University of Thessaly
Department of computer engineering

Virtual Network management and control SDN parenting in a wireless testbed environment

Διαχείριση και Έλεγχος εικονικών δικτύων χρησιμοποιώντας την SDN μεθοδολογία σε ασύρματη πειραματική πλατφόρμα

ΔΗΜΟΠΟΥΛΟΣ ΔΗΜΗΤΡΙΟΣ

Περίληψη

Αυτή η διπλωματική εργασία αναφέρεται στη διαχείριση και τον έλεγχο εικονικών δικτύων χρησιμοποιώντας την SDN (Software Defined Networking) μεθοδολογία. Συγκεκριμένα, θα μελετήσουμε δίκτυα τύπου SDN, καθώς και την αρχιτεκτονική τους, και θα δούμε τα οφέλη και την καινοτομία που προσφέρει η SDN μεθοδολογία στον χώρο των δικτύων. Θα μιλήσουμε για το OpenFlow πρωτόκολλο το οποίο είναι απαραίτητο εργαλείο για τα SDN δίκτυα, καθώς και τον OpenDaylight, τον κεντρικό Controller του δικτύου μας που ρυθμίζει ουσιαστικά την κίνηση και την ροή των πακέτων. Στη συνέχεια, θα εξετάσουμε το Mininet, έναν εξομοιωτή δικτύου, που θα μας βοηθήσει να δημιουργήσουμε μια τοπολογία εικονικού δικτύου πάνω στην οποία θα εκτελέσουμε το πείραμά μας. Τέλος, τρεχουμε το πείραμά μας το οποίο στο πρώτο σκέλος πετυχαίνει Load Balancing μεταξύ των host του δικτύου μας χρησιμοποιώντας το αντίστοιχο service του OpenDaylight Controller, και έπειτα μια JAVA εφαρμογή η οποία επικοινωνεί κατευθείαν με τις συσκευές του δικτύου μας και ρυθμίζει την ροή των πακέτων στο δίκτυό μας.

Contents

PART I

1. Introduction

1.1 Virtual Networking

Virtual Networking is a technology that facilitates the control of one or more remotely located computers or servers over the Internet. A Virtual Network is an interconnected group of networks that appears as one large network to the user. The users of a virtual network are able to transparently communicate locally and remotely across similar and not similar networks through a simple user interface.

Virtual Networking also facilitates consolidation of different services and devices on a single hardware platform. The centralization of control reduces the cost and the complexity of maintaining hardware and software compared with many separate devices in different geographical locations. This means that the users of a virtual network can install device drivers, perform tests and generally solve problems on the remote machines from a single location. But, to take advantage of this technology it is necessarily to install virtual networking software on the remote computers or servers. Many vendors, including Microsoft, VMWare and VM VirtualBox, offer virtual networking software. Some vendors offer comprehensive virtual networking services, allowing business network administrators to outsource resources to the vendor.

Nowadays, many companies are using virtual networks in order to cut down on costs an increase performance and security. Virtual Networks can connect remote users to a larger centralized network. This fact, along with the universal appeal of the Internet, has made virtual technology more accessible and financially viable for large and small corporations. The result is remote access that is quicker, more secure and wider in scope.

On the other hand, one major limitation of traditional Virtual Networks is that they are point-to-point, and do not tend to support or connect broadcast domains. The meaning of this is that communication, software and networking may not be fully supported exactly as they would on a real LAN. Next, we analyze Software Defined Networking (SDN) which provides virtual network services.

1.2 Software Defined Networking (SDN)

1.2.1 SDN's definition and architecture

Software defined networking (SDN) is a new approach to designing, building and generally controlling networks.

To start with, SDN enables a network administrator (controller) to shape and handle network traffic and control network services, without having to touch individually switches, bridges or routers in the forwarding plane. The goal of SDN is to allow network administrators respond quickly to changing business requirements in a flexible manner without touching the hardware of the network.

The basic idea of SDN is the separation of networking into the control and the data plane.

- The **data plane** includes all the devices (switches, routers, bridges) that allow packets' transaction from a point A to a point B.

- The **control plane** is a set of management servers that communicate with all of the different network devices of the data plane and decide the data move (traffic) through them.In few words, control plane contains the servers tha are responsible for the data management.

- The **management plane** includes methods of configuring the control plane in order to check the functionalty of its servers.

Thus, in a common network, if you want to execute some operations (e.g. on a router) regarding, control, management and forwarding plane, all these operations occur within the same device, and each node in the network operates autonomously to make its own forwarding decisions based on its local configuration. Here comes the intelligence of SDN, which separates the control plane from the forwarding plane and offloads its functions to a centralized controller

.

Specifically , with SDN we can dynamically model and shape traffic in our network depending on what we need to do. In few words , we try to reduce the hardware work and keep the intelligence of the hardware via software networking.

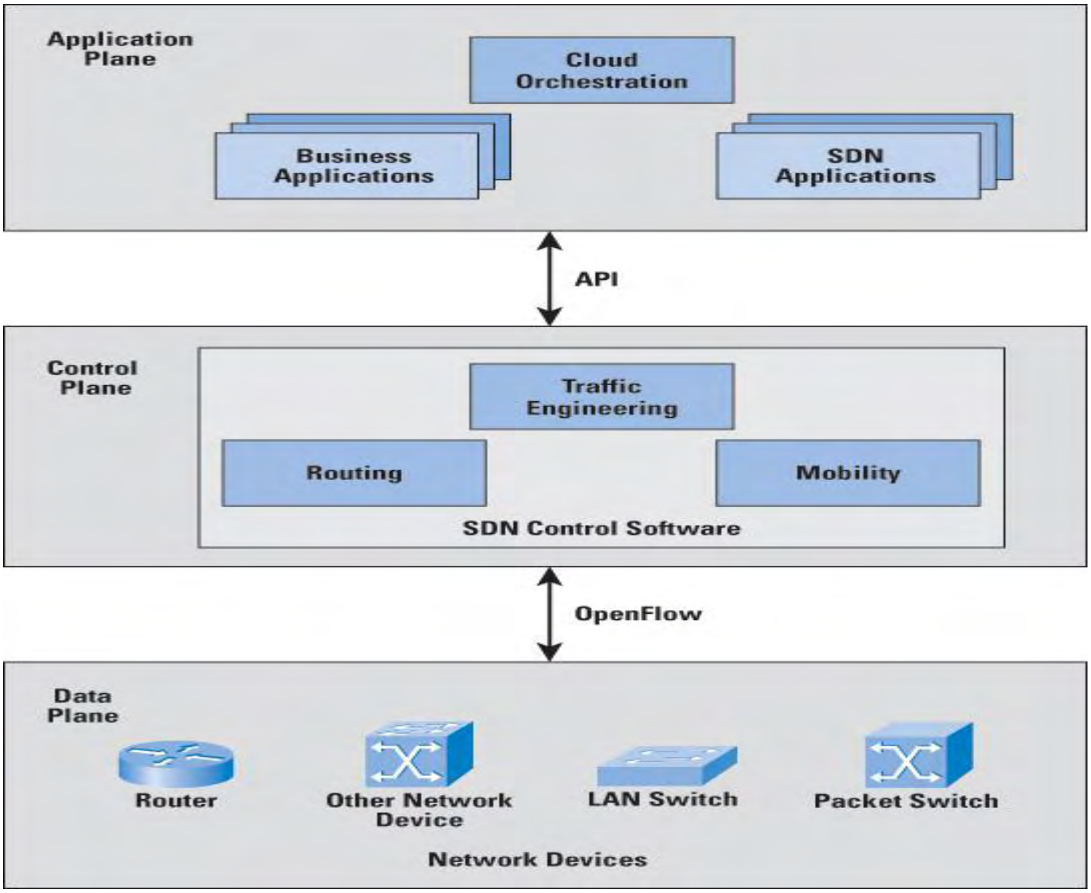


Figure 1: A three-layer SDN architecture.

Figure 1 illustrates the SDN framework, which consists of three layers. The lowest layer is the **infrastructure layer** (**data plane**). It contains the forwarding network elements. The responsibilities of the forwarding plane are mainly data forwarding, as well as monitoring local information and gathering statistics.

One layer above, we find the **control layer**, also called the **control plane**. It is responsible for programming and managing the forwarding plane. It makes use of the information provided by the forwarding plane and defines network operation and routing. It contains one or more software controllers that communicate with the forwarding network elements through standardized interfaces, which are referred to as southbound interfaces (OpenFlow protocol). OpenFlow (*see section2*), which is one of the mostly used southbound interfaces, mainly considers switches, whereas other SDN approaches consider other network elements, such as routers.

The **application layer** contains network applications that can introduce new network features, such as security and manageability, forwarding schemes or assist the control layer in the network configuration. The application layer can receive an abstracted and global view of the network from the controllers and use that information to provide appropriate guidance to the control layer. The interface between the application layer and the control layer is referred to as the northbound interface.

1.3 Programmable data plane – OpenvSwitch

As we saw SDN (Software Defined Networking) is changing the way we design and manage networks since the user is able to control and shape the traffic of the network via a centralized controller. Making computer networks more programmable enables innovation in network management and lowers the barrier to deploying new services. The main characteristic of SDN is that separates the control plane (which decides how to handle the traffic) from the data plane (which forwards traffic according to the control plane). This separation makes network virtualization possible because you're no longer executing all the command or control rules on the hardware itself.

The **data plane** enables data transfer to and from clients, handling multiple conversations through multiple protocols (OpenFlow), and manages conversations with remote peers. Data plane traffic travels through routers or switches so that the hosts of the network can communicate each other. SDN implements the control plane in software, fact that enables programmatic access to make network administration much more flexible. A network administrator can shape traffic from a centralized control console without having to touch individual switches. The administrator can change any network switch's rules when necessary with a very granular level of control.

In SDN the data plane contains all the devices of our network (switches, routers, hosts) which devices are programmable devices and can communicate with the centralized controller via OpenFlow protocol.

An **OpenFlow switch** is a software program or hardware device that forwards packets in a software-defined networking environment. OpenFlow switches are either based on the OpenFlow protocol or compatible with it. An OpenFlow switch consists of at least three basic parts:
(1) A **flow table**, with an action to tell the switch how to process the flow,
(2) A **secure channel** that connects the switch to a remote controller, allowing the packets to be sent between the controller and the switch,
(3) The **OpenFlow protocol**, with which the controller communicates the switch.

The flow table has **flow entries**. Each flow entry has an action associated with it. The three basic actions are:

(1) Forward this flow's packets to the given port or ports,
(2) Forward and encapsulate this flow's packets to a controller (Secure Channel),
(3) Drop this flow's packets. Each flow entry contains:

Header Fields	**Counters**	**Actions**

- **header fields** to match against packets
- **counters** to update for matching packet, are maintained per-table, per- row, per-port and per queue
- **actions** to apply to matching packets. Each row entry is associated with zero or more actions that dictate how the switch handles matching packets. If there are no forward actions, the packet is dropped.

Some OpenFlow software switches are NetFGPA Switch (offers line-rate performance for 4 Gigabit ports, regardless of packet size, via hardware acceleration), Open WRT(convert a cheap commercial wireless router and access point into an OpenFlow-enabled switch with a WebUI and a CLI) and **Open vSwitch**.

Open vSwitch is a production quality, multilayer virtual switch licensed under the open source Apache 2.0 license. It is designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocol (OpenFlow, NetFlow etc.). Open vSwitch can operate as a soft switch running within the hypervisor and been ported to multiple virtualization platforms and switching chipsets. It is the default switch in XenServer 6.0 and also supports VirtualBox (we use VirtualBox for our experiment). The kernel datapath is distributed with Linux and packages are available for Ubuntu, Debian and Fedora. Open vSwitch is used in multiple products and runs in many large production environments.

Below are some important Open vSwitch's **features and characteristics** that make it reasonable to use.

- Open vSwitch has support for both configuring and migrating both slow (configuration) and fast network state between instances.
- Open vSwitch supports a number of features that allow a network control system to respond and adapt as the environment changes. Open vSwitch also supports OpenFlow as a method of exporting remote access to control traffic.
- Open vSwitch provides QoS (Quality of Service) such as traffic queuing and traffic shaping and also is secure providing VLAN isolation and traffic filtering.

- Open vSwitch provides hardware integration. Open vSwitch's forwarding path is designed to be amenable to "offloading" packet processing to hardware chipsets. This allows for the Open vSwitch control path to be able to both control a pure software implementation or a hardware switch. If physical switches also expose the Open vSwitch control abstractions, both bare-metal and virtualized hosting environments can be managed using the same mechanism for automated network control.

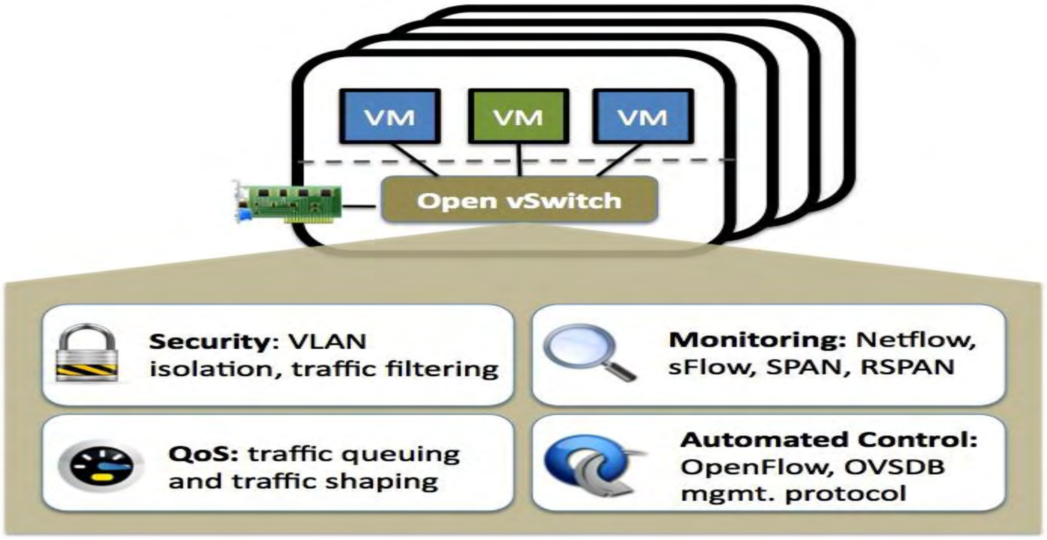

VM
VM
VM
Open vSwitch
Security: VLAN isolation, traffic filtering
Monitoring: Netflow, sFlow, SPAN, RSPAN
QoS: traffic queuing and traffic shaping
Automated Control: OpenFlow, OVSDB mgmt. protocol

Figure 2: Open vSwitch, advantages and specifications

In many ways, Open vSwitch targets a different point in the design space than previous hypervisor networking stacks, focusing on the need for automated and dynamic network control in large-scale Linux-based virtualization environments. The goal with Open vSwitch is to keep the in-kernel code as small as possible and to re-use existing subsystems when applicable.

