POLICY CARRY-OVER FOR MOBILITY IN SOFTWARE-DEFINED NETWORKING

A Thesis

Presented to

The Faculty of the Department of Engineering Technology

University of Houston

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science in Engineering Technology

By

Kiran Vemuri

December 2014

Acknowledgments

I would like to express my deep and sincere gratitude to my advisor, Dr. Deniz Gurkan. Her invaluable advice and guidance have provided a good basis for this thesis. I am very thankful for her ideas, comments and consistent support throughout this work.

POLICY CARRY-OVER FOR MOBILITY IN SOFTWARE-DEFINED NETWORKING

An Abstract of a Thesis

Presented to

The Faculty of the Department of Engineering Technology

University of Houston

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science in Engineering Technology

By

Kiran Vemuri

December 2014

Abstract

Due to increase in the number of mobile devices that are connected to a network, it is getting harder by the day to manage these devices while they hop between different networks. Network management often involves implementation of policies that are generic to all the devices connected to the network as well as the ones that are specific to individual devices. In order to support this type of network management, static policies have to be set up across the networks using middle box technologies like firewalls, network policy servers, authentication servers etc. These middle boxes control the activity of the devices connected to the network using the policies but are often home for a huge number policies that complicate the network setup and make policy management a herculean task.

In recent years, the development of Software Defined Networking practices has made it simple and intuitive to instantiate programmable networks and automate different network functions. We propose a solution to automate the process of network policy management for mobility in a Software Defined Network. Using this approach, we can implement a solution that can dynamically carry the policies across the network as a host or device moves from one place to another in a network, without any action from the network administrator. Also, with the help of this implementation, we can automate the process of policy repair in the case of network changes or errors.

Table of Contents

Chapter 1 – Introduction

Due to increase in the number of mobile devices that are connected to a network, it is getting harder by the day to manage these devices while they hop between different networks. Network management often involves implementation of policies that are generic to all the devices connected to the network as well as the ones that are specific to individual devices. In order to support this type of network management, static policies have to be set up across the networks using middle box technologies like firewalls, network policy servers, authentication servers etc. These middle boxes control the activity of the devices connected to the network using the policies but are often home for a huge number policies that complicate the network setup and make policy management a herculean task.

In recent years, the development of Software Defined Networking practices has made it simple and intuitive to instantiate programmable networks and automate different network functions. We propose a solution to automate the process of network policy management for mobility in a Software Defined Network. Using this approach, we can implement a solution that can dynamically carry the policies across the network as a host or device moves from one place to another in a network, without any action from the network administrator. Also, with the help of this implementation, we can automate the process of policy repair in the case of network changes or errors.

1.1 Software Defined Networking

Software Defined Networking defines a new paradigm shift in the field of computer networks. It can be explained as the separation of the control plane from the data plane. The separation of the control plane allows us to define a new approach to accessing lower level network services through a layer of abstraction. This abstraction layer will allow us to develop applications to replace network functions that binds multiple hardware vendors. From upper part of abstraction layer, the control mechanism of different hardware vendors will be same. The aim of SDN is to provide open interfaces that enable the development of software that can control the connectivity provided by a set of network resources and the flow of network traffic though them, along with possible inspection and modification of traffic that may be performed in the network.

Concepts of the Software Defined Networking include Programmability, Agility, Centrally Managed, Vendor-Agnostic, Open standards. These points brought us new network programmability options. Without the limitations of legacy networking technologies, we can design more open architectures and push innovation further. We have multiple components as a part of a network while using a Software Defined Networking approach. They include Network Applications, Controller, Data plane, Southbound Interface, and Northbound Interfaces. Figure 1.1 illustrates the different components of the Software Defined Networking architecture.

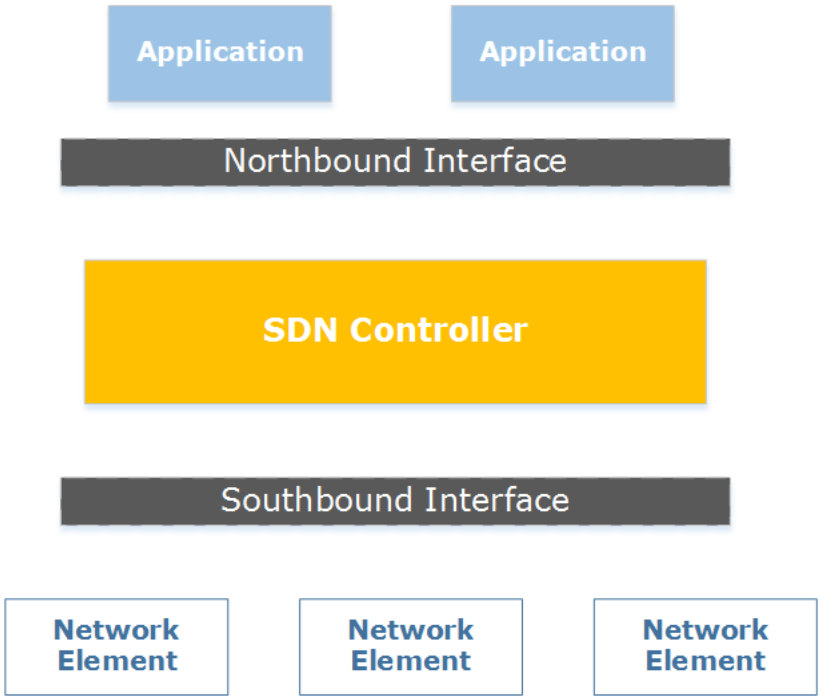

Figure 1.1: Software Defined Networking Architecture

1.2 OpenFlow Protocol

OpenFlow is a standard communications interface defined between the control and forwarding layers of SDN architecture i.e., an SDN southbound interface. OpenFlow allows direct access to and manipulation of the forwarding plane of network elements such as switches and routers, both physical and virtual (hypervisor-based). OpenFlow can be compared to the instruction set of a CPU. The OpenFlow protocol is implemented on both sides of the interface between network infrastructure devices and the SDN control software. OpenFlow uses the concept of flows to identify network traffic based on pre-defined match-action rules that can be statically

or dynamically programmed by the SDN control software. These rules are termed as flow rules. It also allows a network administrator to define how traffic should flow through the network elements based on parameters such as usage patterns, applications, and available resources. Since OpenFlow allows the network to be programmed on a per-flow basis, an OpenFlow-based SDN architecture provides extremely granular control, enabling the network to respond to real-time changes at the application, user and session levels.

