SYMPHONY - A CONTROLLER FOR HYBRID SOFTWARE DEFINED NETWORKS

A thesis submitted by **Vijaya Durga Chemalamarri**
in fulfilment of the requirements for the award of the degree
Master of Science in Computing Sciences (Research)

Faculty of Engineering and Information Technology
University of Technology Sydney
March 2016

Certificate of Original Authorship

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Production Note:
Signature removed prior to publication.

[illegible]

Acknowledgement

I would like to acknowledge and thank for all the encouragement, guidance and support provided by my supervisors Dr.Karla Felix Navarro and Dr.Priyadarshi Nanda.

I am grateful for having a family that installed in me the value of knowledge since early age. I am very grateful to have my sister Rohini as a constant source of motivation. I cannot insist enough on the support provided by my husband Deba and my in-laws.

I also thank all my friends for cheering and motivating me.

Abstract

Software Defined Networks (SDN) is currently an active area of research. As enterprises migrate to SDN, an inevitable network transitional state is a brownfield state, where both Software Defined and Legacy networks coexist. To achieve interoperability between legacy and Software Defined Networks and to leverage the existence of OpenFlow devices in the traditional network to improve existing network state, a Hybrid SDN controller is a desirable addition to any brownfield deployment of SDN.The thesis of this work aims to further the knowledge in the area of Hybrid Software Defined Networks by highlighting the requirements and challenges to be addressed while integrating legacy and Software Defined Networks. The requirements and challenges discussed in this thesis focus on path computation, packet forwarding, and centralised policy application. This is achieved by building a Hybrid SDN Controller. The main controller components - controller application that runs on POX, a route server to store legacy network information, a next-hop module, a path discovery module and a policy module. We also discuss three use cases of SYMPHONY hybrid SDN controller to demonstrate the controller's application and usefulness. We also perform few key tests to determine the efficiency of deploying a hybrid SDN controller in a network.

Publications Supporting this Thesis

The following is a list of publications resulting from this thesis.

Conferences

Fourth European Workshop on Software Defined Networks

1. Chemalamarri, V.D., Nanda, P. and Navarro, K.F., 2015, September. SYMPHONY—A Controller Architecture for Hybrid Software Defined Networks. In Software Defined Networks (EWSDN), 2015 Fourth European Workshop on (pp. 55-60). IEEE.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software Defined Networks (SDN) is an emerging network model. While existing networks make a transition to SDN by deploying SDN compatible devices in a legacy network, there is a need to bridge both legacy and Software Defined Networks. The work of this thesis investigates brownfield deployments of Software Defined Networks, analyses challenges involved in the co-existence of both OpenFlow (OF) and legacy devices in a network, design and finally propose a prototype of a Hybrid SDN controller. OpenFlow(OF) protocol serves as communication protocol within the OF data plane while OSPF (Open Shortest Path First) protocol operates within the legacy networks, making well-informed path decisions to route the packets. We further develop and build various use cases to demonstrate essential networking scenarios such OSPF neighbour relation establishment over OF domains, inter domain communication, communication during link failures and policy enforcements on Hybrid networks.

1.1 KeyWords

- Software Defined (SD) Network (SDN) is a network paradigm where data plane is decoupled from the control plane. While the data plane serves as a physical forwarding plane, the control plane configures the forwarding plane to forward or filter packets. The control and data plane use southbound APIs such as OpenFlow for communication. Network functions run as northbound applications enable the controller plane to program the data plane resulting in desired network state.
- Controller is a logically centralised control entity that controls the data plane by computing flows and adding, modifying or deleting flow table entries by using a control protocol such as OpenFlow.

- Hybrid Software Defined Network is a brownfield or mixed network state that allows coexistence of traditional and SD Networks. Communication between the SDN and legacy domain is established at Media Access Control (MAC) layer or Internet Protocol (IP) layer.
- SDN data plane is an abstract used to describe a set of OpenFlow-enabled devices.
- OpenFlow (OF) is communication protocol between the SDN control and SDN data plane.

1.2 Background

The first node connected to the 'Internet'in 1969, then known as ARPANET. As more and more nodes connected to the Internet by adopting packet switching technology, a variety of network devices were invented to perform various functions and introduced into the network. Examples of such devices are switches, routers, firewalls and Intrusion Detection System. The control planes of multiple similar devices coordinated using signalling mechanisms such as routing protocols, spanning tree protocols resulting in a consistent view of the network. The distributed nature of traditional networks had multiple advantages such as scalability, reasonable convergence, resiliency, and stability.

Traditional internetworking equipment, on the other hand, had concerns such as being vendor specific, offering fixed set of features, requiring per-box management, cumbersome planning and deployment phases and high chances of human error. These concerns limited the implementation of innovative concepts such as network virtualisation or on—demand provision of Network as a Service (NaaS). Also, traditional networks did not offer a much granular level of control.

With the above factors influencing the direction of evolution of networks, the paradigm of Software Defined Networks (SDN) evolved. It would be superficial to attribute the development of SDN only to the recent years. SDN is an evolving field with initial efforts dating as back as 1995 [14]. SDN allows the existence of vendor neutral devices, provides flexibility of programming data plane and the provision of a centralised control to manage the network. Moreover, SDN allows space for much needed innovation in the area of networking. On-demand applications, network virtualisation and on-demand provisioning of a logical network using the underlying infrastructure (Network as a Service) are to name few. By physically separating the forwarding plane and the control plane of a network node, SDN allows more network innovation and develop technologies for emerging network infrastructures.

Though SDN might look like a one stop solution for all network problems, SDN still needs to address few of the known concerns such as scalability and security amongst others as will be discussed in Chapter3. A prudent proposition thus is to maintain a brownfield deployment where both traditional and SDN-based technologies can co-exist and work towards a feasible solution. Thus the work of this thesis is to investigate the brownfield deployments of Software Defined Networks.

1.3 Aims and Objectives

This thesis aims to:

- (A1) Identify and address challenges for establishing inter domain communication between traditional and software defined networks.
- (A2) Establish inter domain communication between traditional networks and software defined networks.
- (A3) Enhance policy enforcement on legacy devices.

The objectives thus are to:

- (O1) Propose requirements to address a set of challenges identified while deploying OpenFlow-enabled network devices alongside a traditional network. (A1)
- (O2) Design and prototype a Hybrid SDN controller to enable inter domain communication by performing packet handover. (A2)
- (O3) Perform traditional policy based routing using the newly deployed architecture using OpenFlow data plane. (A3)

1.4 Significance

This research is significant to various stakeholders mainly network engineers and enterprises who have a keen interest to migrate from a traditional network to SDN and also those considering a brownfield deployment. While businesses assess the value of deploying SDN, Mixed/Hybrid SDN is a logical step in the process of transitioning from a traditional network to SDN. Thus, a Hybrid SDN controller is required to establish inter domain communication between legacy and OF enabled devices.

A centralised controller manages OpenFlow devices deployed to build an SDN network. That is in contrast to how current traditional networks with distributed control work. Thus, a set of challenges arises while establishing communication between the nodes such as path computation and forwarding loops. Thus the Hybrid SDN controller is required to address the challenges faced while integrating traditional and SDN control planes. The proposed Hybrid SDN controller 'SYMPHONY', computes efficient end-to-end path in a hybrid network during various networks conditions such as the presence of redundant links, failure of links, etc.

Finally, manual configuration and troubleshooting of routing policies are cumbersome activities for network engineers. Not only are the policies applied on a per-box basis, but the configurations are also prone to human error. This research enables the network engineers to configure policies centrally, thus eliminating the need for a per-box configuration. Finally, the Hybrid SDN controller functions as a central repository that can be further upgraded to function as a centralised management unit.

Research Questions

Thus, significance of this research work lays in answering research questions of:

- Is it feasible to model and build an efficient, scalable and easy to manage controller for Hybrid SDN?
- Can such a controller make real-time route computations for establishing data communication between SDN and legacy network domains?
- How does the existence of such a controller aid in resolving issues those plague legacy networks?

This thesis looks into the possibility of building and operating brownfield networks, or networks in the transition from existing infrastructure, where both SDN and legacy devices can coexist. Such coexistence demands an interoperability and integration between IP control plane of legacy networks and control plane of SDN networks.

Research Proposal

The researcher intends to answer the research questions by identifying the challenges, developing requirements, designing the Hybrid SDN controller and building the prototype to address the challenges.

1.5 Research Methodology

A Spiral/Agile methodological approach has been followed while conducting the research. The spiral methodology illustrated in Figure 1.1 consists of four activities and can be stated as following:

- Discover
- Design
- Develop
- Test

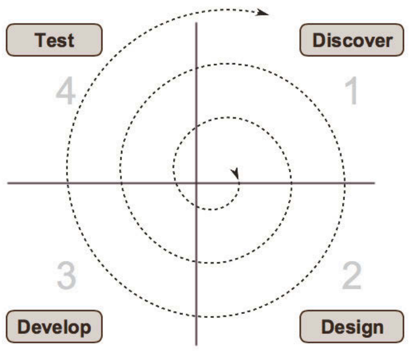


Figure 1.1: Agile Model

Step I (Discover): Conduct literature review.

A thorough literature review was conducted to determine the extent of knowledge available in the fields of Hybrid SDN and SDN in general and to revisit the principles of operation of legacy networks. The outcome of this activity is an understanding of series of challenges which need to be addressed while deploying SDN-enabled devices alongside traditional networks.

Step II (Design): Design the requirements and framework.

This stage involves developing requirements based on the challenges identified in Step I, followed by designing the first prototype of Hybrid SDN controller. The design has then been gradually enhanced to provide extended functions such as policy management. The environment for prototyping the hybrid controller was also set up during this stage.

Design Environment:

Python has been chosen as the programming language for prototyping. Python imposes lesser language constraints while allowing rapid prototyping. SDN controller used for prototyping is a Python based controller—POX. Mininet network emulator has been used for the project. MiniNExT extension for Mininet is used to emulate the traditional network nodes. MiniNExT provides individual file systems for each device; thus multiple instances of routing engine can be run to emulate routers. Finally Quagga, an open source routing engine provides the routing capability to the nodes. A Linux container running Quagga serves as a routing engine. The environment has been briefly listed in Appendix A.

Step III (Develop): Prototype of the proposed architecture.

The initial prototype was built to test if inter domain communication could be established. Once successful, the prototype has been incrementally updated and finally evolved into a more advanced system through a testing and feedback integration cycle. The prototype developed was tested using following two topologies:

- Topology with SDN island
- Topology with distributed OF devices.

Step IV (Testing): Testing at each stage.
Each design and development iteration resulted in the inclusion of new functionality that is tested by developing use cases. Thus, preliminary testing has been performed to test the functionality of the designed use case. To validate the objectives, following use cases have been defined:

- Use case I: Establish communication
- Use case II: Centralised policy application
- Use case III: Traffic management using the TCP ECN bit

1.6 Key Contributions

Following are the key contributions of our research:

- Identify various challenges for building Hybrid SDN controller
- A generic framework for building Hybrid SDN controller
- A functional prototype of Hybrid SDN controller
- Designing and experimenting with different use cases

1.7 Thesis Structure

Chapter1 of the thesis introduces the aims and objectives of the research. The chapter highlights the significance of our work, defines the research proposal and research methodologies to conduct the research. The research then applies agile methodology to discover, design, develop and test a prototype for Hybrid SDN controller. The chapter concludes by listing the key contributions of this work and presenting the reader with the structure for rest of the thesis.

Chapter2 introduces the reader to the topic of Software Defined Networks. It is in this chapter, the three components of SDN are discussed which are the data plane, the control plane and the communication protocol between the data plane and the control plane. Then the topic of brownfield network state–Hybrid Software Defined Network(Hybrid SDN) where both SD and legacy devices co-exist is introduced. Finally, the existing proposals and frameworks for Hybrid SDN are listed and briefly discussed.

Chapter3 presents the core of this research work. The chapter opens by detailing the challenges involved in building Hybrid SDN. The chapter then progresses into framing the requirements for designing and developing a Hybrid SDN controller to solve the challenges identified. The later section of the chapter introduces the reader to the proposed architecture and its components while detailing the functionality of each component such as the Legacy Routing Server, the main packet processor module, topology and path discovery modules, policy handling module among others.

Chapter4 details the experiments and use cases conducted to address the challenges discussed in earlier chapters. Three main use cases are discussed. OpenFlow switches can be deployed either as an SDN island or as a data plane for the existing L3 infrastructure. The topologies used in each experiment are described and detailed. Each of the three use cases (connectivity establishment, application of policy and congestion handling mechanisms) are then described and are tested using a specific topology as described in the experiments.

Chapter5 develops and conducts tests on the developed use cases. We have developed three different test cases. The first test case compares OSPF convergence time between the legacy network and OpenFlow network. The second test case tests how routes learnt by Legacy Route Server vary with the number of legacy routers connected and the third test case verifies the application of policy.

Chapter6 is the final and conclusive chapter of this research. While it concludes the research, it also presents the limitations and scope of future work.

1.8 Conclusions

This chapter discussed the motivation for the research work. Aims and objectives have been discussed, and research proposal has been put forth. The significance of this work and research methodologies used to conduct the research have been reviewed. Key contributions expected from this work have been listed and finally the chapter lays out the structure of this thesis. The upcoming chapter discusses the background and works related to this research work.

Chapter 2

Background and Related Work

The focal point of Software Defined Networks is providing network programming capability to the end user such as a network engineer. One of the early efforts made in the direction of programmable networks were Active Networks[39]. The next equally important focal point is the separation of control plane and data plane within a network. ForCES[43], Routing Control Platform[7], Path Control Element[13] are earlier frameworks prior to SDN, which proposed the separation of data and control plane. Thus SDN is a network paradigm to achieve a centrally programmable network by separating data and control plane. Formally, Open Networking Foundation defined SDN as 'the physical separation of network's control plane from the forwarding plane where a control plane controls several devices'.

2.1 Why Software Defined Networks?

As introduced in the previous section, primary motivation fuelling SDN deployment is programmability. Programmability introduces innovative ways in which a user (network engineer) interacts with the network. In traditional networks, device management was mostly on per-box basis via Command Line Interface (CLI). Managing the devices included activities such as the application of Access Control Lists (ACLs), Policy-Based Routing (PBRs) or enabling Traffic Policing functionality. Any modification to the policy would require the network engineer to update the configuration on each device participating in handling packets for a particular flow manually. A centrally programmable controller which applies network policy immediately solves this by not only identifying the right network policy to use but also through the identification of the devices to update the policy. Such centralised action is faster and efficient when compared to a human operator manually updating the policy.

Second, the new era of data communication calls for various data analysing techniques and a centralised control can prove to be pivotal in collecting various network statistics in real time. Currently, traditional networks are monitored using by employing network monitoring tools such as Nagios[2]. SDN control plane, on the other hand, is modular, flexible and logically centralised. The centralised SDN control plane makes the routing decision and can be extended to operate as a network traffic analyser or a policy engine. For instance, the controller can continuously analyse network traffic patterns and integrate feedback into the decision-making process in real-time. In the current state, networks are incapable of conducting such a feedback based routing.

2.2 SDN Architecture

As discussed in the previous section, the main components, which make SDN architecture, are a logically centralised control plane and a programmable data plane. For the control plane to control and manage the data plane, a communication interface between the control and data plane is required. This interface is known as Southbound interface. Unlike the Southbound interface that establishes communication between the control and data plane, Northbound interfaces (Northbound APIs) allow the controller to run multiple network services as applications, thus enabling close interactions amongst various network services resulting in a single cumulative flow table. Figure 2.1 shows SDN architecture as below.

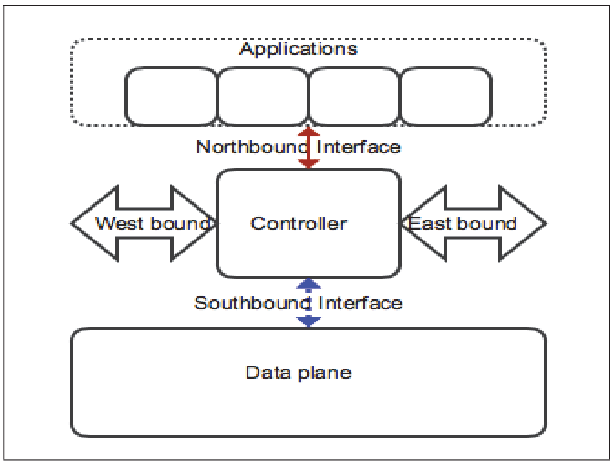


Figure 2.1: SDN Architecture

2.2.1 Control plane

An SDN controller is a single computing entity or several entities running controller software. The controller controls and manages the data plane by manipulating flow table of switches. Several networking functions such as forwarding, filtering, policy management and monitoring run as controller applications and several such applications can run simultaneously and are interfaced to the control plane via any Northbound API. The controller communicates with the forwarding plane by establishing an out-of-band TCP based TLS or simple TCP secure channel. A controller can control multiple switches by establishing multiple simultaneous connections, and a switch can be managed by a single controller or by multiple controllers for reliability.

2.2.2 Data plane

SDN separates control plane from the data plane, and thus reduces the network devices to solely forwarding entities, which can route packets as directed by the controller which is in stark contrast to the current set of legacy network devices, which are assembled to maintain both data and control plane in a single box. These new set of devices use flow table entries to forward or filter packets. The data plane forwarding elements are enabled with a communication interface to receive instructions and forward requests to the controller.

2.2.3 OpenFlow

OpenFlow[28] is an Open Software platform that provides an open interface for the controller to program flow tables on various switches. OpenFlow(OF) enables communication between the control and data plane by 1) defining a communication protocol between controller and data plane 2) maintaining a secure communication channel and 3) by provisioning an OpenFlow switch. A typical OpenFlow switch consists of flow tables, a secure channel to communicate with the controller and switching fabric to route the packets as necessary. The flow tables consist of flow identifiers and action associated with each flow. OpenFlow Architecture is shown in Figure 2.2

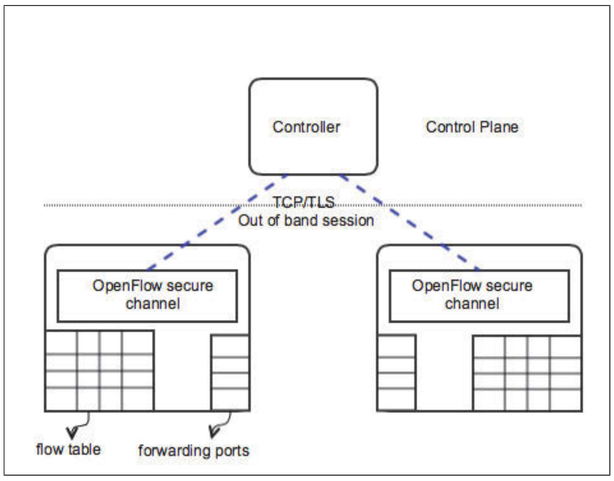


Figure 2.2: Interaction between OpenFlow Control plane and Data plane

Operation

Flow is an important attribute for OpenFlow operation. Communication stream between any set of source and destination, which matches some predefined match criteria is known as a flow. An OpenFlow switch's flow table is a table to store the flows. Incoming packets are matched against these flows based on matching criteria. The criterion is in turn based on the packet header fields. Upon receiving a packet matching none of the flows in the flow table, the OpenFlow switch encapsulates and forwards the packet to the controller. Thus, the first packet of an unmatched flow is always forwarded to the controller for handling instructions. The controller processes the incoming packet, builds a flow::action entry and updates the OpenFlow switch flow table with the new rule. The subsequent packets are matched against this flow-entry, and the corresponding action is taken. The flow table also keeps track of the number of packets (statistics) that have matched against a particular flow-entry. The structure of OpenFlow table is shown in Table 2.1.

Table 2.1: flow table

Header Match fields	Counter	Action
...	...	...
...	...	...

OpenFlow specification v.1.0[37] specifies the header fields, which the packets are matched against. This field-set has been expanded in following versions . Below Table 2.2 lists the match fields.

Table 2.2: Match Fields

Protocol	Match feilds
OpenFlow	(Ingress Port)
Ethernet	(Src Port, Dst Port, VLAN ID, VLAN Priority,Type)
IP	(Src IP,Dst IP, Protocol, TOS Bits)
TCP/UDP	(Src Port,Dst Port)

The flow table also has at least one action against each flow (flow::action). Actions can be reclassified as 1) required actions - {Forward, Drop} and 2) optional actions -{Enqueue, Modify-Field}. Below Table 2.3 lists and describes the OF actions

Table 2.3: Types of Actions

Action	Description
Forward	Forward all the packets matching a specific flow to a destination port. Destination port can either be a single/multiple physical ports or virtual ports. Virtual Ports - ALL, CONTROLLER, LOCAL, TABLE, IN_PORT
Drop	Drop the packet that matches a flow with no action.
Enqueue	Configuration of a queue attached to a port dictates the forwarding behaviour.
Modify-Fields	Modify fields of packets before forwarding packets. Fields which can be modified are -VLAN ID, VLAN priority, VLAN header, Eth src MAC addr, Eth dst MAC addr, IPv4 src IP addr, IPV4 dst IP addr, IPV4 TOS, TCP/UDP src port, TCP/UDP dst port

The following subsections briefly list and describe the OF messages, events, and handlers relevant to this work.

