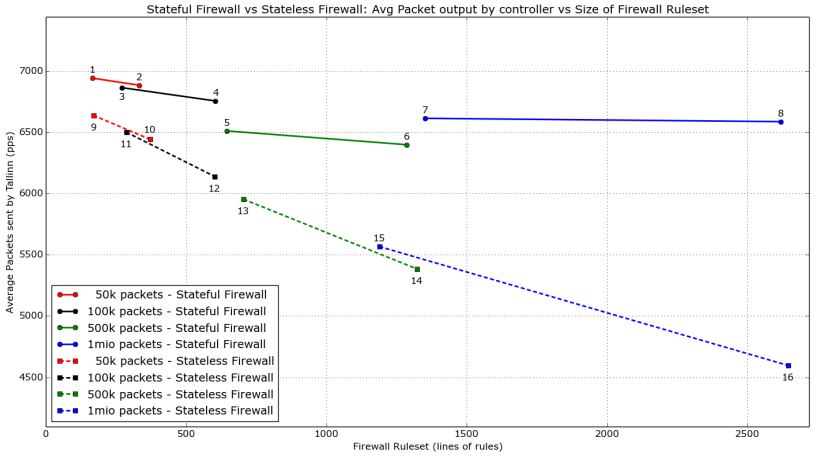


Figure 3.21: Comparison between Stateful and Stateless application

Graph 3.21 is drawn to measure average number of packets Tallinn sends out with respect to size of Firewall Ruleset. It can be seen from the graph that, for Stateful testcases, the slope remains almost negligible or zero. E.g. for 50k pcap file, slope of line joining point 1 and 2, is almost negligible. Similarly, line 7-8 that represents tests with 1mio pcap file has slope of zero. Interestingly, the Stateless testcases represent different results where slope is negative & nonnegligible. Compared to its Stateful counterpart, number of packets sent by Tallinn with 50k pcap file & 10% matching rules(point 9) are quite higher than the test with 50k pcap file & 20% matching rules(point 10). It happens for all pcap files. With almost same size of ruleset, Tallinn sends more packets with Stateful testcases than Stateless testcases. The reason for such strange controller behavior lies within the design of these applications. With Stateful applications, we first check packets against connection tracking mechanism. If we do not find a match, only then we check against original Firewall ruleset. Such design is desirable. For a packet belonging to some existing connection, it smooths up packet inspection at the controller. Since there will be few such tracked connections, it relatively speeds up overall packet processing. With Stateless applications, we do not differentiate packet inspection by checking it against connection tracking mechanism. Each packet is checked directly against Firewall ruleset which is

definitely larger than tracked connections. Hence, as the number of Firewall rules grow, controller's rate of packet inspection reduces down gradually with Stateless applications.