2. OpenFlow protocol and SDN Controllers

2.1 OpenFlow Protocol- Overview

The **OpenFlow** protocol is an over-the-wire protocol such as TCP or IP with specific behaviors and specifications. In few words, OpenFlow is another dynamic approach to SDN. The word “open” in OpenFlow protocol indicates that it’s an open source protocol (an OpenFlow controller) which can be used in programming the behavior of the forwarding plane of an OpenFlow switch. OpenFlow is the first interface designed specifically for SDN, providing high performance, traffic control across network's devices.

The OpenFlow architecture consists of three basic concepts. (1) Network devices like OpenFlow switches that compose the data plane, (2) the control plane consists of one or more OpenFlow controllers, (3) a secure channel that connects the switches with the control plane.

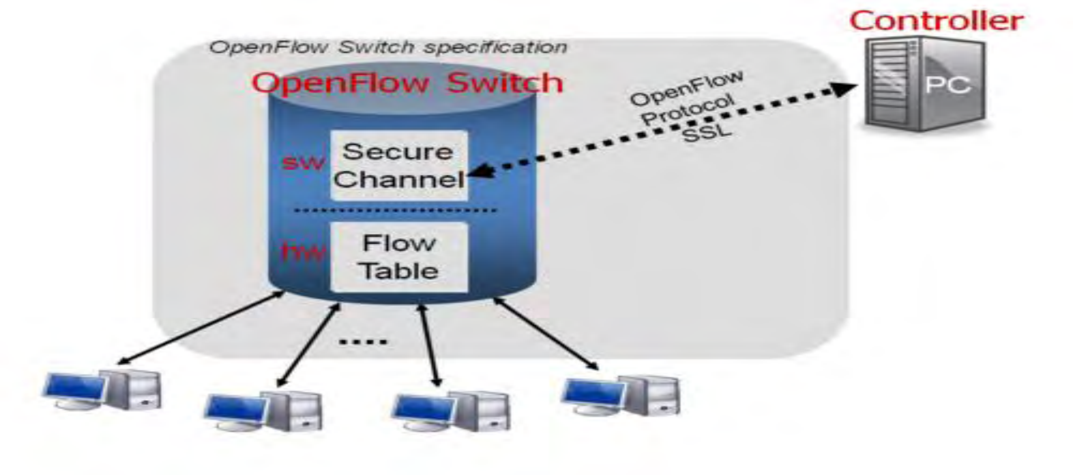
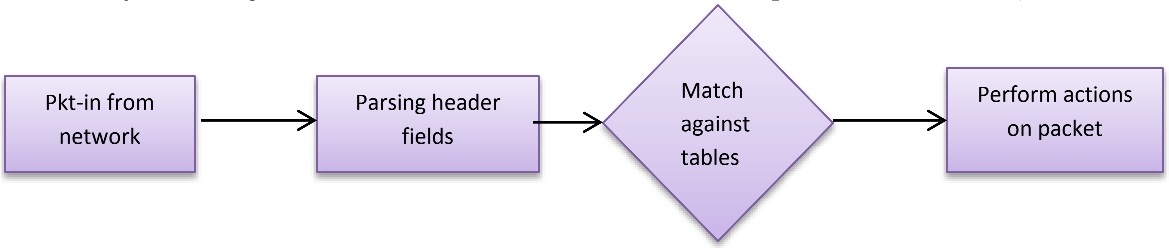


Figure 3: OpenFlow Architecture in SDN

With this interesting protocol, a centralized major controller can program and control all the physical and virtual switches in a network. Despite the fact this can be done with only one controller, a lot of companies provide high availability clusters with more than one controller. Thus, by using an OpenFlow switch all the control functions and services of a traditional switch run at the central OpenFlow controller. Nowadays, the creation of an OpenFlow switch is a simple procedure since manufacturers produce chips that can fully control the OpenFlow protocol. This can be implemented with some simple software code without high processing and memory requirements.

2.1.1 How Does OpenFlow Works?

An OpenFlow based SDN creates a physical network that separates the control plane from the data plane of the physical network. The data path portion remains on the switch while control path portions (high level routing decisions) are moved to a separate Controller. The OpenFlow switch and the Controller communicate each other via OpenFlow protocol which defines messages such as “packet received”, “send-packet-out” and “modify forwarding table”. Initially, the data path of an OpenFlow Switch presents a clean flow-table, and each flow entry contains a set of packets fields to match, plus an action. When a new packet is received by the OpenFlow switch and there is not a matching flow entry at the flow table this packet is sent to the Controller. The Controller makes the decision on how to handle this packet (it can drop it, or add a flow entry directing the switch on how to forward a similar packet in the future).



2.1.2 OpenFlow Channel

The OpenFlow channel is the interface that connects each OpenFlow switch to a controller. Through this interface, the controller configures and manages the switch, receives events from the switch and sends packets out the switch. All OpenFlow channel messages must be formatted to the OpenFlow protocol. The OpenFlow channel is encrypted using TLS or TCP.

The OpenFlow protocol supports three messages types:

1. Controller to switch messages. These messages are initiated by the controller and may or may not require a response from the switch.
2. Asynchronous messages are sent without a controller soliciting them from a switch. Switches sent asynchronous messages to the controllers to denote a packet arrival, change of the switch state or an error. The four asynchronous messages are: Packet-in, Flow-Removed, Port-status, and Error.
3. Symmetric messages, which are sent in either direction without solicitation.

The OpenFlow channel is used to exchange messages between an OpenFlow switch and an OpenFlow controller. A typical OpenFlow controller manages multiple OpenFlow channels, each one to a different OpenFlow switch. An OpenFlow switch may have one OpenFlow channel to a single controller, or multiple channels for reliability, each to a different controller.

The switch must be able to establish communication with a controller at a user-configurable IP address, using a user-specified port. If the switch knows the IP address of the controller, the switch initiates a standard TLS or TCP connection to the controller. Traffic to and from the OpenFlow channel is not run through the OpenFlow pipeline. Thus, the switch must identify incoming traffic as local before checking the flow tables.

The switch and controller may communicate through a TLS connection. The TLS connection is initiated by the switch on startup to the controller, which is located by default on TCP port 6633. The switch and controller may optionally communicate using plain TCP. The TCP connection is initiated by the switch on startup to the controller, which is located by default on TCP port 6633. When using plain TCP, it is recommended to use alternative security measures to prevent controller impersonation or other attacks on the OpenFlow channel.

2.1.3 OpenFlow uses and specifications

OpenFlow is the first standard communications interface defined between the control and forwarding layers of an SDN architecture. OpenFlow allows direct access to and manipulation of the forwarding plane of network devices such as switches and routers, both physical and virtual. Also, OpenFlow provides a standardized hook to allow researchers to run experiments without requiring vendors to expose the internal workings (hardware) of their network devices. It is also allows to easily deploy innovative routing and switching protocols in your network. It is used for a lot of applications; some of these are high-security networks or virtual machine mobility. A protocol like OpenFlow is needed to move network control out of the networking switches to logically centralized control software. The OpenFlow protocol is implemented on both sides of the interface between network devices and the SDN control software. This protocol is a key enabler for software-defined networks and currently is the only standardized SDN protocol that allows direct management and control of the forwarding plane of network devices. OpenFlow-based SDNs can be deployed on existing networks, both physical and virtual. Network devices can support OpenFlow-based forwarding. This is an easy way for enterprises and carriers to progressively introduce OpenFlow-based SDN technologies.

The first OpenFlow version, 1.0, was developed at Stanford University and was widely implemented. Switches using OpenFlow 1.0 forwarding model cannot perform more than one operation during the packet forwarding process. They must match the input port and destination MAC address in a single flow rule Version 1.1 of the OpenFlow protocol was released on February 28, 2011, and new development of the standard was managed by the Open Networking Foundation (ONF). In December 2011, the ONF board approved OpenFlow version 1.2 and published it in February 2012. OpenFlow 1.3 significantly expands the functions of the specification (version 1). **OpenFlow Version 1.3** is likely to become the stable base upon which future commercial implementations for OpenFlow will be built. The OpenFlow 1.4 Switch Specification was officially finalized last October by the Open Networking Foundation. The last version of OpenFlow is 1.5, released last December and the OpenFlow 1.5.1 version approved on March 26th 2015.

2.1.4 Comparison of OpenFlow specifications (OpenFlow v1.3)

OpenFlow 1.3 introduces new features for monitoring and operations and management. To that end, the meter table is added to the switch architecture. A meter is directly attached to a flow table entry by its meter identifier and measures the rate of packets assigned to it. A meter band may be used to rate-limit the associated packet or data rate by dropping packets when a specified rate is exceeded. Instead of dropping packets, a meter band may optionally recolor such packets by modifying their differentiated services (DS) field. Thus, simple or complex QoS frameworks can be implemented with OpenFlow 1.3 and later specifications.

The support for multiple controllers is extended. With OpenFlow 1.2, only fault management is targeted by a master scheme. With OpenFlow 1.3, arbitrary auxiliary connections can be used to supplement the connection with the master controller and the switch. Thereby, better load balancing in the control plane may be achieved. Moreover, per-connection event filtering is introduced. This allows controllers to subscribe only to message types they are interested in. For example, a controller responsible for collecting statistics about the network can be attached as the auxiliary controller and subscribes only to statistics events generated by the switches.

OpenFlow 1.3 supports IPv6 extensions headers. This includes, e.g., matching on the encrypted security payload (ESP) IPv6 header, IPv6 authentication header, or hop-by-hop IPv6 header.

2.2 OpenFlow Controllers

An **OpenFlow Controller** is an application that manages flow control in a SDN environment. Most current SDN controllers are based on the OpenFlow protocol.
The SDN controller serves as a sort of operating system (OS) for the network. All the communications between applications and devices have to go through the controller. The controller uses the OpenFlow protocol to configure network devices and choose the best paths for application traffic. Due to the fact that the network control plane is implemented in software, rather than the hardware devices, network traffic can be managed more dynamically by the user. Also, because of the software implementation, the controller facilitates network management and makes it easier to administer and integrate business applications.

Here are some of the most common OpenFlow controllers and some information about them:

NOX: is the original-first OpenFlow controller, and facilitates development of fast C++ or Python controllers on Linux.
POX: has a high-level SDN API including a topology graph and support for virtualization and is a Python based open source implementation for on Windows, Mac OS, or Linux.
BEACON: is a fast, cross-platform, modular, Java-based OpenFlow controller. Beacon is an open source implementation and claims to have been used in several research projects, networking classes, and trial deployments.
FLOODLIGHT: is an Apache-licensed, Java-based OpenFlow Controller. Floodlight supports a broad range of virtual and physical OpenFlow switches. It can manage multiple islands of OpenFlow switches, a common deployment scenario.
TREMA: is a full-stack framework for developing OpenFlow controllers in Ruby and C.
MAESTRO: (Java) Maestro is an OpenFlow "operating system" for orchestrating network control applications.

We will examine in detail **OpenDaylight** Controller in next section since is the controller we used for our experiment.

3. OpenDaylight

OpenDaylight is a project which promotes the Software Defined Networking (SDN). It is an open source project with a modular and flexible controller platform at its core. OpenDaylight officially started on 8th April of 2013 and it is based on Eclipse Public License. This controller is implemented strictly in software and is contained within its own Java Virtual Machine (JVM). It runs on any hardware and OS platform that supports JAVA. OpenDaylight also aims to accelerate adoption of Software Defined Networking and create a solid foundation for Network Functions Virtualization (NFV) for a better approach that provides innovation and reduces risk. Additionally, OpenDaylight is an open, industry-supported framework, consisting of code and it provides innovation, adoption, and a more transparent approach to SDN.

3.1 OpenDaylight Controller

3.1.1 Architecture of OpenDaylight

The OpenDaylight project is directly connected with Software Defined Networking and OpenFlow, and can be used in physical or virtual environments.

The project contains:

-**The controller**: A modular, scalable and multi-protocol SDN controller based on OSGI (this framework is a module system and service platform for the Java programming language that implements a complete and dynamic component model)

-**OpenFlow Plugin**: Integration of OpenFlow protocol library in controller Service Abstraction Layer

-**OpenFlow Protocol Library**: OpenFlow 1.3 protocol library implementation.

-Open vSwitch Database (OVSDB): configuration and management protocol support, e.g. for Open vSwitch.

-YANG Tools: Java-based NETCONF and YANG tooling for OpenDaylight projects.

OpenDaylight uses the following software tools:

Maven: a project management tool that is used in OpenDaylight to hand the builds and dependencies.

OSGi: This framework allows dynamically loading bundles and packaged Jar files, and binding bundles together for information exchange. An OSGi bundle has a manifest file that declares the various packages that are imported and exported.

Java interfaces: Java Interfaces are used for event listening, specifications and forming patterns.