Types of OF packets

The below are the packets used to build SYMPHONY. The packet descriptions are limited to OFv1.0. More detailed description of the same is listed in [37]

ofp_packet_in

ofp_packet_in message is an asynchronous message generated by the switch. The data-path uses this message to inform the controller about an incoming packet. The message structure is listed in Table 2.4

Table 2.4: ofp_packet_in message structure

Field	**Purpose**
header (ofp_header)	Default ofp header
buffer_id	ID of the buffer where packet is buffered on the datapath
total_len	full length of the frame
reason	OFPR_TABLE_MISS, OFPR_APPLY_ACTION, OFPR_INVALID_TTL, OFPR_ACTION_SET, OFPR_GROUP, OFPR_PACKET_OUT
in_port	port on which the packet arrived
data	Ethernet frame

ofp_flow_mod

ofp_flow_mod message is a *controller–to–switch* message. The controller uses this message to modify a flow. The modification can be either adding or deleting a flow or modification of existing flow table. Structure of the *ofp_flow_mod* message is listed in Table 2.5

Table 2.5: ofp_flow_mod message structure

Field	**Purpose**
header (ofp_header)	Default ofp header
cookie	Opaque Controller issued identifier
command	OFPFC_ADD, OFPFC_MODIFY, OFPFC_MODIFY_STRICT, OFPFC_DELETE, OFPFC_DELETE_STRICT
idle_timeout	idle time before discarding a flow
hard_timeout	max time before discarding a flow
priority	priority level of flow insertion
buffer_id	buffer id of the buffered packet or OFP_NO_BUFFER
out_port	port to queue packet on.
action	list of actions that define packet processing by the datapath
flags	bitmap of any OFPP message flags
match(ofp_match)	fields to match (ref pg.)

ofp_packet_out

ofp_packet_out message is a *controller–to–switch* message. The controller uses this message to instruct a data path or switch to send out a packet. Structure of the message is listed in Table 2.6

Table 2.6: ofp_packet_out message structure

Field	**Purpose**
header (ofp_header)	Default ofp header
buffer_id	buffer id of the buffered packet
in_port	ingress port or OFPP_NONE
actions_len	size of action array
pad	padding
actions (ofp_action_header)	list of actions that define packet processing by the datapath
data	packet data

ofp_port_status

ofp_port_status message is a asynchronous message sent by the switch to the controller. Whenever ports are added, removed or modified the controller is updated using the *ofp_port_status* message. The structure of the message is listed in Table 2.7

Table 2.7: ofp_port_status message structure

Field	**Purpose**
header (ofp_header)	Default ofp header
reason	OFPR_ADD, OFPPR_DELETE, OFPPR_MODIFY
pad	padding
actions_len	size of action array
pad	padding
desc (ofp_port)	port description port_no, length, pad, hw_Addr, pad2, config, state

Events

Every OF datapath establishes communication with the controller. In this work, we use POX controller[30]. OF messages received from the OF data plane raise events on the POX controller, which in turn are handled by the event handlers configured in the controller application. Typically every message is associated with an event. Table 2.8 list of OpenFlow events handled by SYMPHONY, which can be raised on the POX controller.

Table 2.8: List of Events

Event	**Description**
ConnectionUp	The event is triggered when a new OpenFlow switch connects to the pox controller. Attributes: {connection, dpid, ofp}
PacketIn	*ofp_packet_in* message triggers this event Attributes: {connection, dpid, ofp, port, data, parsed, ofp_packet_in,}
PortStatus	*ofp_port_status* message triggers this event. Attributes: connection, dpid, ofp

2.3 Hybrid Software Defined Networks

> The marriage of dynamic routing and signalling and a centralised view is incredibly powerful. It enables the fastest possible protection in the event of a failure, the greatest resiliency, and the ability to place services into a network in one command. The two technologies working together is really a major step forward that wasn't previously in our toolbox.–[31]

Traditional networks based on IP and Ethernet based deployments are mature and operate well as a distributed platform. The SDN controller can only control those devices with an OpenFlow/similar interface, and most traditional devices do not support OpenFlow. Due to custom built ASICs of legacy devices, migrating these networks to Software Defined Networks (SDN) would need a complete overhaul of existing network devices or upgrading the firmware (if possible) to support OpenFlow protocol. It might not prove profitable for an enterprise to replace its existing infrastructure entirely to implement SDN thus impeding overall adoption of SDN. On the other hand, gradual deployment of OpenFlow-enabled devices in traditional networks would not need legacy equipment to be made redundant.

Interoperability between legacy IP and SDN control planes opens up a variety of opportunities for various stakeholders and to the network in general. For example, an integrated control plane makes network planning, configuration, and management effortless for the network engineers by providing a single plane of operation. Also,'tying'critical IP devices with OpenFlow devices provides the traditional devices with a new data plane, which can be configured centrally, thus attaining SDN-like control over traditional network devices. For instance, by focusing on IP and SDN integration, IP specific functionalities, especially those with policy configuration such as traffic policing, ACLs, and policy-based routing, could be computed centrally and applied to networks in a more flexible and easier to manage fashion. The controller can compute and transform an end-to-end policy into OpenFlow flow table entries and apply to the entities in the data plane, thus eliminating per-box policy configuration.

Hybrid Switches that include legacy control plane (non-OF enabled switches) and an OF compatible data plane are now commercially available. Since traffic segregation is upon the switch, the ability of the SDN controller to influence the legacy network is compromised as the hybrid operation is localised to a particular node.

Apart from the above factors others such as investment on skill might prove to be a case for the existence of Hybrid SDNs. The current workforce is highly skilled in managing a distributed network and adopting an incremental deployment approach might provide the necessary time for the workforces to up-skill themselves. By diligently deploying OpenFlow-enabled devices, investment in new equipment can be drastically reduced, and a complete overhaul of the existing legacy network can be avoided. Finally, few enterprises might be willing to try SDN on a trial basis and deploy SDN alongside with their traditional networks and not migrate completely. All the factors mentioned above make a strong case to deploy Hybrid SDN.

The following section briefly discusses Some of the previous work in the area of Hybrid Software Defined Networks.

2.3.1 Existing Frameworks for deploying Hybrid Software Defined Networks

Control planes of traditional networks and SDN can be integrated either at layer 2, by using VLAN technology or at layer3 by using IP protocols. Irrespective of their layer of operation, few of the existing frameworks proposed for Hybrid SDN are investigated below.

RouteFlow[32] builds a virtual L3 topology by mapping every OpenFlow-enabled node to a Virtual Machine (VM) running a routing engine. The architecture consists of a slave daemon rfslaved running on each VM, a routing engine Rfserver and a controller, which runs routeflowc daemon. The Rfserver communicates with the VM via Rfprotocol and computes the corresponding flow modification commands. The controller uses these flow modification commands to configure the forwarding plane using OpenFlow. While interacting with legacy L3 nodes, messages initialised in the virtual plane are converted and transmitted to the physical OpenFlow domain via the Rfserver and the controller, whereas routing control messages from physical OF domain, are switched to the virtual domain by the controller and the Rfserver.

I2RS[20] is an IETF standard aiming to consolidate routing, based on flow, policy, more aware of application, time and external events and network state while making use of existing software where possible. The framework consists of a routing element, which runs an I2RS agent and applications, on the clients. Any given client can control multiple agents running in the routing element. The I2RS agents can communicate with one or more I2RS clients running on the application. While OpenFlow modifies the FIB table in the data plane, I2RS works within the control plane while interacting with the L3-forwarding engine and the routing table to modify the RIB tables. The I2RS interface can also be leveraged to provide connectivity to legacy nodes.

Panopticon[24] proposes a transitional model of network architecture consisting of legacy switches (non-OpenFlow) and OpenFlow switches. The incremental deployment model requires a fewer OpenFlow switches or a subset of legacy devices to be upgraded to OpenFlow. The framework enforces waypoint enforcement where flows between any two hosts traverse at least one OpenFlow node. The framework groups the set of non-OpenFlow nodes as cells, the set of OpenFlow nodes adjacent to cells as frontiers nodes, and finally builds safe paths/Solitary Confinement Trees (SCT) based on running STP within a cell which includes at least one frontier node. A VLAN ID is assigned to each SCT to maintain isolation. Traffic originating from legacy or non SDN-policed ports is forwarded based on legacy principles and might be policed by an OpenFlow node if encountered while traversing a path.

CARDIGAN[38] builds a distributed router using OpenFlow. The project is typically one of the use cases of RouteFlow, where the mesh of OF-enabled nodes are aggregated as one single virtual router and exchanges routes with the legacy network devices through the RF proxy as described in [32].

LegacyFlow[12] enables the SDN controller to manage non-OpenFlow devices through an intermediate layer to translate rules into legacy node configuration syntax. The framework introduces a virtual data path LegacyFlow Datapath(LD) which operates as a proxy, between the legacy node and controller. The LD communicates with the LegacyFlow switch controller using OpenFlow protocol and applying the corresponding OpenFlow actions to the legacy nodes using CLI/SNMP or Web Service mechanism. Each legacy node maps to a corresponding LD and the corresponding ports on each of these legacy nodes are subsequently mapped to virtual ports on the LD.

HybNET[26] visualises a set of legacy switches as virtual links between OpenFlow switches. Hyb-

NET controller computes the commands for a given operation, separates the OpenFlow commands from legacy commands and dissipates the OpenFlow commands to OpenFlow controller via REST and to legacy nodes using NETCONF. The architecture has three main components: a physical topology, the pathfinder, and a controller. The framework leverages network virtualisation principles to create paths between hosts using VLANs for legacy switches and switch slices for OpenFlow nodes. Once the path computation component computes the path, computed ingress and egress ports on legacy switches are configured in a VLAN and a switch slice on OpenFlow nodes. Consecutive VLANs and switch slices are joined to create a pipe for the data packet to travel.

SUMA[9] is an intelligent middle-box filtering layer to control and provide management abstractions amongst SDN controllers, legacy NMS, and OpenFlow nodes. SUMA has three primary modules to handle all monitoring operations. The filtering and common processing module that performs smart packet/flow filtering, event and alarm mitigation, traffic classification and forwarding based on network events. The transformation and adaptation module communicates with remote management controllers and requests policies, among other elements. Finally, the dynamic event monitor module performs operations such as switch health inspection and traffic analysis among others. The middle-box directs the events to respective controllers (legacy or OF) for further processing.

iNMCS [36] proposes a framework, in which traditional network management tools interact with the SDN controller to provide network management capability for the complete mixed/hybrid network. The framework controls a set of hybrid switches (which run both legacy and SDN protocols). The control decision engine is the main component, which makes policy-based control decisions based on the inputs from various sources as the State API (monitors state of users/devices), Event traps and the Policy Manager. The flows that need to adhere to QoS requirements are forwarded to the controller by the Hybrid OF switches; whereas other flows are handled by legacy protocols such as STP.

Traffic-engineering model proposed in [1] for a hybrid network leverages OSPF_TE. Traffic is categorised as either controlled traffic and uncontrolled traffic. The controlled traffic passes through SDN Forwarding Elements (SDN_FE) whereas the uncontrolled traffic does not pass through SDN_FEs while traversing the shortest path. The controller peers with SDN_FEs running OSPF_TE and maintains the routing table information on behalf of SDN_FEs to perform traffic engineering on the controlled traffic.

OSHI [35] proposes running a routing daemon within the data plane. By interfacing OpenVSwitch with the physical interfaces, OSHI forwards all IP packets to the routing daemon via Linux forwarding and the rest of the packets to the SDN controller.

[22] proposes an architecture called Routing Interoperability with SDN Flow-map(RISF). This solution suggests a RISF agent and RISF SDN application . The RISF SDN application translates legacy messages into OpenFlow flow mod messages and vice versa and installs the flows on the switches,

The following are the few of the proposed BGP-based solutions. The authors in[18] propose inter-domain routing controller to inter-operate with BGP.The framework as suggested in[19] implements a software-defined IPX controller where the controller implements multiple pipelines for each of policy processing, route server interaction with BGP neighbours. RCP proposed by[7] runs a Routing Control Server (RCS) daemon to calculate routes for the complete network taking various inputs such as BGP routes, network-wide policies, and customer specific routing policies.

Other works in the area such as [40], [36], [4] also discuss centralised network control and policy management models. The Inter-domain Management Layer architecture proposed by [40] decouples routing and policy domains for better interoperability between multiple SDN ASs and with traditional AS. The authors of [21] suggest a global level policy transformation within an SDN domain. Procera[42] is an event-based dynamic policy management framework. The network/policy state can be monitored via SNMP and other management services. Events are then raised on the policy engine, which re-computes the policies while network controller deploys them.

Hardware based solution proposed in [8] introduces a new device *SDN Shim* to mediate messages between SDN controller and legacy devices. The authors suggest that the solution is applicable for low traffic environments. OpenRouter [15] is another hardware-based solution proposed to embed an OpenFlow module within the router to be controlled by the SDN controller and modify the information stored in the routing table of the router. Works such as [3] propose a hardware abstraction layer for non-OpenFlow compatible devices.

Above discussed frameworks partially meet the devised expectations presented in this thesis. While RouteFlow bridges legacy and OF enabled SDN domain by interconnecting OpenFlow-enabled and non-OpenFlow devices at Layer3, complexity, multiple points of failure and complete isolation of OpenFlow domain in case the RF server goes down are some of the concerns which need to be investigated further. I2RS framework[20] also addresses most of the challenges, but adoption of yet another new communication protocol. Cardigan[38] framework is an implementation of RouteFlow in one of the operational modes and hence similar issues apply. Panopticon[24] on the other hand assumes that routers are only limited to behave like gateways and doesn't seem to be addressing challenges faced while interfacing to Layer3 devices. Also, policy enforcement such as ACLs is still done in the traditional way on Non-SDN-policed traffic. The network is consolidated to single logical SDN switch. Thus the effectiveness of policy-based computations is based on the number of SDN devices deployed. LegacyFlow[12] seems to be currently operating at Layer2 and for an enterprise network having a large number of nodes, an equal number of virtual instances need to be spurred up. Also, as mentioned in [12], reactive mode of packet forwarding is compromised. Also, the LegacyFlow data path should be capable of interpreting multiple vendor devices to transform OF commands into appropriate equipment syntax. Furthermore, it is unclear if other legacy devices such as routers will have their LegacyFlow data path configured. HybNET [26] on the other hand operates at Layer2, and the disadvantage of this model lies in the fact that maximum number of VLANs what can be configured is 4096. SUMA[9] is a unified monitoring middle-box, which unifies Legacy NMS, SDN controller, and OpenFlow switches. This framework does not suggest communication integration of legacy network and SDN network. Unlike [25], the thesis of this work proposes a separate routing server to form adjacencies with legacy routers using OSPF.

2.4 Conclusion

In this chapter, the background for this thesis has been laid. The related work in the area of Hybrid SDN has been discussed. The chapter introduces SDN technology and briefly discusses the OpenFlow protocol. The chapter then proceeds to introduce the concept of Hybrid SDN and briefly outlays the existing work in the field of Hybrid SDN. Various frameworks have been studied and investigated including Panopticon, LegacyFlow, HybNet, RouteFlow, and Cardigan. Thus the motive of this thesis to bridge legacy and SD networks beyond establishing inter-domain communication has been reinforced. The thesis demonstrates how functions earlier implemented on routers can now be deployed in a centralised manner. These functions are discussed in coming chapters of 3 and 4.

Chapter 3

SYMPHONY–A Controller for Hybrid SDN

As discussed in the previous chapter, SDN control plane, and data plane interaction mechanisms do not apply to traditional networks and vice versa. Also, given that SDN aims at centralising the network control in contrast to the existing distributed control in a legacy network. There is thus a need to revisit some of the fundamental networking requirements such as reachability and reliability of routes within the realm of a Hybrid SDN. Also, as previously highlighted, a centralised hybrid controller would not only enable seamless inter domain communication between SDN and legacy network but also open up newer avenues for improvement in traditional networks and vice versa. In this chapter, the architecture, components and operation of Hybrid SDN controller SYMPHONY are presented. The following section lists some of the key requirements for building Hybrid SDN controller followed by addressing the challenges in meeting the requirements. Finally, a prototype of our proposed controller meeting the requirements for enabling Hybrid SDN is presented.

3.1 Need for Hybrid SDN Controller

Following requirements have been defined for building a Hybrid SDN controller.These requirements are based on an earlier set of requirements as proposed by [33] and [17].

Route validity and path visibility

In the context of Hybrid SDN, a route is considered valid if there exists a consistent path between any two hosts and the path terminates at the desired destination. That implies that consistent paths not only need to exist between the hosts in the legacy network and hosts in SDN but also between a host in legacy network and a host in SDN. The controller plays a vital role in propagating valid routes across the two domains, thus providing path visibility.

Fault tolerance, detection and rectification

Any link failure or congestion within the network (either in the traditional or the SDN domains) should prompt the controller to compute alternate paths and update flow tables accordingly. The convergence time of a hybrid network in such cases should be at least the same as in traditional networks. Also, controller redundancy and its placement should be given a complete thought in this context.

Planning, testing, deployment and active network management to benefit the existing legacy networks

Current network deployment models leave much of the network planning, configuration and validation activities to the network engineer. SDN controller stores topology and policy information of the entire network, and can be put to use during the planning-to-deployment cycle. Such application improves the success rate of any given network implementation. Moreover, network-monitoring tools can be made to be more dynamic to allow network engineers to perform tests on active network 'maps' to analyse the effect of changes. Through the seamless incorporation of SDN into a legacy network domain, the controller can be cloned into a test environment, and relevant changes/upgrades can be performed, conflicts in policy can be verified, rectified and deployed on the main controller.

Simplified policy

A centralised controller can simplify the task of network management. While an SDN controller manages only OpenFlow (OF) enabled nodes, a centralised Hybrid SDN controller can maintain policies throughout the network by perceiving the hybrid network as one single logical homogeneous domain. The controller can handle any prevalent anomalies within the policy computation such as conflicting routing policies, ACLs, and Policy Based Routing (PBR) rules. The controller should then be capable of offering improved operation, administration, and management (including policy dispatching) capabilities concerning traffic engineering, and QoS adherence among others.

3.2 Hybrid SDN Controller Design Challenges

This section highlights the challenges to be addressed while building a framework for Hybrid SDN. The challenges can be broadly categorised as

- control plane challenges

- management plane challenges.

The control plane challenges develop on the routing pathologies put forth by V. Paxson in 1996 [33] and J. Rexford [17]. Control plane challenges affect functions of path computation, routing states maintenance, and routing protocols operation among others. By addressing the challenges, communication between the SDN control plane and legacy control plane is accomplished. A Hybrid SDN island topology as in Figure 3.1 is used to illustrate the challenges. As shown below R1, R2 are legacy routers and O1, O2, O3, O4, O5 are OpenFlow devices.

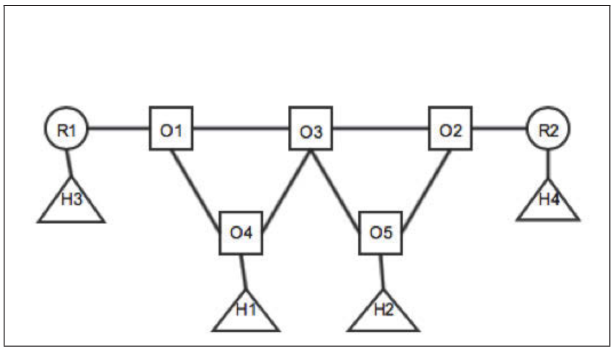


Figure 3.1: Hybrid SDN

Legacy control traffic

As depicted in Figure 3.1, For R1 and R2 to exchange OSPF packets, it is important that the route signalling packets between R1 and R2 are exchanged over the SDN setup using OF switches O1, O3, O2 or any other alternative SDN path. The routing protocol control messages between routers R1, R2 are forwarded by nodes O1, O3, and O2 to the controller for processing as any other packets with unknown action. This is followed by the installation of flow table entries by the controller. Exchange of control packets (OSPF LSAs) should complete within the timers set for legacy protocols for routers to maintain adjacency. Any delays incurred during the OSPF hello packets exchange process could prevent legacy routers from forming OSPF adjacencies. Thus, the controller should handle legacy control traffic appropriately.

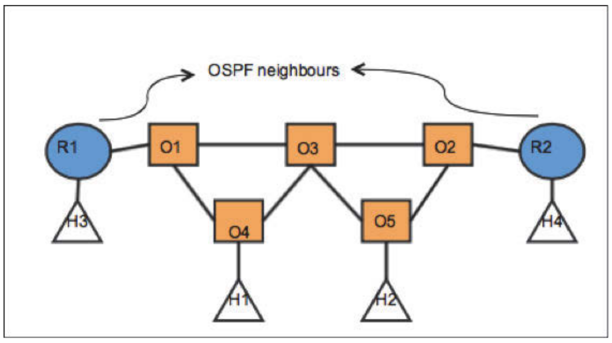


Figure 3.2: Legacy control traffic

Selecting better egress nodes to lead to choosing an optimal route

In the case of a topology based hybrid architecture [41], if a single link connects SDN to the legacy network, a static route can be used to bridge both domains. Contrastingly, in the presence of multiple links (ex:(R1, O1),(R2, O2)) as in Figure 3.2 the controller, should compute not only the best path within the OF domain but also have visibility on how the legacy network handles the packets. For instance, as depicted in Figure 3.3 let R1 be the default gateway for H1. Within the legacy network, R2 has a better path to the destination node Hn. Hence the packets from H1 destined to Hn, should traverse via R2 and not be using the configured default gateway (R1).

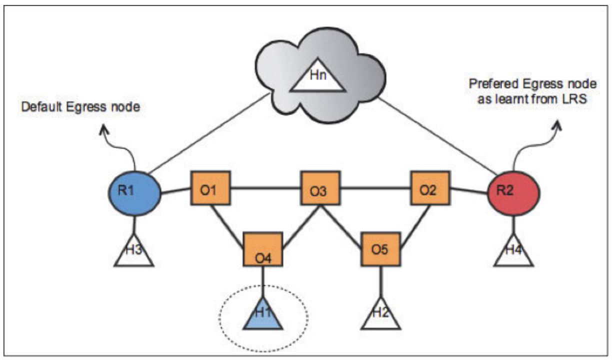


Figure 3.3: Selecting an egress node

Management plane functions such as the configuration of policies for traffic engineering, ACLs, and network availability monitoring can also affect path/route selection. When the controller maintains a global, unified view of the hybrid network, these challenges can be addressed. The other management challenges put in perspective of Hybrid SDN are discussed below

Invalid Routes due to ill-transformed policies between centralised and distributed domains

Centralised controller can only control OpenFlow devices, and has no visibility over the policies configured on traditional nodes. In Figure 3.4 R1 is set up with an ACL to deny specific traffic on the interface connecting to O1. As the controller is unaware of the deny policy configured on R1, the controller might determine O1 as the edge OF node and install flow table entries on O4 and O1 resulting in packet forwarding on the port connecting to R1, which is configured with a deny policy. Since the controller is unaware of any policies configured in the legacy domain, the controller has no way to quarantine this path and use O2 as the exit node and compute O4, O3, O2 or O4, O3, O5, O2 as the best path for packets destined to remote network from H1.