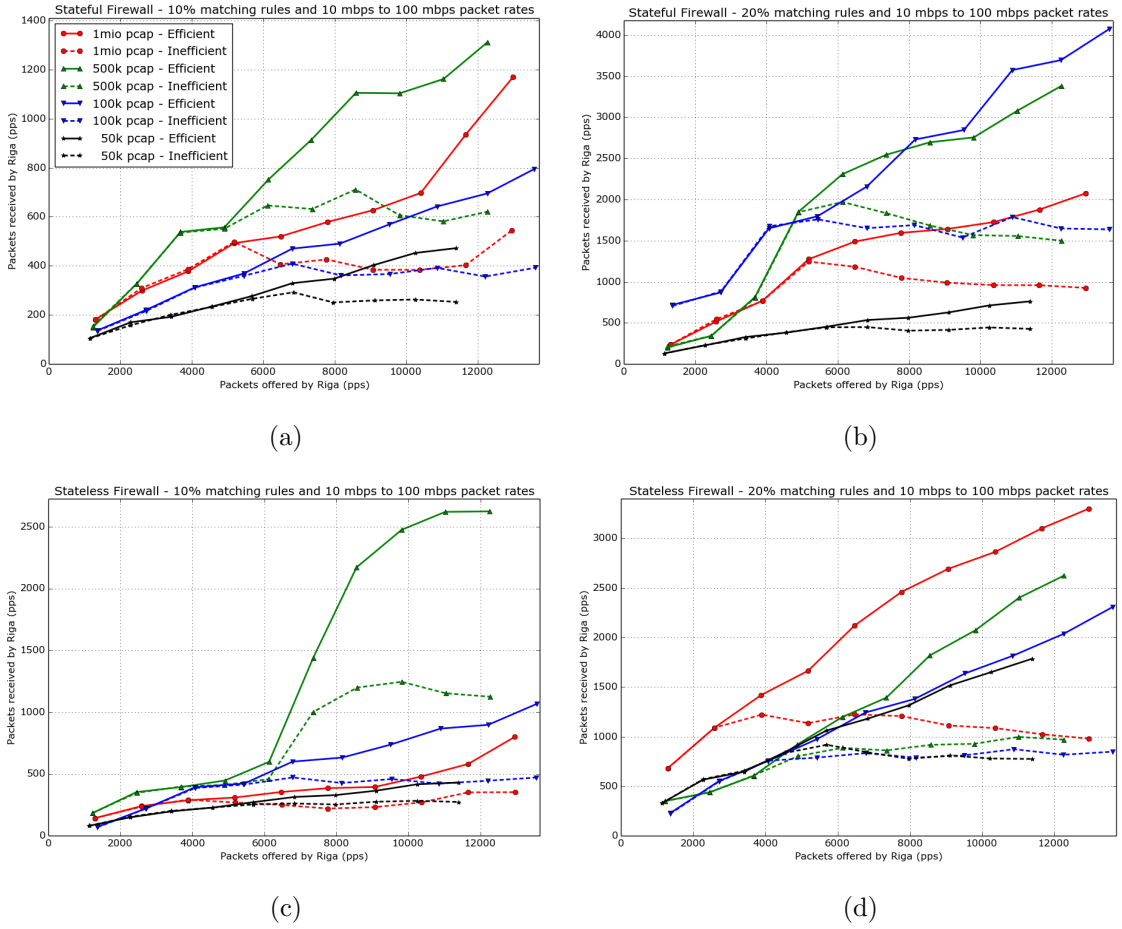
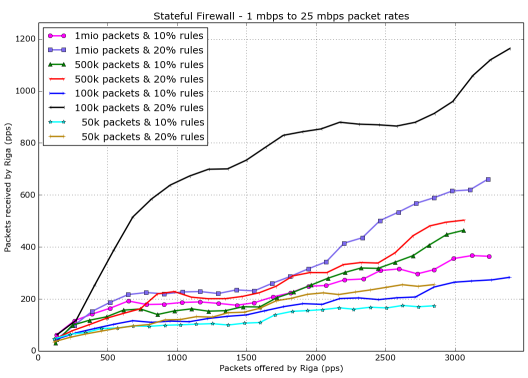


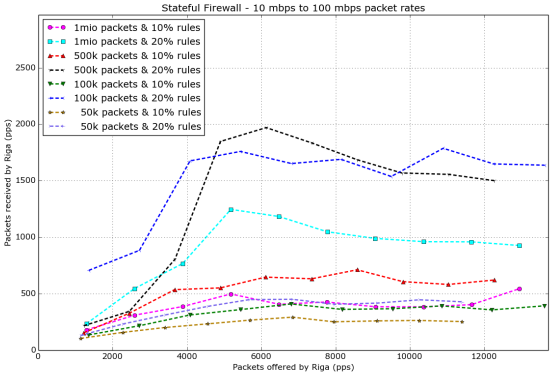
Figure 3.22: Recevied packets: Inefficient vs Efficient Firewall applications

When tests are conducted with Efficient Firewall applications, Riga has received quite higher number of packets. A comparison graph can be found in Figure 3.22. The Figure is divided into four subgraphs, each of which represents number of sent & received packets on Riga with respect to both types of applications. Such results are conceivable because of Flow Table entries. With Inefficient Firewall applications, switch forwards all incoming packets to the controller that leads to quick & high congestion in the OpenFlow channel. The congestion creates high number of packet-drops at the switch. This mechanism affects controller's handling capacities and reduces down overall receiving packet rate at Riga. It can be seen from the subgraphs that the receiving rate remains constant with Inefficient Firewall applications over all testcases even though sending rate increases.

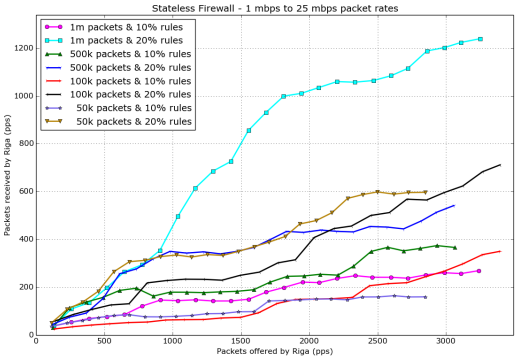
As opposed to this, switch implements productive mechanism with Efficient Firewall applications for forwarding the packets. It handles traffic on its own by using Flow Table entries and forwards less number of packets to the controller. As a consequence, this slows down the rate of congestion on the channel with time, and in turn, the packet-drops too. With such mechanism, controller's handling capacities can be increased by delaying the TCP congestion control. Hence, the receiving packet rate increases gradually with the sending rate. The subgraph results show that receiving rate with Efficient Firewall applications outnumbers the receiving rate with Inefficient Firewall applications by the factor of two to three.



