



**ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ**

---

**UNIVERSITY OF PIRAEUS**

Department of Digital Systems

Postgraduate Programme "Digital Communications & Networks"

## **Software Defined Networks**

### **Reactive Flow Programming and Load Balance switching**

Supervisors: Prof. Dr. Demestichas Panagiotis

Prof. Dr. Tsagaris Konstantinos

Presented by,

Kallianiotis Nikolaos

**Piraeus 2017**

## Preface

This project serves as a Master Thesis as the requirements of the master's programme Master of Digital Communications and Networks. It proposes load balancing algorithms applied to Software-Defined Networks to achieve the best possible resource utilisation of each of the links present in a network. The open-sources Opendaylight project and Floodlight project are used as SDN controllers, and the network is emulated using Mininet software.

## Acknowledgements

Special thanks go to my parents, Christos and Sophia, my girlfriend Despoina and my friend Periklis that have always supported me.

Finally I would like to dedicate this thesis to my grandfather Panagiotis that although he has passed away, he is alive in my memories which make me march with courage, patience and confidence. I will never forget his pleasure and his satisfaction, because of my words when I hold my balance riding the bike which had made me gift. "I learned grandfather. I learned."

## Table of Contents

1.	Introduction .....	5
1.1.	Motivation.....	5
1.2.	Related Work .....	6
1.3.	Problem formulation.....	8
2.	Introduction to Software Defined Network .....	9
2.1.	Software-Defined Networking .....	13
2.2.	Network Virtualization .....	18
2.3.	OpenFlow Protocol.....	20
2.3.1.	Benefits of OpenFlow-Based Software-Defined Networks .....	26
2.4.	Mininet.....	28
2.5.	Software Defined Networking (SDN) Controllers .....	28
3.	OpenDaylight Helium Application Developers' Tutorial .....	33
3.1.	Introduction to OpenDaylight .....	37
3.2.	Basic steps to writing an OpenDayLight Application .....	38
3.3.	Writing an AD-SAL OpenDayLight Application .....	39
3.3.1.	Receiving events.....	39
3.3.2.	Parsing packets .....	40
3.3.3.	Sending messages to switch .....	40
3.3.4.	Managing multiple modules.....	41
3.4.	Sample AD-SAL application: Learning Switch.....	42
3.5.	Workflow, dependencies and information sharing.....	43
3.5.1.	Maven and OSGI.....	43
3.5.2.	Life of a packet.....	44
3.6.	AD-SAL OpenDaylight Web UI.....	44
3.7.	Writing an MD-SAL OpenDayLight Application.....	45
3.8.	Karaf.....	48
3.9.	Debugging.....	51
4.	Floodlight .....	53
4.1.	Introduction to Floodlight .....	53
5.	Reactive Flow Programming with OpenDaylight.....	56
5.1.	Registering Required Services and Subscribing to Packet-in Events .....	57
5.2.	Handling Packet-in Events.....	61
5.3.	Programming Flows .....	66
5.4.	Forwarding Packets .....	67

5.5. Implementation.....	69
6. Load Balancing.....	79
6.1. Implementation Approach.....	79
6.2. Running The Program (OpenDaylight) .....	79
6.3. Running The Program (Floodlight) .....	83
6.4. iPerf .....	87
6.5. Results We Achieved.....	88
Bibliography.....	90
Appendices .....	92
Codes .....	92

# 1. Introduction

## 1.1. Motivation

In the remaining twenty years networks' requirements were changing continuously. The quantity of traffic has been increasing exponentially and more demanding quit-to-stop desires are wished.

But, the networks architectures were unchanged, growing the complexity and hindering its configuration. if you want to adapt to the new wishes, new network paradigms consisting of software-described Networking (SDN), cognitive networks or automatic networks are rising fast due the pursuits of providers and ISPs.

The first activities is to analyse which can be the problems of the prevailing networks due to the fact that they have emerge as a barrier to growing new, progressive services, and a good large barrier to the ongoing growth of the internet.

Analysing the data plane within the ongoing networks architecture we discover properly defined layers for exceptional purposes, making them independent. for instance the physical layer (fibre, copper, radio...) is unbiased of the link layer (ATM, X.25, PPP...), the delivery layer (TCP, UDP...) or the utility layer (HTTP, FTP, telnet...), what lets in the evolution of each of them independently. Because of the fact of dividing the trouble in tractable portions, networks had been able to evolve, increasing many importance changes in phrases of speed, scale or variety of makes use of. Alternatively, there may be the control plane which, unlike the data plane, has no abstraction layers in any respect. Within the control plane there are several protocols to configure the network elements, which includes Multiprotocol Label Switching (MPLS) or NETCONF. Also there are others routing protocols like: Shortest Path Bridging (SPB), Enhanced Interior Gateway Routing Protocol (EIGRP) and Routing Information Protocol (RIP).