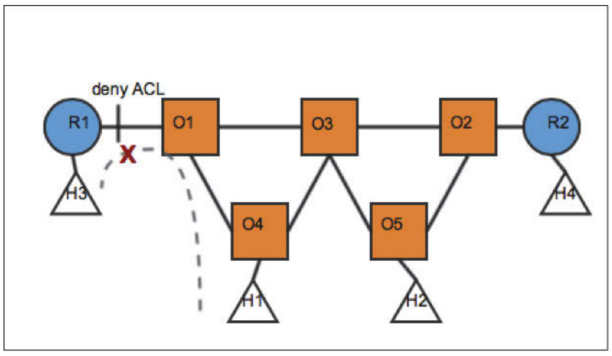


Figure 3.4: Invalid routes

Inconsistent policy configuration (non-optimal path routes)

In Figure 3.5, consider a communication instance between host H1 in an OF domain and a remote host Hn in a legacy network. Both inefficient and inconsistent policy configuration can not only lead to packet delivery on a longer and inefficient route but sometimes also in packet loss. For a flow between H1, Hn a deny policy at the remote router and a permit policy on R2 leads to conflicting, inconsistent and multiple policy configurations for same flow. None or partial policy information available to the controller might result in O2 being chosen as the egress node. These packets eventually get dropped at the remote router as shown in Figure 3.5. Such an inconsistency can lead to a packet loss.

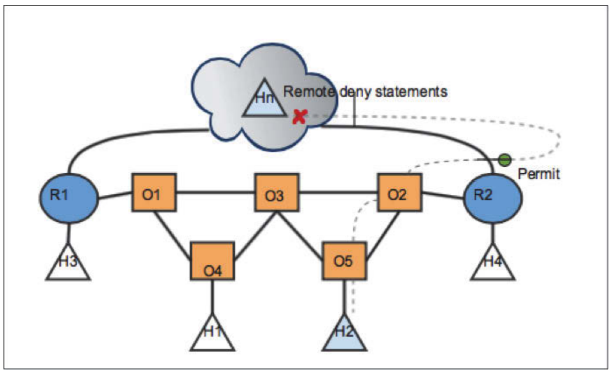


Figure 3.5: Application of inconsistent policy

Invalid Routes caused by inconsistent policy configuration

Inconsistent policy configuration can also end in looping of data. An instance where R1, R2 are configured with ill-formed policies can lead to a routing loop in the topology shown in Figure 3.3.

3.3 HYBRID SDN CONTROLLER ARCHITECTURE

The hybrid SDN controller proposed as an outcome of the previous section allows seamless data communication between legacy domain and SDN domain by establishing communication between the control planes of SDN and legacy network. The proposed controller architecture aims to orchestrate legacy and SDN control domains, hence the name –SYMPHONY.

The controller programs OF data plane not only to handle intra OF traffic but also to perform other (legacy) functions to support inter domain (between SDN, legacy) communication. The controller computes and installs end-to-end flows on the OF data plane proactively and avoids delays caused by reactive flow computation and installation. The controller performs legacy functions with the aid of a Legacy Route Server (LRS). Controller, together with LRS is capable of computing end-to-end flows between any two given hosts. Thus, two main components that build SYMPHONY are a Legacy Routing Server (LRS) and a Packet-Forwarder controller application built on SDN controller. The other components include Path-finder, LLDP, Next-hop and Policy modules as seen in Figure 3.6

The components Path-finder, LLDP, Next-hop and Policy interact with the main component–the Packet-Forwarder application. LRS is a Linux container [27] running Quagga [34] routing engine. The Packet-Forwarder along with LRS is referred to as the Hybrid SDN controller. Other passive components such as Hosts.txt, Switches.txt and Edges.txt store topology information and are referred to by the controller to compute and install flows proactively. The architecture and interaction among various components of the Hybrid SDN are illustrated in Figure 3.6.

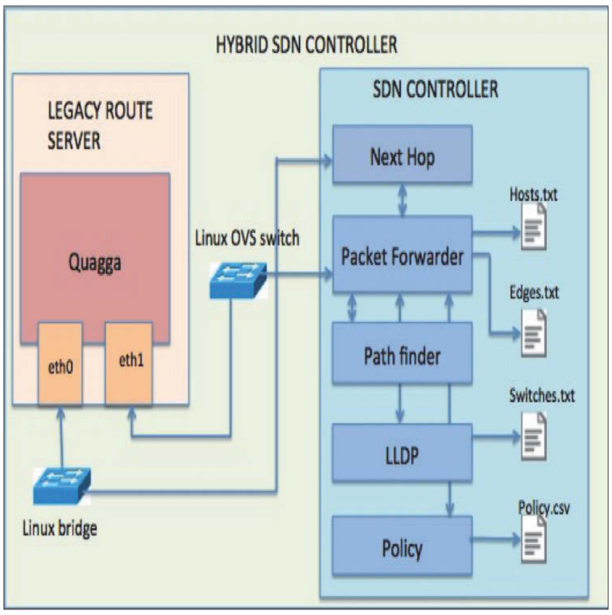


Figure 3.6: SYMPHONY Controller

3.3.1 Packet-Forwarder

The Packet-Forwarder component runs as a controller application that listens and handles incoming OF events raised by the data plane. In this prototype the Packet- Forwarder module listens and handles ConnectionUp, PacketIn and PortStatus type of OF events. This is the main controller application that performs legacy functions and enables the controller to replicate the behaviour of a legacy L3 router such as deciding the next best L3 hop, multicast and unicast packets and apply static, dynamic policies. These functions are performed with the aid of LRS. Apart from performing legacy functions, the controller also handles OF events from the data plane, thus making a well-informed packet handling the decision. These decisions are then conveyed to the data plane through various activities such as queuing the packet to out-queues of switches, inserting or deleting flow table entries using different OF messages as discussed in Chapter2.

Event Handlers

Event handlers are defined to handle the events raised by OF packets on the controller. These event handlers are simply functions that get executed upon receiving a trigger. SYMPHONY handles ConnectionUp, PacketIn, Port Status packets using the below detailed event handlers.

handle_ConnectionUp

When a new connection is established, a ConnectionUp event is triggered on the controller. SYMPHONY uses these ConnectionUp events to collect information about the network. Specifically, the controller uses the ofp attribute type ofp_switch_features to collect information about the newly connected switch such as the switch's dpid, ports and stores the same in the switch_ports dictionary. This information is later used to build the topology table. It is also during the handling of ConnectionUp event that the controller inserts default flows in the switch's flow table. As of now the default flow table entries inserted by the controller instruct the OF switches to forward the LLDP and OSPF packets to the controller. The controller uses these packets for topology discovery. Structure of switch_ports dictionary is shown in below Figure 3.7.

```
{sw1:[(p1,mac1),(p2,mac2)...],sw2:[(p1,mac3),(p2,mac4)..],...}
```

Figure 3.7: Switches_port dictionary

handle_PacketIn

Upon receiving an ofp_packet_in message from the data plane, a PacketIn event is raised on the controller. The handle_PacketIn event handler handles the PacketIn event by parsing the incoming packet and the headers to determine the packet type (IP, Ethernet, ICMP, OSPF, TCP, HTTP and others). The handler identifies if the packet is a unicast packet or a multicast packet by verifying the destination details in the OF packet payload. Based on the packet information the handler arrives at a sequence of actions such as dropping, forwarding the packet or manipulating the payload header fields before forwarding. These actions are then conveyed to the data plane in the form of updates to the flow tables. The result of this process is a single/multiple flow table entries, which are installed on a single/multiple OF switches.

Multicast

The multicast method of the controller exhibits similar behaviour as the multicast in legacy networks and one of the advantages of SDN is the ability to override of a legacy function with a new function. Both OSPF multicast packet and an ARP broadcast packet are handled as multicast packets by the controller. The controller chooses a set of OF switches to forward the packet. This set of OF switches (multicast group) is decided based on the type of packet received. For example, in case of OSPF multicast packets, the controller chooses edge OF nodes as target switches. The controller then determines the set of intermediate OF switches based on the path computed between the source and destination OF switches. Thus flows for packets of different flows can be installed in different set of OF devices. Multicast process is illustrated in Figure 3.8.

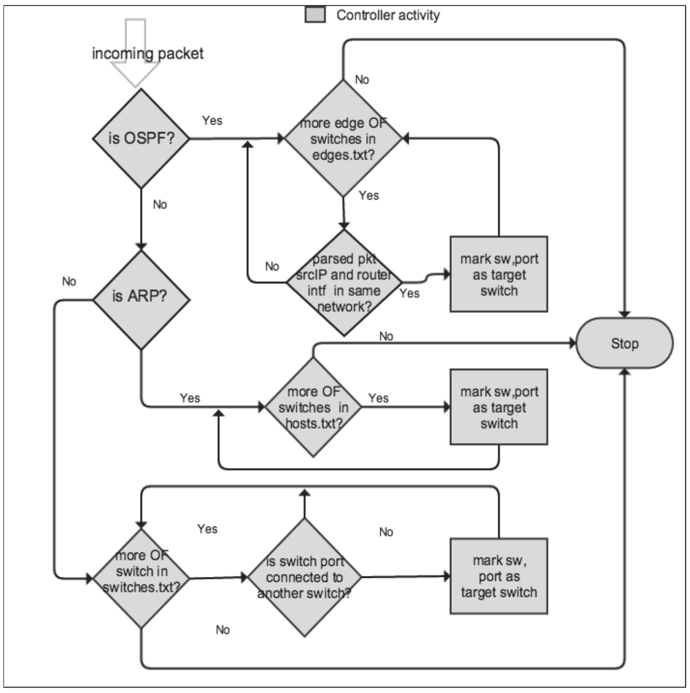

Controller activity
incoming packet
is OSPF?
Yes
No
more edge OF switches in edges.txt?
No
Yes
parsed pkt srcIP and router intf in same network?
No
Yes
mark sw,port as target switch
No
is ARP?
Yes
No
more OF switches in hosts.txt?
Yes
No
mark sw,port as target switch
Stop
Yes
more OF switch in switches.txt?
No
is switch port connected to another switch?
No
mark sw, port as target switch

Figure 3.8: Multicast process to determine packet destination

Unicast

The unicast method of the controller exhibits similar behaviour as that of unicast in legacy network. When a packet of non-multicast or broadcast destination arrives, the controller determines the destination OF switch to either queue the packet or install flow table entries. Based on the destination address, the target OF switch can vary. Unicast decision-making process is illustrated in below Figure 3.9.

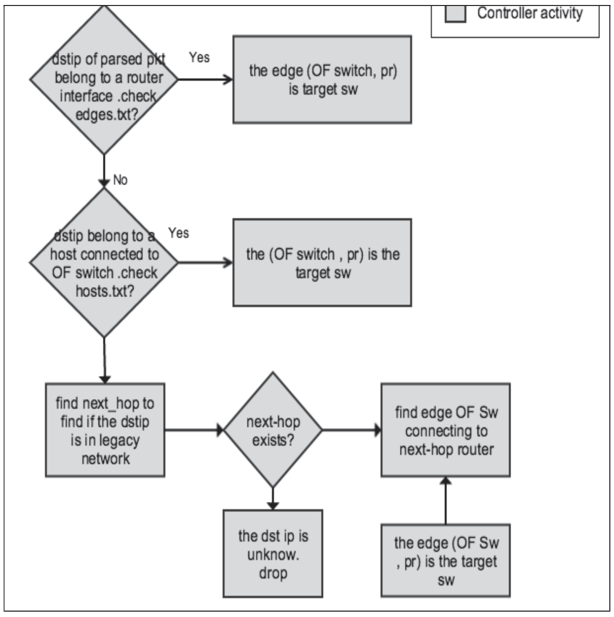

Controller activity
dstip of parsed pkt belong to a router interface .check edges.txt?
Yes
the edge (OF switch, pr) is target sw
No
dstip belong to a host connected to OF switch .check hosts.txt?
Yes
the (OF switch , pr) is the target sw
find next_hop to find if the dstip is in legacy network
next-hop exists?
find edge OF Sw connecting to next-hop router
the dst ip is unknow. drop
the edge (OF Sw , pr) is the target sw

Figure 3.9: Unicast process to determine destination

Updating packet payload header information

Information exchange between any two network nodes is carried as OF payload. Thus this payload can be extracted and processed. SYMPHONY controller manipulates legacy packet headers to suit the prevalent network conditions. This feature is immensely useful while maintaining communication between SDN and legacy network. For instance, a flow can be added to OF switch to modify the OF packet payload to choose a different boundary node over the pre-configured default gateway or choose an optimal egress node in presence of multiple default gateways. Manipulation of packet headers at the OF data plane creates a secondary data plane for the edge routers. This discussion will be presented in more detail Chapter4 where experimental topologies and corresponding use cases will be discussed.

handle_PortStatus

PortStatus messages indicate a change in the port status of a given port of a switch. A new connection established to the switch or an existing connection might be down. The controller updates its topology table information for the port as notified by the PortStatus message.

3.3.2 Legacy Routing Server

The Legacy Routing Server (LRS) is a Linux container running Quagga routing engine. LRS connects to the host computer with both generic ethernet bridge and an OVS switch. The advantage of this setup that the OVS switch can be controlled using the SDN controller. To be precise, interface eth0 on LRS communicates with the host machine via a Linux bridge, whereas interface eth1 of the LRS connects to the OVS, which is controlled by the SDN controller. Figure 3.10 shows the setup. LRS forms OSPF neighbour relation with edge routers via the interface eth1. By establishing OSPF neighbouring relationships with legacy network, LRS maintains a logical view of the legacy network. The controller consults LRS to choose the next best hop for a remote destination host residing in the legacy domain. More the number of edge routers LRS connects to, more precise is the path computed. Figure 3.11 illustrates LRS connectivity to the edge routers respectively.

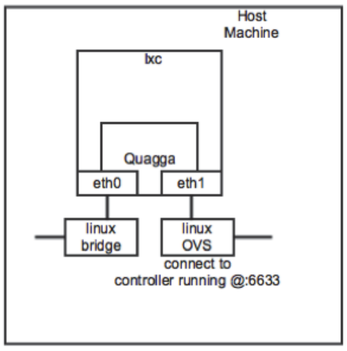


Figure 3.10: LRS connectivity with host

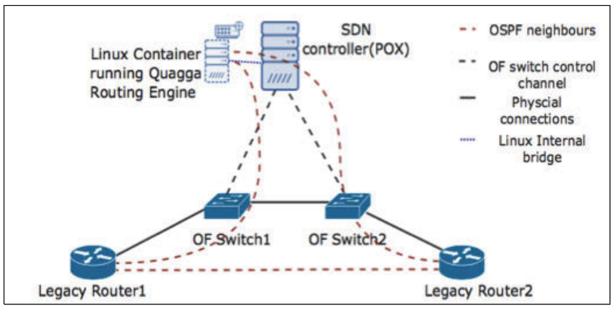


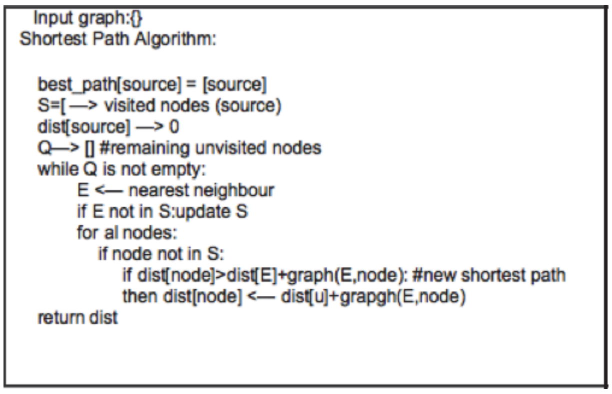
Figure 3.11: LRS connectivity with egress routers

3.3.3 Path-finder

As the name suggests the Path-finder module assists the controller to find optimal paths within the OF domain by running Dijikstra's Shortest Path First (SPF)[29] algorithm on switches.txt as input. Switches.txt is a network graph built with the aid of the LLDP module. The Path-finder component returns a set of OF switches to install flows between source and destination nodes. As of now, we have implemented SPF on equal cost paths. Dijikstra's algorithm is presented in the Figure 3.12

3.3.4 Next-hop

In our proposed scheme, LRS learns the remote legacy networks via one or more edge routers. When multiple edge routers report a single remote network, the LRS chooses the routing update from the edge router, which has the least cost. The functionality of the Next-hop module is to



```
 Input graph:{}
Shortest Path Algorithm:

  best_path[source] = [source]
  S=[ —> visited nodes (source)
  dist[source] —> 0
  Q—> [] #remaining unvisited nodes
  while Q is not empty:
        E <— nearest neighbour
        if E not in S:update S
        for al nodes:
           if node not in S:
              if dist[node]>dist[E]+graph(E,node): #new shortest path
              then dist[node] <— dist[u]+grapgh(E,node)
  return dist
```

Figure 3.12: Dijikstra's algorithm

query the routing table of LRS for any possible routes to the remote networks. Upon a match, the corresponding edge router is returned as the best next hop to reach a remote destination. The Packet-Forwarder module then nominates the corresponding edge OF switch as mapped in Edges.txt connected to the edge router as target (or destination) OF switch, followed by the compiling of a set of intermediate switches between the source and destination OF switches to forward the packets. The controller then installs appropriate flow table entries on the determined OF Switches leading to the destination node.

3.3.5 LLDP module for OF data plane discovery

LLDP module frames Link Local Discovery Protocol [11] packets. It receives the LLDP packets and builds the OF topology graph. The final graph is typically a python dictionary of the format as shown in Figure 3.13

```
{sw1:[[p1,(sw2,p2),. . . ],. . . ],sw2:[[p2,(sw1,p1),. . . ],. . . ],. . . }
```

Figure 3.13: Sample switches.txt

3.3.6 Policy Transformation

The Policy module of the controller inserts, deletes, reads and writes ACL like policy within the Policy.csv file of the controller. The inserted policy is by default in the forwarding direction of the flow between source and destination IP address (srcip, dstip). A second policy can be configured for return traffic with updated source and destination IP addresses for symmetrical application of the policy. In presence of multiple policies, the last configured policy is chosen as latest policy to be transformed. Each such policy is transformed into an OF switch flow entry and controller configures the flow on OF Switch with dpid = dpid and port = port. The action field allows the controller to configure a 'permit 'flow or a 'drop 'flow. A policy can be either static or dynamic type. While static policies are configured to perform policy-based routing, a dynamic policy is computed by the controller based on current network conditions. A sample Policy.csv is listed in Table 3.1

Table 3.1: Sample Policy.csv

s.no	source	destination	protocol	action	dpid	port	type
...	...	...	...	...	...	...	...

3.3.7 Topology Discovery

The controller categorises any given network into three subgroups. Nodes that make OF network as OF nodes, those nodes which connect to devices in different domains as edge/boundary/egress nodes and finally nodes in remote networks. The controller employs LLDP to discover the OF nodes, OSPF protocol to discover edge/boundary nodes and the remote networks through the LRS. LRS in turn discovers the remote networks using OSPF. The hosts connected within the OF domain are discovered using the ARP packets. Though POX controller provides host discovery module, SYMPHONY discovers hosts on-demand without using the modules provided by POX controller. Similarly, SYMPHONY uses custom built LLDP discovery module. Below are the three components of topology discovery process.

Discovery of OF switches using LLDP

LLDP (Logical Link Discovery Protocol) [11] is a Layer2 discovery protocol. This work uses LLDP protocol to discover OF switches. It has to be noted that OF switches are typically uninformed of any such discovery process, hence do not run LLDP or discover the topology intrinsically. To have a complete view of the topology, it becomes the responsibility of the controller to trigger the topology discovery process. This is done with the help of the LLDP module discussed in the previous section. LLDP module builds and processes incoming LLDP packets and assists the controller to build an OF switch topology graph.LLDP packet structure is shown in figure 3.14.

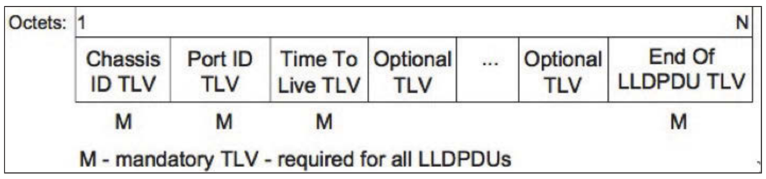


Figure 3.14: LLDP packet structure

When an OF switch (new connection) connects to the controller, the controller 1) builds LLDP packet 2) queues the LLDP packet on all possible connected ports on the switch 3) inserts a default flow on the switch to forward all incoming LLDP packets to the controller.