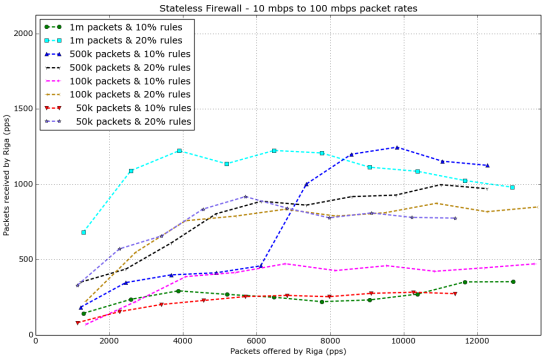
(a)



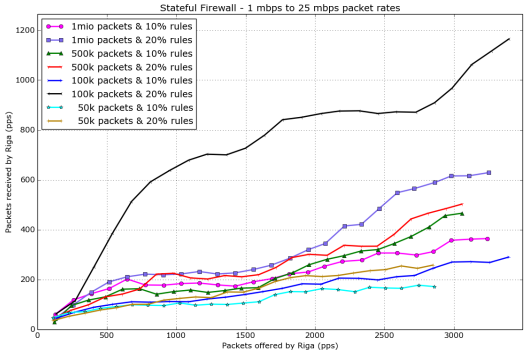
(b)



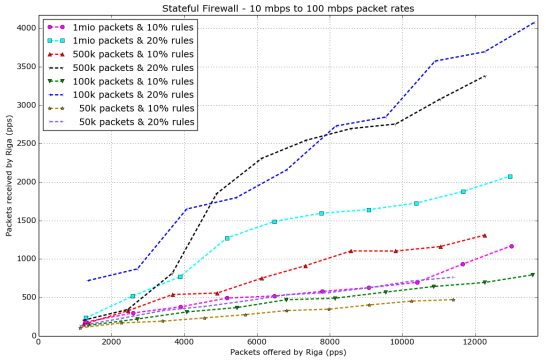
(c)



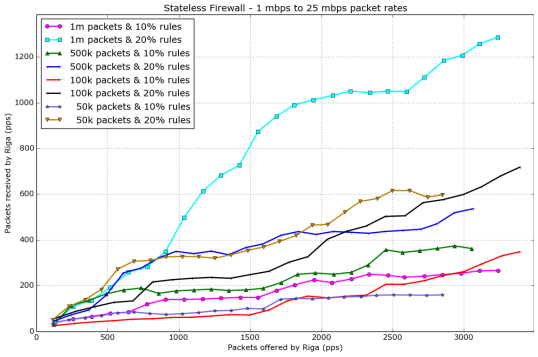
(d)



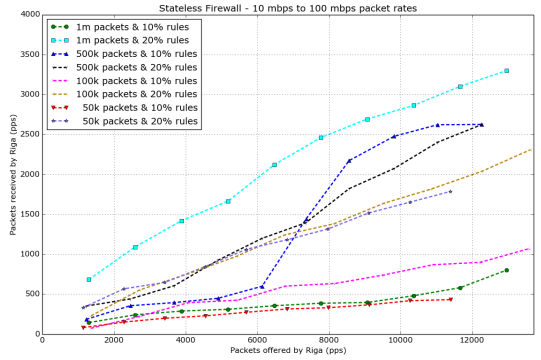
(e)



(f)



(g)



(h)

Figure 3.23: Comparison: received packets with 10% rules vs 20% rules

Even though such Firewall rules were created in purely random manner, it is observed that Riga has received more number packets with 20% matching rules than 10% matching rules. Such behavior is desirable which determines credibility of the testing environment and the working of Firewall applications. The size of 20% matching rules is almost double than the size of 10% matching rules. This difference plays important role with the tests. Since 20% matching rules have more number of lines, more number of incoming packets will have a matching entry. Hence, more number of packets will be allowed by the Firewall rules which results into higher receiving packet rate on Riga. Though PF_ RING sends packets randomly from the Pcap, the overall receiving rate is found to be higher with 20% matching rules.