1.3 Network Policies

Network policies are sets of conditions, constraints, and actions that are implemented by a network administrator to control different aspects of a network. Network policies can be viewed as rules. Each rule has a set of conditions and actions. Whenever there is a new request in the network, the conditions of the rule are compared to the properties of requests. If a match occurs between the rule and the request, the actions defined in the rule are applied to the request. When multiple network policies are configured, they are an ordered set of rules. The policy implementing system checks each request against the first rule in the list, then the second, and so on, until a match is found.

Different types of policies like User Policies, IT Policies, and Security policies are generally defined in a network. User Policies define what users can and must do to use the network. It defines what limitations are put on users to keep the network

secure such as whether they can access specific websites, types of network services they can use and how they can access data. Some examples of such policies include internet usage, VPN and remote access, firewall policies, access control lists etc.

IT and Security Policies include general policies for the IT department which are intended to keep the network secure and stable. Some examples of such policies include intrusion detection, containment and removal, identity management, wireless, router and switch security, DMZ policy, network management, network configuration and maintenance policies etc.

While using a Software Define Networking approach, these policies can be defined as flow rules over different network elements in the network.

1.4 Mobility

Prior to the advent of wireless networks, all the devices connected to a network or provisioning a network were fixed and immobile. With the invention of Wi-Fi and other wireless networking technologies, people started moving towards a more agile system of networks that allows users to move from place to place in a network while keeping them continuously connected to the network. With the implementation of protocols like DHCP (Dynamic Host Configuration Protocol) in wireless networks, the network configurations of the user devices are changed automatically without any user intervention as he moves from one part of the network to the other. This illustrates the concept of mobility in computer networks.

Migration in a network can be defined as a scenario where a service or a machine is being moved from one network to another. Migration in a virtual environment can be defined as the process of moving a running virtual machine or application between different physical machines without disconnecting the client or application. Memory, storage, and network connectivity of the virtual machine are transferred from the original host machine to the destination.

1.5 Thesis Work

SDN technologies and tools allow the implementation of network policies as flow rules in a network. In this thesis, we implemented an automated solution to support different types of network management tasks in a Software Defined Network. We use the SDN concept of flows and flow rules to implement network policies over different elements of a network. By using this type of implementation, we hope to reduce the usage of middle box technologies to a large extent and hence reducing the complexity of the network and the network management procedures. Also, by integrating a graph database technology with a controller application using the SDN northbound interfaces, we implemented an automated solution to handle the network policy management tasks for events like mobility, changes and errors in a network.

We have accomplished the automated network access policy carry-over for mobile hosts within same and different domains. Furthermore, we have implemented

our solution over an orchestrated test network to measure the performance for various network loads on such policy management support across services.

1.6 Thesis Overview

This thesis has a total of 6 chapters. We present the literature survey in chapter 2. Next, we present our hypothesis and proposed research investigations in chapter 3. Chapter 3 explains about the problem we are trying to solve and our proposed solution. Chapter 4 explains about the different tools and technologies we have used to setup a sample experiment to test the performance of our solution. Also, we explain about the design and implementation of architecture, topology, and data model of our proposed solution. In chapter 5, we describe the test scenarios and performance measurements along with measurement setup and results. Chapter 6 concludes the thesis work and we describe about what else could be added to this solution as extensions and future work.

Chapter 2 – Literature Survey

Mobility has been a problem for the network administrators since the beginning of mobile technologies. While we are still working on network protocols to support IP based local and global mobility [13], we also need to focus on the network management issues that arise because of mobility. Policies that govern service access and user applications carry over between network domains through configuration changes. Such a configuration change is one of the most error-prone elements of effective network management [14]. To implement such policies, various network middlebox technologies [18] have been implemented in the network topology. But because of the increasing complexity of the networks, having multiple middleboxes in the network increases the complexity of the networks as well as the network management procedures.

With the development of Software Defined Networking paradigm [17][19], facilitation and programmatic management of networks has become a simple task. SDN can be explained as the separation of the control plane from the data plane. The separation of the control plane allows us to define a new approach to accessing lower level network services through a layer of abstraction. This abstraction layer will allow us to develop applications to replace network functions that bind multiple hardware vendors. From upper part of abstraction layer, the control mechanism of different hardware vendors will be same. The aim of SDN is to provide open interfaces that

enable the development of software that can control the connectivity provided by a set of network resources and the flow of network traffic though them, along with possible inspection and modification of traffic that may be performed in the network.

Software defined networking is based on previous network control systems such as RCP [20], 4D [21], and Ethane [19]. Recent work has introduced the implementation of southbound and northbound interfaces. The southbound interface can be referred as the interface or protocol between programmable switches (SDN-capable switches) and the SDN controller. The northbound interface refers to the interface above the SDN controller that interacts with applications to make decisions about the underlying network. OpenFlow [22] is one of the common southbound protocols. Many vendors currently support and implement in their switching hardware that is available in the market.

Although there has been lots of research in defining and implementing a southbound API, there has been relatively less effort towards a network management solution that uses the northbound interfaces [23]. Our solution provides a mechanism to implement some of such network management tasks.

In our experiment setup, we have used RYU SDN Framework [24] as our controller platform because of its easy to use and well documented python API. We also implemented a graph database Neo4j and a SQL like querying system to programmatically interact with the database [10][11][15][16]. Furthermore, we have used a network prototyping system, mininet [25] to implement and test our solution. To

implement mobility in a Software Defined Network, we have considered implementations like OFHIP [26][27] but, later implemented controller applications to make use of OpenFlow control messages to identify and monitor network changes.

Chapter 3 – Hypothesis

To achieve automation of policy carry-over for mobility in a Software Defined Network, a controller application that uses the northbound interfaces to detect changes in the network is needed. Once a change in the network is detected, the application has to respond with the respective actions. To achieve this, we implement a central repository or a database that stores the information about the network changes and the policies in the underlying network. With this information available, whenever the controller recognizes a new change or a policy error in a network, the application can retrieve the respective policy from the central repository, modify it to be compatible with the new change in the network and push it onto the network.

So, When a host moves from one network to another or when the controller recognizes a change in the network that is in violation with a previously existing policy, the controller recognizes this change and retrieves the policies related to that specific device and responds with the necessary changes to the new network without the intervention of the network administrator so that the policies assigned to that specific devices are obliged.

3.1 Problem Statement

Policies are defined in a network to ensure the network is secure and stable, avoid misuse of networks and protect the network and the connected devices from a

wide range of threats, attacks and malicious activity. In the case of legacy networks, these policies are defined at multiple points in the network as access control lists at routers, policies while using a middle box technologies like firewalls and intrusion detection systems, policies at network policy servers or access and authentication servers. These hardware systems have the overhead of enforcing the policies defined by the network administrator over a large network form a single point and hence increasing the complexity of the network as well as creating limitations in automatically handling changes in the network. Also, to support these type of policies for mobile devices in the network, these policies have to be replicated and implemented at multiple points in the network so that when a device hops from one part to another part of the network, the policies are already in place so that there are no inconsistencies in the network.