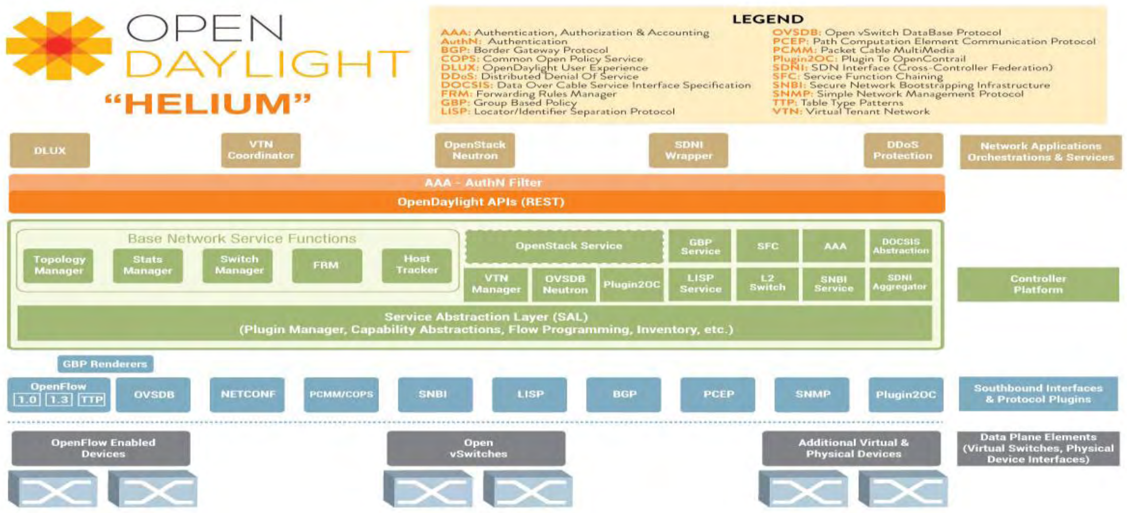

OPEN DAYLIGHT
"HELIUM"
LEGEND
AAA: Authentication, Authorization & Accounting
AuthN: Authentication
BGP: Border Gateway Protocol
COPS: Common Open Policy Service
DLUX: OpenDaylight User Experience
DDoS: Distributed Denial Of Service
DOCSIS: Data Over Cable Service Interface Specification
FRM: Forwarding Rules Manager
GBP: Group Based Policy
LISP: Locator/Identifier Separation Protocol
OVSDB: Open vSwitch DataBase Protocol
PCEP: Path Computation Element Communication Protocol
PCMM: Packet Cable MultiMedia
Plugin2OC: Plugin To OpenContrail
SDNI: SDN Interface (Cross-Controller Federation)
SFC: Service Function Chaining
SNBI: Secure Network Bootstrapping Infrastructure
SNMP: Simple Network Management Protocol
TTP: Table Type Patterns
VTN: Virtual Tenant Network
DLUX
VTN Coordinator
OpenStack Neutron
SDNI Wrapper
DDoS Protection
Network Applications Orchestrations & Services
AAA - AuthN Filter
OpenDaylight APIs (REST)
Base Network Service Functions
Topology Manager
Stats Manager
Switch Manager
FRM
Host Tracker
OpenStack Service
GBP Service
SFC
AAA
DOCSIS Abstraction
VTN Manager
OVSDB Neutron
Plugin2OC
LISP Service
L2 Switch
SNBI Service
SDNI Aggregator
Controller Platform
Service Abstraction Layer (SAL)
(Plugin Manager, Capability Abstractions, Flow Programming, Inventory, etc.)
GBP Renderers
OpenFlow 1.0 1.3 TTP
OVSDB
NETCONF
PCMM/COPS
SNBI
LISP
BGP
PCEP
SNMP
Plugin2OC
Southbound Interfaces & Protocol Plugins
OpenFlow Enabled Devices
Open vSwitches
Additional Virtual & Physical Devices
Data Plane Elements (Virtual Switches, Physical Device Interfaces)

Figure 4: OpenDaylight architecture framework

Figure 4 shows the architecture of OpenDaylight in a SDN network which is commonly described in layers.

Network Applications and Orchestration: The top layer consists of business and network logic applications that control and monitor network behavior.

Controller Platform: The middle layer is the framework in which the SDN abstractions can manifest, providing a set of common APIs to the application layer (commonly referred to as the northbound interface) while implementing one or more protocols for command and control of the physical hardware within the network (typically referred to as the southbound interface).

Physical & Virtual Network Devices: The bottom layer consists of the physical & virtual devices (switches, routers etc.) that make up the connective fabric between all endpoints within the network.

The controller exposes open **northbound APIs** which are used by applications. OpenDaylight supports the OSGi framework and bidirectional REST for the northbound API. The OSGi framework is used for applications that will run in the same address space as the controller while the REST API is used for applications that do not run in the same address space (or even necessarily on the same machine) as the controller. These applications use the controller to gather network intelligence, run algorithms to perform analytics and then use the controller to organize the new rules throughout the network.

The controller platform itself contains a collection of dynamically pluggable modules to perform needed network tasks. There are a series of base network services for such tasks as understanding what devices are contained within the network and the capabilities of each (e.g. statistics gathering).

The **southbound** interface is capable of supporting multiple protocols (e.g. OpenFlow 1.0, OpenFlow 1.3, BGP-LS, etc.). These modules are dynamically linked into a Service Abstraction Layer (SAL) (see next section).

3.1.2 OpenDaylight APIs - Service Abstraction Layer (SAL)

The **Service Abstraction Layer** (**SAL**) is at the heart of the modular design of the controller and is a crossbar connecting the protocol plugins (OpenFlow, BGP, OVSDB) to the Network Function Modules (Topology Manager, Switch Manager, Statistics Manager etc.). In other words, SAL allows the controller to support multiple protocols on the Southbound and providing consistent services for modules and Apps. The SAL provides basic services like Device Discovery which are used by modules like Topology Manager to build the topology and device capabilities. Services are constructed using the features exposed by the plugins (based on the presence of a plugin and capabilities of a network device). Based on the service request the SAL maps to the appropriate plugin and thus uses the most appropriate Southbound protocol to interact with a given network device. Each plugin is independent of each other and are loosely coupled with the SAL. There are two approaches for Service Abstraction Layer: the **Model Driven** Service Abstraction Layer (**MD-SAL**) and the **API Driven** Service Abstraction Layer (**AD-SAL**).

API Driven SAL: is an approach of the SAL created for the Controller project. Intent of this layer is to provide abstracted view of all the southbound to all the northbound applications irrespective of the underlying protocol. Also, AD-SAL has been created in order to control two different types of network elements, by accessing them programmatically. The AD-SAL has both northbound and southbound APIs and there is a standard REST API for each northbound or southbound plugin. The SAL APIs in AD-SAL, request routing between consumers and providers, and data adaptations are all statically defined at compile time.

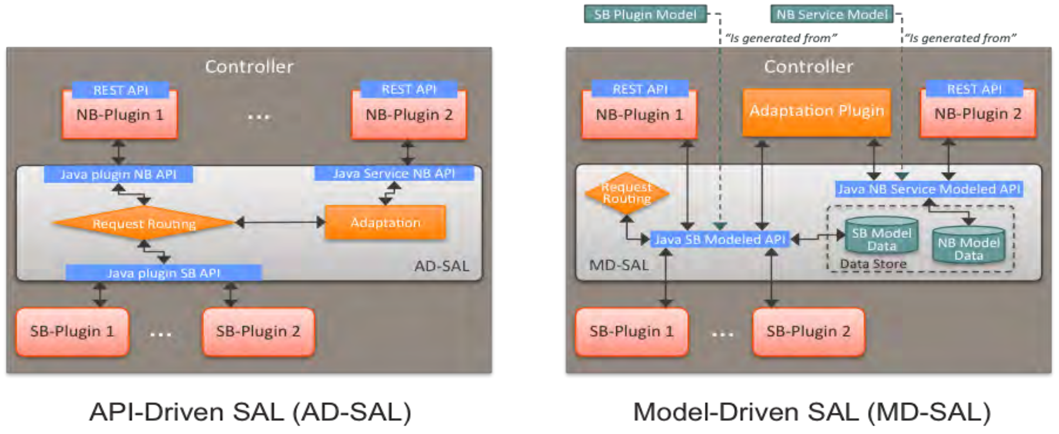


Figure 5: AD-SAL & MD-SAL comparison

Model Driven SAL: In the MD-SAL, the SAL APIs and the request routing between consumers and providers are defined from models, and also data adaptations are provided by internal adaptation plugins. The API code is generated from models when a plugin is compiled. When the OSGI bundle is loaded into the controller, the API code is also loaded into the controller along with the others plugins containing the model. Also, MD-SAL does not provide service adaptation itself because service adaptation is provided by plugins. This adaptation plugin, which is a regular plugin, provides data to the SAL and consumes data from the SAL through APIs generated from models. Additionally, the MD-SAL can store data for models defined by plugins. In this way, provider and consumer plugins can exchange data through the MD-SAL storage.

The difference between southbound and northbound plugins is that southbound plugins talk protocols to network nodes and northbound plugins talk application APIs to controller applications. As far as the **SAL** is concerned, there is really no north or south. The SAL is basically a data exchange & adaptation mechanism between plugins. A producer implements an API and provides the API's data; a consumer uses the API and consumes the API's data.

3.1.3 OpenDaylight Project List

Below is a list with some projects of OpenDaylight controller. A developer can create his own project and if his project be approved by TSC (OpenDaylight Technical Steering Committee) will be automatically listed in the project list of OpenDaylight in wiki page. Here is the latest project list according to https://wiki.opendaylight.org/view/Project_list :

- AAA:Main
- Affinity Metadata Service:Main
- ALTO:Main
- BGP LS PCEP:Main
- CAPWAP:Main
- Controller Core Functionality Tutorials:Main
- CrossProject:Integration Group
- Defense4All:Main
- DIDM:Main
- Discovery:Main
- Documentation
- Group Based Policy (GBP):Main
- IoTDM:Main
- L2 Switch:Main
- LACP:Main
- Network Intent Composition:Main
- NeutronNorthbound:Main
- ODL Root Parent:Main
- ODL-SDNi App:Main
- Open DOVE:Main
- OpenDaylight Controller:Main
- OpenDaylight dlux:Main
- OpenDaylight Lisp Flow Mapping:Main
- OpenDaylight OpenFlow Plugin:Main

- OpenDaylight Toolkit:Main
- OpenDaylight Virtual Tenant Network (VTN):Main
- Openflow Protocol Library:Main
- OpFlex:Main
- OVSDB Integration:Main
- PacketCablePCMM:Main
- Persistence:Main
- RelEng/Autorelease
- RelEng/Builder
- Reservation:Main
- SecureNetworkBootstrapping:Main
- Service Function Chaining:Main
- SNMP Plugin:Main
- SNMP4SDN:Main
- Southbound plugin to the OpenContrail platform:Main
- SXP:Main
- Table Type Patterns
- TCPMD5:Main
- Topology Processing Framework:Main
- TSDR:Main
- USC:Main
- VPNService:Main
- YANG Tools:Main

Clicking on the link of every project you are able to see project information or the architecture of the project you are interested in, and also you can follow guides and tutorials in order to download and install the respective project.

PART II
(Experiment Implementation)

Part A: Mininet theory & code

A.1 Mininet Information

Mininet is a network emulator that creates a realistic virtual network, running real kernel and application code on a single machine. Further, mininet runs a collection of hosts, switches, routers and links on a Linux kernel. It uses lightweight virtualization to make a single system looks like a complete network, running the same kernel and system. The entire network can be packaged as a VM, so that others can download, run, examine and modify it. A host at Mininet behaves like a real machine. You can easily approach this host (via *Ssh* command) and run arbitrary programs (also you can run anything that is installed on the underlying Linux system).

The most important thing on mininet is its virtual hosts, switches, links and controllers. These virtual devices are simply created using software instead of hardware, and their behavior is similar to discrete hardware work and elements.

A.1.1 How Mininet creates the virtual network?

The answer is that mininet internally uses **Linux containers** to emulate hosts, switches and links. A Linux container is an operating system (OS) level virtualization method for running multiple isolated Linux systems on a single control host. Linux also supports **network namespaces**, a lightweight virtualization feature that provides individual processes with separate network interfaces, routing tables and ARP tables. For each virtual host, mininet creates a container connected to a network namespace. Network namespaces are containers for network state. Mininet uses the lightweight virtualization mechanisms built into the Linux Operating System, and also 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 (veth).

Specific mininet creates:

Hosts: Network namespace provide processes with exclusive ownership of interfaces, ports, and routing tables (such as ARP and IP). A host in Mininet is simply a shell process moved into its own network namespace. Each host has its own virtual Ethernet interface(s) (veth) and a pipe to a parent Mininet process, mn, which sends commands and monitors output.

Switches: Software OpenFlow switches act like hardware switches. Both user-space and kernel-space switches are available.

Controllers: Controllers can be anywhere on the real or simulated network, as long as the machine on which the switches are running has IP-level connectivity to the controller. For Mininet running in a VM, the controller could run inside the VM, on the host machine, or in the cloud.

Links: A virtual Ethernet pair acts like a wire connecting two virtual interfaces. Packets sent through one interface are delivered to the other, and each interface appears as a fully functional Ethernet port to all system and application software. Veth pairs may be attached to virtual switches such as the Linux Bridge or a software OpenFlow switch.

A.1.2 Mininet Features

Mininet includes:

- A command line launcher (**mn**) to instantiate networks.
- A handy Python API for creating networks of varying topologies.
- Examples (in the **examples/** directory) to help you get started.
- Full API documentation via Python **help ()** docstrings.
- Parametrized topologies (tree, single, linear etc.) using the mininet object.
- A command-line interface (CLI class) which provides useful diagnostic commands (like ***iperf*** and ***ping***), as well as the ability to run a command to a node.

- A "cleanup" command (***mn –c***) to get rid of junk (interfaces, processes, files) which might be left around by Mininet or Linux.