The process starts with the controller building the LLDP packet. The chassis ID TLV in the LLDP packet is updated with the dpid of newly connected OF switch and port ID TLV as any operational port on the switch. This packet is queued on all the connected ports (with updated port ID TLV)using the ofp_packet_out message discussed in previous chapter. As the OF switch queues and forwards the packets on all its ports, the connected OF switches forward the packet to the controller using ofp_packet_in message due to the default flow to forward all incoming LLDP packets to the controller. Based on the incoming ofp_packet_in message's dpid and port attributes and the chassis ID and port ID TLV within the packet payload, the controller discovers the OF data plane network and builds a graph Switches.txt in process to store all the information. The process of OF network discovery using LLDP is illustrated in Figure 3.15

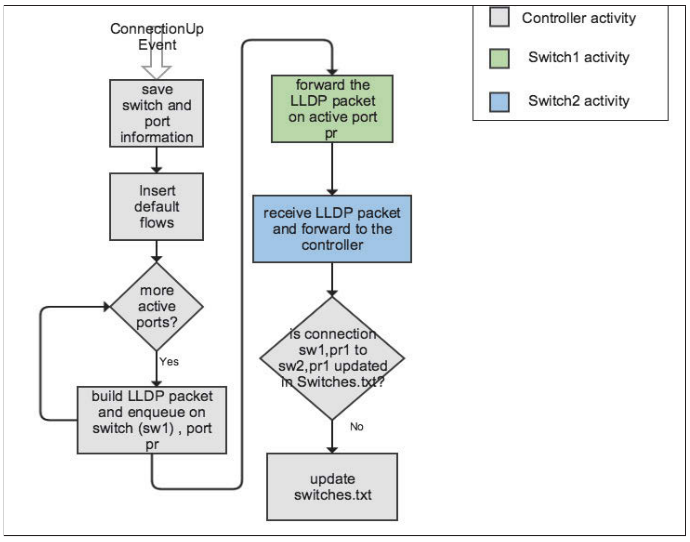

ConnectionUp Event
save switch and port information
Insert default flows
more active ports?
Yes
build LLDP packet and enqueue on switch (sw1) , port pr
forward the LLDP packet on active port pr
receive LLDP packet and forward to the controller
is connection sw1,pr1 to sw2,pr1 updated in Switches.txt?
No
update switches.txt
Controller activity
Switch1 activity
Switch2 activity

Figure 3.15: OF node discovery process

Discovery of Boundary devices

A connection between a legacy router and an OF switch is referred as boundary link between the legacy and OF domain. The connected devices (router, OF Switch) are referred as border/edge devices. The border legacy routers act as egress/edge nodes to the legacy networks and LRS forms neighbour relation with these routers. To discover boundary devices (including routers and OF switches) a default flow is added to the OF switches while handling ConnectionUp event. The flow instructs the OF device to match incoming packet payload protocol(protocol number: 89) and to forward all the matching packets (OSPF) to the controller.The controller uses the event.dpid and event.port information from the incoming OF packet and srcIP information from the packet payload to build Edges.txt. Since LRS is connected to the controller via OVS switch, the controller discovers LRS as an edge router. The Edges.txt thus built can be used to handle other OSPF routing update packets which are unicast/multicast packet and can be enqueued to the edge OF switches without being forwarded/enqueued to all the OF switches thus reducing overall traffic in the network. Thus existence of Edges.txt reduces the route computation time. Boundary nodes discovery process is illustrated in Figure 3.16

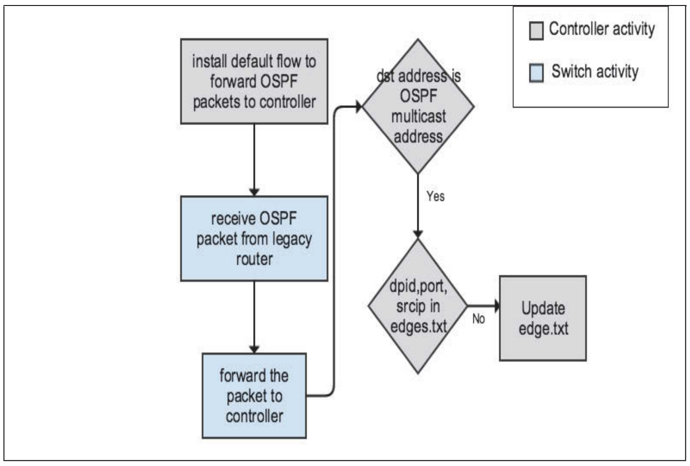

Controller activity
Switch activity
install default flow to forward OSPF packets to controller
receive OSPF packet from legacy router
forward the packet to controller
dst address is OSPF multicast address
Yes
dpid,port, srcip in edges.txt
No
Update edge.txt

Figure 3.16: Boundary nodes discovery process

The outcome of this discovery is an Edges.txt that stores the information about the border nodes in format shown in Figure 3.17.

```
{sw1:[( p1,rip1),. . . ], sw2:[(p2,rip2),. . . ],. . . }
```

Figure 3.17: Sample Edges.txt

Discovery of Hosts

As in case of any legacy networks, ARP is used to discover hosts in OF network. Typically, all packets with unknown actions are forwarded to the controller. The controller then conducts a source-based learning of all the hosts connected to the OF network. Upon receiving ARP broadcast message forwarded by an OF switch, the controller updates Hosts.txt with the information from the incoming OF packet's event.dpid and event.port and payload field srcIP. Figure 3.18 shows the host discovery process. Sample Hosts.txt is listed in Figure 3.19:

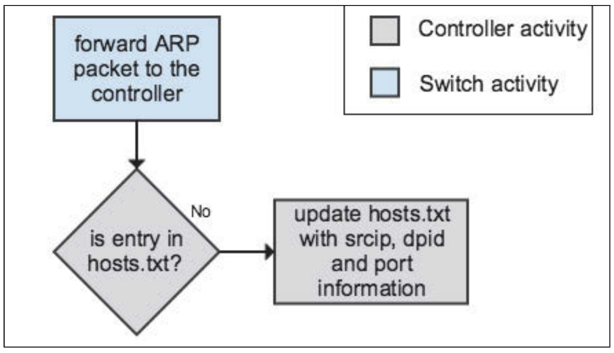

forward ARP packet to the controller
Controller activity
Switch activity
is entry in hosts.txt?
No
update hosts.txt with srcip, dpid and port information

Figure 3.18: Hosts discovery process

```
{ip1: (sw1,pr1),ip2: (sw2,pr2),...}
```

Figure 3.19: Sample Hosts.txt

By applying the topology discovery mechanism describes above to the topology shown in 3.20, the controller discovers {O1, O2, O3, O4 and O5} via LLDP, the boundary/edge routers R1 and R2 and links {(R1, O1), (R2, O2)} through OSPF and the remote networks (grey nodes) via OSPF. The end result of such a discovery process is three separate graphs in the form of python

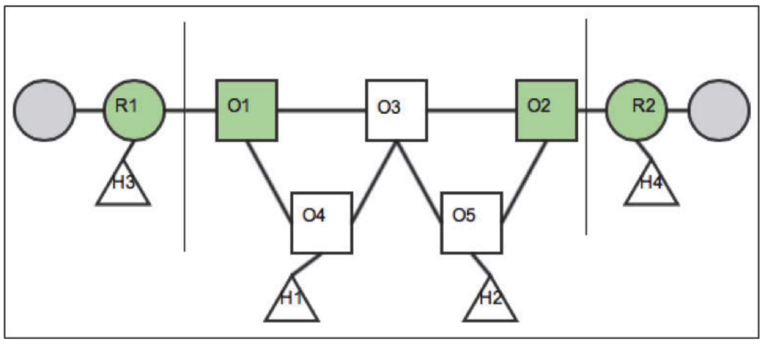


Figure 3.20: Identification of nodes in Hybrid SDN

dictionaries which can be stored as individual files. Python dictionaries are hash tables that store information in form of (key:value) pairs and offer efficient lookup. Figure 3.21 lists the simulation

```
R1 R1-eth0:sw1-eth1 R1-eth1:h1-eth1
R2 R2-eth0:sw2-eth1 R2-eth1:h2-eth1
h1 h1-eth1:R1-eth1
h2 h2-eth1:R2-eth1
h3 h3-eth1:sw4-eth3
h4 h4-eth1:sw5-eth3
sw1 lo: sw1-eth1:R1-eth0 sw1-eth2:sw3-eth2 sw1-eth3:sw4-eth1
sw2 lo: sw2-eth1:R2-eth0 sw2-eth2:sw3-eth1 sw2-eth3:sw5-eth2
sw3 lo: sw3-eth1:sw2-eth2 sw3-eth2:sw1-eth2 sw3-eth3:sw4-eth2 sw3-eth4:sw5-
eth1
sw4 lo: sw4-eth1:sw1-eth3 sw4-eth2:sw3-eth3 sw4-eth3:h3-eth1
sw5 lo: sw5-eth1:sw3-eth4 sw5-eth2:sw2-eth3 sw5-eth3:h4-eth1
```

Figure 3.21: Emulated network topology

Switches.txt built using LLDP for the topology in 3.21 is listed in Figure 3.22

```
{1: [[3, (4, 1)], [2, (3, 2)]], 2: [[2, (3, 1)], [3, (5, 2)]], 3: [[1, (2, 2)], [2, (1, 2)], [3,
(4, 2)], [4, (5, 1)]], 4: [[1, (1, 3)], [2, (3, 3)]], 5: [[2, (2, 3)], [1, (3, 4)]]}
```

Figure 3.22: Switches.txt to store OpenFlow topology

Edges.txt is listed in the Figure 3.23 :

```
{1: [(1, IPAddr('11.11.11.33'))], 2: [(1, IPAddr('11.11.11.44'))],
169955848697678: [(1, IPAddr('11.11.11.11'))]}
```

Figure 3.23: Edges.txt to store boundary connections and nodes

OPERATION

In this thesis, the term 'destination OF'indicates the last OF switch in the path towards the final destination. This can either be an OF switch connected to the final destination or the edge switch. A destination OF is also referred to as target OF switch. The OF switches that constitute the path to the target OF switch are referred to as intermediate OF switches. As mentioned earlier, SYMPHONY handles PacketIn, ConnectionUp and PortStatus OF messages only.

The controller processes an incoming control plane packet by verifying the packet payload header fields and by determining the course of action for handling the packet. If incoming control packet is an LLDP packet, the controller builds the Switches.txt as described in earlier sections. If the incoming packet's payload is a routing control multicast packet (OSPF hello packets), the controller determines edge OF switches and OVS connecting to LRS as target OF switches to queue the packet. Thus, edge routers connected to an OF switch form neighbour relationships. LRS also receives OSPF updates from the legacy L3 network and the Quagga routing engine running on LRS listens to OSPF routing updates (on the interface eth1 of LRS) forms OSPF adjacencies with all the edge L3 routers and builds a legacy topology table. In instances where the packet is LSA update destined to a particular destination, the controller evokes the unicast component, resulting in a path discovery process between the source and destination, followed by proactive installation of flow table entries on all the intermediate OF switches. Finally, the Edges.txt file stores details of edge OF switches and the corresponding connected edge L3 routers. The above functionality can be juxtaposed to OSPF neighbouring relationships establishment over a LAN setup. For other L2 broadcast packets such as ARP, all the switches and access ports of them connecting to hosts are chosen as target switches and target ports to forward the packet.

The following section describes the functionality when the incoming packet is a data packet i.e. a transport or application layer packet. The process triggers off by validating if the incoming packet matches any of the policies configured in the Policy.txt. Incase of a positive match and the port as in match is operational; the packet header details are updated, and the packet is forwarded based on the policy configured. In traditional routers the control plane of sender device is uninformed of the port status of adjacent device, thus might forward packets onto a link, which might potentially be down. This can be addressed using a centralised control as in a Hybrid SDN with the control plane working in coherence with the data plane of various devices and building the paths centrally along the active ports. Hence an alternative path can be computed.

Next, the packet is validated for L4 headers. L4 transport protocols are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). In TCP header, TCP ECN flag is a field validated by SYMPHONY. Explicit Congestion Notification(ECN) [16] is a feedback mechanism used by the hosts to inform the sender about congestion in the network. OpenFlow v1.0 does not support ECN. Hence SYMPHONY uses the ECN notification information in the TCP header of the OF packet payload. When a packet with ECN bit enabled is received by a data plane, the centralised controller computes an alternate path in form of a dynamic policy and reroutes the packet based on the policy. Chapter 4 details the uses cases, which demonstrate the functionality of the controller as described here. This is again in contrast with the behaviour of the legacy devices, where ECN bit is used to inform the sender host to reduce the TCP window.

Based on the packet destination, the target OF switch and set of intermediate OF switches are chosen. With the aid of the Hosts.txt, it can be determined if the host is connected to an OF switch. If the host is configured within the OF domain, the OF switch connected to the host is nominated as target switch, and a path is computed. In case the destination node is not provisioned within OF domain, the controller performs a look-up on Edges.txt to determine if the packet is destined to an edge router. If the packet is destined to an edge router, the controller determines the connecting OF switch as target switch. The Edges.txt here acts as a cache for the SDN controller to query before referring to the LRS module. This feature is better for time bound packet processing activity such as the establishment of OSPF neighbouring relationships. If the destination node is neither connected to an OF switch nor to an edge router then, POX refers to the LRS module to determine if the address is located in the legacy network, in which case a suitable next hop edge router and the corresponding connecting edge OF switch (target OF switch) are determined. The preferred path is computed between the source OF switch, and the target OF switch and flows are proactively installed on all the intermediate OF switches.

It has to be noted that the controller at no point broadcasts any packets. Instead, broadcast packets are forwarded only to the intended set of devices. For example, ARP requests are not forwarded onto the switch interfaces connecting other switches, but only on those interfaces connecting to end hosts. This simple validation avoids L2 looping and broadcast storms in the network. Moreover, in case the packet is destined to the legacy network, the controller updates the packet with the appropriate edge router details, thus ensuring that a best and valid path is chosen. Upon link failures, the controller determines the best alternative path to route the packets. SYMPHONY's packet handling mechanism is illustrated in the Figure 3.24

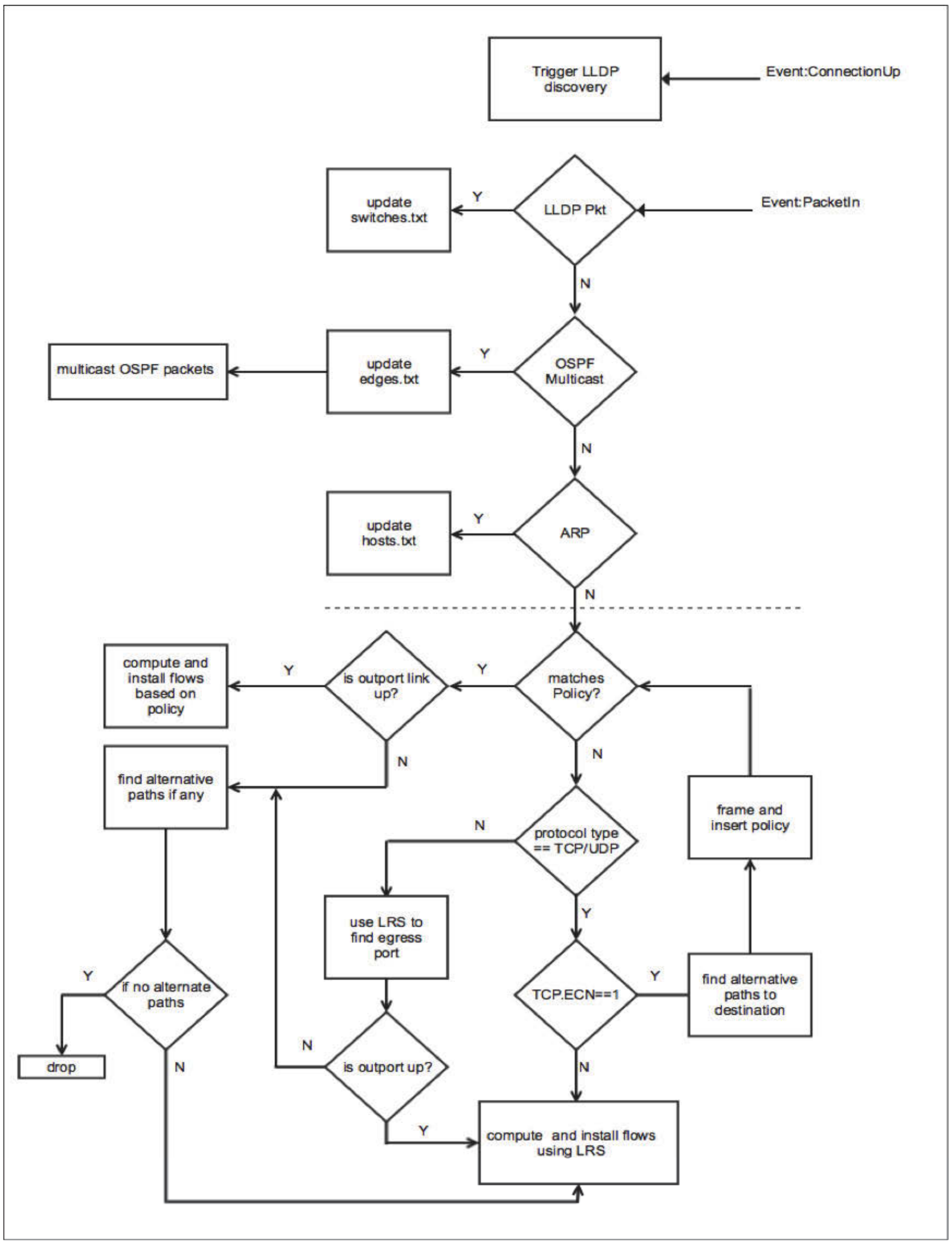


Figure 3.24: FlowControl of SYMPHONY Controller for Hybrid SDN

3.4 Conclusion

This chapter presented the architecture of SYMPHONY. SYMPHONY is a Hybrid SDN controller built with an aim to establish inter domain communication between legacy and Software Defined Network. To enable such an communication, SYMPHONY uses a Legacy Route Server(LRS) to learn and store legacy network information. Using LRS, SYMPHONY controller establishes end-to-end path communication. Apart from establishing communication, SYMPHONY can also be used to enhance legacy networks. In this architecture, OF data plane is used as secondary data plane for legacy network and related policy can be applied centrally on OF data plane and not isolated to individual routers, thus providing end-to-end policy application visibility. SYMPHONY can also be used to load balance traffic to reduce traffic congestion. To do so, SYMPHONY uses explicit congestion notification. The following chapter describes different use cases to validate our scheme where SYMPHONY controller can be used to manage Hybrid SDN.

Chapter 4

Implementation and Use Cases

This chapter presents the usefulness of SYMPHONY through three use cases. The uses cases have been simulated using Mininet[23] SDN emulator and miniNExT[19] extension of Mininet to emulate routers. The environment setup is briefly listed in Appendix A. Through the first use case, we demonstrate the primary functionality of intra and inter domain communication. The first use case details on three different communication scenarios between any hosts connected to OF switches, between hosts connected to the legacy networks over an OF island and between a host connected to the legacy network and a host connected to OF switch respectively. The second use case describes a new centralised policy application mechanism on legacy networks using OF switches. This is done by mapping legacy devices to an OF data plane. The third and final use case builds upon the policy use case(use case-II) and describes how network congestion can be addressed by loading traffic load balancing. We employ ECN feedback mechanism provided by TCP for realising the third use case.

4.1 Use Case I: Establish communication

The first use case illustrates data exchange between any given pair of source and destination hosts. For data exchange to be possible in a Hybrid SDN network, it is required that the SDN controller is aware of the complete topology. One way for the LRS to learn the legacy routing information (legacy control traffic) is by exchanging OSPF routing updates with a subset of routers in the legacy network. By forming OSPF neighbour relation with boundary routers, LRS learns complete network topology in legacy network and compute best next hop for packets destined for remote legacy networks.

The use case has been illustrated using topology in Figure 4.1

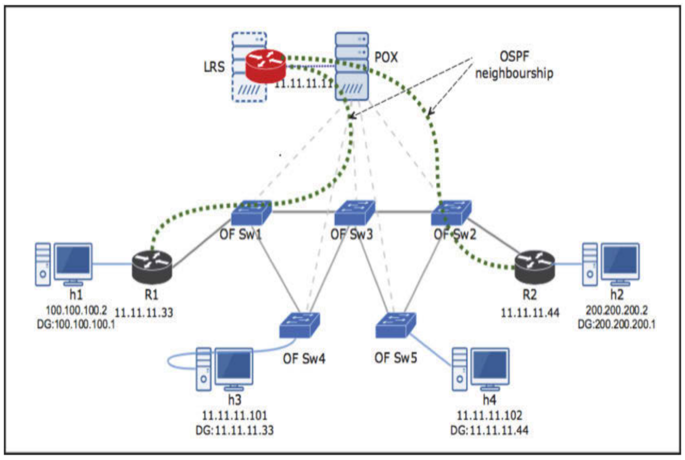


Figure 4.1: OSPF Neighbour relation between LRS and edge routers

Table 4.1 represents OSPF neighbour table on LRS. Here, LRS has two neighbours - boundary routers R1 and R2.

Table 4.1: OSPF Neighbour table of LRS

Neighbor ID	Pri	State	Dead Time	Address	Interface	RXmtL	RqstL	DBsmL
3.3.3.3	1	Full/DROther	36.279s	11.11.11.33	eth1:11.11.11.12	0	0	0
4.4.4.4	1	Full/Backup	36.281s	11.11.11.44	eth1:11.11.11.12	0	0	0

Upon successful LSA exchange between boundary and LRS, legacy networks are learnt by the LRS. Figure shows the routing table:

```
RS2# sh ip route

Codes: K - kernel route, C - connected, S - static, R - RIP,

O - OSPF, I - IS-IS, B - BGP, A - Babel,

>- selected route, * - FIB route


K>* 0.0.0.0/0 via 10.0.3.1, eth0

O>* 3.3.3.3/32 [110/20] via 11.11.11.33, eth1, 00:04:52

O>* 4.4.4.4/32 [110/20] via 11.11.11.44, eth1, 00:04:52

C>* 10.0.3.0/24 is directly connected, eth0

O 11.11.11.0/24 [110/10] is directly connected, eth1, 00:06:16

C>* 11.11.11.0/24 is directly connected, eth1

O>* 100.100.100.0/24 [110/20] via 11.11.11.33, eth1, 00:04:52

C>* 127.0.0.0/8 is directly connected, lo

O>* 200.200.200.0/24 [110/20] via 11.11.11.44, eth1, 00:04:52
```

We test three such different communication scenarios that arise between OF and legacy domain. We test reachability using ICMP ping.