If we consider implementation of such scenario in a Software Defined Network, the function of the middle box technologies can be replaced by the north bound applications that run over the centralized controller to implement a specific function over the network. In this case, the SDN approach would eliminate the need for having multiple middle box technologies in place at different points of the network and implement complex policies that complicates the network administrator's policy management tasks. But, we need to figure out a solution that can automate the process of policy repair and dynamic policy carry over for mobility or changes in the Software Defined Network.

3.2 Proposed Solution

We propose a solution that uses the Software Defined Networking approach. Using the principles of SDN, the control plane can be decoupled from the data plane, hence giving us an ability to manage the network from a centralized location. This centralized location is what is defined as an SDN controller which is capable of handling multiple networks connected to it using a southbound interface like OpenFlow protocol. The SDN control plane gives us a chance to programmatically handle the forwarding and network management tasks of the underlying networks. This approach drastically decreases the complexity of the network and since all the decision making tasks are dealt by the controller, a controller application can be developed that is capable of implementing a network function over the underlying network. Hence, this eliminates the need for software or hardware middle boxes in the network.

To achieve automation of policy management in the network, we propose having a central repository which acts as a storage point for the details about the underlying network and the policies defined over different networks and sub networks in a domain. We implement a northbound application that runs over different controllers in the network, to handle the following tasks:

- Survey the entire underlying network and store the details of the network onto the database.

- Gathering different policies defined by the network administrator for a specific device or the entire network of the multiple networks that are attached to the controller.
- Pushing these policies as metadata for the previously available node or network information on the central repository (database).
- Monitoring and keeping track of different changes happening in the network.
- In case of any change in the network, retrieving the desired policy information from the central repository and making necessary modifications to the network.
- Once the necessary modifications are made, update the network and policy information on the central repository (database).

Chapter 4 – Experiment Setup

To implement a highly efficient solution to support mobility and policy repair in a Software Defined Network, we have to design a use case for the experiment and create a testbed where the solution can be implemented to handle different policy management tasks while different scenarios of network changes are simulated.

4.1 Tools

The experiment setup is created using virtualization technologies to simulate the hosts and network elements. We used Mininet to generate the required network topologies. We used Open vSwitch as the software switch in the topology that supports different versions of OpenFlow as the southbound interface and RYU SDN Framework for the controller and the controller applications. We chose to implement the central repository function using a graph database with Neo4j as a graph database eliminates the need for indexing and complex querying process like recursive searching for path traversal. Hence, making improving the space efficiency and improving the processing speed of the application.

4.1.1 Mininet

Mininet is a network emulator which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support different versions of OpenFlow protocol for highly flexible custom routing and Software-Defined Networking.

Every operating system virtualizes computing resources using a process abstraction. Mininet uses process-based virtualization to run many switches and hosts on a single OS kernel. Linux has supported network namespaces, a lightweight virtualization feature that provides individual processes with separate network interfaces, routing tables, and ARP tables. The full Linux controller architecture adds process and user namespaces, and CPU and memory limits to provide full OS-level virtualization, but Mininet does not require these additional features. Mininet can create kernel or user-space OpenFlow switches, controllers to control the switches, and hosts to communicate over the simulated network. Mininet connects switches and hosts using virtual Ethernet pairs. Mininet's code is almost entirely Python, and hence it is easy for us to implement a python application that can interact with Mininet using its python API.

Mininet:

- Provides a simple and inexpensive network testbed for developing OpenFlow applications.

- Enables complex topology testing, without the need to wire up a physical network.
- Supports arbitrary custom topologies, and includes a basic set of parameterized topologies.
- Provides a straightforward and extensible Python API for network creation and experimentation.

Mininet networks run real code including standard Unix/Linux network applications as well as the real Linux kernel and network stack. Because of this, the code we develop and test on Mininet, for an OpenFlow controller, modified switch, or host, can move to a real system with minimal changes, for real-world testing, performance evaluation, and deployment. Importantly this means that a design that works in Mininet can usually move directly to hardware switches for line-rate packet forwarding.

4.1.2 Open vSwitch

Open vSwitch is an open source production quality, multilayer virtual switch. It can operate both as a software switch running within the hypervisor, and as the control stack for switching silicon. It has been ported to multiple virtualization platforms and switching chipsets. Open vSwitch took Linux bridge module as a foundation and fixed its bugs and added more features. It is developed with C language and it is using the Apache License 2.0. Open vSwitch has a kernel module

and uses a database daemon ovsdb. Open vSwitch can also operate, entirely in user space, without assistance from a kernel module. This user space implementation should be easier to port than the kernel-based switch. It is also capable of OpenFlow version 1.0 and higher while also supporting a learning legacy switch mode when required. To provide support for different switches that are using different OpenFlow protocol versions, we have developed our application to be able to interact with OpenFlow protocols 1.0 to 1.3 on the controller end of the setup.

4.1.2 RYU SDN Framework

Ryu is an open source component-based Software Defined Networking framework. Ryu provides software components with well-defined API that make it easy for developers to create new network management and control applications. Ryu supports various protocols for managing network devices, such as OpenFlow, Netconf, OF-config, etc. About OpenFlow, Ryu supports fully 1.0, 1.2, 1.3, 1.4 and other Nicira Extensions. Ryu is fully written in Python.

Ryu SDN framework provides us with built in API to support applications for a learning switch, router, firewall and other quality of service features. Ryu also supports a REST API. Representational state transfer (REST) is an abstraction of the World Wide Web architecture. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the

constraints upon their interaction with other components, and their interpretation of significant data elements.

Since Ryu is completely written in python, we make use of the python call back functions for the controller to pass a call to the northbound application whenever a desired event occurs in the underlying network. A callback function is passed as an argument in another method and which is invoked after some kind of event. The 'call back' nature of the argument is that, once its parent method completes, the function which this argument represents is then called; that is to say that the parent method 'calls back' and executes the method provided as an argument.

4.1.3 Graph Databases and Neo4j

A graph database uses graph like structures with nodes, edges, and properties to represent and store data. A graph database provides index-free adjacency. This means that every element contains a direct link to its adjacent elements and no indexing is necessary. A relational database is faster while operating on huge records as it stores indexes for every record and hence its complexity and large storage requirements. Since the graph databases do not need any indexes, it would require less storage. Also, when using a relational database, recursively searching across tables using joins would make the querying a complex and a slow process. Since the graph databases have their basis as relationships, following paths and relationships to get the desired results would be quick and less complex process.