A.1.3 Mininet Advantages & uses

If we compare Mininet with other full virtualized based approaches, we configure that mininet **boots faster** (in seconds instead of minutes). It is also provides to user the ability to create hundreds of hosts, switches , links ,while in another virtualized system you can only create a single digit of devices. Also, mininet can be **easily installed** as prepackaged Virtual Machine and it runs on VMware or **VirtualBox** (on Windows, Linux or Mac) with OpenFlow v1.0 tools already installed. Further, **provides more bandwidth** (almost 2Gbps total bandwidth).

Now, compared to hardware testbeds (platforms for experimentation and implementation of large developments projects) mininet is **inexpensive** and always available. Also is quickly **restartable** and **reconfigurable**.

In addition, Mininet **runs application code**, OS kernel code and control plane code (you can easily run via a simple command both OpenFlow controller code and Open vSwitch code). Finally, Mininet offers you **interactive performance** due to the fact that you can type at it.

Mininet is distributed as a VM, runnable on common virtual machines such as VirtualBox. The virtual machine provides a convenient container for distribution. Thus, Mininet provides you the ability to **share your network** to others. It can also be installed natively ***(sudo apt-get install)*** on Ubuntu.

Once a design works on Mininet, it can be **deployed on hardware** for real-world use and testing. Every emulated component must act in the same way as its corresponding physical one; the virtual topology should match the physical one. Virtual Ethernet pairs must be replaced by link-level Ethernet connectivity. Hosts emulated as processes should be replaced by hosts with their own OS image. In addition, each emulated OpenFlow switch should be replaced by a physical one configured to point to the controller. This way, the CLI (command line interface) enables interaction with the network in the same way as before.

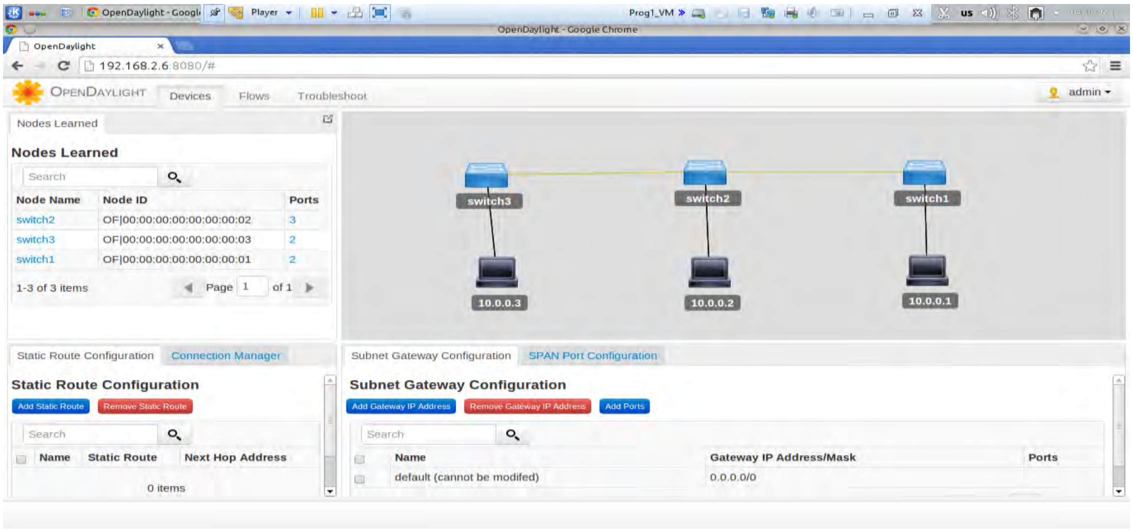
A.1.4 Mininet Disadvantages & Limitations

The main disadvantage for mininet is that you will probably need to use *slower links*, due to the fact that packets are forwarded by software switches (such as Open vSwitch) that share CPU and memory resources and usually have lower performance than switching hardware. More specific, if your server has 3GHz CPU, this CPU has to be shared and balanced among mininet's virtual hosts and switches. Thus we have lower performance in our network. Additionally mininet *does not provide an OpenFlow controller*. If you want to control your network and your switching behavior you have to search (or develop) for a controller with these specifications (such as OpenDaylight). Finally, mininet uses a single Linux kernel for all virtual hosts and this means that you *can't run software that depends on other OS kernels.*

A.2 Mininet code & commands- Topology examples

A.2.1 How to Create Topologies in Mininet

1. A way to create a custom topology on Mininet is to run the topology within the '**mn**' command.
The command you type from Mininet CLI is:
sudo mn –topo "topology type" ,"number of hosts" where the topology type can be single, linear or tree.
For example with the command: *sudo mn –topo linear,3* mininet creates a virtual network with three switches and three hosts where the connectivity between the switches and the hosts is linear. To see this topology in graphics we have to connect to an OpenFlow controller (OpenDaylight) and open its web page.
This is implemented by the command:
sudo mn –topo linear, 3 –controller=remote,ip="localhost"
Then, we open our browser and we type "*localhost:8080*" where *localhost* is the ip where our controller is running.



Nodes Learned

Node Name	Node ID	Ports
switch2	OF|00:00:00:00:00:00:00:02	3
switch3	OF|00:00:00:00:00:00:00:03	2
switch1	OF|00:00:00:00:00:00:00:01	2

1-3 of 3 items Page 1 of 1

Static Route Configuration

Name	Static Route	Next Hop Address

0 items

Subnet Gateway Configuration

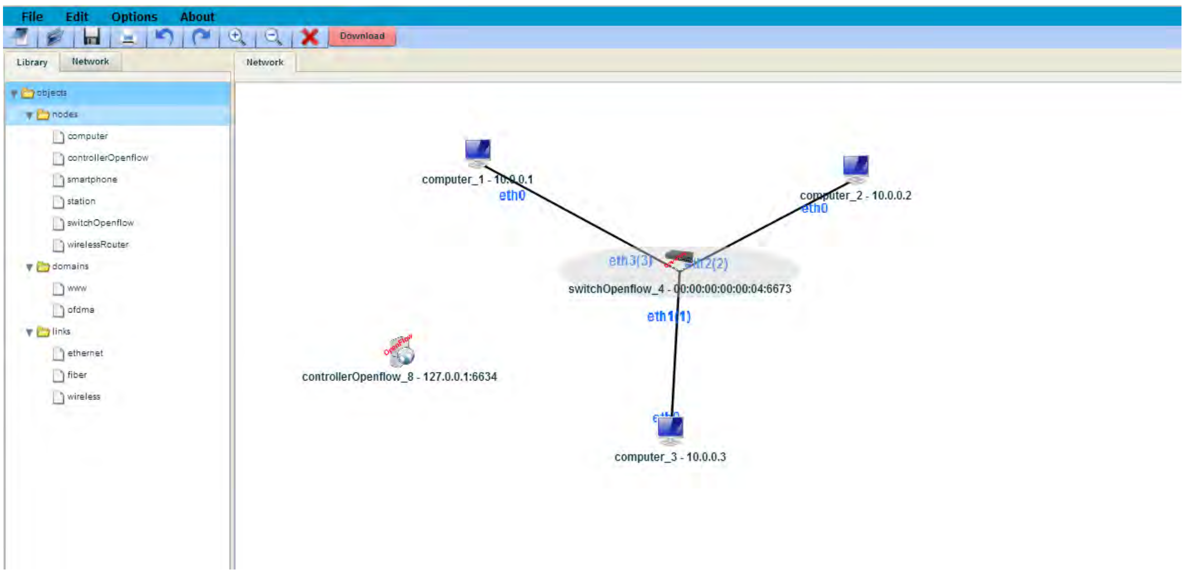
Name	Gateway IP Address/Mask	Ports
default (cannot be modifed)	0.0.0.0/0	

2. Another way to create a custom topology is by a python script. You can open your mininet console and go to the directory mininet/examples where you can find the file ***emptynet.py***. Adding the controller's ip and the right port we have the code below:

```
from mininet.net import Mininet
from mininet.node import Controller
from mininet.cli import CLI
from mininet.log import setLogLevel, info
def emptyNet():

    net = Mininet( controller=Controller )

    info( '*** Adding controller\n' )
    net.addController( 'c0',controller=Controller,ip="10.0.2.15",port=6633 )

    info( '*** Adding hosts\n' )
    h1 = net.addHost( 'h1',ip="10.0.0.1")
    h2 = net.addHost( 'h2',ip="10.0.0.2")

    info( '*** Adding switch\n' )
    s3 = net.addSwitch( 's3' )

    info( '*** Creating links\n' )
    net.addLink( h1, s3 )
    net.addLink( h2, s3 )

    info( '*** Starting network\n')
    net.start()
  info( '*** Running CLI\n' )
    CLI( net )
   info( '*** Stopping network' )
    net.stop()
if __name__ == '__main__':
   setLogLevel( 'info' )
   emptyNet()
```

With the code above I create a virtual network with two hosts (h1 and h2) connected to a switch (s3). Now, we run the python script alone without having to run within “mn”. We type the command: *sudo python emptynet.py* and we notice that our network is created. Typing the controller’s ip (port 8080) in our browser we see a graph of our topology.

3. A third way to create a mininet topology is visiting the site www.ramonfontes.com/vnd where we can graphically create our virtual network adding manually icons representing hosts, switches, controllers and Ethernet or Wireless links in order to connect our devices.
Next, by clicking on File→Export→Export to Mininet we can export our topology to mininet and download the *mininet.sh* file. Then, we can simple run our python script (like we did above) and create our custom topology.
Picture 1 shows an example of a network created in www.ramonfontes.com/vnd.



A.2.2 Mininet Commands (CLI)

Here some basic commands in Mininet (CLI):

COMMAND	MEANING
$sudo mn	creates default-minimal topology
$sudo mn --switch ovsk --controller ref –topo tree, depth=2, fanout=8 --test pingall	tree topology of depth 2 and fanout 8 (thus , 64 hosts connected to 9 switches) using Open vSwitch switches under the control of the OpenFlow/Stanford reference controller, and runs the pingall test to check connectivity between every pair of nodes
$sudo mn –test iperf	creates minimal-default topology, ran an iperf server on one host, ran an iperf client on the second host, and parsed the bandwidth achieved.
$ sudo mn --test pingpair	creates a minimal topology, started up the OpenFlow reference controller, ran an all-pairs-ping test, and tore down both the topology and the controller.
$sudo mn -c	Clean mininet up, if it crashes.

We **interact with Hosts and Switches** of the network and check their connectivity with the following commands. We are able to control and manage our network from the CLI of Mininet.

mininet>nodes	Display networks' nodes.
mininet>net	Display networks' links.
mininet>dump	Dump information about all nodes. We see the switches, hosts, controller listed.
mininet> h1 ifconfig -a	We should see the host's h1-eth0 and loopback (lo) interfaces.
mininet> s1 ifconfig -a	This will show the switch interfaces, plus the VM's connection out (eth0).

mininet> h1 ping h2	Test connectivity between h1 and h2
mininet>exit	Exit the command line interface(CLI)

Part B: Implementation and Demo Setup

B.1 Experiment description (Load Balancing)

In this section we describe the procedure we followed in order to succeed load balancing between two machines. We show step by step the procedures we followed to download and install the necessary Virtual Machines, OpenDaylight controller and Mininet. Also, we see how we create a virtual network (custom topologies) in Mininet and how this topology is connected to OpenDaylight controller via OpenFlow protocol. In addition, we explain the Java-based application we developed in order to succeed load balancing between two, three and five hosts in Mininet and finally, we compare the results of load balancing (bandwidth, data transferred) between hosts depending on time.

B.2 Experiment Implementation – Mininet

B.2.1 Topology Details & OpenDaylight

Here are the steps followed in order to implement the experiment:

Step 1 (Download OpenDaylight and Mininet): We downloaded an Ubuntu 14.04 virtual machine (64-bit/3GB) that has some SDN software and tools already installed; SDN controller (OpenDaylight), Open vSwitch v 2.3.0 with support of OpenFlow (1.2, 1.3 and 1.4), Mininet to create and run example topologies. We import the VM into VirtualBox and boot it.

Step 2 (Install and run OpenDaylight): In this step, we run the OpenDaylight controller which is already downloaded in VM. First we open a terminal on *controller/opendaylight* directory and run the command *mvn clean install*.("*mvn*" command uses Apache Maven to build the tutorial code. It compiles code based on the pom.xml file in that directory. "*install*" is essential for compilation. "*clean*" cleans the temporary build files. Next, we go to directory distribution/opendaylight/target/opendaylight and we type the command sudo ./run.sh and after some seconds openDaylight controller is running in our system.

Step 3 (Create topology-mininet): We open mininet console and we type the command *sudo mn –topo single,3 –controller=remote,ip=10.0.2.15*. Now, we are connected to OpenDaylight controller and a topology with three hosts and a switch is created. This virtual network appears in Picture 1 in previous section (page 26).

Step 4 (Load Balancer Service): Next, we will call the load balancer service of OpenDaylight Controller between hosts. Load balancer application balances traffic to backend servers based on the source addresses and source port of each incoming packet. The service reactively installs OpenFlow rules to direct all packets with a specific source address and source port to one of the appropriate backend servers. The servers are chosen using a **round robin policy**.

At this point, we have our controller running on localhost:8080 and we have created a topology (mininet) connected to the controller. On mininet CLI, if we make a *pingall* check we notice that all the hosts are reachable from each other.

In Appendix (Code I), we present the code we wrote in Java, on Netbeans IDE, in order to succeed Load balancing between hosts. We created three classes. First one creates a load balancer pool, second one creates a load balancer VIP and the third one adds pool members to the load balancer pool.