Communication between two OF nodes

This scenario is quite common and a prerequisite for conducting further route reachability tests. SYMPHONY employs SPF algorithm as described in earlier chapters to complete the shortest path within the OF domain. Under normal circumstances, when all the links are functional, the controller uses the Path-finder module to compute a complete path between a given source and destination within the OF island. In topology illustrated in Figure 4.2 ,for h3@11.11.11.101 to communicate with h4@11.11.11.102, the controller computes a set of intermediate switches (OFSw4, OFSw3, OFSw5) as the path for packet traversal. In case the link (OFSw3, OFSw5) goes down, an alternate path is constructed along OFSw4, OFSw3, OFSw2, and OFSw5. Flows for $h3 \leftrightarrow h4$ under normal circumstances while link (OFSW3,OFSw5) is operational traverse the path $OFSw4 \rightarrow OFSw3 \rightarrow OFSw5$. The flow table entries are listed in Appendix B Figure B.1 When link (OFSW3,OFSw5) is down, the new path computed is along $OFSw4 \rightarrow OFSw3 \rightarrow OFSw2 \rightarrow OFSw5$ as shown in Figure 4.2.

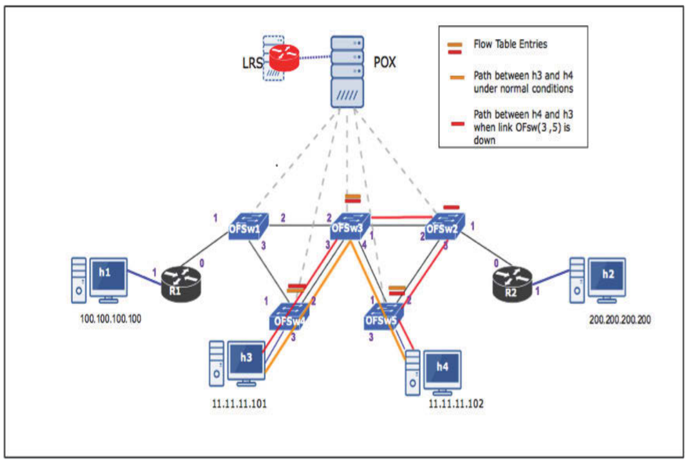


Figure 4.2: Communication between two hosts in a OF domain

Flow table entries can be found in Appendix B Figure B.2

Communication between two legacy nodes over OF switches

To enable communication between legacy hosts over the OF network, the controller must make a decision about the location of the destination node. The controller does so by querying LRS. In topology illustrated in Figure 4.3, OFsw1 switch forwards the first IP packet received from h1 for the flow $h1 \leftrightarrow h2$ to the controller with reason as unknown action. Since the packet is destined to a host in the legacy network, the controller queries the LRS to find the next best hop router (border router) to forward the packet. In this case, the next best hop router is R2. Knowing the next best border router to forward packets to, the controller determines the connecting edge OF Switch which in this case—OFsw2 as target OF Switch. The controller which is now aware of the source OF Switch (OFSw1) and the destination OF Switch (OFSw2) computes the best path between the source and destination, which in this case is $OFsw1 \rightarrow OFsw3 \rightarrow OFsw2$. The controller fully frames the flows and proactively install the flows in all the switches, which constitute the path. Thus, instead of all the intermediate OF switches (in this case OFsw3) forwarding packets to the controller for reactive forwarding, the controller pre-computes flows and installs the flows in a proactive fashion.The flow table entries for communication between h1 and h2 along the path $OFSw1 \rightarrow OFSw3 \rightarrow OFSw2$ can be found in Appendix B Figure B.3.

Incase of link (OFsw1,OFsw3) failure, alternate path $OFSw1 \rightarrow OFSw4 \rightarrow OFSw3 \rightarrow OFSw2$ is now choosen as shown in Figure 4.3. Corresponding flow table entries are listed in Appendix B Figure B.4

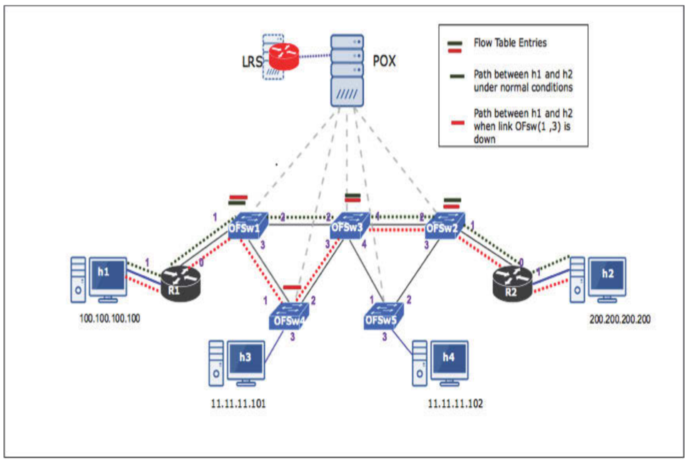


Figure 4.3: Alternate paths incase of link failures

Communication between a legacy node and an OF node

The third scenario details the communication between a legacy and OpenFlow node. Choosing the right edge nodes is one of the challenges to be addressed while enabling communication between legacy and OF node. This can be further broken down into smaller functions, which are listed below:

Function1: Forwarding the packet to the configured default gateway (boundary router)
Under normal circumstances when all of the links are functional, the controller (and LRS) must compute a complete path between a given source and destination OF switches. Consider a com-

munication instance between host h3 and h1 for topology illustrated in Figure 4.4. The default gateway of h3 is R1. As in other scenarios, OFSw4 forwards the first packet for the flow $h3 \rightarrow h1$ to the controller with reason as an unknown destination. The controller (in this instance, the LRS) computes the next best hop to reach h1 as R1 and concludes that OFsw1 is a suitable OF egress node. Thus, a path between OFSw4 and OFSw1 is computed, and flows are installed. In this instance, since OFSw1 and OFSw4 are directly connected, the path is $OFSw4 \rightarrow OFSw1$. Resulting flow table entries can be found in Appendix B in Figure B.5

Function2: Forwarding the packet to a different boundary router and not the default gateway

As discussed earlier, for data exchange between hosts in different domains, there can be multiple border routers. Thus, a suitable border router is determined based on the next hop information provided by the LRS. The following scenario illustrates this. For a packet sourced at h3 and destined at h2, boundary router R1 is the default gateway for host h3. But R2 has the best route to h2 and thus LRS determines router R2 as the best exit node and the controller, computes OFSw2 as the best OF node towards R2. The packet headers are updated with the new next hop MAC details and are thus forwarded to router R2 instead of the default gateway of h3 that is router R1. The controller installs appropriate flows on all the OF switches in the path ($OFSw4 \rightarrow OFSw3 \rightarrow OFSw2$) to route the packet to router R2. In case the link (OFSw3, OFSw2) goes down, the alternate path is computed via $OFSw4 \rightarrow OFSw3 \rightarrow OFSw5 \rightarrow OFSw2$ and router R2. This scenario is shown in Figure 4.4. As it can be noted, the more boundary nodes LRS is connected to, better routes within the legacy networks are learnt by the LRS. Flow table entries for the flow can be found in Appendix B Figure B.6.

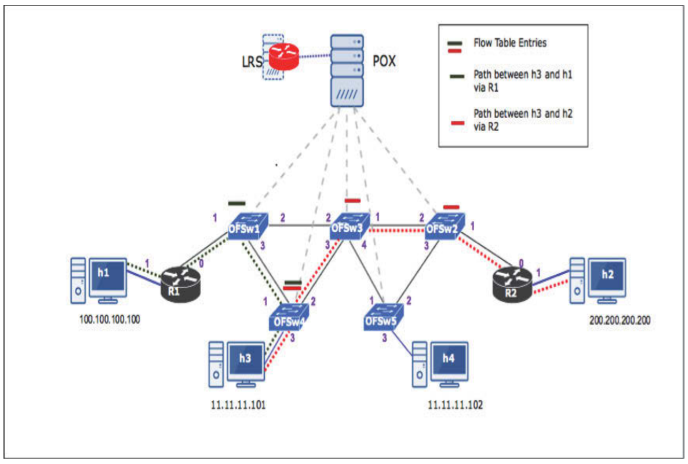


Figure 4.4: Choice of different edge routers for different remote destinations

4.2 Use Case II:Centralised policy application

Using two fundamental principles that OF switches are configurable data planes and policies in routers are applied on data planes; the second use case has been developed. The setup involves mapping LAN interfaces of a router to an OF data plane. This is illustrated in Figure 4.6. Ports 1,3 of OF Switch connect to routers R1, R2 and ports 2,4 connect R1, R3. Port 5 connects R1 to LRS on a dedicated subnet. This allows LRS to form neighbour relation with a router on a dedicated subnet and learn legacy routes. To ensure that data traffic is not routed via the LRS, maximum OSPF administrative distance has been assigned for the link connecting to LRS. Flow table entries of the OF device are centrally configured by the controller that is capable of manipulating and rerouting packets independent of packet routing decision made by legacy routers. Thus, we were able to create a centrally configurable data plane for the legacy routers by 'tying'the OpenFlow switches to the routers as shown in Figure 4.6. The corresponding legacy topology is shown in Figure 4.5. Thus, a packet arriving on interface1 of the OF Switch, need not have to be routed on to interface3 to R2. Instead, the packet header can be updated and be routed towards R3. Any centralised policy (either statically configured or dynamically calculated) can be applied on the OF Switch to enforce traffic engineering.

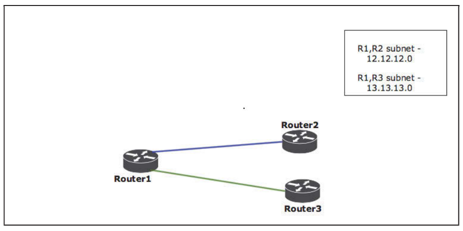


Figure 4.5: Legacy topology

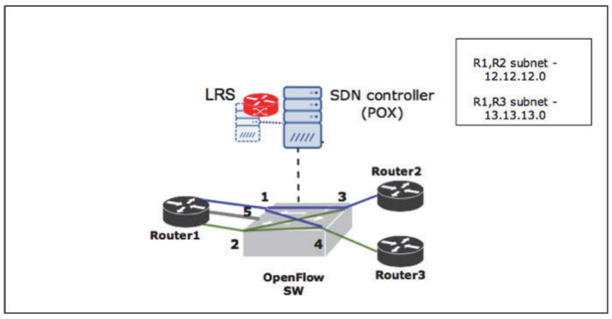


Figure 4.6: Connectivity setup for policy application

This use case aims to demonstrate policy based routing by doubling an OF Switch to function as a data plane for a router. The use case also shows how SYMPHONY controller and Hybrid SDN can benefit legacy networks. For the purpose of prototyping, this work uses a static central policy file(policy.csv), which can be manually or dynamically updated. Packet handling mechanism matches all incoming packets with the policies configured in the policy.csv. Upon a successful match, the controller transforms the policy into a flow table entry and configures the flow on the OF Switch acting as router data plane, irrespective of the policy configured on the routers. To validate this scenario, we have built a simple network as shown in Figure 5.9 below. The network has been set up such that the traffic between h1 and h2 chooses path $R1 \rightarrow R3 \rightarrow R4$. Firstly, with no policy configured, OFSw1 forwards the packet onto the default next hop, i.e., R3. If a central policy has been configured to reroute the traffic via R2 as shown in Table 4.2, the controller installs a flow on the OFSw1 to forward the packet via the interface connecting to R2 instead. This behaviour is similar to the application of routing policies in traditional networks. SYMPHONY goes beyond this traditional set up by falling back onto the default route in instances when the link (R1, R2) fails (specifically R2 port connecting to R1 is down). In a typical legacy network, the policy is enforced irrespective of the port and neighbour status unless other mechanisms such as tracking are enabled. With SYMPHONY, the policy is applied only when the neighbour port/link are up, thus ensuring that routers do not forward the packets into a void. Thus, in spite of policy in place, the packets are routed via OFSw1 to R3. The immediate advantage of such centralised policy application lies in the avoidance of having to configure policies on multiple independent devices, and more importantly also avoiding traffic black hole in case of links with policies configured on fail.

Table 4.2: Policy Table

s.no	source	destination	protocol	action	dpid	port	type
1	60.60.60.100	40.40.40.100	1	permit	1	4	static
2	40.40.40.100	60.60.60.100	1	permit	2	4	static

Figure 4.7 illustrates the flow of packets upon application of policy in a hybrid network.

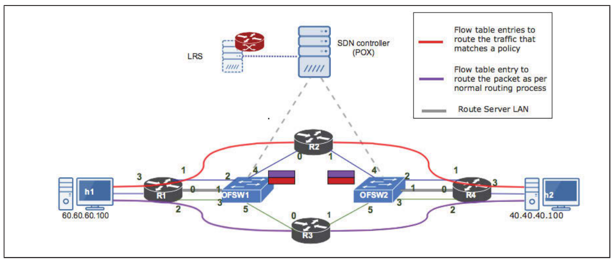

LRS
SDN controller (POX)
Flow table entries to route the traffic that matches a policy
Flow table entry to route the packet as per normal routing process
Route Server LAN
R2
R1
OFSW1
OFSW2
R4
R3
h1
h2
60.60.60.100
40.40.40.100

Figure 4.7: Policy application in action

While flow table entries for flow between h1 and h2 with policy in place and all links operational can be found in Appendix B Figure B.7 and Figure B.8 are flow table entries during link failure between R1, R2.

4.3 Use Case III: Using the TCP-ECN bit to divert traffic dynamically onto other links.

Various congestion control mechanisms employed by networks include source based and gateway based schemes [6]. All methods use mechanisms to notify the sender about congestion using some feedback. Feedbacks can be either i) explicit or ii) implicit. With implicit feedback mechanism, the end hosts adjust the window size/throughput/sending rate based on external observations such as packet loss, large RTT and so on. In case of explicit feedback, the host adjusts its sending rate upon receiving a feedback notification from a congested router to slow down. Explicit Congestion Notification (ECN) [16] was proposed in RFC 3168 to allow end-to-end notification of network congestion and uses two bits (6 and 7) in the IP TOS field. One bit is set by the end host to indicate if its ECN capable, whereas the other bit is enabled by a congested transit router. ECN is a feedback notification, which results from a growing congestion in a network. Thus, ECN works in conjuncture with queue management mechanisms such as RED. Once ECN flag is set, it is not reset until the intended sending host receives the feedback. After receiving ECN notifications is that the host reduces the window size and starts sending packets at a reduced rate to alleviate congestion.

This use case demonstrates how traffic engineering can be performed using the ECN bits and TCP header fields. Ideally, in Figure 4.8 when a transit router's queues are full, the ECN flag is set in the IP TOS field and forwarded to the sender, in this case, h2. The TCP ACK packet with ECN flag on is sent by h2 to h1 as a feedback mechanism to inform h1 about the congestion in the network. As a response to this notification, h1 reduces the window size. In typical scenarios, there is no provision on the network devices to reroute the traffic on different paths and to avoid reduction of the TCP window size prematurely. We use this use case to demonstrate how OF data plane of the routers perform load balancing and reroute the traffic on lesser congested paths using lesser-congested nodes.

Again, for the purpose of showing the functionality of the prototype, instead of creating enormous traffic to trigger generation of ECN notifications, we use a packet generating software scapy [5]. We create a packet with ECN flag enabled. Upon receiving such TCP packet, the controller finds an alternate path. An important assumption made here is that there are alternative routes available for routing the packet. The controller then frames and inserts a dynamic policy in the policy file to route the packets. Thus, a congested link is treated as a failed link and packets are rerouted on alternate paths. In normal circumstances the traffic from h1 to h2 is routed via R3 as can be seen from the diagram depicted in Figure 4.8

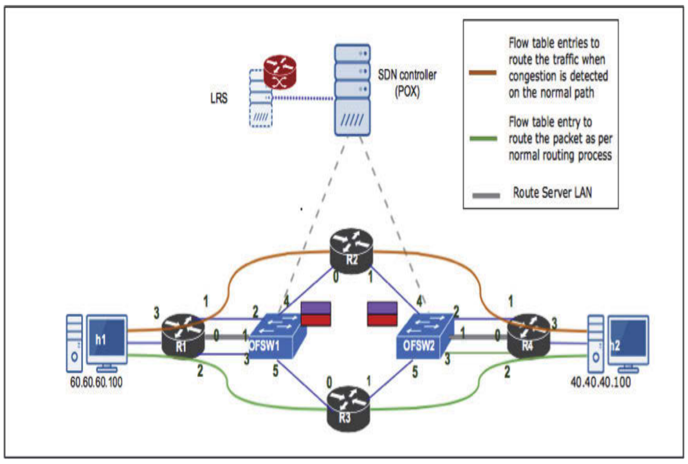
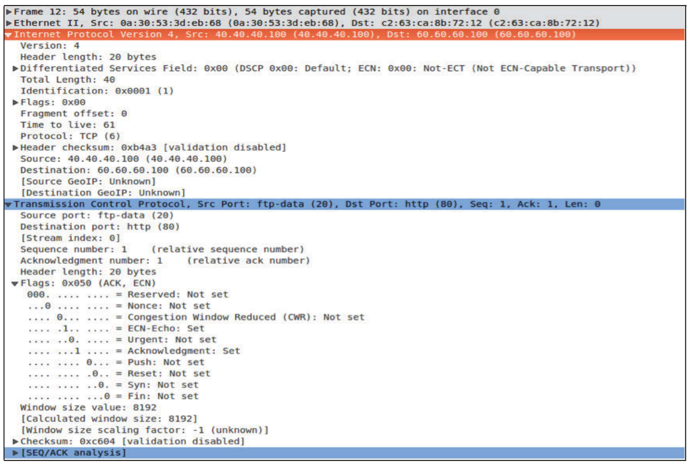

Flow table entries to route the traffic when congestion is detected on the normal path
Flow table entry to route the packet as per normal routing process
Route Server LAN
SDN controller (POX)
LRS
R1
R2
R3
R4
OFSW1
OFSW2
h1
h2
60.60.60.100
40.40.40.100

Figure 4.8: Effect of ECN

flow table entries for this communication can be found in Appendix B figure B.9 Using Scapy packet builder program , a TCP packet has been sent from to with ECN flag on indicating congestion. Packet generated from scapy as sniffed using wireshark is shown in Figure 4.9 below:



```
▶Frame 12: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
▶Ethernet II, Src: 0a:30:53:3d:eb:68 (0a:30:53:3d:eb:68), Dst: c2:63:ca:8b:72:12 (c2:63:ca:8b:72:12)
▼Internet Protocol Version 4, Src: 40.40.40.100 (40.40.40.100), Dst: 60.60.60.100 (60.60.60.100)
  Version: 4
  Header length: 20 bytes
 ▶Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
  Total Length: 40
  Identification: 0x0001 (1)
 ▶Flags: 0x00
  Fragment offset: 0
  Time to live: 61
  Protocol: TCP (6)
 ▶Header checksum: 0xb4a3 [validation disabled]
  Source: 40.40.40.100 (40.40.40.100)
  Destination: 60.60.60.100 (60.60.60.100)
  [Source GeoIP: Unknown]
  [Destination GeoIP: Unknown]
▼Transmission Control Protocol, Src Port: ftp-data (20), Dst Port: http (80), Seq: 1, Ack: 1, Len: 0
  Source port: ftp-data (20)
  Destination port: http (80)
  [Stream index: 0]
  Sequence number: 1    (relative sequence number)
  Acknowledgment number: 1    (relative ack number)
  Header length: 20 bytes
 ▼Flags: 0x050 (ACK, ECN)
   000. .... .... = Reserved: Not set
   ...0 .... .... = Nonce: Not set
   .... 0... .... = Congestion Window Reduced (CWR): Not set
   .... .1.. .... = ECN-Echo: Set
   .... ..0. .... = Urgent: Not set
   .... ...1 .... = Acknowledgment: Set
   .... .... 0... = Push: Not set
   .... .... .0.. = Reset: Not set
   .... .... ..0. = Syn: Not set
   .... .... ...0 = Fin: Not set
  Window size value: 8192
  [Calculated window size: 8192]
  [Window size scaling factor: -1 (unknown)]
 ▶Checksum: 0xc604 [validation disabled]
 ▶[SEQ/ACK analysis]
```

Figure 4.9: ECN enabled packet as captured in wireshark

Upon receiving the ECN packet, the controller finds alternate paths to route the packet and inserts a dynamic policy in the policy file to reroute the packets. Table 4.3 below shows the dynamic policy added by the controller to divert the traffic for a flow off the congested link to a lesser-congested link via R2.

Table 4.3: Dynamic policy inserted in policy table

s.no	source	destination	protocol	action	dpid	port	type
1	60.60.60.100	40.40.40.100	1	permit	1	4	dynamic

Since the router's OF data plane(OF Switch) can be programmed to divert the traffic to different path, the host does not receive any further ECN enabled packets as a result of reduced congestion, thus actually increasing the duration to reach the threshold set which in turn prompts the host to reduce the congestion window. When all the links get congested, the router can no longer perform load balancing; as whatever link is chosen, the receiver will eventually notify the sender and once the sender receives more of the ECN enabled packets/real packets, the threshold is reached, and the window is reset. The flow table entries for this flow can be found in Appendix B table B.10

4.4 Conclusion

In this chapter, this thesis presented three different use cases to demonstrate the use of our proposed Hybrid SDN controller. In the first use case, three basic but different communication scenarios have been discussed. The first scenario verifies communication between two hosts within OF domain, and the second scenario confirms communication between two hosts connected to legacy routers and the final scenario is on verifying communication between a host connected to OF domain and a host connected to legacy network. The second use case demonstrates how centralised policy can be applied in Hybrid SDN and how an OF data plane can function as a secondary data plane for legacy routers. The third use case demonstrates how traffic engineering can be achieved. In this use case, we compute dynamic policies based on the congestion feedback mechanism and divert traffic on other links. The next chapter discusses various tests that have been conducted and analyse the results.

Chapter 5

Testing and Results

This chapter details on the test cases devised to assess the proposed framework. We evaluate the OSPF neighbour relation establishment time, validate if the number of egress nodes affect the quality of routes learnt by LRS and verify the duration of convergence after policy application. We also make a proposition for testing some variation to TCP congestion window upon congestion feedback using Explicit Congestion Notification (ECN).