Result of the tests is shown in Figure 3.23. The Figure is divided into eight subgraphs that illustrates comparison between different testcases. Graph 3.23a, 3.23b, 3.23c & 3.23d represent test-results of Inefficient Firewall applications while graph 3.23e, 3.23f, 3.23g & 3.23h represent test-results of Efficient Firewall applications. The result produces constant receiving rate for 1 Mbps to 25 Mbps tests. However, with 10 Mbps to 100 Mbps of packet rate, tests with 20% matching rules produce almost double receiving rate than tests with 10% matching rules.

Analysis of the tests with real-world traffic draws some important conclusions which are listed below.

(1) Efficient Firewall applications scale very well with the time. When implemented on controller, it was noticed that switch had handled approximately 50% of the requests by its own. Compared to their Inefficient counterparts, Efficient Firewall applications boost performance by 200% after some testcases.

(2) Stateful applications are good options for implementing Firewall in Software Defined Networking environment. They reduce down the burden of packet-inspection on controller by managing tracking mechanism. The mechanism delays TCP congestion on the OpenFlow channel and allows controller to handle more packets. The test-results show that controller handles roughly 8000 packets per second before TCP congestion comes into picture. These results depend upon the tracked states managed by the application. In contrast, Stateless applications can handle between 5000 - 8000 packets per second before TCP congestion affects the OpenFlow channel. Firewall rulesize and relevant entries play important role with Stateless applications. More rules with less relevant entries results into drastic performance whereas, less rules with more relevant entries delivers higher performance.

(3) Efficient Stateful application delivers highest performance among all applications. Since it uses connection tracking mechanism and Flow Table entries, one should consider it as a first choice where performance is the biggest concern.

4. Summary

In this document, SDN applications are presented as a way to implement Firewall functionality (Stateful and Stateless) in SDN environment. These applications are tested with local machine to determine their correct operational behavior. Later, they are deployed on real network to determine their robustness against real-world traffic with varying transmission rates. Applications are able to implement advanced Firewall feature like connection tracking which speeds up packet processing in general compared to traditional way. It is also noted that the overall performance of the applications is highly influenced by the underlying protocol (TCP) upon which OpenFlow is implemented. In general, the test results show that the applications take advantage of Flow Table entries to fasten up packet processing as well as provide security to the protected network. The test results also show that applications can work effectively with 10-100 Mbps speed but it is also capable of working with Gigabit Ethernet. However, TCP congestion mechanism limits the handling capacities of the applications; hence implementing them with Gigabit Ethernet does not affect the results.

E. Appendix

E.1 MAC address learning

Throughout the experiments, association between MAC addresses and switch ports was kept static. That is, in this IDP, controller was made aware of the topology in advance rather than discovering it dynamically. It is known that controller is communicating with forwarding devices that deal with Layer 2(MAC) addresses. In order to identify the network topology, controller keeps track of switch ports and the hosts connected to them. Whenever a packet is forwarded to controller, it learns the incoming switch port which has received this packet (for example Port1) and the sender‘s Layer 2 address. It stores this information in 2 dimensional tabular data structure like python dictionary.

After that, it checks in the table to find the correct switch port associated with destination MAC address (for example Port2). This way, controller will tell the switch to forward this packet to its particular port (in this case: Port2). Upon not finding an entry, controller will tell switch to flood this packet to all the ports except the received one (in this case: Port1).

This mechanism is vulnerable and can breach the Firewall. What if controller does not know the forwarding switch port for given destination IP address? With this mechanism, it will simply flood the packet to all ports. So, for example, if controller receives a TCP SYN packet matching the Firewall rule, then it will flood this packet to all switch ports. This can raise the issue of spoofed TCP connection in which attacker can create a spoof TCP SYN ACK packet with legitimate source IP address and can easily by pass the Firewall rule.

To overcome this vulnerability, the mechanism can be changed a bit such that controller will drop all the IP packets until it has learned about the switch port associated with destination host's MAC address. This way, we can block all TCP SYN packets to be flooded and exposed to all devices that can bring serious violation to the Firewall.

E.2 No direct support for TCP flag based Match Rule

Open vSwitch and OpenFlow Protocol support various protocol header fields like Ethernet addresses, Ethertype, IP addresses, IP protocol, TCP ports and more for constructing a Flow Table entry. However, OFP 1.3 does not provide support for TCP flags or any higher level protocol header fields directly. OFP has introduced its support for TCP flag

matching from version 1.5. (through OFPXMT_ OFB_ TCP _ FLAGS) [21]. On the other end, OVS has just started matching packets based on TCP flags in Flow Table entries [22]. But, since OFP and Ryu do not support TCP flag based matching directly [23]; implementing TCP flag based matching rule on OVS would become unnecessary task in such environment.