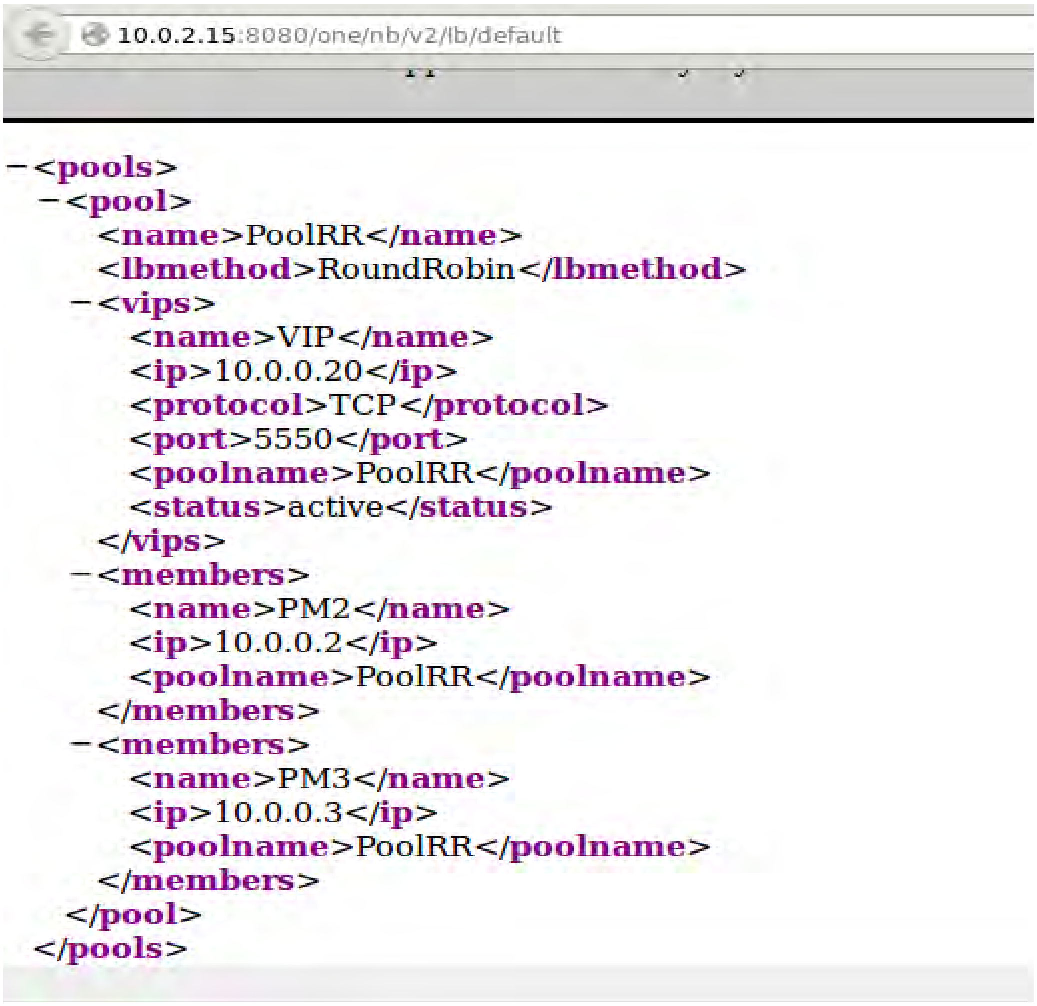
Specifically, class *CreatePool* creates a Load Balancer pool called *PoolRR* with *Round Robin* load balancing policy. At line 38, we call a REST API of the OpenDaylight's load balancer service. At line 40, we make a json string which contains the parameters (pool's name and load balancing method) of the HttpPost url we created at line 39. Lines 46-58 create the Http connection and in this way we make an HTTP POST request via using Apache HttpClient library API. The response we take after the code run is 201 when "pool created successfully", 404 when the "container name is not found" and 503 when "Load balancer service is unavailable".

In the same way, class *CreateLB_VIP* makes an HTTP POST request using HttpClient library API. To use load balancer service, a virtual IP (or VIP) should be exposed to the clients of this service and used as the destination address. A VIP is an entity that

comprises of a virtual IP, port and protocol (TCP or UDP). Only one server pool (host) can be assigned to a VIP. All the flow rules are installed from the service with an idle timeout of 5 seconds.
Now, in the code we call the “create/vip” REST API and at line 66 we add header of HTTP POST. So, a new Virtual IP is created called “VIP”, with an ip 10.0.0.20, port 5550 and protocol TCP. VIP is added at the pool named “PoolRR”.

Finally, in class *Create_Pool_Member1* we add the hosts of our network (pool members) to load balancer pool named PoolRR. We add a host named PM2 with ip 10.0.0.2 to the PoolRR. To continue with, we add all the hosts of our network except one host (e.g. h1) which will be used as a source/client that will send traffic to the VIP.

Now we will try load balancing for two hosts. If we run the loadbalancer.java file we take as response 201 from the four classes (*Create_Pool*, *CreateLB_VIP*, *Create_PoolMember1/2*) which means that a load balancer pool, a load balancer VIP and two pool members are created. In Picture 1, we see the xml file with the pool created by typing in a browser *localhost:8080/one/nb/v2/lb/default*.

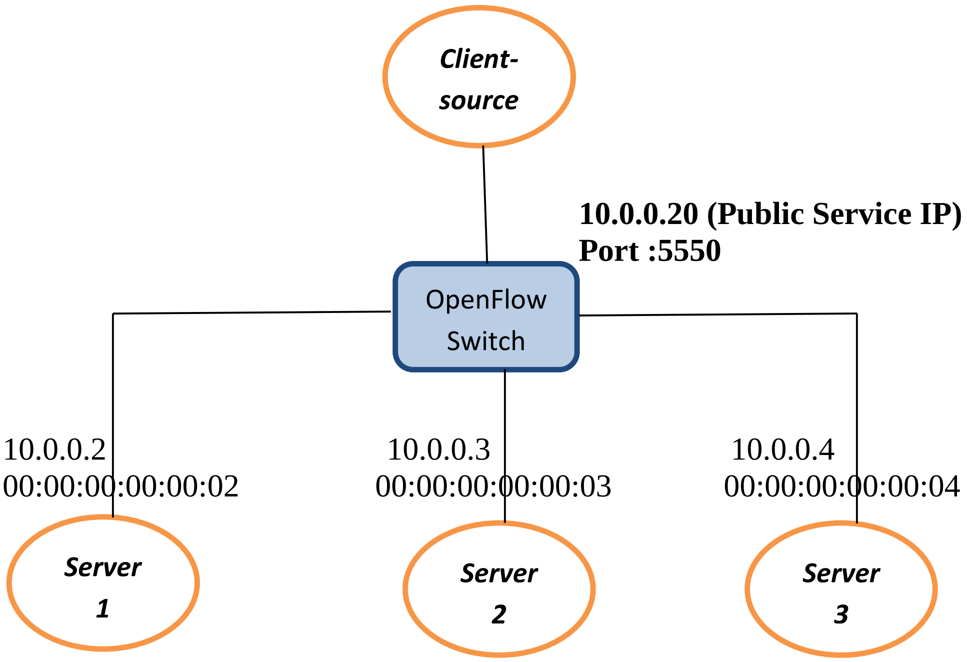

10.0.2.15:8080/one/nb/v2/lb/default

```
-<pools>
  -<pool>
     <name>PoolRR</name>
     <lbmethod>RoundRobin</lbmethod>
   -<vips>
      <name>VIP</name>
      <ip>10.0.0.20</ip>
      <protocol>TCP</protocol>
      <port>5550</port>
      <poolname>PoolRR</poolname>
      <status>active</status>
    </vips>
   -<members>
      <name>PM2</name>
      <ip>10.0.0.2</ip>
      <poolname>PoolRR</poolname>
    </members>
   -<members>
      <name>PM3</name>
      <ip>10.0.0.3</ip>
      <poolname>PoolRR</poolname>
    </members>
  </pool>
</pools>
```

Picture 1

Since VIP created, doesn't exist in the network, OpenDaylight controller won't be able to resolve ARP for IP of the VIP. Load balancer application assumes that if VIP is configured and exposed by user to a network, packets that are destined to the VIP's IP address will get routed to it through external mechanisms. For this experiment, we can locally resolve the ARP for the VIP's IP, by adding static entry to the ARP table of the client host.
Thus, we add static ARP entry for VIP's IP address in the host h1 arp cache. We execute the command: *arp -s 10.0.0.20 00:00:10:00:00:20* in h1 xterm window to add the ARP entry to the ARP table of h1. Next step is to start **iperf server** instances that listen on port 5550 on hosts h2 and h3. We execute the command *iperf –s –i 1 -p 5550*

on h2, h3 xterms. "-i 1" means that the server will report the data transferred and the bandwidth per one second. On h1, we start **iperf client** that sends traffic to the virtual ip address with the command *iperf -c 10.0.0.20 –t 10 -p 5550*. *"-t"* means that h1 send traffic for 10 seconds every time we trigger iperf client.
We notice that iperf client connects to the iperf server running on host h2. Once the previous test of iperf finishes, type the iperf client command again, and this time the client will connect to the iperf server running on host h3. Similarly in next iteration it will connect to the h2 in round robin fashion.



Graph 1: Load Balancing with round robin policy (3 servers , 1 client)

B.2.2 Experiment results for 2, 3, 5 hosts

Two hosts: This example executed for 20 seconds. From the servers' xterm console (h2 & h3) we start iperf server with report every one second. Then, we start iperf client from the client xterm console (h1) and send traffic every 5 seconds. We write down the results and we notice in figure 1 the load balancing between the machines. For example, in first five seconds we see that the client sends traffic to h2 server (bandwidth in h3 is zero), and in next 5 seconds client sends traffic to h3 server. Another comment in Figure 1 is that the time between 5th and 6th second (every time client sends traffic) one server’s bandwidth tends to zero and the other server’s increased.

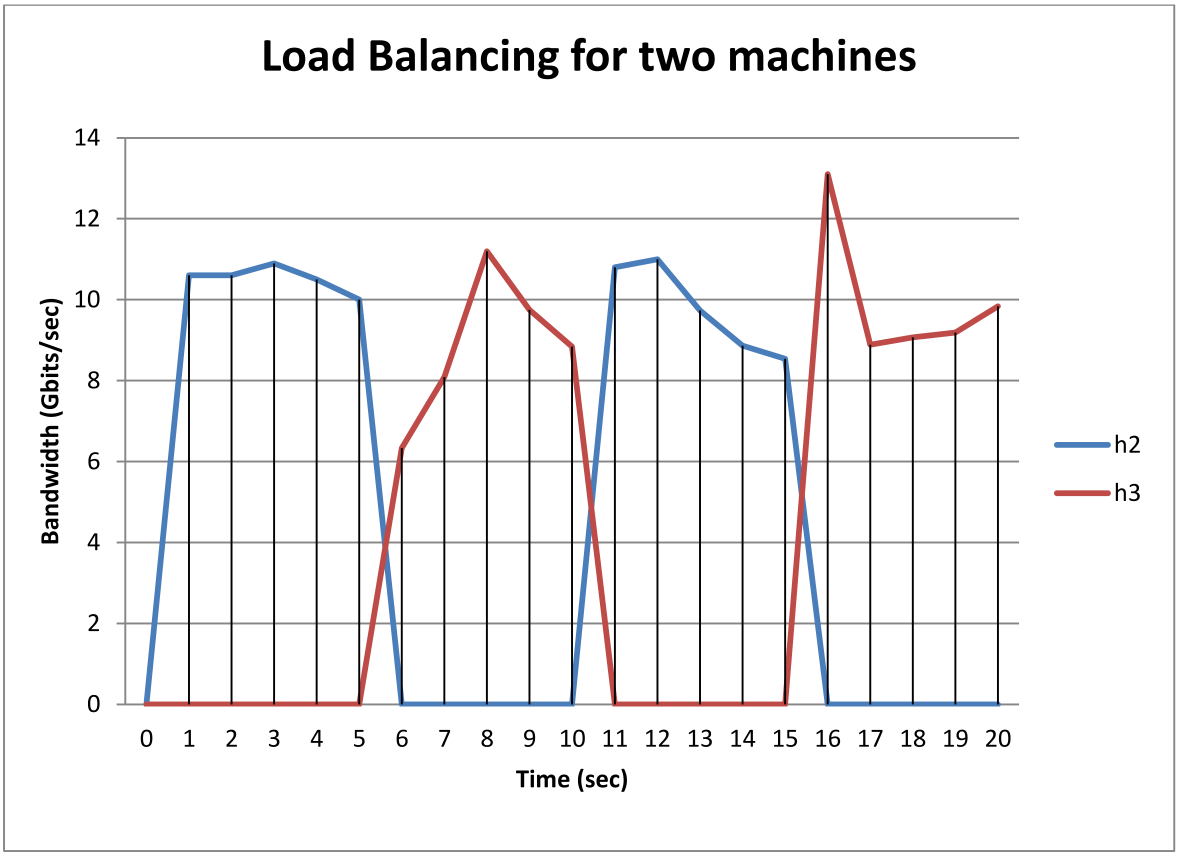


Figure 1

<u>Three hosts:</u> This example is executed for 40 seconds between three machines. Traffic is sent every 5 seconds by the client host (h1) and we notice on Figure 2 that the load balancing is working as the three servers send bandwidth every 5 seconds with round robin policy.