5.1 Test Case1: Convergence over OF domain

This test case verifies the time taken by the edge nodes to form OSPF neighbour relation over an OF island topology. Firstly, in traditional networks, L2 switches implement Spanning Tree Protocol (STP) to eliminate loops in a network. STP translates a looped network into a tree network by putting few ports in the forwarding state and others in blocking state. This not only eliminates loops but also creates longer paths between nodes. SYMPHONY uses LLDP to discover OF network and store the OF topology information in a topology database thus reducing the dependency on STP to maintain a loop-free network.

5.1.1 Spanning Tree Protocol

STP is implemented in L2 networks to draft an acyclic (specifically a tree) topology from a cyclic topology. STP triggers election of a Root Bridge followed by setting ports in one of the various states (blocking, listening, learning, forwarding, disabled). Only after listening and learning stages, the ports transitions to the forwarding state. Total cumulative time spent to perform each step of STP approximates to 45 seconds. Hence, variations of STP such as Rapid Spanning Tree Protocol (RSTP) have been implemented to address the slow convergence caused due to STP. The convergence time of STP also depends on the number of switches present in the network. More the number of bridges in a network more delay in convergence can be noticed. As per [10] forwarding delay of an L2 network can be calculated theoretically with the formula mentioned in below

= (End-to-end BPDU_propagation_delay(A) + Message_age_overestimate(B) + Maximum_frame_lifetime (C) + Maximum_transmission_halt_delay(D))/ 2

A brief description of the formula can be found in Appendix C. Based on the above formula, the time taken for STP convergence can be obtained as below Table 5.1

Table 5.1: STP Convergence time

No of switches	A	B	C	D	Total time
3	10	2	3.5	1	8
5	12	4	5.5	1	11
7	14	6	7.5	1	14
14	21	13	14.5	1	25
21	28	20	21.5	1	35
28	35	27	28.5	1	46

OSPF hello packets are exchanged only after STP convergence is complete. Hence, its safe to assume that time taken by legacy routers to fully converge on a L2 network is greater than the STP convergence time. The topologies used for testing are illustrated in Appendix C C.1. Below Table 5.2 compares the OSPF convergence time in legacy and in Hybrid SD networks.

Table 5.2: OSPF Convergence time

No of switches	Time over legacy L2	Time over OF
3	>8	13
5	>11	17
7	>14	13
14	>25	15
21	>35	14
28	>46	12

The column 'Time over OF 'column indicates the time taken by legacy routers to form FULL neighbour relationship over an OF island from the time the controller started. We timed the controller using the time() module in python to register accurate time, we also used debug functions on routers to determine the exact time of receipt of OSPF packets. 'Time over legacy L2'column indicates the time taken for the legacy routers to reach FULL state. As can be deduced, in case of an OF domain, there is not too much variation in the time measured but in case of legacy as the number of switch increases, the time taken for STP convergence increases thus affecting the overall time taken for completion of OSPF process. Hence, it can be inferred that routers, when connected over traditional L2 networks, have longer OSPF convergence time as compared to the routers connected over OF switches. Thus, SYMPHONY not only allows OSPF convergence over OF network but also compares better than convergence timings than the legacy network. Graph in Figure 5.1 illustrates the Table 5.2

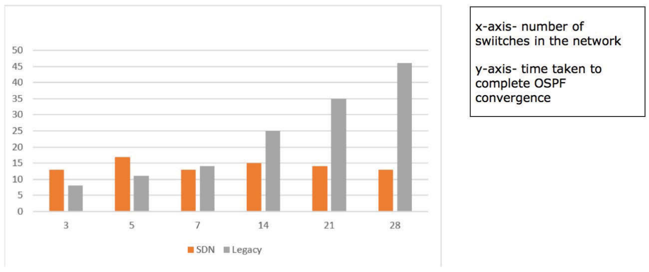


Figure 5.1: OSPF convergence Comparison graph

5.2 Test Case2: Effect of presence of multiple egress nodes on routes learnt by LRS

The test has been conducted to reflect that the number of egress routers connected to an SDN domain can impact the type of routes LRS learns. Single edge connectivity though allows inter-communication between traditional and SDN networks is ineffective. It can be both a bottleneck in the network and a single point of failure. Also, such connectivity does not fully expose the legacy network to allow LRS to determine best the end-to-end path. Consider the network in Figure 5.2, in the legacy plane the best next hop with least cost from the destination network (100.100.100.0/24) is R5 with a cost of 20, but a cumulative cost to source 140. The next best hop node is R2 with total cost 160. But since the OF domain is connected to a single egress/edge router (R1), LRS learns all the routes within the legacy domain via R1. This not only creates a bottleneck at R1 but also leads LRS to view the legacy domain as mapped by R1.

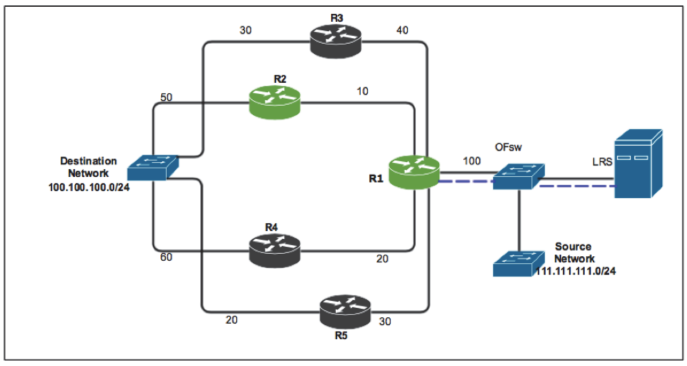
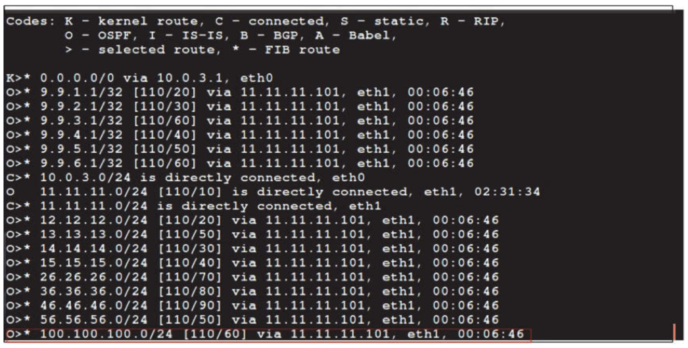

R3
30
40
R2
50
10
Destination Network 100.100.100.0/24
R1
100
OFsw
LRS
R4
60
20
Source Network 111.111.111.0/24
20
R5
30

Figure 5.2: Hybrid topology with a single edge router

The routing table of LRS as shown in Figure 5.3 :



```
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, A - Babel,
       > - selected route, * - FIB route

K>* 0.0.0.0/0 via 10.0.3.1, eth0
O>* 9.9.1.1/32 [110/20] via 11.11.11.101, eth1, 00:06:46
O>* 9.9.2.1/32 [110/30] via 11.11.11.101, eth1, 00:06:46
O>* 9.9.3.1/32 [110/60] via 11.11.11.101, eth1, 00:06:46
O>* 9.9.4.1/32 [110/40] via 11.11.11.101, eth1, 00:06:46
O>* 9.9.5.1/32 [110/50] via 11.11.11.101, eth1, 00:06:46
O>* 9.9.6.1/32 [110/60] via 11.11.11.101, eth1, 00:06:46
C>* 10.0.3.0/24 is directly connected, eth0
O   11.11.11.0/24 [110/10] is directly connected, eth1, 02:31:34
C>* 11.11.11.0/24 is directly connected, eth1
O>* 12.12.12.0/24 [110/20] via 11.11.11.101, eth1, 00:06:46
O>* 13.13.13.0/24 [110/50] via 11.11.11.101, eth1, 00:06:46
O>* 14.14.14.0/24 [110/30] via 11.11.11.101, eth1, 00:06:46
O>* 15.15.15.0/24 [110/40] via 11.11.11.101, eth1, 00:06:46
O>* 26.26.26.0/24 [110/70] via 11.11.11.101, eth1, 00:06:46
O>* 36.36.36.0/24 [110/80] via 11.11.11.101, eth1, 00:06:46
O>* 46.46.46.0/24 [110/90] via 11.11.11.101, eth1, 00:06:46
O>* 56.56.56.0/24 [110/50] via 11.11.11.101, eth1, 00:06:46
O>* 100.100.100.0/24 [110/60] via 11.11.11.101, eth1, 00:06:46
```

Figure 5.3: LRS routing table

As stated in earlier chapters, LRS is a Linux container running Quagga routing engine, which learns legacy destinations by establishing OSPF neighbour relation with edge routers. By utilising more routers as edge routers, LRS can be enabled to choose alternate paths, in this case, R5 or R2. Upon OSPF neighbour relation establishment between LRS and R5, LRS can choose alternate path between the routes learned via R1 and R5 for legacy destinations and update the routing table. This arrangement not only enables LRS to choose better paths for traffic originating from OF domain to legacy plane but also automatically load balances the traffic on various paths for different destinations. Such a hybrid topology with multiple egress nodes is shown in Figure 5.4.

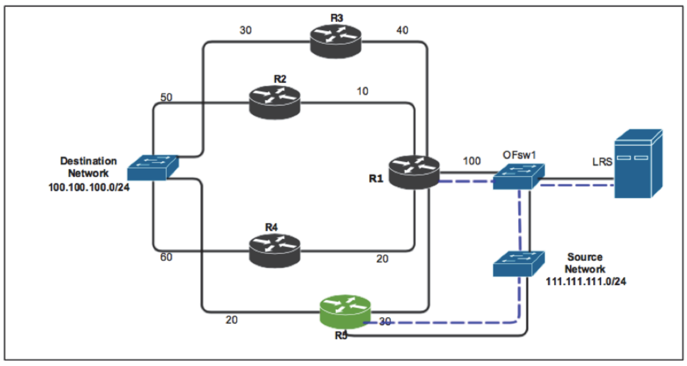
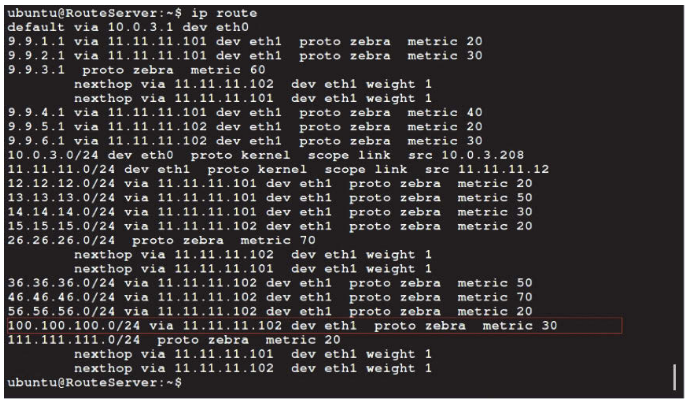


Figure 5.4: Hybrid topology with multiple edge routers

LRS routing table as shown in Figure 5.5 now shows alternate(better) next hop selection for remote network 100.100.100.0/24



```
ubuntu@RouteServer:~$ ip route
default via 10.0.3.1 dev eth0
9.9.1.1 via 11.11.11.101 dev eth1  proto zebra  metric 20
9.9.2.1 via 11.11.11.101 dev eth1  proto zebra  metric 30
9.9.3.1  proto zebra  metric 60
        nexthop via 11.11.11.102  dev eth1 weight 1
        nexthop via 11.11.11.101  dev eth1 weight 1
9.9.4.1 via 11.11.11.101 dev eth1  proto zebra  metric 40
9.9.5.1 via 11.11.11.102 dev eth1  proto zebra  metric 20
9.9.6.1 via 11.11.11.102 dev eth1  proto zebra  metric 30
10.0.3.0/24 dev eth0  proto kernel  scope link  src 10.0.3.208
11.11.11.0/24 dev eth1  proto kernel  scope link  src 11.11.11.12
12.12.12.0/24 via 11.11.11.101 dev eth1  proto zebra  metric 20
13.13.13.0/24 via 11.11.11.101 dev eth1  proto zebra  metric 50
14.14.14.0/24 via 11.11.11.101 dev eth1  proto zebra  metric 30
15.15.15.0/24 via 11.11.11.102 dev eth1  proto zebra  metric 20
26.26.26.0/24  proto zebra  metric 70
        nexthop via 11.11.11.102  dev eth1 weight 1
        nexthop via 11.11.11.101  dev eth1 weight 1
36.36.36.0/24 via 11.11.11.102 dev eth1  proto zebra  metric 50
46.46.46.0/24 via 11.11.11.102 dev eth1  proto zebra  metric 70
56.56.56.0/24 via 11.11.11.102 dev eth1  proto zebra  metric 20
100.100.100.0/24 via 11.11.11.102 dev eth1  proto zebra  metric 30
111.111.111.0/24  proto zebra  metric 20
        nexthop via 11.11.11.101  dev eth1 weight 1
        nexthop via 11.11.11.102  dev eth1 weight 1
ubuntu@RouteServer:~$
```

Figure 5.5: Better routes learnt by LRS

Thus SYMPHONY architecture allows not only multiple points of connectivity between SDN and Legacy networks but also establishes an efficient communication channel between both the domains by choosing optimal best paths (within OF domain as well as legacy domain) from OF domains to remote networks.

5.3 Test Case3 :Applying Centralised Policy

This test verifies that upon the failure of an active link on which a policy is applied, alternate paths can be utilised by enabling SYMPHONY controller make forwarding decision based on both policy and link state into consideration. Hence, this test assumes that alternate paths are available for rerouting the traffic. We evaluate the time taken for a re-routing the packet over an alternate path available upon failure of links with policy applied.

In legacy topology shown in Figure 5.6, policy is applied on R1 to forward all traffic towards h2 via R2. The policy is applied on interface '3'of R1.

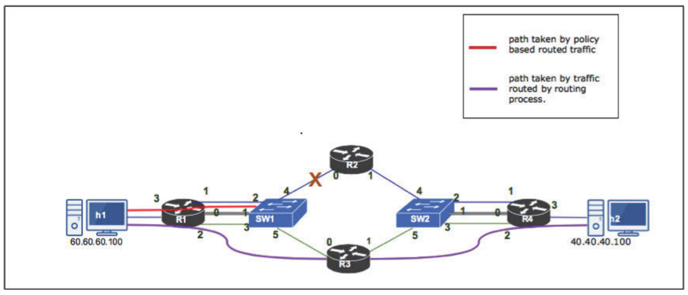

path taken by policy based routed traffic
path taken by traffic routed by routing process.
R1
R2
R3
R4
SW1
SW2
h1
h2
60.60.60.100
40.40.40.100

Figure 5.6: Traffic blackhole in legacy networks incase of link failures

R1 is unaware of link (Sw1, R2) failure and routes the traffic is via $R2 \rightarrow R4$ as configured in the policy map, thus leading to a network black hole, where the intended traffic doesn't reach destination. This is handled by either manually disabling the policy or by enabling tracking for a policy. Tracking only tracks next hop connectivity and not end-to-end connectivity as such. With a centralised controller and a controlled data plane as in SDN, packet forwarding is a centralised decision-making process. SYMPHONY employs this feature of SDN to enhance the policy based routing as performed in legacy routers. With centralised policy data base and programmable data plane, SYMPHONY is capable of choosing alternate paths when link (Sw1, R2) goes down as shown in the Figure 5.7

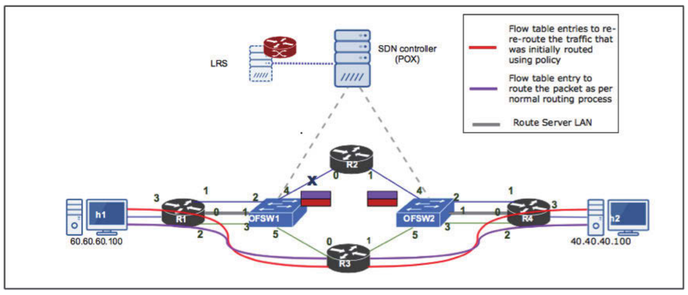
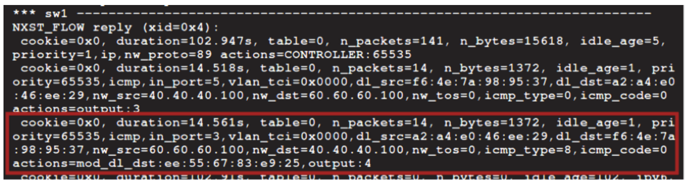

LRS
SDN controller (POX)
Flow table entries to re-route the traffic that was initially routed using policy
Flow table entry to route the packet as per normal routing process
Route Server LAN
R1
R2
R3
R4
OFSW1
OFSW2
h1
h2
60.60.60.100
40.40.40.100

Figure 5.7: Alternate routes selected in SDN incase of link failures

Figure 5.8 are flow table entries in sw1. The output port in this scenario is coherent with the policy configured to steer traffic from 60.60.60.100 destined to 40.40.40.100 to outport 4. A sample policy table entry can be found in Table 5.3. The link (OFSw1,R2) is functional while executing the test.

Table 5.3: Policy Table

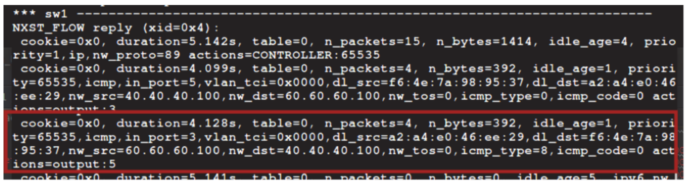
s.no	source	destination	protocol	action	dpid	port	type
1	60.60.60.100	40.40.40.100	1	permit	1	4	static



```
*** sw1 ------------------------------------------------------------------------
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=102.947s, table=0, n_packets=141, n_bytes=15618, idle_age=5,
priority=1,ip,nw_proto=89 actions=CONTROLLER:65535
 cookie=0x0, duration=14.518s, table=0, n_packets=14, n_bytes=1372, idle_age=1, pri
ority=65535,icmp,in_port=5,vlan_tci=0x0000,dl_src=f6:4e:7a:98:95:37,dl_dst=a2:a4:e0
:46:ee:29,nw_src=40.40.40.100,nw_dst=60.60.60.100,nw_tos=0,icmp_type=0,icmp_code=0
actions=output:3
 cookie=0x0, duration=14.561s, table=0, n_packets=14, n_bytes=1372, idle_age=1, pri
ority=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=a2:a4:e0:46:ee:29,dl_dst=f6:4e:7a
:98:95:37,nw_src=60.60.60.100,nw_dst=40.40.40.100,nw_tos=0,icmp_type=8,icmp_code=0
actions=mod_dl_dst:ee:55:67:83:e9:25,output:4
 cookie=0x0, duration=102.91s, table=0, n_packets=0, n_bytes=0, idle_age=102, ipv6,
```

Figure 5.8: Application of Policy-1

In case the link (OFSw1,R2) goes down as has been shown in Figure 5.7 , SYMPHONY architecture leverages the presence of OF switches and re-compute an alternate path to divert the traffic to. The alternate path thus computed is via R3 towards R4. Thus the modified flow table entry destined to 40.40.40.100 from 60.60.60.100 is via outport 5. The new flow table entries are shown in Figure 5.9



```
*** sw1 ------------------------------------------------------------------------
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=5.142s, table=0, n_packets=15, n_bytes=1414, idle_age=4, prio
rity=1,ip,nw_proto=89 actions=CONTROLLER:65535
 cookie=0x0, duration=4.099s, table=0, n_packets=4, n_bytes=392, idle_age=1, priori
ty=65535,icmp,in_port=5,vlan_tci=0x0000,dl_src=f6:4e:7a:98:95:37,dl_dst=a2:a4:e0:46
:ee:29,nw_src=40.40.40.100,nw_dst=60.60.60.100,nw_tos=0,icmp_type=0,icmp_code=0 act
ions=output:3
 cookie=0x0, duration=4.128s, table=0, n_packets=4, n_bytes=392, idle_age=1, priori
ty=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=a2:a4:e0:46:ee:29,dl_dst=f6:4e:7a:98
:95:37,nw_src=60.60.60.100,nw_dst=40.40.40.100,nw_tos=0,icmp_type=8,icmp_code=0 act
ions=output:5
 cookie=0x0, duration=5.141s, table=0, n_packets=0, n_bytes=0, idle_age=5, ipv6,nw
```

Figure 5.9: Application of policy-II

Figure 5.10 show the ping dump during the interval when the link (OFSw1, R2) goes down. Ideally, the ping should time out as the traffic is now being blackhole due to non-availability of the link. But, the ping responses show a spike in the RTT. The spike in RTT is the time taken by the controller to detect the link down and recomputed an alternate path to divert the traffic on, after which the ping responses continue to be received.

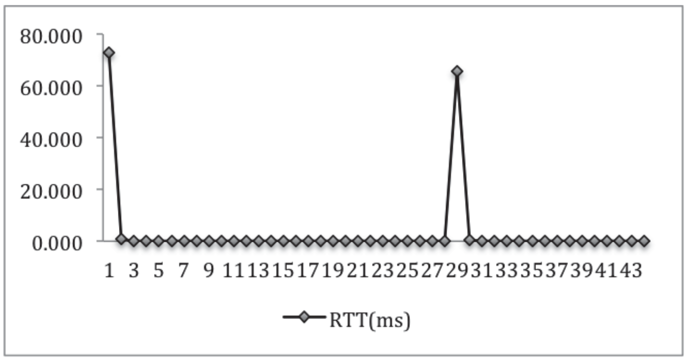


Figure 5.10: Response time

Thus this test proves that a Hybrid SDN setup can be used to enhance the policy handling mechanism of legacy network.