Protocols have a tendency to be described in isolation, but, with each solving a specific hassle and without the gain of any fundamental abstractions. This has ended in one of the primary barriers of nowadays's networks: complexity. as an instance, to add or circulate any device, IT ought to contact more than one switches, routers, firewalls, internet authentication portals, and many others. And replace ACLs, VLANs, Quality of Services (QoS), and other protocol-based totally mechanisms the use of tool-level control tools. Further, network topology, dealer switch model, and software model all ought to be taken into account. Due to this complexity, today's networks are relatively static because it seeks to minimise the threat of provider disruption.

On the way to cope with all that mess, masses of community tracking and control tools have appeared looking to assist the networks managers to have a control in their networks.

On the identical time that era advances, the communications requirements also evolve.

The claims that these days are wished in the exclusive services that customers demand, along with VoIP or streaming video in excessive exceptional, in which improbable while the architecture changed into designed.

All the ones elements have result in an over-provisioning of the networks, increasing the cost, and losing sources due the difficulty to optimise the manage and adapt the bodily sources to be had to the necessities of each instant.

So it's time to take a step forward and evolve to a more most suitable architecture, clean to manage, evolve and understand.

Here is wherein the SDN method involves play. Having a clean abstraction layers and a centralized control plane, it is plenty easier to make a more effective and dynamic management and control of the network, since we have a global overview of the network, and his absolute totalitarian control.

The motivation of this grasp's Thesis is to take advantage of the new centralized networking approach of SDN to broaden a load balancing algorithm that adapt the route of each waft depending at the cutting-edge state of the network with a view to obtain a better aid allocation, decreasing the overall cost and adapting its behaviour to the traffic growth.

## 1.2. Related Work

There is plenty of papers about traffic engineering trying to adapt the routing rules to the network conditions with the aim of taking the maxim profit of the available resources, and avoid traffic congestion. One of the ways to readapt the networks is using OSPF, by adjusting the state of the links thus the traffic can be adjusted to the current conditions .

They compare OSPF versus MPLS. Trying many different ways to adapt the weights setting to achieve a performance closer to the optimal in MPLS routing, but getting as a result that MPLS can achieve better performance. However, they also mention some advantages of OSPF against MPLS, since it is much simpler due that the routing is completely determined by one weight of each link, what avoids making decision per each source-destination pair. Also, if a link fails, the weights on the remaining links immediately determines the new routing. MPLS has many great advantages in supporting Traffic Engineering. MPLS allows efficient explicit routing of Label Switching Paths, which can be used to optimise the utilisation of network resources and enhance traffic oriented performance characteristics. For example, many several paths can be used at the same time to improve performance from a given source to destination. Is because of that MPLS has been a matter of study in a huge number of technical reports. They present three possible algorithms for non-priority unicast traffic that can be applied to MPLS networks, to map traffic flows to different routes to get the maximal network throughput and simultaneously enhance the total network resource utilisation.

On the other hand, Software-Defined Networking have become a pretty hot topic lately, and several research groups around the globe are investigating about its possibilities. Some are trying to propose some modifications of the most used protocol used nowadays to communicate the data plane with the control plane (i.e. OpenFlow) to achieve a more efficient flow management, since they think that OpenFlow introduces to much overhead and this suppose a problem of scalability. The prototyping and emulation of SDN networks have been also an important field of study, trying to provide the best way to allow everyone to experiment with this new networking approach in a realistic environment without the need of spending large amounts of money on it.

Other studies closest to the goal of the present project propose multi-path methods for a better utilisation of the resources, while maintaining a certain level of quality.

SDN is still in a primitive state, however, the amount of studies about this new networking approach is increasing every day, and the interest and new applications that are appearing leads to thing that it will be the future.

#### Possible Opportunities

One of the traits that is frequently associated with any basically new approach to technology is that there may be confusion approximately the opportunities that can be addressed by that new approach.

With a purpose to correctly evaluate and adopt a new approach to technology such as SDN, IT organizations want to identify which opportunity or oppotunities that are critical to the company are satisfactory addressed by means of that new approach.

After all the SDN-related discussions that have come about over the past couple of years, the subsequent have emerged because of the maximum probably set of possibilities that SDN can deal with.

- Aid the dynamic movement, replication and allocation of virtual resources.
- Ease the administrative burden of the configuration and provisioning of capability including QoS and security.
- More without difficulty installation, deployment and scalability of network functionality.
- Traffic engineering performance with an end-to-end view of the network.
- Better make use and exploitation of network resources.
- Reduce Operating Expense (OPEX).
- Have network functionality evolve faster than expected based on a software development lifecycle.
- Allow applications to dynamically request services from the network.
- Put in force extra effective security functionality.
- Reduce complexity.



### 1.3. Problem formulation

Data traffic in networks has been increasing exponentially since the beginning of networking at the same time that new kind of services and connection requirements appear. Those factors have led to a complex networks that cannot satisfy the needs of carriers nor users.

As explained in this introduction chapter, the current networks have several limitations to adapt to end-to-end goals requirements for the different sort of services without wasting the existing resources.

Determining resource allocation per class of service must be done with knowledge about traffic demands for the various traffic classes, keeping a fixed amount of bandwidth for each class, which results in a poor utilisation of resources. However, the traffic that the network has to carry change constantly in an unpredictable way.