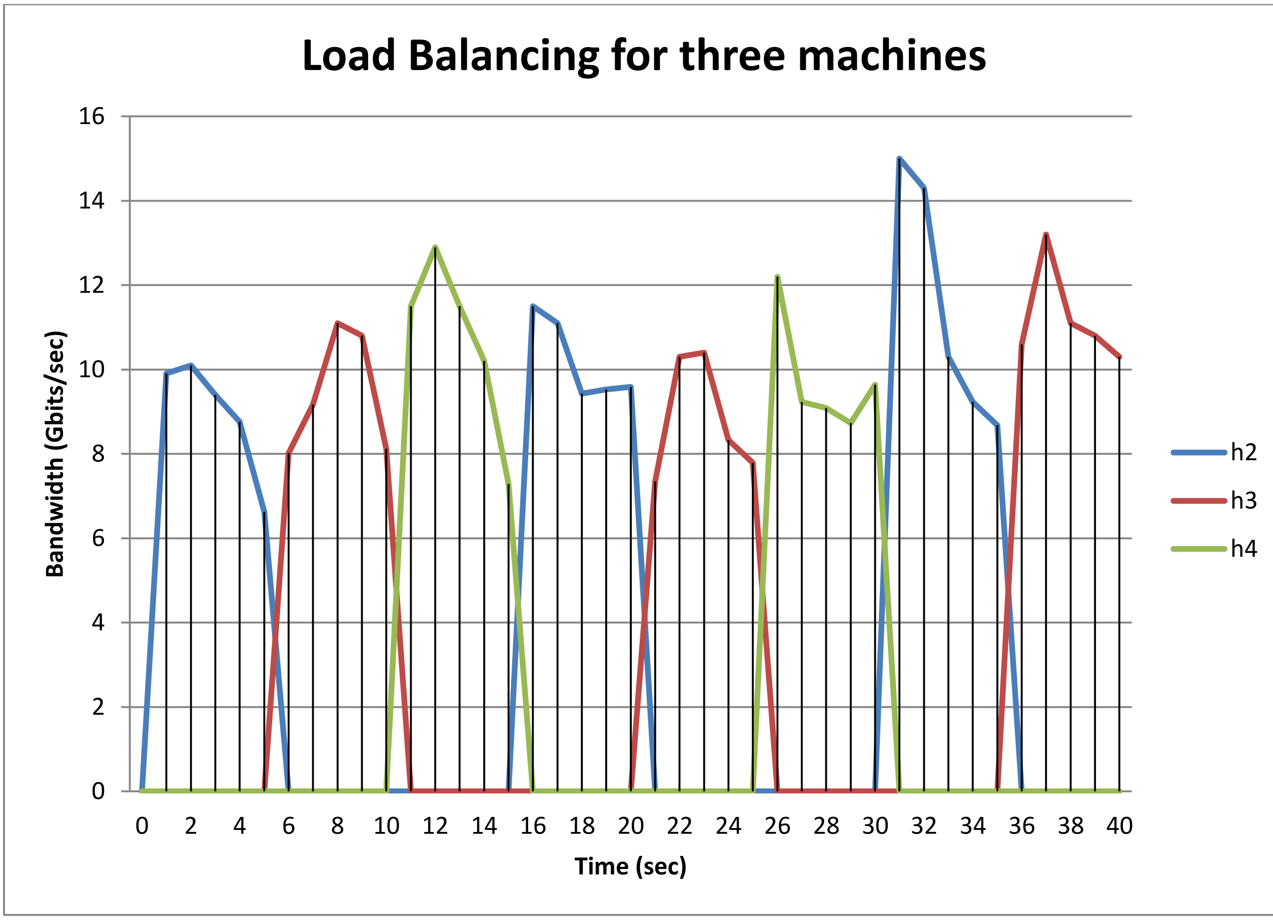


Figure 2

<u>Five Hosts:</u> This example is executed for 60 seconds between five machines. Traffic is sent every 5 seconds by the client host (h1) and we notice on Figure 2 that the load balancing is working as the five servers (h2, h3, h4, h5 & h6) send bandwidth every 5 seconds with round robin policy.

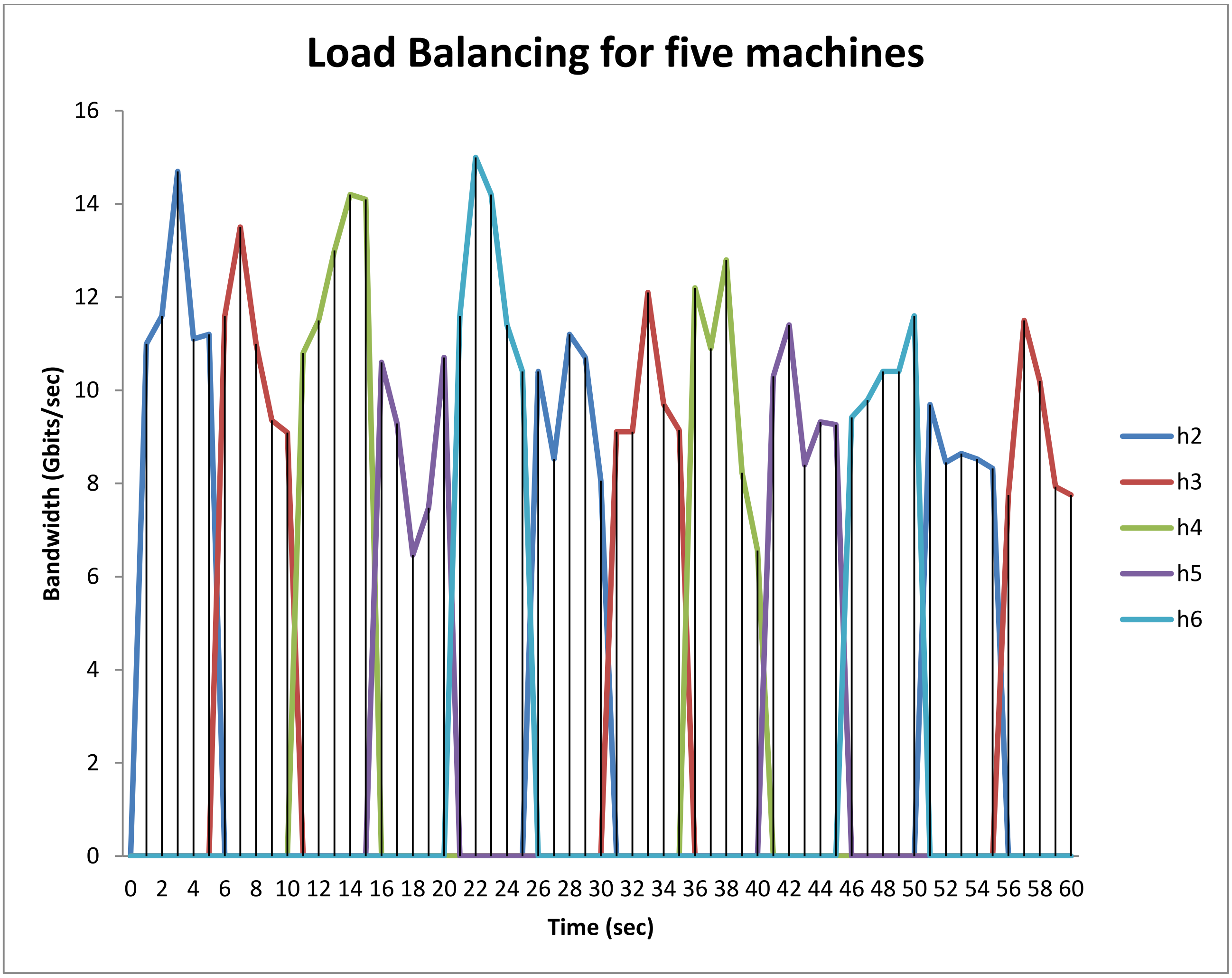


Figure 3

B.2.3 Alternative Experiment Implementation – Java

In B.2.1 & B.2.2 we ran our experiment after we called the Load Balancer service, which service exists in the controller's platform in the core of OpenDaylight controller.

Another way to succeed load balancing or any other service is by creating a Java application over the controller. This application will not call any service of the controller.

We wrote another Java Application (Appendix Code II) in order to install flows in OpenDaylight simply by running a Java program. The topology we use in this example is two hosts connected to a OpenFlow switch. The hosts communicate each other but after the flow installation (Action: DROP) the communication stops. To continue with, if we delete the flow we installed we notice that the two hosts start communicating again.

Class ***installFlow*** creates a url (line 29) using the Flow Programmer Rest API. Then creates an http connection (line 36) with the right connection properties (lines 38-47) and installs a flow in OpenFlow switch (with id : 00:00:00:00:00:00:00:01 and type:OF). This flow has an action DROP and its data is post in class Tester (page 42). Next, we run the program and we point our browser to *localhost:8080* where we select tab *Flows* and we see our flow that has been installed. Now, if we go at mininet console and type a pingall command we see that there is no communication between hosts (if we call *installFlow* class from our main).

Similarly, class ***deleteFlow*** creates a url, an http connection with the appropriate connection properties and deletes the flow that is installed in the switch (flowName and postData the parameters of the class). Also, if we call class deleteFlow we see that the two hosts communicate each other, since the flow with the DROP action has been deleted from the flow table of the OpenFlow switch.

Finally, in page 42 we see class ***Tester*** that contains the data of the flow we install. Specifically, we create a JSONObject variable named postData in line 14, and then we define the data (source IP, Destination IP, Ethernet Type, flow’s name, action) and the node that the flow will be installed and deleted.

Part C: Conclusion and Future Work

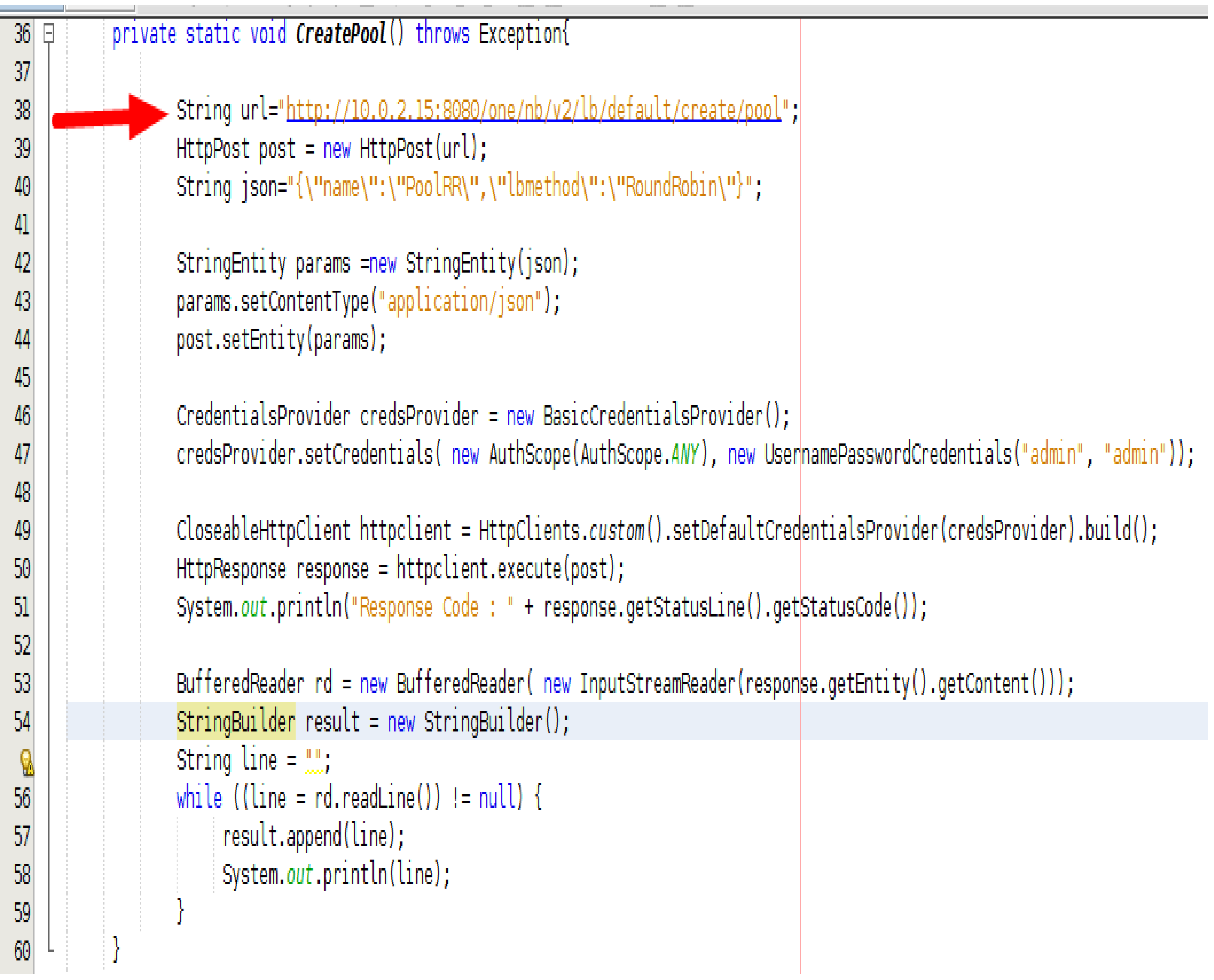
This part will explain the conclusions extracted after doing this thesis and the future work we are going to do. The future of networking will rely more and more on software, which will accelerate the pace of innovation for networks as it has in the computing and storage domains. SDN promises to transform today's static networks into flexible, programmable platforms with the intelligence to allocate resources dynamically, the scale to support enormous data centers and the virtualization needed to support dynamic, highly automated, and secure cloud environments. With its many advantages, SDN is on the way to becoming the new norm for networks. It helps to simplify operations by automating and centralizing network management tasks. SDN as a Networking Technology is poised to meet the demands of Operators and Enterprises with tremendous improvement in cost, flexibility and maintenance. OpenFlow as a technology is rapidly evolving and has reached a state of maturity, where major vendors are offering competitive platforms and solutions. Furthermore, is a serious alternative to software Ethernet Switching or IP Routing because it does not just do layer-2 and layer-3 forwarding, but also can do port forwarding and layer-4 forwarding, so we can consider it more flexible and configurable.
Also, we saw OpenDaylight that is a community-led, open, industry-supported framework, for accelerating adoption, fostering new innovation, reducing risk and creating a more transparent approach to Software-Defined Networking.
Finally, we introduced Mininet that is a unique open-source network simulator developed to support research and education in SDN. Mininet seems to be most suitable to researchers who will implement new controller software and are experienced with the Python scripting language.