Neo4j is an open-source NoSQL graph database implemented in java and scala. Neo4j implements the Property Graph Model down to the storage level. As opposed to graph processing or in-memory libraries, Neo4j provides full database characteristics including ACID (Atomicity, Consistency, Isolation, and Durability) compliance, cluster support, and runtime failover, making it suitable to use graph data in production scenarios. In Neo4j, everything is stored in form of either an edge, a node or an attribute. Each node and edge can have any number of attributes. Both the nodes and edges can be labelled. Labeling is useful, because you can narrow down your searching area using the labels. In the previous versions of Neo4j node indexing was supported. But the current version does not have any such provision.

Neo4j graph database has the following advantages.

- Materializing of relationships at creation time, resulting in no penalties for runtime queries.
- Constant time traversals for relationships in the graph both in depth and in breadth due to double-linking on storage level between nodes and relationships.
- All relationships in Neo4j are equally important and fast, making it possible to materialize and use new relationships later on to “shortcut” and speed up the domain data when new needs arise.
- Compact storage and memory caching for graphs, resulting in efficient scale up and billions of nodes in one database on moderate hardware.

Neo4j uses a new type of querying language to facilitate an interface for the user to create and retrieve data called Cypher. Cypher is a declarative, SQL inspired language for describing patterns in graphs. It allows us to describe what we want to select, insert, update or delete from a graph database without requiring us to describe exactly how to do it.

To interact with the Neo4j database using the Cypher query language through a northbound application written in python, we use a python library Py2neo. Py2neo is a simple and pragmatic Python library that provides access to the popular graph database Neo4j via its RESTful web service interface. With no external dependencies, installation is straightforward and getting started with coding is easy. The library is actively maintained on GitHub, regularly updated in the Python Package Index and is built uniquely for Neo4j in close association with its team and community.

4.2 Solution Architecture

The architecture of our solution has Neo4j graph database as a central repository. The northbound application that resides over the SDN controller talks to Neo4j using a python library Py2neo and cypher query language. The northbound application talks to the controller using REST api and python callback functions. RYU SDN framework is used as the SDN controller and runs a firewall application that allows the switches to work with firewall rules that are implemented as flow rules. This firewall application is programmed to support different versions of the

OpenFlow protocol (1.0, 1.1, 1.2, and 1.3). The application adjusts itself to the protocol version that is compatible with the switch in an automated manner without any input from the network administrator. The network is generated using Mininet network emulator. Mininet uses Open vSwitch to create virtual switches and uses Linux containers to simulate hosts. The network administrator can choose the OpenFlow protocol version that is implemented on a switch created with Mininet by using Open vSwitch configuration commands. The controller has a northbound application that keeps track of the events in the network and exchanges data with a centralized database (Neo4j graph database). Figure 4.1 shows the Policy Carry-over solution architecture as implemented.

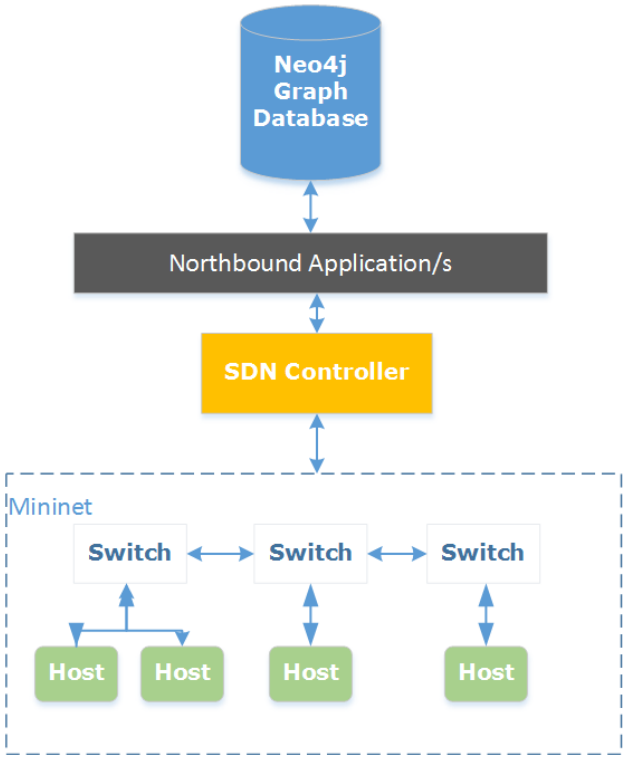

Figure 4.1: Policy Carry-Over Solution Architecture

4.2.1 Topology Setup Using Mininet

The above mentioned topology is simulated in Mininet using the following steps.

- Start mininet with a linear topology

  $ sudo mn --topo=linear --controller=remote,ip=127.0.0.1,port=6633

- Create a new host

  mininet> py net.addHost(‘h3’)

- Add link between the new host and the switch

  mininet> py net.addLink(s1, net.get('h3'))

- The addLink() function creates a new link by creating a new interface on the switch and linking it to the interface on the host. Now, to attach the newly created switch interface to the switch

  mininet> py s1.attach(‘s1-eth3’)

- Adding an IP address to the newly created host

  mininet> py net.get(‘h3’).setIP(‘10.0.0.3’)

4.3 Data Model

The policy carry-over database stores information about the network policies and the topology information like

- Switch datapath id

- Host IP address
- Hosr MAC addresses
- Flow rules/policies that are being implemented in the networks connected to the contoller.

The database stores the data as properties of nodes and relationships of the graph that represents the topology of the network that is connected to the controller. The policy carryover application constantly keeps track of the changes happening in the network and responds to those changes with necessary actions. Whenever a change is made or detected in the underlying network, the policy carry-over applications modifies the data in the database. The data model for the policy carry-over database is shown in figure 4.2.

In the policy carry-over database, every host node has the host IP address, host MAC address, datapath id of the switch it is connected to and the port on which it is connected to as metadata. The switch has the switch datapath id and the OpenFlow protocol version it is using as metadata. In a graph database, we can assign properties to the relationships or the links as well. So, the link between a switch and a host has the information about the port on which the host is connected and the link between a switch and another switch has the information about the port on switch1 and port on switch 2 as its metadata.