Therefore, initial TCP handshake packets are forwarded to the controller and matched against Firewall rule. This could lead to serious performance degradation as all packets need to be forwarded to the controller for identifying the flow. If TCP flag based matching is implemented then only TCP SYN packet from the flow can be seen at the controller and rest of the packets can be handled directly by the switch as controller can add Flow Table entries based upon TCP flags.

E.3 Firewall Rules and Flows

It is for sure the most important thing to check the relation between traffic flows and constructed Firewall rules. Of course, the more the flows match the rules, the more precisely we can measure the performance of the system. Hence, we can conclude that trade-off between Firewall rules and matching flows is a parameter which cannot be neglected while measuring the performance and concluding the result.

E.4 Commands for Mininet

To create Mininet topology:
sudo mn –topo single,3 –mac –switch ovsk –controller remote -x

Set up OVS to adapt to OFP 1.3
ovs-vsctl set Bridge br0 proto=OpenFlow13

To check installed Flow Table entries on OVS.
ovs-ofctl -O OpenFlow13 dump-flows br0

References

[1] Software-Defined Networking : The New Norm for Networks. White Paper, Open Networking Foundation, April 2012.

[2] Software-Defined Networking and Network Programmability Use Cases for Defense and Intelligence Communities. White Paper, Cisco Inc., January 2014.

[3] S.H. Yeganeh, A. Tootoonchian, and Y. Ganjali. On Scalability of Software-Defined Networking. *Communications Magazine, IEEE*, 51(2):136–141, February 2013.

[4] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN: An Intellectual History of Programmable Networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, April 2014.

[5] Robert Colin Scott, Andreas Wundsam, Kyriakos Zarifis, and Scott Shenker. What, Where, and When: Software Fault Localization for SDN. Technical Report UCB/EECS-2012-178, EECS Department, University of California, Berkeley, Jul 2012.

[6] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou. Feature-based comparison and selection of Software Defined Networking (SDN) controllers. In *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on*, pages 1–7, Jan 2014.

[7] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[8] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 85–90. ACM, 2012.

[9] OpenFlow Channel Connection. In *OpenFlow Switch Specification - Version 1.3.0*. Open Networking Foundation, June 2012.

[10] Open vSwitch FAQs. [Online] Available: https://github.com/openvswitch/ovs/blob/master/FAQ.md.

[11] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. OFLOPS: An Open Framework for Openflow Switch Evaluation. In *Proceedings of the 13th International Conference on Passive and Active Measurement*, PAM'12, pages 85–95. Springer-Verlag, 2012.

[12] Ryu website. [Online] Available: http://osrg.github.io/ryu/certification.html.

[13] Ryu: SDN Framework - slides. [Online] Available: http://www.slideshare.net/yamahata/ryu-sdnframeworkupload.

[14] RYU: SDN Framework - Book. In *Using OpenFlow 1.3, RYU SDN Framework*. RYU project team.

[15] Application Code. [Online] Available: https://github.com/jms30/SDN_Firewall/tree/master/Latest.

[16] Mininet - An Instant Virtual Network on your Laptop or other PC. [Online] Available: http://mininet.org/.

[17] Ryu - VM image. [Online] Available: http://archive.openflow.org/wk/index.php/OpenFlow_Tutorial#Controller_Choice:_Ryu_.28Python.29.

[18] Measurement- and model-based performance Evaluation and speed-up communications of Multiprocessor PC systems in High-Speed networks. [Online] Available: http://www.net.in.tum.de/de/projekte/dfg-memphis/.

[19] PFSEND and PFRING. [Online] Available: https://svn.ntop.org/svn/ntop/trunk/PF_RING/doc/UsersGuide.pdf.

[20] Open vSwitch - Packet Drop at High Packet Ins. [Online] Available: http://openvswitch.org/pipermail/discuss/2013-November/012110.html.

[21] OFP version 1.5. In *OpenFlow Switch Specification Version 1.5.0*. The Open Networking Foundation, December 2014.

[22] Open vSwitch: Newsletter version 2.1.0. [Online] Available: http://openvswitch.org/releases/NEWS-2.1.0, March 2014.

[23] RYU - mailing list. [Online] Available: http://sourceforge.net/p/ryu/mailman/message/32402649/.