Future Work

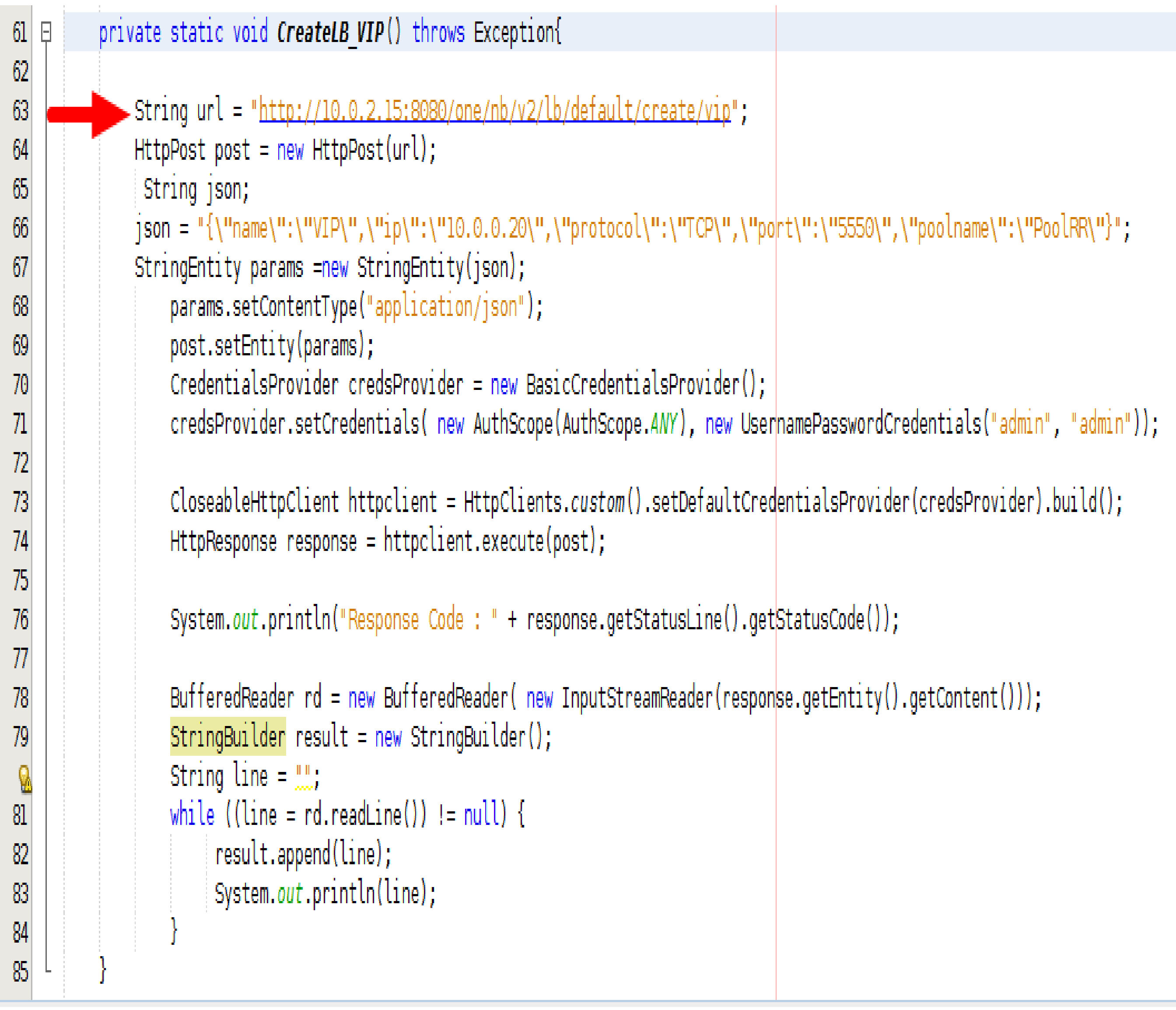
As far as future work concern, we intend to run our load balancing Java project into real nodes, using the appropriate OpenFlow switches. The experiment will take place in NITOS lab that has the equipment to support our work and we will have two servers, that will implement load balancing, and one client that will send traffic.

Appendix

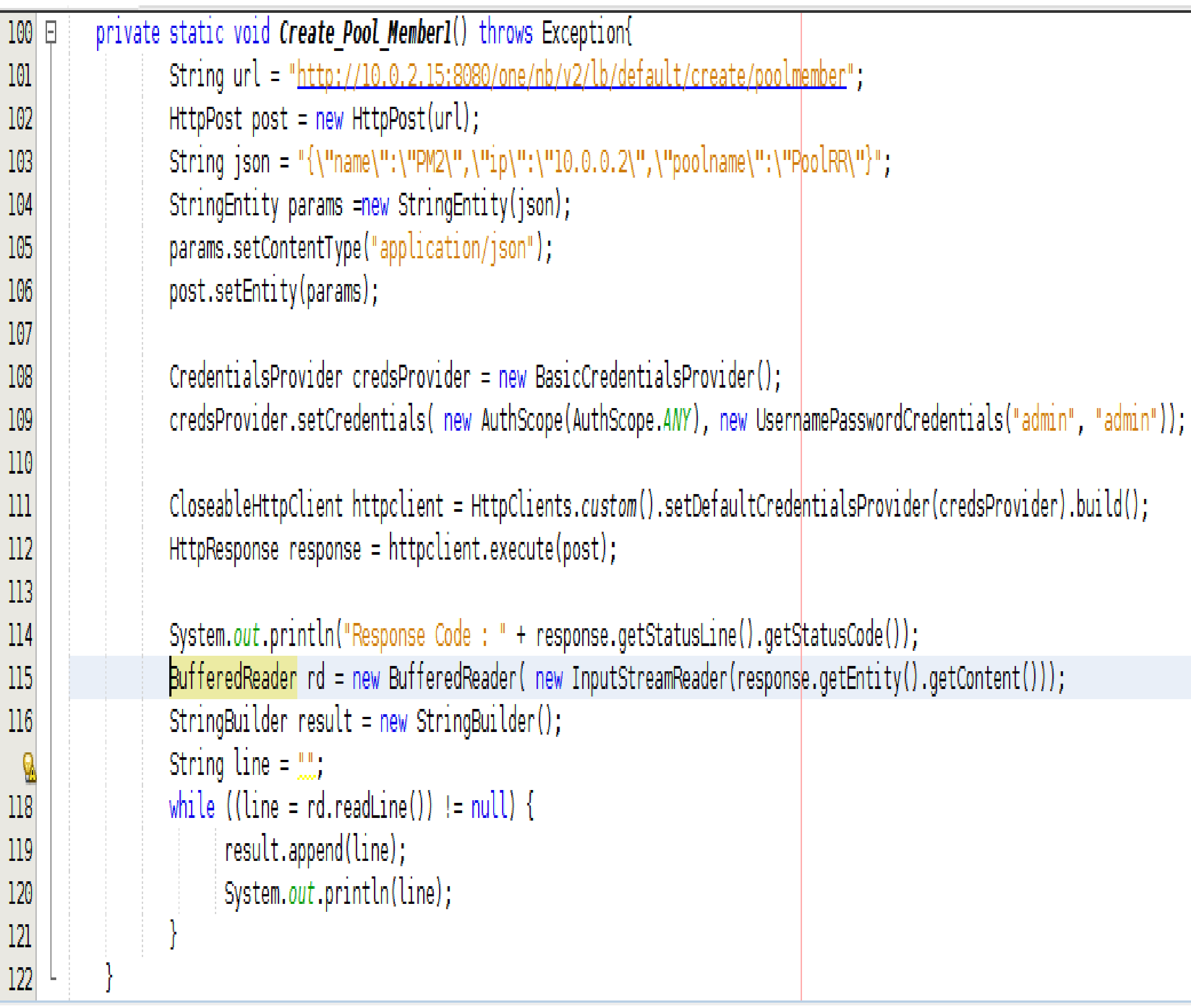
Code I (Load Balancer)



```
private static void CreatePool() throws Exception{

    String url="http://10.0.2.15:8080/one/nb/v2/lb/default/create/pool";
    HttpPost post = new HttpPost(url);
    String json="{\"name\":\"PoolRR\",\"lbmethod\":\"RoundRobin\"}";

    StringEntity params =new StringEntity(json);
    params.setContentType("application/json");
    post.setEntity(params);

    CredentialsProvider credsProvider = new BasicCredentialsProvider();
    credsProvider.setCredentials( new AuthScope(AuthScope.ANY), new UsernamePasswordCredentials("admin", "admin"));

    CloseableHttpClient httpclient = HttpClients.custom().setDefaultCredentialsProvider(credsProvider).build();
    HttpResponse response = httpclient.execute(post);
    System.out.println("Response Code : " + response.getStatusLine().getStatusCode());

    BufferedReader rd = new BufferedReader( new InputStreamReader(response.getEntity().getContent()));
    StringBuilder result = new StringBuilder();
    String line = "";
    while ((line = rd.readLine()) != null) {
        result.append(line);
        System.out.println(line);
    }
}
```

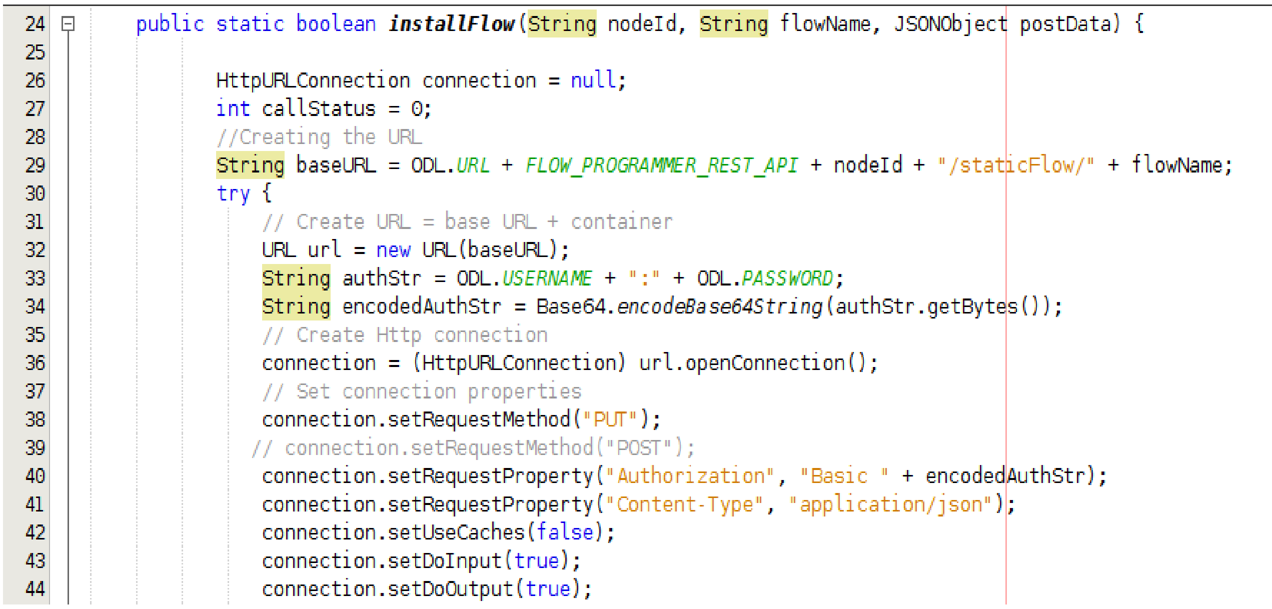


```java
    private static void CreateLB_VIP() throws Exception{

        String url = "http://10.0.2.15:8080/one/nb/v2/lb/default/create/vip";
        HttpPost post = new HttpPost(url);
         String json;
        json = "{\"name\":\"VIP\",\"ip\":\"10.0.0.20\",\"protocol\":\"TCP\",\"port\":\"5550\",\"poolname\":\"PoolRR\"}";
        StringEntity params =new StringEntity(json);
            params.setContentType("application/json");
            post.setEntity(params);
            CredentialsProvider credsProvider = new BasicCredentialsProvider();
            credsProvider.setCredentials( new AuthScope(AuthScope.ANY), new UsernamePasswordCredentials("admin", "admin"));

            CloseableHttpClient httpclient = HttpClients.custom().setDefaultCredentialsProvider(credsProvider).build();
            HttpResponse response = httpclient.execute(post);

            System.out.println("Response Code : " + response.getStatusLine().getStatusCode());

            BufferedReader rd = new BufferedReader( new InputStreamReader(response.getEntity().getContent()));
            StringBuilder result = new StringBuilder();
            String line = "";
            while ((line = rd.readLine()) != null) {
                result.append(line);
                System.out.println(line);
            }
    }
```