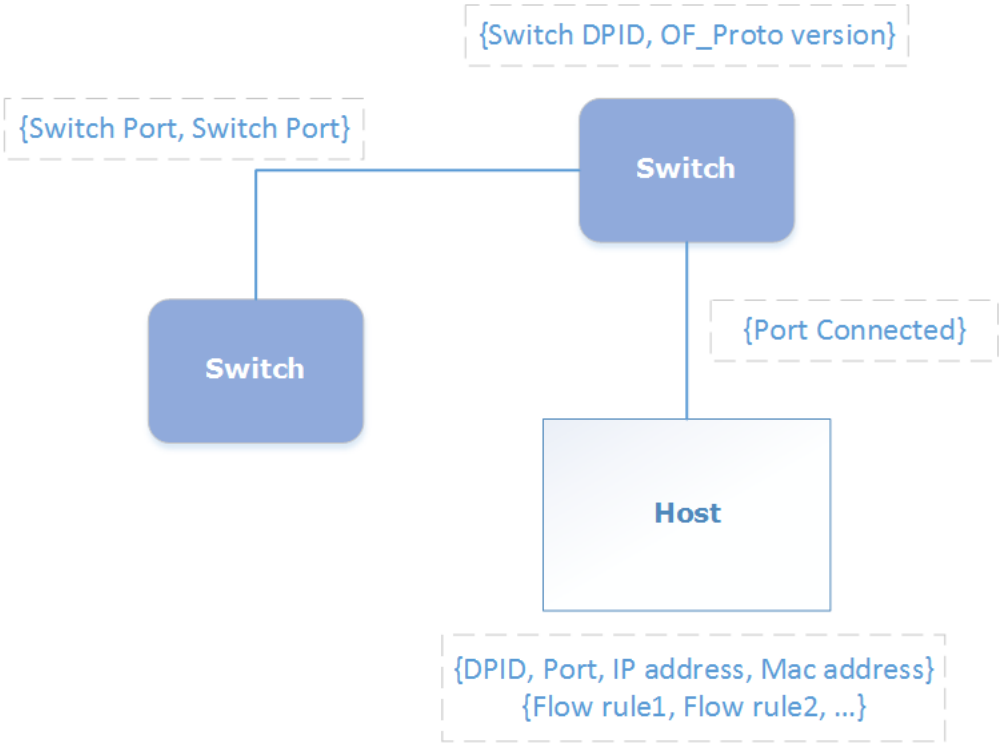

Figure 4.2: Policy Carry-Over Data model

4.4 Packet Exchange

In an SDN network, whenever a switch gets a new flow of traffic, it sends the first packet to the controller as a Packet_In message. The controller reads this message and makes a decision of what to do with that incoming flow and sends the packet back to the switch as a Packet_Out message to the switch. This decision is generally a match-action pair. The controller can also send a Flow_Mod message along with the Packet_Out message so that all the rest of the packets of that flow automatically are forwards according to the flow rule. The switch stores these flow rules in a table called flow tables. A flow rule stays in the flow table till it expires after a certain timeout period or when it is deleted by the network administrator. If the

switch gets a new incoming flow, it matches it against all the flow rules in the flow table. If a match is found, the corresponding action is taken and the communication to the controller is not initiated. If a match is not found, the communication to the controller is initiated.

4.4.1 Controller - Switch

- Upon session establishment, the controller sends an OFPT_FEATURES_REQUEST message. The switch responds with an OFPT_FEATURES_REPLY message. This reply message contains details about the switch like switch datapath id, max packets buffered, max number of tables supported etc.
- The controller is able to set and query configuration parameters in the switch with the OFPT_SET_CONFIG and OFPT_GET_CONFIG_REQUEST messages, respectively.
- The switch responds to a configuration request with an OFPT_GET_CONFIG_REPLY message.
- Modifications to a flow table from the controller are done with the OFPT_FLOW_MOD message.
- Whenever the switch receives a new flow, it checks its flow tables for a flow rule that describes what to do with that particular flow. If a matching flow rule

is not found, the switch sends an OFPT_PACKET_IN message to the controller along with the details of the first packet of the flow.

- The switch takes a look at the Packet_In message and responds with an action for that specific packet by sending a OFPT_PACKET_OUT message.
- The controller and switches exchange many other types of packets along with OFPT_Hello messages that act as keep alive messages.

4.5 Northbound Application

In software-defined networking, the control plane of the network is decoupled from the underlying infrastructure. In many scenarios, the control plane is consolidated into a centralized controller that uses the OpenFlow protocol, or alternate communication methods, to control each node and traffic flow on the network. Applications called northbound applications sit to the top of the controller. The northbound API presents a network abstraction interface to the applications and management systems at the top of the SDN stack. The information from these applications is passed along through a southbound interface. The southbound interface allows a controller to define the behavior of switches at the bottom of the SDN "stack”.

The Policy_CarryOver application relies on the messages exchanged between the controller and the switch to keep track of the network events. The following process takes place to publish the switch and host details onto the database.

1. Whenever there's a new host connected to the network and is trying to send some traffic out onto the network, the switch sends a Packet_In message to the controller. The northbound application reads all the Packet_In messages received by the controller, fetches the switch and source and destination details from the header fields.
2. The application then checks for any existing records with the newly found switch and host information on the database by sending a query to the database. If any record is found, the application ignores the newly found data. If the database query resulted in no records, the application then publishes new records onto the database.
3. Whenever a network admin pushes a policy onto the network or a specific device, the northbound application fetches this information from the controller and retrieves the policy information and publishes it onto the database as a property of the respective network element.

To support the above described process in the controller application, we use the python callback functions. In computer programming, a callback is a piece of executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at some convenient time. The invocation may be immediate as in a synchronous callback or it might happen at later time, as in an asynchronous callback. In our case, we use a synchronous callback from the

controller application to the controller when different events occur during the experiment flow.

4.6 Mobility

Mobility in network describes the process of a host or a device moving from place to place and hence transitioning from one network to another during the process. When a network element is mobile, it often disconnects from one network and joins another network. This might render the policies defined for that specific device in the previous network obsolete as the device has moved to a different network. To address this issue, the network administrator has to either repeat the procedures of define policies in the new network or define policies over the complete domain. Doing either of them would only result in redundancy or increased complexity. So, we came up with an automated solution that can seamlessly carry over the policies along with the network changes.

4.6.1 Mobility Using Mininet

To simulate mobility in the networks using Mininet, we used the following procedure.

1. Setup a topology and create a new host as described in the topology section.
2. To move the host to a new network, we first create a connection between the host and another switch.

mininet> py net.addLink(s2, net.get(‘h3’))

mininet> py s2.attach(‘s2-eth3’)

3. The above step creates a new interface on host h3 with a new MAC address. To simulate the same MAC addressing moving from first network to the second, we copy the MAC address of the first interface and set it as the MAC address of the second inteface.
    1. To fetch the MAC address of the first interface

       mininet> py net.get(‘h3’).intf(‘h3-eth0’).MAC()
    2. To set the MAC address of the second interface

       mininet> py net.get(‘h3’).intf(‘h3-eth1’).setMAC(‘xx:xx:xx:xx:xx:xx’)
4. Disabling the link between host h3 and switch s1

   mininet> py net.configLinkStatus(‘h3’,’s1’,’down’)
5. Setting an IP to the new interface of host h3

   mininet> py net.get(‘h3’).intf(‘h3-eth1’).setIP(‘10.0.0.3’)
6. Setting host h3’s new interface as the default interface

   mininet> h3 route add -net default h3-eth1

4.7 Implementation

The experiment setup has the following elements.