5.4 Test Case 4: Traffic Engineering using Explicit Congestion Notification

This test is aimed at verifying the effect of congestion in a network. In legacy networks, as discussed in chapter4, congestion in the network can be notified to the sender through feedback notifications. Upon receiving such a notification, host adjusts the congestion window to alter the number of packets being sent. Using Use case-III, we showed how alternate path could be chosen to conduct load balancing in the network. Such response to the feedback notification reduces bottleneck in the network and also prolongs the congestion window for the host to send packets at maximum capacity. SYMPHONY Hybrid SDN controller achieves this by verifying for alternate paths and inserting dynamic policy in the policy table to divert the traffic on alternate paths. This test case has been simulated in use case -III. However, further investigation will be carried out as future work of this thesis.

5.5 Conclusion

In this chapter, functionality test results have been presented. Three test cases built upon the use cases discussed in chapter4 have been tested. It has been noticed that using SYMPHONY Hybrid SDN controller; OSPF neighbour relation can be established between legacy routers and LRS. This is important to enable LRS to learn legacy network. Since SYMPHONY doesn't require STP to avoid loops in OF domain, the OSPF neighbour establishment time is considerably less and remains consistent over a variable number of OF switches. In test2, we verified that the number of edge routers connected improves the quality of routes learnt by the LRS. Finally in test3, an existing legacy functionality of policy enforcement has been improved with the help of SYMPHONY SDN controller.

Chapter 6

Conclusions

6.1 Summary

This thesis proposes controller architecture for building and maintaining Hybrid SDN. As networks migrate from distributed to centralised control planes, there is a need to establish inter domain communication within a brownfield deployment consisting of both SDN and legacy devices. Chapter 1 of the thesis highlights the research questions and details upon the research methodology. The primary research question that motivated this research is whether it feasible to model and build an efficient and easy to manage controller for Hybrid SDN and can such controller make real-time route computations for establishing data communication between SDN and legacy networks domains? We also investigated how such a hybrid network deployment model along with the Hybrid SDN controller can improve legacy network deployments.In Chapter2 of this thesis, the concept of Hybrid SDN has been introduced and an extensive literature review to understand the current state of research in Hybrid SDN has been listed. We have studies various frameworks such as RouteFlow, Panopticon, Cardigan, RISF to name few.

Chapter3 of this thesis proposes requirements and identify challenges for building controllers for Hybrid SDN networks. Based on the requirements and challenges identified, SYMPHONY –a controller for Hybrid SDN is proposed. The main components of SYMPHONY consist of the SDN controller (POX) application Packet-Forwarder and a Legacy Route Server (LRS). Topology information is stored in three parts –OF switches, boundary/edge nodes and remote networks. LLDP module assists the Packet-Forwarder application to discover OF topology. LRS forms neighbour relations with edge routers and discovers remote networks. LRS acts as a legacy route repository to store information of the legacy network and assist the main POX controller in achieving inter domain routing. Next-hop module enables Packet-Forwarder to query LRS for edge routers. Policy module reads and writes policies and Path-finder module applies SPF to find the shortest path between source and target OF Switch. Depending on the type of packet, a single target OF Switch (unicast) or multiple OF switches (multicast) can be chosen as target switches. Packet-Forwarder application performs functions such as unicast, multicast, maintain and apply static or dynamic policy for a flow. While a static policy is configured to perform policy-based routing, a dynamic policy is configured by the controller to reroute packets based on network conditions. POX controller along with LRS performs intra SDN and inter domain routing.

Chapter4 of this thesis describes various use cases demonstrating the need and efficiency of our proposed architecture SYMPHONY. The first use case illustrates that seamless communication between hosts in both the domain can be easily established. The second use case demonstrates how by maintaining a central policy repository a secondary data plane can be built for legacy routers. This use case also presents the fallback mechanism in case the policy enforcement fails. The third and final use case demonstrates how traffic engineering can be performed using a centralised controller. This use case shows how TCP ECN feedback flag can be used to choose alternate routes. While the first use case answers the first part of the research question, use cases 2,3 address the second research question and demonstrate how deploying a Hybrid SDN along with an SDN controller can enhance performance and better manage network components.

In Chapter5, tests have been conducted to determine the efficiency of SYMPHONY. The first test shows the effectiveness of SYMPHONY controller in establishing the communication channel over OF domain. The time taken for OSPF convergence is almost consistent with no dependency on the number of OF switches. The test case 2 shows how efficiently LRS can discover legacy network by forming OSPF relation with more edge routers. The test case 3 demonstrates that by deploying a centralised Hybrid SDN controller, not only communication can be established but also, legacy functions such as policy-based routing can also be implemented. This is an enhancement to legacy functions as this test shows that alternate paths can also be used to route the packets. We test the functionality of ECN using packet generator scapy to observe the controller reaction. However, this work can be extended in future to investigate the capabilities of our proposed scheme for more congested networks with different conditions.

6.2 Future Work

While the work of this thesis contributes to the field of Hybrid SDN, the proposed model can be extended to many interesting future works in the field. We are listing three important continuations of the work below:

Active Maps

Hybrid SDN controller stores complete topology information in various files. These files can be processed, and topology maps can be created to replicate existing topologies. These replicas can be used for experimentation purposes. Currently, no such feature exists in legacy networks. Legacy monitoring tools are limited to static maps.

Network feedbacks

Feedback between the source and destination to allow efficient use of network resources. SYMPHONY demonstrates a use case based on ECN. ECN is an optimal feature what network devices choose to support. Also, incase a router in a network doesn't support ECN, the ECN feedback loop fails. Hence, a different network feedback mechanism can be utilised.

Policy enforcement

The current version of SYMPHONY, the policy file is used to reroute the traffic. The feature can be extended further, by allowing the dynamic policies to expire after a change in network conditions. Also, a feature to check the policies for validity on regular intervals will make the better policy enforcement. Another area for improvement is the validation of policy enforcement based on end-to-end path availability. -

Appendices

Appendix A

Chapter3

Setting up the environment

1. Install mininet 2.1.0 and pox (from package)
2. OpenVSwitch comes preinstalled with ubuntu
3. Install dependencies for mininext
4. Install mininext

Building installation of Route server using lxc

1. Download lxc
2. create container
3. create second set of interface on lxc to connect to ovs switch
4. Set pox as controller for the ovs switch
5. install quagga
6. add a loopback Ip exclusively for routing purposes

Appendix B

Chapter4

```
*** sw1 ——————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=2782.609s, table=0, n_packets=299, n_bytes=25922, idle_age=9, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
** sw2 ——————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=2782.609s, table=0, n_packets=297, n_bytes=25734, idle_age=9, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
** sw3 ——————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=2782.615s, table=0, n_packets=0, n_bytes=0, idle_age=2782, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=5.498s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=2,priority=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=62:53:6f:a2:2b:ba,
dl_dst=7a:db:8e:c0:b8:f0, nw_src=11.11.11.101,nw_dst=11.11.11.102,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:4
cookie=0x0, duration=5.491s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=2,priority=65535,icmp,in_port=4,vlan_tci=0x0000,dl_src=7a:db:8e:c0:b8:f0,
dl_dst=62:53:6f:a2:2b:ba,nw_src=11.11.11.102,nw_dst=11.11.11.101,nw_tos=0,
icmp_type=0,icmp_code=0 actions=output:3
** sw4 ——————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=2782.594s, table=0, n_packets=0, n_bytes=0, idle_age=2782, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=5.493s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=2,priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=7a:db:8e:c0:b8:f0,
dl_dst=62:53:6f:a2:2b:ba, nw_src=11.11.11.102,nw_dst=11.11.11.101,nw_tos=0,
icmp_type=0,icmp_code=0 actions=output:3
cookie=0x0, duration=5.506s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=2,priority=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=62:53:6f:a2:2b:ba,
dl_dst=7a:db:8e:c0:b8:f0,nw_src=11.11.11.101,nw_dst=11.11.11.102,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:2
** sw5 ——————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=2782.619s, table=0, n_packets=0, n_bytes=0, idle_age=2782, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=5.503s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=2,priority=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=62:53:6f:a2:2b:ba,
dl_dst=7a:db:8e:c0:b8:f0, nw_src=11.11.11.101,nw_dst=11.11.11.102,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:3
cookie=0x0, duration=5.501s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=2,priority=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=7a:db:8e:c0:b8:f0,
dl_dst=62:53:6f:a2:2b:ba,nw_src=11.11.11.102,nw_dst=11.11.11.101,nw_tos=0,
icmp_type=0,icmp_code=0 actions=output:1
```

Figure B.1: Flow table entries for flow between h3,h4 when all the links are operational

```
*** sw1 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=4381.976s, table=0, n_packets=463, n_bytes=40038, idle_age=9, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
** sw2 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=4381.976s, table=0, n_packets=461, n_bytes=39850, idle_age=9, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=6.728s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=62:53:6f:a2:2b:ba,
dl_dst=7a:db:8e:c0:b8:f0, nw_src=11.11.11.101,nw_dst=11.11.11.102,nw_tos=0,icmp_type=8,
icmp_code=0 actions=output:3
cookie=0x0, duration=6.721s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=7a:db:8e:c0:b8:f0,
dl_dst=62:53:6f:a2:2b:ba, nw_src=11.11.11.102,nw_dst=11.11.11.101,nw_tos=0,icmp_type=0,
icmp_code=0 actions=output:2
** sw3 ———————————————————————————— NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=18.959s, table=0, n_packets=0, n_bytes=0, idle_age=18, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=6.735s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=62:53:6f:a2:2b:ba,
dl_dst=7a:db:8e:c0:b8:f0, nw_src=11.11.11.101,nw_dst=11.11.11.102,nw_tos=0,icmp_type=8,
icmp_code=0 actions=output:1
cookie=0x0, duration=6.721s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=7a:db:8e:c0:b8:f0,
dl_dst=62:53:6f:a2:2b:ba, nw_src=11.11.11.102,nw_dst=11.11.11.101,nw_tos=0,icmp_type=0,
icmp_code=0 actions=output:3
** sw4 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=4381.96s, table=0, n_packets=0, n_bytes=0, idle_age=4381, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=6.722s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=7a:db:8e:c0:b8:f0,
dl_dst=62:53:6f:a2:2b:ba, nw_src=11.11.11.102,nw_dst=11.11.11.101,nw_tos=0,icmp_type=0,
icmp_code=0 actions=output:3
cookie=0x0, duration=6.739s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=62:53:6f:a2:2b:ba,
dl_dst=7a:db:8e:c0:b8:f0, nw_src=11.11.11.101,nw_dst=11.11.11.102,nw_tos=0,icmp_type=8,
icmp_code=0 actions=output:2
** sw5 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=18.963s, table=0, n_packets=0, n_bytes=0, idle_age=18, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=6.739s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=62:53:6f:a2:2b:ba,
dl_dst=7a:db:8e:c0:b8:f0, nw_src=11.11.11.101,nw_dst=11.11.11.102,nw_tos=0,icmp_type=8,
icmp_code=0 actions=output:3
cookie=0x0, duration=6.735s, table=0, n_packets=4, n_bytes=392, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=7a:db:8e:c0:b8:f0,
dl_dst=62:53:6f:a2:2b:ba, nw_src=11.11.11.102,nw_dst=11.11.11.101,nw_tos=0,icmp_type=0,
icmp_code=0 actions=output:2
```

Figure B.2: Flow table entries for flow between h3,h4 when link between OFSw3 and OFSw5 goes down

```
*** sw1 ——————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=73.488s, table=0, n_packets=29, n_bytes=2618, idle_age=8, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=4.344s, table=0, n_packets=3, n_bytes=294, idle_timeout=10,
idle_age=2,priority=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=1e:1b:0b:8b:dc:54,
dl_dst=ba:62:98:74:8b:81, nw_src=100.100.100.100,nw_dst=200.200.200.200,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:2
cookie=0x0, duration=4.297s, table=0, n_packets=1, n_bytes=342, idle_timeout=10,
idle_age=4,priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=ba:62:98:74:8b:81,
dl_dst=1e:1b:0b:8b:dc:54, nw_src=200.200.200.200,nw_dst=100.100.100.100,nw_tos=0,
icmp_type=0,icmp_code=0 actions=output:1
** sw2 ——————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=73.485s, table=0, n_packets=28, n_bytes=2612, idle_age=8, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=4.287s, table=0, n_packets=3, n_bytes=294, idle_timeout=10,
idle_age=2,priority=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=ba:62:98:74:8b:81,
dl_dst=1e:1b:0b:8b:dc:54, nw_src=200.200.200.200,nw_dst=100.100.100.100,nw_tos=0,
icmp_type=0,icmp_code=0 actions=output:2
cookie=0x0, duration=4.331s, table=0, n_packets=3, n_bytes=294, idle_timeout=10,
idle_age=2,priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=1e:1b:0b:8b:dc:54,
dl_dst=ba:62:98:74:8b:81, nw_src=100.100.100.100,nw_dst=200.200.200.200,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:1
** sw3 ——————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=73.486s, table=0, n_packets=0, n_bytes=0, idle_age=73, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=4.29s, table=0, n_packets=3, n_bytes=294, idle_age=2, prior-
ity=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=ba:62:98:74:8b:81,
dl_dst=1e:1b:0b:8b:dc:54, nw_src=200.200.200.200,nw_dst=100.100.100.100,nw_tos=0,
icmp_type=0,icmp_code=0 actions=mod_dl_dst:1e:1b:0b:8b:dc:54,output:2
cookie=0x0, duration=4.351s, table=0, n_packets=0, n_bytes=0, idle_age=4, prior-
ity=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=1e:1b:0b:8b:dc:54,
dl_dst=ba:62:98:74:8b:81, nw_src=100.100.100.100,nw_dst=200.200.200.200,nw_tos=0,
icmp_type=8,icmp_code=0 actions=mod_dl_dst:ba:62:98:74:8b:81,output:1
** sw4 ——————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=73.494s, table=0, n_packets=0, n_bytes=0, idle_age=73, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
** sw5 —————————————————————————————— NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=73.5s, table=0, n_packets=0, n_bytes=0, idle_age=73, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
```

Figure B.3: Flow table entries for flows between h1, h2 when all links are operational

```
*** sw1 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=7.63s, table=0, n_packets=1, n_bytes=86, idle_age=7, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=6.904s, table=0, n_packets=5, n_bytes=490, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=66:ca:ae:f0:f2:11,
dl_dst=d2:e7:9e:e5:e8:2c, nw_src=200.200.200.200,nw_dst=100.100.100.100,nw_tos=0,
icmp_type=0,icmp_code=0 actions=output:1
cookie=0x0, duration=6.981s, table=0, n_packets=5, n_bytes=490, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=d2:e7:9e:e5:e8:2c,
dl_dst=66:ca:ae:f0:f2:11, nw_src=100.100.100.100,nw_dst=200.200.200.200,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:3
** sw2 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=81.696s, table=0, n_packets=36, n_bytes=3488, idle_age=7, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=8.97s, table=0, n_packets=7, n_bytes=686, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=d2:e7:9e:e5:e8:2c,
dl_dst=66:ca:ae:f0:f2:11, nw_src=100.100.100.100,nw_dst=200.200.200.200,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:1
cookie=0x0, duration=8.943s, table=0, n_packets=7, n_bytes=686, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=66:ca:ae:f0:f2:11,
dl_dst=d2:e7:9e:e5:e8:2c, nw_src=200.200.200.200,nw_dst=100.100.100.100,nw_tos=0,
icmp_type=0,icmp_code=0 actions=output:2
** sw3 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=7.633s, table=0, n_packets=0, n_bytes=0, idle_age=7, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=6.93s, table=0, n_packets=5, n_bytes=490, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=1,vlan_tci=0x0000, dl_src=66:ca:ae:f0:f2:11,
dl_dst=d2:e7:9e:e5:e8:2c, nw_src=200.200.200.200,nw_dst=100.100.100.100,nw_tos=0,
icmp_type=0,icmp_code=0 actions=output:3
cookie=0x0, duration=6.959s, table=0, n_packets=4, n_bytes=392, idle_age=3, prior-
ity=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=d2:e7:9e:e5:e8:2c,
dl_dst=66:ca:ae:f0:f2:11,nw_src=100.100.100.100,nw_dst=200.200.200.200,nw_tos=0,
icmp_type=8,icmp_code=0 actions=mod_dl_dst:66:ca:ae:f0:f2:11,output:1
** sw4 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=81.703s, table=0, n_packets=0, n_bytes=0, idle_age=81, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=6.994s, table=0, n_packets=5, n_bytes=490, idle_timeout=10,
idle_age=3,priority=65535,icmp,in_port=1,vlan_tci=0x0000, dl_src=d2:e7:9e:e5:e8:2c,
dl_dst=66:ca:ae:f0:f2:11, nw_src=100.100.100.100,nw_dst=200.200.200.200,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:2
cookie=0x0, duration=8.85s, table=0, n_packets=6, n_bytes=588, idle_age=3, prior-
ity=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=66:ca:ae:f0:f2:11,
dl_dst=d2:e7:9e:e5:e8:2c,nw_src=200.200.200.200, nw_dst=100.100.100.100,nw_tos=0,
icmp_type=0,icmp_code=0 actions=mod_dl_dst:d2:e7:9e:e5:e8:2c,output:1
** sw5 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=81.71s, table=0, n_packets=0, n_bytes=0, idle_age=81, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
```

Figure B.4: Flow table entries for flows between h1 , h2 when the link between OFSw1, OFSw3 is down.

```
*** sw1 ——————————————————————————————-
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=69.658s, table=0, n_packets=33, n_bytes=2990, idle_age=3, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=5.929s, table=0, n_packets=4, n_bytes=401, idle_timeout=10,
idle_age=3, priority=65535, icmp,in_port=3,vlan_tci=0x0000, dl_src=ca:
46:f5:9b:5f:dd,dl_dst=16:60:2c:cc:f2:7c,nw_src=11.11.11.101,nw_dst=100.100.100.100,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:1
cookie=0x0, duration=5.911s, table=0, n_packets=3, n_bytes=294,
idle_timeout=10, idle_age=3, priority=65535, icmp,in_port=1,vlan_tci=0x0000,
dl_src=16:60:2c:cc:f2:7c,dl_dst=ca:46:f5:9b:5f:dd,nw_src=100.100.100.100,nw_dst=11.11.11.101,nw_tos=0,
icmp_type=0,icmp_code=0 actions=output:3
cookie=0x0, duration=69.621s, table=0, n_packets=0, n_bytes=0, idle_age=69,
ipv6,nw_proto=0,nw_tos=0 actions=drop
cookie=0x0, duration=69.621s, table=0, n_packets=16, n_bytes=5472, idle_age=4, udp,tp_src=68
actions=drop
cookie=0x0, duration=69.621s, table=0, n_packets=0, n_bytes=0, idle_age=69, udp,tp_src=53
actions=drop
** sw2 ———————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=69.652s, table=0, n_packets=32, n_bytes=2932, idle_age=3, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=69.623s, table=0, n_packets=0, n_bytes=0, idle_age=69,
ipv6,nw_proto=0,nw_tos=0 actions=drop
cookie=0x0, duration=69.623s, table=0, n_packets=13, n_bytes=4446, idle_age=0, udp,tp_src=68
actions=drop
cookie=0x0, duration=69.623s, table=0, n_packets=0, n_bytes=0, idle_age=69, udp,tp_src=53
actions=drop
** sw3 ———————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=69.658s, table=0, n_packets=0, n_bytes=0, idle_age=69, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=69.628s, table=0, n_packets=0, n_bytes=0, idle_age=69,
ipv6,nw_proto=0,nw_tos=0 actions=drop
cookie=0x0, duration=69.628s, table=0, n_packets=29, n_bytes=9918, idle_age=2, udp,tp_src=68
actions=drop
cookie=0x0, duration=69.628s, table=0, n_packets=0, n_bytes=0, idle_age=69, udp,tp_src=53
actions=drop
** sw4 ———————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=69.663s, table=0, n_packets=0, n_bytes=0, idle_age=69, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=5.92s, table=0, n_packets=2, n_bytes=196, idle_age=3, prior-
ity=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=ca:46:f5:9b:5f:dd,
dl_dst=16:60:2c:cc:f2:7c,nw_src=11.11.11.101,nw_dst=100.100.100.100,
nw_tos=0,icmp_type=8, icmp_code=0 actions=mod_dl_dst:16:60:2c:cc:f2:7c,output:1
cookie=0x0, duration=5.918s, table=0, n_packets=3, n_bytes=294, idle_timeout=10, idle_age=3,
priority=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=16:60:2c:cc:f2:7c,
dl_dst=ca:46:f5:9b:5f:dd,nw_src=100.100.100.100,nw_dst=11.11.11.101,
nw_tos=0, icmp_type=0,icmp_code=0 actions=output:3
cookie=0x0, duration=69.63s, table=0, n_packets=0, n_bytes=0, idle_age=69,
ipv6,nw_proto=0,nw_tos=0 actions=drop
cookie=0x0, duration=69.63s, table=0, n_packets=14, n_bytes=4788, idle_age=0, udp,tp_src=68
actions=drop
cookie=0x0, duration=69.63s, table=0, n_packets=0, n_bytes=0, idle_age=69, udp,tp_src=53 ac-
tions=drop
** sw5 ———————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=69.669s, table=0, n_packets=0, n_bytes=0, idle_age=69, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=69.634s, table=0, n_packets=0, n_bytes=0, idle_age=69,
ipv6,nw_proto=0,nw_tos=0 actions=drop
cookie=0x0, duration=69.634s, table=0, n_packets=16, n_bytes=5472, idle_age=1, udp,tp_src=68
actions=drop
cookie=0x0, duration=69.634s, table=0, n_packets=0, n_bytes=0, idle_age=69, udp,tp_src=53
actions=drop
```