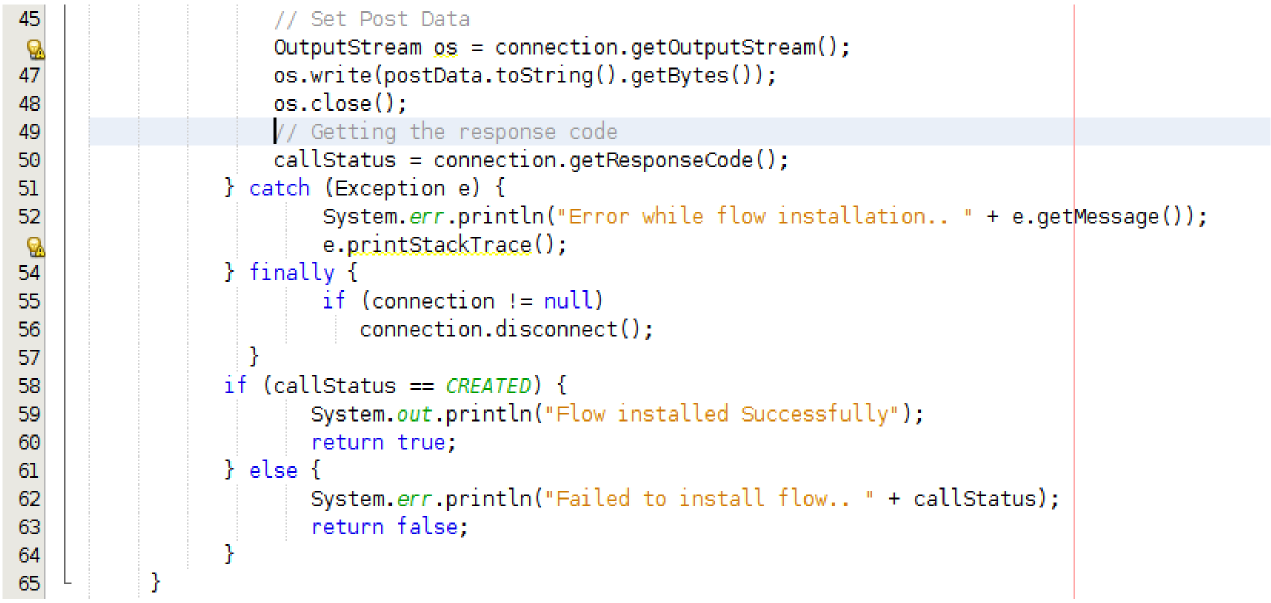


```
    private static void Create_Pool_Member1() throws Exception{
            String url = "http://10.0.2.15:8080/one/nb/v2/lb/default/create/poolmember";
            HttpPost post = new HttpPost(url);
            String json = "{\"name\":\"PM2\",\"ip\":\"10.0.0.2\",\"poolname\":\"PoolRR\"}";
            StringEntity params =new StringEntity(json);
            params.setContentType("application/json");
            post.setEntity(params);

            CredentialsProvider credsProvider = new BasicCredentialsProvider();
            credsProvider.setCredentials( new AuthScope(AuthScope.ANY), new UsernamePasswordCredentials("admin", "admin"));

            CloseableHttpClient httpclient = HttpClients.custom().setDefaultCredentialsProvider(credsProvider).build();
            HttpResponse response = httpclient.execute(post);

            System.out.println("Response Code : " + response.getStatusLine().getStatusCode());
            BufferedReader rd = new BufferedReader( new InputStreamReader(response.getEntity().getContent()));
            StringBuilder result = new StringBuilder();
            String line = "";
            while ((line = rd.readLine()) != null) {
                result.append(line);
                System.out.println(line);
            }
    }
```

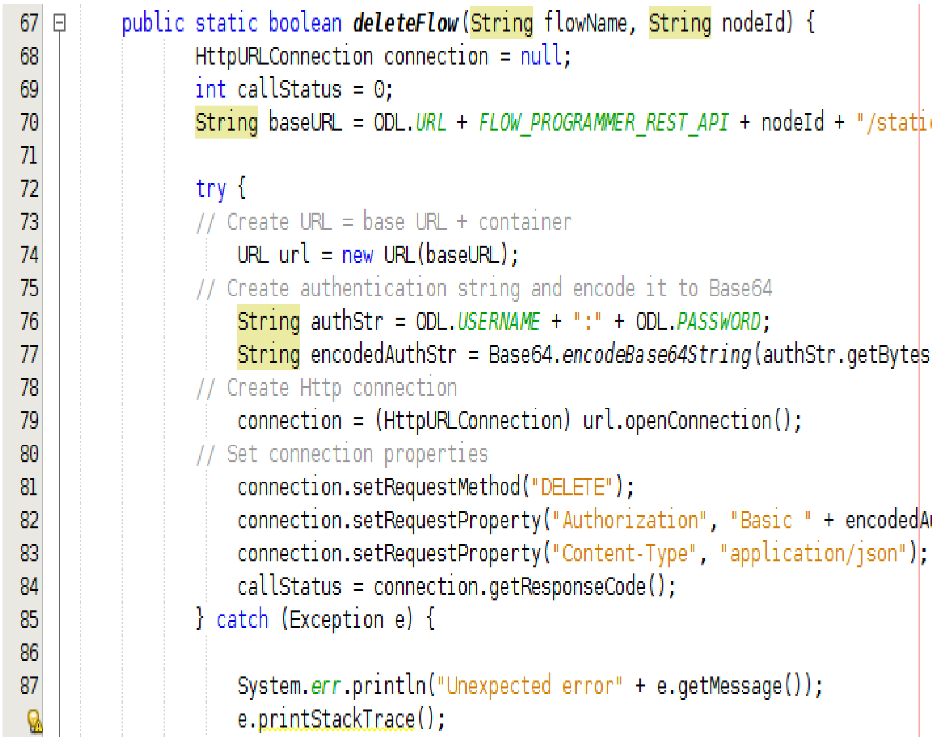
Code II (Add flows using JAVA)



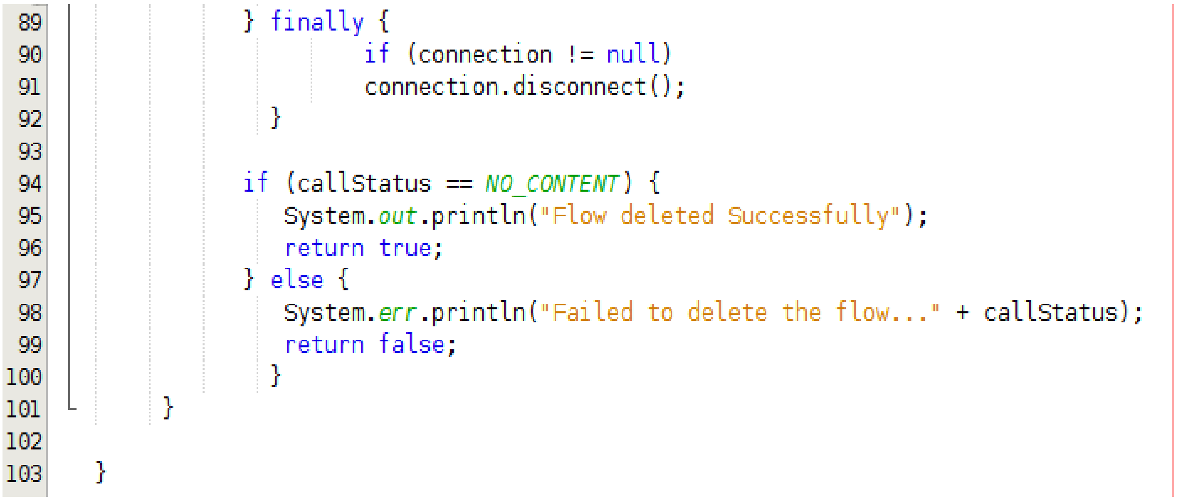
```java
    public static boolean installFlow(String nodeId, String flowName, JSONObject postData) {

        HttpURLConnection connection = null;
        int callStatus = 0;
        //Creating the URL
        String baseURL = ODL.URL + FLOW_PROGRAMMER_REST_API + nodeId + "/staticFlow/" + flowName;
        try {
            // Create URL = base URL + container
            URL url = new URL(baseURL);
            String authStr = ODL.USERNAME + ":" + ODL.PASSWORD;
            String encodedAuthStr = Base64.encodeBase64String(authStr.getBytes());
            // Create Http connection
            connection = (HttpURLConnection) url.openConnection();
            // Set connection properties
            connection.setRequestMethod("PUT");
           // connection.setRequestMethod("POST");
            connection.setRequestProperty("Authorization", "Basic " + encodedAuthStr);
            connection.setRequestProperty("Content-Type", "application/json");
            connection.setUseCaches(false);
            connection.setDoInput(true);
            connection.setDoOutput(true);
```



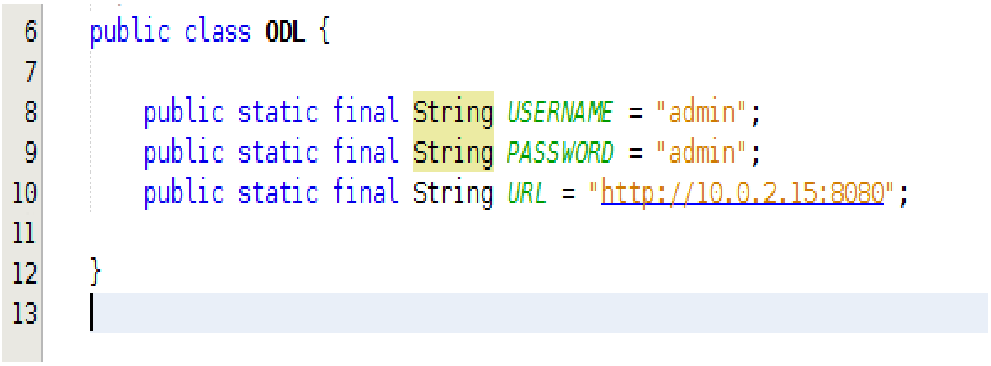
```java
            // Set Post Data
            OutputStream os = connection.getOutputStream();
            os.write(postData.toString().getBytes());
            os.close();
            // Getting the response code
            callStatus = connection.getResponseCode();
        } catch (Exception e) {
                System.err.println("Error while flow installation.. " + e.getMessage());
                e.printStackTrace();
        } finally {
                if (connection != null)
                    connection.disconnect();
          }
        if (callStatus == CREATED) {
               System.out.println("Flow installed Successfully");
               return true;
        } else {
               System.err.println("Failed to install flow.. " + callStatus);
               return false;
        }
    }
```



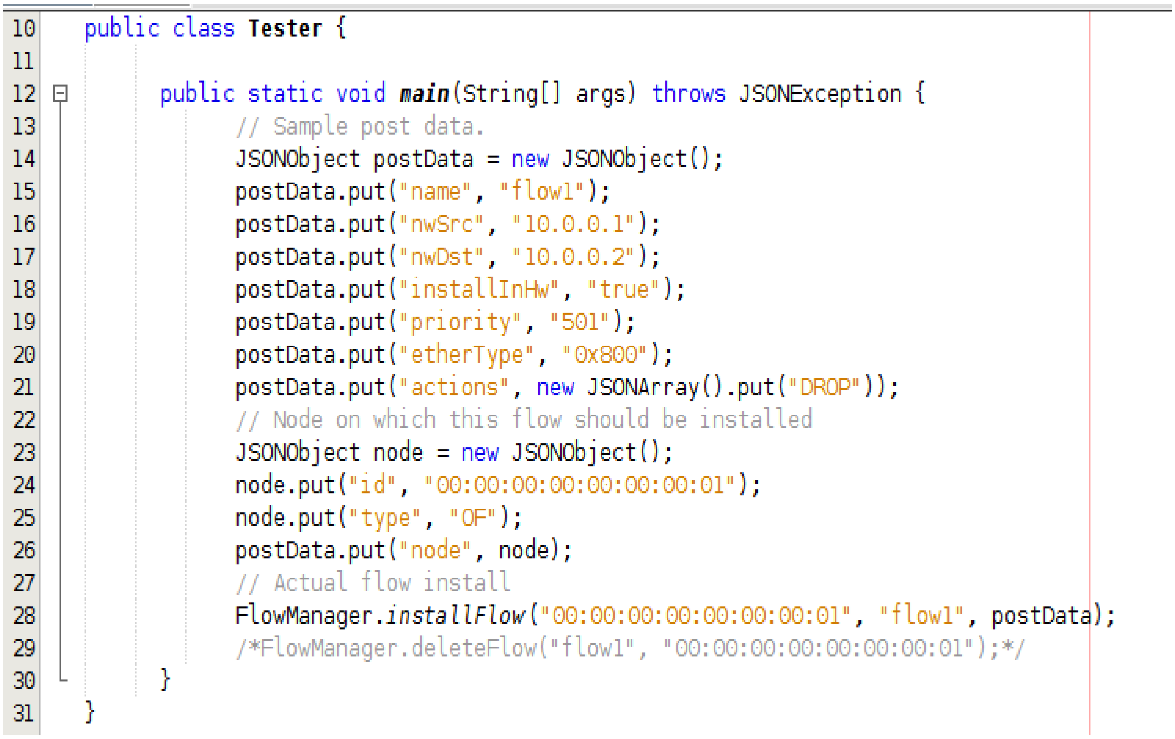
```
    public static boolean deleteFlow(String flowName, String nodeId) {
        HttpURLConnection connection = null;
        int callStatus = 0;
        String baseURL = ODL.URL + FLOW_PROGRAMMER_REST_API + nodeId + "/staticFlow/" + flowName;

        try {
        // Create URL = base URL + container
            URL url = new URL(baseURL);
        // Create authentication string and encode it to Base64
            String authStr = ODL.USERNAME + ":" + ODL.PASSWORD;
            String encodedAuthStr = Base64.encodeBase64String(authStr.getBytes());
        // Create Http connection
            connection = (HttpURLConnection) url.openConnection();
        // Set connection properties
            connection.setRequestMethod("DELETE");
            connection.setRequestProperty("Authorization", "Basic " + encodedAuthStr);
            connection.setRequestProperty("Content-Type", "application/json");
            callStatus = connection.getResponseCode();
        } catch (Exception e) {

            System.err.println("Unexpected error" + e.getMessage());
            e.printStackTrace();
```



```
            } finally {
                    if (connection != null)
                    connection.disconnect();
                }

            if (callStatus == NO_CONTENT) {
                System.out.println("Flow deleted Successfully");
                return true;
            } else {
                System.err.println("Failed to delete the flow..." + callStatus);
                return false;
            }
        }

    }
```



```java
public class ODL {

    public static final String USERNAME = "admin";
    public static final String PASSWORD = "admin";
    public static final String URL = "http://10.0.2.15:8080";

}
```



```java
public class Tester {

        public static void main(String[] args) throws JSONException {
                // Sample post data.
                JSONObject postData = new JSONObject();
                postData.put("name", "flow1");
                postData.put("nwSrc", "10.0.0.1");
                postData.put("nwDst", "10.0.0.2");
                postData.put("installInHw", "true");
                postData.put("priority", "501");
                postData.put("etherType", "0x800");
                postData.put("actions", new JSONArray().put("DROP"));
                // Node on which this flow should be installed
                JSONObject node = new JSONObject();
                node.put("id", "00:00:00:00:00:00:00:01");
                node.put("type", "OF");
                postData.put("node", node);
                // Actual flow install
                FlowManager.installFlow("00:00:00:00:00:00:00:01", "flow1", postData);
                /*FlowManager.deleteFlow("flow1", "00:00:00:00:00:00:00:01");*/
        }
}
```