- RYU SDN Controller
- Policy Carry-Over controller application
- Neo4j graph database
- Mininet

We simulate a linear topology using mininet. Mininet creates a network with OpenFlow capable Open vSwitch switches that are connected to a remote SDN controller.

The implementation of the experiment is a step by step process and is explained in the following list.

- When the first OFPT_Packet_In message from host h1 reaches the controller, the controller recognizes a new device connected to the switch.
- The controller calls the node_publish() method to invoke the "Neo4j CREATE Query" to publish the newly connected host details (MAC, IPV4, Switch_DPID, Port_on_Switch) onto the database.
- When a new policy is applied for host h1 by using CURL to the controller, the controller pushes them onto the corresponding switch.

- It also calls the flow_publish() method to accept the flows that arrive at the controller, parse the flows and write the flows to respective host properties/metadata on the database using a "Neo4j MATCH and MERGE queries".
- The host is disconnected from switch s1 by using "py net.configLinkStatus()" command in Mininet CLI to bring down the link between s1 and h1.
- The host connects to s2 by doing the following steps.
    1. "py net.addLink()" to add link between s2 and h1.
    2. "py s2.attach()" to attach newly created link to switch s2.
    3. "h1.intf().setMAC()/setIP()" to modify the MAC/IP address of the interface.
    4. "h1 route add -net default h1-eth1" to set the default exit point for host h1.
- When the host moves, the controller doesn't know that host moved and connected to s2 until it receives the first OFPT_Packet_In message.
- Upon reception of the OFPT_Packet_In message, the controller looks for an existing entry for the mac address in the database using "Neo4j MATCH query".
- If the "MATCH query" returns any results and if the "DPID" in the new Packet_In message is different from the existing DPID entry.

1. It retrieves all the flows associated with the mac address using "Neo4j MATCH query", parses and modifies them to be compatible with the new switch.
2. It also publishes the modified flow entries onto the database using "Neo4j MATCH and Merge queries".

- The controller gets the modified flow entries from the previous step to be pushed one by one onto the new switch using the controller REST API of the controller using a subprocess() to run the CURL process and hence the policies are carried over along with the mobile host.

This process represents one cycle of the experiment i.e., from the time a host connects to the network to when the network change is detected to the necessary action is issued by the northbound application and controller onto the network. This process keeps on repeating every time a new device connects or moves in a network or a network change is detected in the network.

Chapter 5 - Measurements

To evaluate the performance of the experiment, we have tested the solution implementation with multiple scenarios. Before we get into the details about the setup and the results, let's understand about our end goal and what we are trying to accomplish. Policy carry-over in the case of mobility or network changes is a complex process and for this process to work efficiently, we need to verify the time efficiency of every individual step involved in the process. So throughout the chapter, we will be discussing about the time measurements for different events in the experiment. Various key events that occur during the execution of the experiment are explained in figure 5.1. Among the events mentioned in the following figure, we believe that controller recognizing the change in the network, fetching the corresponding database entries and modifying the policies to be compatible with the change in the network and pushing them onto the corresponding switch to setup the policies play a key role in the evaluation process. Throughout the chapter, we will be discussing about various measurements for the events T4, T5, T6.

T0	Host h1 connects to switch s1	1. When the first OFPT_Packet_In message from host h1 reaches the controller, the controller recognizes a new device connected to the switch. 2. The controller calls the node_publish() method to invoke the "Neo4j CREATE Query" to publish the newly connected host details (MAC, IPV4, Switch_DPID, Port_on_Switch) onto the database.
T1	Policies corresponding to h1 are pushed	1. When a policy is applied for host h1 by using CURL to the controller, the controller pushes them onto the corresponding switch. 2. It also calls the flow_publish() method to accept the flows that arrive at the controller, parse the flows and write the flows to respective host properties/metadata on the database using a "Neo4j MATCH and MERGE queries".
T2	Host h1 is disconnected from switch s1	1. The host disconnected from switch s1 by using "py net.configLinkStatus()" command in Mininet CLI to bring down the link between s1 and h1
T3	Host h1 connects to switch s2	1. The host connects to s2 by doing the following steps. a) "py net.addLink()" to add link between s2 and h1 b) "py s2.attach()" to attach newly created link to switch s2 c) " h1.intf().setMAC()/setIP()" to modify the MAC/IP address of the interface. d) "h1 route add -net default h1-eth1" to set the default exit point for host h1
T4	Controller recognizes that the node has moved	1. When the host moves, the controller doesn't know that host moved and connected to s2 until it receives the first OFPT_Packet_In message. 2. Upon reception of the OFPT_Packet_In message, the controller looks for an existing entry for the mac address in the database using "Neo4j MATCH query".
T5	Controller gets the corresponding policies from the database	1. If the "MATCH query" returns any results and if the "dpid" in the new packet_in message is different from the existing dpid entry, a) it retrieves all the flows associated with the mac address using "Neo4j MATCH query", parses and modifies them to be compatible with the new switch. b) It also publishes the modified flow entries onto the database using "Neo4j MATCH and Merge queries".
T6	Controller pushes the flows for h1 one after another onto switch s2	1. The controller gets the modified flow entries from the previous step to be pushed one by one onto the new switch using the controller REST api of the controller using a subprocess() to run the CURL process and hence the policies are carried over along with the mobile host.

Figure 5.1: Policy Carry-Over Timeline

5.1 Measurement Setup

To gather time measurements as discussed in the previous section, we implemented the python callback functions in the application code so that whenever an event occurs, the application sends a call back to the controller and the controller records the time of the event based on this call. The time results we get from the controller as it responds to the callback functions are saved into a spreadsheet and we use data from this spreadsheet to evaluate the performance of the experiment.

5.2 Assumptions and Constraining Parameters

Mobility in a network is dependent on the person carrying the device and moving from one part to another of the network and we cannot simulate the exact scenario using a program. To simulate mobility, we made use of Mininet's python API to disable a link and move it to a new network. During the measurement or evaluation phase, this process is automated using python scripts and we excluded the time taken for moving a device from one place to another from our evaluation factors.

Also, comparing our automated policy management process to a process that is manually handled by a network engineering team wouldn't be a fair comparison as the latter is dependent on the person handling that task and cannot be efficiently quantified.

Implementing our solution in a physical environment with multiple networks being controlled by our SDN controller needs a lot of hardware resources. Hence, we implemented it in a virtual hypervisor based environment with Mininet simulating the network elements including the hosts. Also, the database, controller and the simulated network reside in the same host and the time taken for the controller to reach the database and time taken for other requests might vary if deployed in a physical environment. So, we couldn't compare our results to any real time solutions or come up with claims about the efficiency of our solution.