Figure B.5: Flow table entries for flow between h3 ,h1

```
*** sw1 ————————————————————————————
NXST__FLOW reply (xid=0x4):
cookie=0x0, duration=259.355s, table=0, n_packets=52, n_bytes=4624, idle_age=3, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535 cookie=0x0, duration=259.318s, table=0,
n_packets=0, n_bytes=0, idle_age=259, ipv6,nw_proto=0,nw_tos=0 actions=drop cookie=0x0,
duration=259.318s, table=0, n_packets=40, n_bytes=13680, idle_age=71, udp,tp_src=68 actions=drop
cookie=0x0, duration=259.318s, table=0, n_packets=0, n_bytes=0, idle_age=259, udp,tp_src=53
actions=drop *** sw2 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=259.35s, table=0, n_packets=51, n_bytes=4566, idle_age=3, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=4.678s, table=0, n_packets=2, n_bytes=196, idle_timeout=10, idle_age=3,
priority=65535,icmp,in_port=1,vlan_tci=0x0000, dl_src=4e:a9:52:b9:69:1a,
dl_dst=ca:46:f5:9b:5f:dd,nw_src=200.200.200.200,nw_dst=11.11.11.101,nw_tos=0,
icmp_type=0,icmp_code=0 actions=output:2
cookie=0x0, duration=4.686s, table=0, n_packets=2, n_bytes=196, idle_timeout=10, idle_age=3,
priority=65535,icmp,in_port=2,vlan_tci=0x0000, dl_src=ca:46:f5:9b:5f:dd,
dl_dst=4e:a9:52:b9:69:1a,nw_src=11.11.11.101,nw_dst=200.200.200.200,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:1
cookie=0x0, duration=259.321s, table=0, n_packets=0, n_bytes=0, idle_age=259,
ipv6,nw_proto=0,nw_tos=0 actions=drop
cookie=0x0, duration=259.321s, table=0, n_packets=41, n_bytes=14022, idle_age=65, udp,tp_src=68
actions=drop
cookie=0x0, duration=259.321s, table=0, n_packets=0, n_bytes=0, idle_age=259, udp,tp_src=53
actions=drop
** sw3 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=259.355s, table=0, n_packets=0, n_bytes=0, idle_age=259, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=4.706s, table=0, n_packets=2, n_bytes=196, idle_age=3, prior-
ity=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=ca:46:f5:9b:5f:dd,
dl_dst=16:60:2c:cc:f2:7c,nw_src=11.11.11.101,nw_dst=200.200.200.200,nw_tos=0,
icmp_type=8,icmp_code=0 actions=mod_dl_dst:4e:a9:52:b9:69:1a,output:1
cookie=0x0, duration=4.676s, table=0, n_packets=2, n_bytes=196, idle_timeout=10, idle_age=3,
priority=65535,icmp,in_port=1,vlan_tci=0x0000,
dl_src=4e:a9:52:b9:69:1a,dl_dst=ca:46:f5:9b:5f:dd,nw_src=200.200.200.200,nw_dst=11.11.11.101,
nw_tos=0, icmp_type=0,icmp_code=0 actions=output:3
cookie=0x0, duration=259.325s, table=0, n_packets=0, n_bytes=0, idle_age=259,
ipv6,nw_proto=0,nw_tos=0 actions=drop
cookie=0x0, duration=259.325s, table=0, n_packets=84, n_bytes=28728, idle_age=65, udp,tp_src=68
actions=drop
cookie=0x0, duration=259.325s, table=0, n_packets=0, n_bytes=0, idle_age=259, udp,tp_src=53
actions=drop
** sw4 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=259.361s, table=0, n_packets=0, n_bytes=0, idle_age=259, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=4.71s, table=0, n_packets=2, n_bytes=196, idle_timeout=10, idle_age=3,
priority=65535,icmp,in_port=3,vlan_tci=0x0000, dl_src=ca:46:f5:9b:5f:dd,
dl_dst=16:60:2c:cc:f2:7c,nw_src=11.11.11.101,nw_dst=200.200.200.200,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:2
cookie=0x0, duration=4.676s, table=0, n_packets=2, n_bytes=196, idle_timeout=10, idle_age=3,
priority=65535,icmp,in_port=2,vlan_tci=0x0000,
dl_src=4e:a9:52:b9:69:1a,dl_dst=ca:46:f5:9b:5f:dd,nw_src=200.200.200.200,nw_dst=11.11.11.101,
nw_tos=0,icmp_type=0,icmp_code=0 actions=output:3
cookie=0x0, duration=259.328s, table=0, n_packets=0, n_bytes=0, idle_age=259,
ipv6,nw_proto=0,nw_tos=0 actions=drop
cookie=0x0, duration=259.328s, table=0, n_packets=40, n_bytes=13680, idle_age=69, udp,tp_src=68
actions=drop
cookie=0x0, duration=259.328s, table=0, n_packets=0, n_bytes=0, idle_age=259, udp,tp_src=53
actions=drop
** sw5 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=259.366s, table=0, n_packets=0, n_bytes=0, idle_age=259, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=259.331s, table=0, n_packets=0, n_bytes=0, idle_age=259,
ipv6,nw_proto=0,nw_tos=0 actions=drop
cookie=0x0, duration=259.331s, table=0, n_packets=42, n_bytes=14364, idle_age=73, udp,tp_src=68
actions=drop
cookie=0x0, duration=259.331s, table=0, n_packets=0, n_bytes=0, idle_age=259, udp,tp_src=53
actions=drop
```

Figure B.6: Flow table entries for flow between h3 , h2

```
*** sw1 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=63.021s, table=0, n_packets=115, n_bytes=13054, idle_age=8, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=4.447s, table=0, n_packets=3, n_bytes=294, idle_age=2, prior-
ity=65535,icmp,in_port=4,vlan_tci=0x0000,dl_src=4a:2c:8b:dd:cf:7e,
dl_dst=fa:8d:43:32:be:00, nw_src=40.40.40.100,nw_dst=60.60.60.100,nw_tos=0,
icmp_type=0,icmp_code=0 actions=output:2
cookie=0x0, duration=4.482s, table=0, n_packets=3, n_bytes=294, idle_age=2, prior-
ity=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=46:b2:0a:c6:4c:37,
dl_dst=5a:c5:57:f5:ff:c4, nw_src=60.60.60.100,nw_dst=40.40.40.100,nw_tos=0,
icmp_type=8,icmp_code=0 actions=mod_dl_dst:4a:2c:8b:dd:cf:7e,output:4
** sw2 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=63.021s, table=0, n_packets=112, n_bytes=13120, idle_age=8, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=4.469s, table=0, n_packets=3, n_bytes=294, idle_age=2, prior-
ity=65535,icmp,in_port=4,vlan_tci=0x0000,dl_src=1a:db:f8:46:fd:f7,
dl_dst=26:c4:37:37:5b:12, nw_src=60.60.60.100,nw_dst=40.40.40.100,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:2
cookie=0x0, duration=4.466s, table=0, n_packets=3, n_bytes=294, idle_age=2, prior-
ity=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=4e:74:15:23:3d:00,
dl_dst=ae:11:db:67:f5:c7, nw_src=40.40.40.100,nw_dst=60.60.60.100,nw_tos=0,
icmp_type=0,icmp_code=0 actions=mod_dl_dst:1a:db:f8:46:fd:f7,output:4
```

Figure B.7: Flow table entries for flows between h1,h2 with policy applied to divert the traffic via R3

```
*** sw1 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=34.121s, table=0, n_packets=12, n_bytes=996, idle_age=8, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=8.035s, table=0, n_packets=3, n_bytes=294, idle_age=6, prior-
ity=65535,icmp,in_port=5,vlan_tci=0x0000,dl_src=1a:6a:14:d4:a6:18,
dl_dst=d6:0c:a3:4d:7d:a7, nw_src=40.40.40.100,nw_dst=60.60.60.100,nw_tos=0,
icmp_type=0,icmp_code=0 actions=output:3
cookie=0x0, duration=8.077s, table=0, n_packets=3, n_bytes=294, idle_age=6, prior-
ity=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=d6:0c:a3:4d:7d:a7,
dl_dst=1a:6a:14:d4:a6:18, nw_src=60.60.60.100,nw_dst=40.40.40.100,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:5
** sw2 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=207.194s, table=0, n_packets=103, n_bytes=9226, idle_age=8, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=8.066s, table=0, n_packets=3, n_bytes=294, idle_age=6, prior-
ity=65535,icmp,in_port=5,vlan_tci=0x0000,dl_src=46:d1:bb:40:20:a5,
dl_dst=fa:41:66:89:28:4c, nw_src=60.60.60.100,nw_dst=40.40.40.100,nw_tos=0,
icmp_type=8,icmp_code=0 actions=output:3
cookie=0x0, duration=8.052s, table=0, n_packets=3, n_bytes=294, idle_age=6, prior-
ity=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=fa:41:66:89:28:4c,
dl_dst=46:d1:bb:40:20:a5, nw_src=40.40.40.100,nw_dst=60.60.60.100,nw_tos=0,
icmp_type=0,icmp_code=0 actions=output:5
```

Figure B.8: Flow table entries for flows between h1,h2 with policy applied to divert the traffic via R3 and link R1, R3 is non operational

```
*** sw1 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=105.228s, table=0, n_packets=150, n_bytes=17100, idle_age=0, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=18.759s, table=0, n_packets=729, n_bytes=50510, idle_age=6, prior-
ity=65535,tcp,in_port=3,vlan_tci=0x0000,dl_src=d2:2a:4e:7f:8a:5d,
dl_dst=6e:e6:d7:7f:74:86, nw_src=60.60.60.100,nw_dst=40.40.40.100,nw_tos=0,tp_src=80,
tp_dst=36652 actions=output:5
cookie=0x0, duration=18.775s, table=0, n_packets=729, n_bytes=1096722, idle_age=6, prior-
ity=65535,tcp,in_port=5,vlan_tci=0x0000,dl_src=6e:e6:d7:7f:74:86,
dl_dst=d2:2a:4e:7f:8a:5d, nw_src=40.40.40.100,nw_dst=60.60.60.100,nw_tos=0,tp_src=36652,
tp_dst=80 actions=output:3
** sw2 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=105.228s, table=0, n_packets=139, n_bytes=16854, idle_age=0, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=18.793s, table=0, n_packets=729, n_bytes=1096722, idle_age=6, prior-
ity=65535,tcp,in_port=3,vlan_tci=0x0000,dl_src=a2:62:16:51:52:ea,
dl_dst=5a:6d:ac:e3:d4:81, nw_src=40.40.40.100,nw_dst=60.60.60.100,nw_tos=0,tp_src=36652,
tp_dst=80 actions=output:5
cookie=0x0, duration=18.747s, table=0, n_packets=729, n_bytes=50510, idle_age=6, prior-
ity=65535,tcp,in_port=5,vlan_tci=0x0000,          dl_src=5a:6d:ac:e3:d4:81,dl_dst=a2:62:16:51:52:ea,
nw_src=60.60.60.100,nw_dst=40.40.40.100,
nw_tos=0,tp_src=80,tp_dst=36652 actions=output:3
```

Figure B.9: Flow table entries under normal conditions

```
*** sw1 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=399.021s, table=0, n_packets=297, n_bytes=29270, idle_age=3, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=24.542s, table=0, n_packets=1, n_bytes=54, idle_age=24, prior-
ity=65535,tcp,in_port=3,vlan_tci=0x0000,dl_src=d2:2a:4e:7f:8a:5d,
dl_dst=6e:e6:d7:7f:74:86, nw_src=60.60.60.100,nw_dst=40.40.40.100,nw_tos=0,tp_src=80,tp_dst=20
actions=mod_dl_dst:06:d6:11:21:fb:fe,output:4
cookie=0x0, duration=24.545s, table=0, n_packets=1, n_bytes=54, idle_age=24, prior-
ity=65535,tcp,in_port=5,vlan_tci=0x0000,dl_src=6e:e6:d7:7f:74:86,
dl_dst=d2:2a:4e:7f:8a:5d, nw_src=40.40.40.100,nw_dst=60.60.60.100,nw_tos=0,tp_src=20,tp_dst=80
actions=output:3

*** sw2 ————————————————————————————
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=399.022s, table=0, n_packets=288, n_bytes=29188, idle_age=3, prior-
ity=1,ip,nw_proto=89 actions=CONTROLLER:65535
cookie=0x0, duration=24.579s, table=0, n_packets=1, n_bytes=54, idle_age=24, prior-
ity=65535,tcp,in_port=3,vlan_tci=0x0000,dl_src=a2:62:16:51:52:ea,
dl_dst=5a:6d:ac:e3:d4:81, nw_src=40.40.40.100,nw_dst=60.60.60.100,nw_tos=0,tp_src=20,tp_dst=80
actions=output:5
cookie=0x0, duration=24.533s, table=0, n_packets=1, n_bytes=54, idle_age=24, prior-
ity=65535,tcp,in_port=4,vlan_tci=0x0000,dl_src=8a:e6:4c:8c:2d:d4,
dl_dst=ce:b5:f6:79:f2:31, nw_src=60.60.60.100,nw_dst=40.40.40.100,nw_tos=0,tp_src=80,tp_dst=20
actions=output:2
```

Figure B.10: Flow table entries when ECN is received by the h1 from h2

Appendix C

Chapter5

= ((lost_message + 1) x hello) + (BPDU_Delay x (diameter -1)) + ((diameter -1) x overestimate_per_bridge) + (diameter x transit_delay)
+ medium_access_delay + Maximum_transmission_halt_delay) / 2 where

- End-to-end BPDU__propagation_delay (A) is the time taken by the BPDU to propagate from one node to another and is given by the formula: ((lost_message + 1) x hello) + ((BPDU_Delay x (diameter–1))

  where

  - Lost_message is the number of BPDU's that can be lost as BDPUs traverse from one end of a network to another.
  - BPDU_Delay is the time taken by a bridge to transmit BPDU received on one port to another port.
  - diameter is the longest distance between two nodes post shortest path computation

- Message_age_overestimate (B) is latency introduced by a switch while forwarding BDPU. Latency can be calculated by using the formula: (diameter –1) x overestimate_per_bridge

  where

  - overestimate_per_bridge is by default 1

- Maximum_frame_lifetime (C) is the time spent by a BPDU in the network before it reaches the final destination. Given by formula: diameter x transit_delay + med_access_delay

  where

- medium_access_delay is the time required by the switch to gain access to the media.

- Maximum_transmission_halt_delay (D) is the time taken to block a port. By default the value is 1.

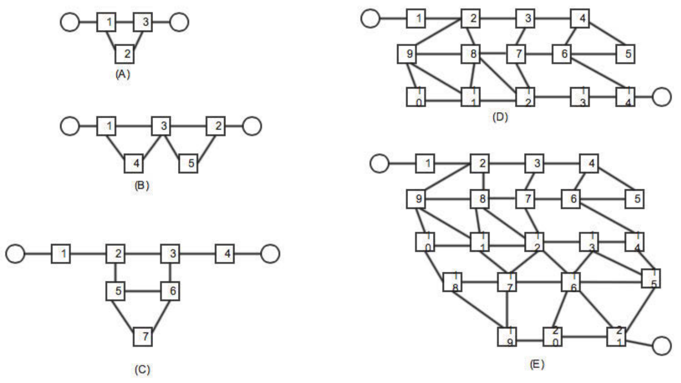


Figure C.1: Topologies used for testcase 1. (A) 3 OF Switches,(B) 5 OF Switches,(C) 7 OF Switches, (D)14 OF Switches,(E) 21 OF Switches

Bibliography

[1] Sankalp Agarwal, Murali Kodialam, and TV Lakshman. Traffic engineering in software defined networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2211–2219. IEEE, 2013.

[2] Wolfgang Barth. *Nagios: System and network monitoring*. No Starch Press, 2008.

[3] Bartosz Belter, Damian Parniewicz, Lukasz Ogrodowczyk, Artur Binczewski, Maciej Stroinski, Victor Fuentes, Jon Matias, Maider Huarte, and Eduardo Jacob. Hardware abstraction layer as an sdn-enabler for non-openflow network equipment. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 117–118. IEEE, 2014.

[4] Ricardo Bennesby, Paulo Fonseca, Edjard Mota, and Alexandre Passito. An inter-as routing component for software-defined networks. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 138–145. IEEE, 2012.

[5] Philippe Biondi. Scapy. *see http://www. secdev. org/projects/scapy*, 2011.

[6] Darius Buntinas. Congestion control schemes for tcp/ip networks. *Dept of CIS, Ohio State University*.

[7] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 15–28. USENIX Association, 2005.

[8] Daniel J Casey and Barry E Mullins. Sdn shim: Controlling legacy devices. In *Local Computer Networks (LCN), 2015 IEEE 40th Conference on*, pages 169–172. IEEE, 2015.

[9] Taesang Choi, Sejun Song, Hyungbae Park, Sangsik Yoon, and Sunhee Yang. Suma: software-defined unified monitoring agent for sdn. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–5. IEEE, 2014.

[10] Cisco. Understanding and tuning spanning tree protocol timers. *see*

http://www.cisco.com/c/en/us/support/docs/lan-switching/spanning-tree-protocol/19120-122.html, 2006.

[11] Link Layer discovery Protocol. Ieee 802.1 ab, 2009.

[12] Fernando Farias, Igor Carvalho, Eduardo Cerqueira, Antônio Abelém, Christian E Rothenberg, and Michael Stanton. Legacyflow: Bringing openflow to legacy network environments, 2011.

[13] Adrian Farrel, Jean-Philippe Vasseur, and Jerry Ash. A path computation element (pce)-based architecture. Technical report, RFC 4655, August, 2006.

[14] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn. *Queue*, 11(12):20, 2013.

[15] Tao Feng, Jun Bi, Peiyao Xiao, and Xiuli Zheng. Hybrid sdn architecture to integrate with legacy control and management plane: An experiences-based study. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 754–757. IEEE, 2015.

[16] Sally Floyd. Tcp and explicit congestion notification. *ACM SIGCOMM Computer Communication Review*, 24(5):8–23, 1994.

[17] Bernard Fortz, Jennifer Rexford, and Mikkel Thorup. Traffic engineering with traditional ip routing protocols. *Communications Magazine, IEEE*, 40(10):118–124, 2002.

[18] Adrian Gämperli. Evaluating the effect of sdn centralization on internet routing convergence. *Master's thesis, ETH Zürich*, 2014.

[19] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. Sdx: A software defined internet exchange. *ACM SIGCOMM Computer Communication Review*, 44(4):551–562, 2015.

[20] Susan Hares and Russ White. Software-defined networks and the interface to the routing system (i2rs). *IEEE Internet Computing*, (4):84–88, 2013.

[21] Nanxi Kang, Joshua Reich, Jennifer Rexford, and David Walker. Policy transformation in software defined networks. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 309–310. ACM, 2012.

[22] Chang Woo Kim, Nikhil Malik, Dipjyoti Saikia, and Sueng Yong Park. An architecture for sdn flowmap inter-operation with legacy protocols. In *Information and Communication*

Technology Convergence (ICTC), 2014 International Conference on, pages 135–137. IEEE, 2014.

[23] 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.

[24] Dan Levin, Marco Canini, Stefan Schmid, and Anja Feldmann. *Panopticon: Reaping the benefits of partial sdn deployment in enterprise networks*. Die Professoren der Fakultät IV, Elektrotechnik und Informatik, 2013.

[25] Pingping Lin, Jun Bi, and Hongyu Hu. Internetworking with sdn using existing bgp. In *Proceedings of The Ninth International Conference on Future Internet Technologies*, page 21. ACM, 2014.

[26] Hui Lu, Nipun Arora, Hui Zhang, Cristian Lumezanu, Junghwan Rhee, and Guofei Jiang. Hybnet: Network manager for a hybrid network infrastructure. In *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, page 6. ACM, 2013.

[27] LXC. lxc @ONLINE, June 2015. URL http://linuxcontainers.org.

[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] Deepankar Medhi. *Network routing: algorithms, protocols, and architectures*. Morgan Kaufmann, 2010.

[30] Ali Al-Shabibi Murphy McCauley. Pox @ONLINE, June 2015. URL https://openflow.stanford.edu/display/ONL/POX+Wiki.

[31] Thomas D Nadeau and Ken Gray. *SDN: software defined networks*. " O'Reilly Media, Inc.", 2013.

[32] Marcelo R Nascimento, Christian E Rothenberg, Marcos R Salvador, Maurıcio F Magalhaes, CN Corrêa, and Sidney C de Lucena. The routeflow approach to ip routing services on software-defined networks. In *Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)*, 2011.

[33] Vern Paxson. End-to-end routing behavior in the internet. *Networking, IEEE/ACM Transactions on*, 5(5):601–615, 1997.

[34] Quagga. Quagga @ONLINE, June 2015. URL http://www.nongnu.org/quagga/i.

[35] Stefano Salsano, Pier Luigi Ventre, Luca Prete, Giulio Siracusano, Matteo Gerola, and Elio Salvadori. Oshi-open source hybrid ip/sdn networking (and its emulation on mininet and on distributed sdn testbeds). In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 13–18. IEEE, 2014.

[36] Parmanand Sharma, Sean Banerjee, Sébastien Tandel, Rui Aguiar, Ronan Amorim, and David Pinheiro. Enhancing network management frameworks with sdn-like control. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 688–691. IEEE, 2013.

[37] OpenFlow Switch Specification. Version 1.0. 0 (wire protocol 0x01), 2009.

[38] Jonathan Philip Stringer, Qiang Fu, Christopher Lorier, Richard Nelson, and Christian Esteve Rothenberg. Cardigan: Deploying a distributed routing fabric. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 169–170. ACM, 2013.

[39] David L Tennenhouse and David J Wetherall. Towards an active network architecture. In *DARPA Active Networks Conference and Exposition, 2002. Proceedings*, pages 2–15. IEEE, 2002.

[40] Peter Thai and Jose Carlos de Oliveira. Decoupling policy from routing with software defined interdomain management: interdomain routing for sdn-based networks. In *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*, pages 1–6. IEEE, 2013.

[41] Stefano Vissicchio, Laurent Vanbever, and Olivier Bonaventure. Opportunities and research challenges of hybrid software defined networks. *ACM SIGCOMM Computer Communication Review*, 44(2):70–75, 2014.

[42] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: a language for high-level reactive network control. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 43–48. ACM, 2012.

[43] Lily Yang, Ram Dantu, T Anderson, and Ram Gopal. Forwarding and control element separation (forces) framework. Technical report, RFC 3746, April, 2004.