5.3 Scenarios and Performance Measurements

In order to evaluate the efficiency of the implementation method, we deployed and tested the experiment using multiple scenarios. These scenarios describe a change in the implementation and experiment setup. The following table, Table 1 explains the changes in the experiment across different scenarios.

Table 5.1: Policy Carry-Over measurement scenarios

SNo	Topology	Switches	Hosts	Policies	Measurements
1	Linear	2	3	• Allow Host1 - Host2	T4,T5,T6

	network			communication • Allow Host1 - Host3 communication • Deny Host2 - Host3 communication	
2	Linear network	3	3	• Allow Host1 - Host2 communication • Allow Host1 - Host3 communication • Deny Host2 - Host3 communication	(T4,T5,T6) X 2

In the experiment, to demonstrate mobility, the host moves from one network to another i.e., from one switch to another. So, in the first scenario, there is only a single set of measurements as the host can only move from switch1 to switch2. As, the numbers of networks increase, we tested the mobility across the networks and every time a host moves from one network to another, there is a new set of measurements.

5.3.1 Policy Retrieval

Whenever the policy carry-over application detects a change in the network, it retrieves the corresponding policies from the central database to modify them so that they are compatible with the new change and then pushes these modified policies onto the network. Figure 5.2 represents a graph plotted with policy retrieval time in seconds on the Y-axis and the number of experiment on the X-axis.

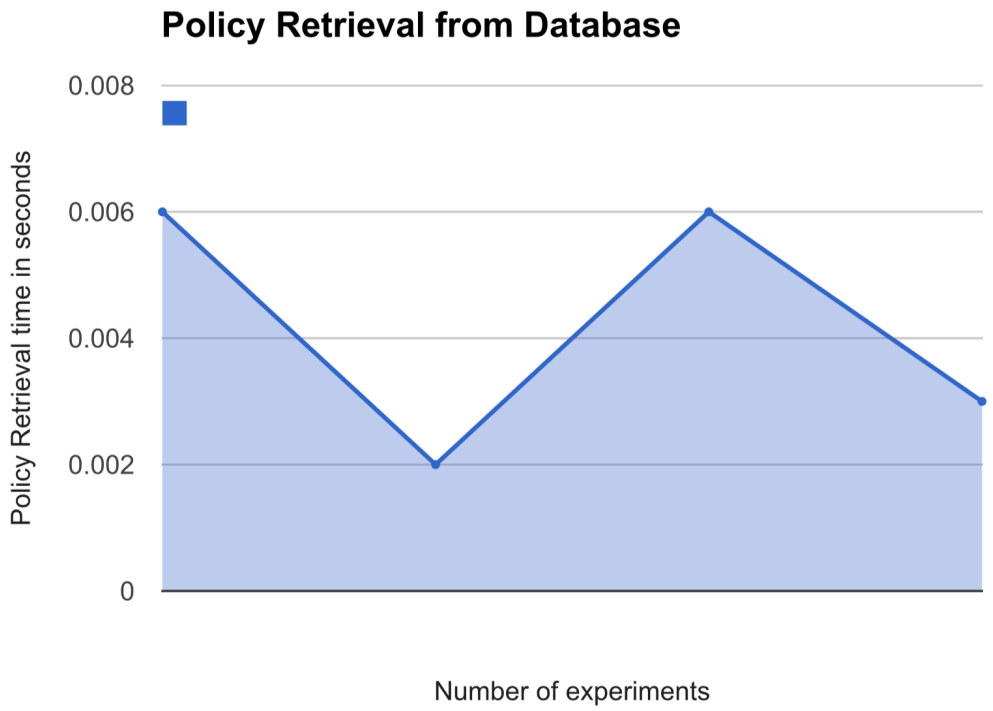

Figure 2.2: Policy retrieval from the database

5.3.2 Policy Modification and Repair

As mentioned in the previous section, whenever there is a change in the network, the controller application detects the change and takes the necessary action to maintain the stability of the network. In this case, the action the controller application takes is to modify the previously existing policy rule to be compatible with the new change and push it onto the network. Figure 5.3 represents a graph plotted with time taken to modify and push flows onto the network on Y-axis and number of experiments on X-axis.

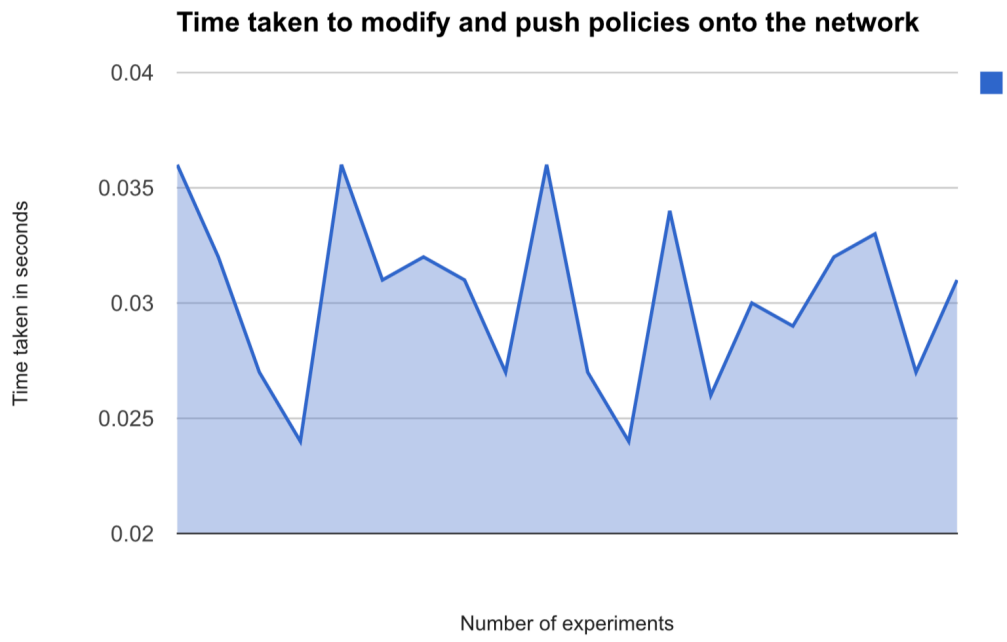

Figure 5.3: Time taken to repair the policies

5.3.3 Packet Loss

Mobility in the networks leads to some disturbance to all the connections to/from the mobile device. This might be because of a delay in the policy repair or due to delay in connecting to the new network. Figure 5.4 represents a graph plotted with packet loss on Y-axis and Number of experiments on X-axis. The graph also shows a comparison between the packet loss for the following cases.

- Packet loss during transmission from the mobile host to another device
- Packet loss during transmission from a device in the network to the mobile host.

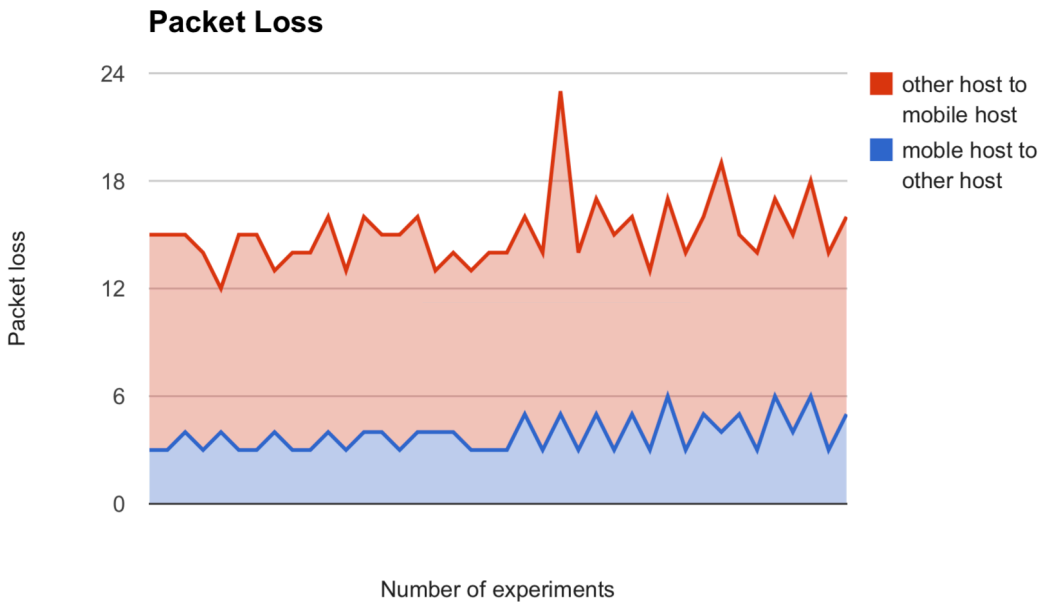

Figure 5.4: Packet loss during mobility

The data packets lost during the time taken by the controller to repair the policies or carry over the policies can be recovered using the following methods.

- If the traffic that is being received during the time of packet loss is TCP traffic, the TCP protocol takes care of detecting the packet loss and retransmission of the lost packets.
- If the traffic that is being transmitted is UDP, the application might or might not need the retransmission of the lost packets.
- If the reception of the data packets is critical to the host or the application running on the mobile host, we can use the SDN features to recognize the type of traffic being transmitted in the network based on the protocol being used or the packet headers and pay load and buffers can be implemented to store the packets that are being lost and re transmit as the connection to the mobile device is re established.

Chapter 6 - Conclusion and Future Work

We demonstrated a novel approach to automate the process of policy carryover during mobility in Software Defined Networks. The reason behind this approach is to improve the efficiency of the network management tasks by automating them as well as reduce the chance of human error. We could achieve the desired efficiency by having a centralized data collection point that is used to collect details of network changes and policies issued. The Policy_CarryOver controller application keeps track of the network changes by using the SDN controller as its management point and automatically responds to the changes with necessary modification to the policies. We limited this approach to RYU SDN Framework but it could be easily implemented with other SDN controllers available in the market by porting the logic to the respective programming languages.

References

[1] SDN Architecture, Open Networking Foundation: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf

[2] OpenFlow Switch Specification: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf

[3] Callback Functions: http://en.wikipedia.org/wiki/Callback_(computer_programming)

[4] Mininet: http://mininet.org/

[5] Mininet python API reference manual : http://mininet.org/api/classmininet_1_1topo_1_1Topo.html

[6] Open vSwitch: http://openvswitch.org/

[7] RYU SDN Framework: http://osrg.github.io/ryu/

[8] RYU SDN Framework reference manual: http://osrg.github.io/ryu-book/en/Ryubook.pdf

[9] Fielding, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine. (2000). REST API: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[10] Neo4j graph database: http://neo4j.com/

[11] Neo4j Cypher query language: http://neo4j.com/developer/cypher-query-language/

[12] Neo4j python library. Py2neo: http://nigelsmall.com/py2neo/1.6/

[13] Kempf, James. "Problem statement for network-based localized mobility management (NETLMM)." (2007).

[14] Bieszczad, Andrzej, Bernard Pagurek, and Tony White. "Mobile agents for network management." Communications Surveys & Tutorials, IEEE 1, no. 1 (1998): 2-9.

[15] Webber, Jim. "A programmatic introduction to Neo4j." In Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity, pp. 217-218. ACM. (2012).

[16] Holzschuher, Florian, and René Peinl. "Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j." In Proceedings of the Joint EDBT/ICDT 2013 Workshops, pp. 195-204. ACM. (2013).

[17] McKeown, Nick. "Software-defined networking." *INFOCOM keynote talk* (2009).

[18] Carpenter, Brian, and Scott Brim. *Middleboxes: Taxonomy and issues*. RFC 3234, February, (2002).

[19] Casado, Martin, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. "Ethane: Taking control of the enterprise." *ACM SIGCOMM Computer Communication Review* 37, no. 4 (2007): 1-12.

[20] Feamster, Nick, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Jacobus Van Der Merwe. "The case for separating routing from routers." In *Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pp. 5-12. ACM, (2004).

[21] Greenberg, Albert, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. "A clean slate 4D

approach to network control and management." *ACM SIGCOMM Computer Communication Review* 35, no. 5 (2005): 41-54.

[22] McKeown, Nick, 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, no. 2 (2008): 69-74.

[23] Kim, Hyojoon, and Nick Feamster. "Improving network management with software defined networking." *Communications Magazine, IEEE* 51, no. 2 (2013): 114-119.

[24] Shalimov, Alexander, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, and Ruslan Smeliansky. "Advanced study of SDN/OpenFlow controllers." In Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia, p. 1. ACM, (2013).

[25] Lantz, Bob, 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*, p. 19. ACM, (2010).

[26] Namal, Suneth, Ijaz Ahmad, Andrei Gurtov, and Mika Ylianttila. "Enabling Secure Mobility with OpenFlow." In Future Networks and Services (SDN4FNS), 2013 IEEE SDN for, pp. 1-5. IEEE, (2013).

[27] Erickson, David, Glen Gibb, Brandon Heller, David Underhill, Jad Naous, Guido Appenzeller, Guru Parulkar et al. "A demonstration of virtual machine mobility in an OpenFlow network." (2008).