Nowadays methodologies are static, or need specific network equipment, which leads to a non-scalable and expensive systems. A new centralised control plane paradigm approach, with a global view and control of every element in the network, can improve significantly the management of the network. Allowing to decide the route for every single flow, taking into account the characteristics of these in terms of bandwidth utilisation or other parameters, to accomplish a much better utilisation of the resources.

The aim of this project is to analyse the possibilities that Software-Defined Networking brings to us to develop an application able to sense the state of the network and adapt its behaviour in order to achieve a better performance and better resource utilisation of the network.

However, resource utilisation is not the unique important factor. In order to guarantee a certain quality for the carried traffic, It's crucial to keep some parameters, such as delay or jitter, within a bounded limits.

Thus, this project tries to assess the possibilities brought by SDN to manage the route per each flow in order to take the maximum advantage of the available bandwidth in the network.

## 2. Introduction to Software Defined Network

Conventional network architectures are not enough suited to fulfill the requirements of today's organizations, companies, and users.

Way to a broad industry effort spearheaded through the Open Networking Foundation (ONF), Software-Defined Networking (SDN) is transforming networking structure. Inside the SDN structure, the control and data planes are decoupled, network intelligence and state are logically centralized, and the basic network infrastructure is supported from the applications. As a result, organizations and vendors advantage unprecedented programmability, automation, and network control, enabling them to construct highly scalable, flexible networks that easily adapt to converting enterprise needs. The ONF is a non-profit enterprise consortium that tends to the advancement of SDN and standardizing critical factors of the SDN structure along with the OpenFlow protocol, which structures communication among the control and data planes of supported network devices.

OpenFlow is the primary standard interface designed in particular for SDN, providing high-performance, granular traffic control throughout more than one companies' network devices.

OpenFlow-based SDN is currently being spread in an variety of networking devices and software, delivering important benefits to both vendors and enterprises, such as:

- Centralized control and manipulate of networking devices from more than one vendors.
- Improved automation and management by means of the usage of common APIs to summary the basic and most important networking details from the orchestration and provisioning systems and applications.
- Speedy innovation via the capability to deliver new network skills and services without the demand to configure individual devices or await vendor releases.
- Agility: Summarizing control from forwarding leaves administrators dynamically adjust network-wide traffic to satisfy changing needs.
- Directly programmable: Network control is directly programmable because it is decoupled from forwarding features.
- Programmability with the aid of operators, organizations, unbiased software companies, and customers (not only equipment manufacturers) using common programming environments, which gives all parties new possibilities to drive profits and differentiation.. SDN gives to network managers the ability to configure, control, secure, and optimize network equipment really fast via dynamic, automated SDN software programs which they can write themselves since the programs do not belong on proprietary software.
- Increased network security and reliability because of centralized and automated management of network devices, homogenous policy enforcement, and less configuration mistakes.
- More fragmented network control with the capability to use comprehensive and wide-ranging policies at the, user and device levels and the session and application layers.

- Open standards-based and vendor-neutral: When implemented with open standards, SDN makes the network design and the operation much more simpler because instructions are provided by SDN controllers instead of multiple, specific vendor's devices and protocols.
- Better end-user experience : Applications use centralized network state information which limit the potential barriers to adapt network behavior to users needs.
- SDN offers a dynamic and flexible network architecture that protects current investments while proofing the network for the future. With SDN, these days's static network can evolve into an extensible service shipping platform capable of responding unexpectedly to changing business, market, and end-user needs.

The explosion of mobile devices and content, server virtualization, and introduction of cloud services are the various trends driving the networking industry to reexamine conventional network architectures. Many traditional networks are hierarchical, constructed with levels of Ethernet switches arranged in a tree structure. This design matched better when client-server computing was prevalent, but the sort of static structure is unwell-perfect to the dynamic computing and storage requirements of these days' s business enterprise data centers, campuses, and service environments.

Some of the most important computing tendencies riding the need for a new network paradigm include: Changing traffic patterns: Within the business unit data center, traffic patterns have modified substantially. In contrast to client-server applications wherein the bulk of the communication takes place between one client and one server, today's applications get entry to distinct databases and servers, putting the bases of "east-west" machine-to-machine traffic before returning data to the end user device inside the conventional "north-south" traffic pattern. At the equal time, users are changing network traffic patterns as they push for get entry to corporate content and applications from any type of device (such as the one belongs to them), connecting from everywhere, at any time. Finally, many enterprise data centers managers are thinking of a utility computing model, which may consist of a private cloud, public cloud, or a few blend of both, resulting in additional traffic across the wide area network.

The "consumerization of IT": The number of users who employ mobile personal devices such as smartphones, tablets, and notebooks to access the corporate network, is constantly increasing. The services provided for these personal devices in a fine-grained manner, make IT under pressure since at the same time they have to protect corporate data and intellectual property and meeting compliance mandates.

The rise of cloud services: Enterprises have enthusiastically embraced each public and personal cloud services, resulting in exceptional growth of these services. Enterprise business units now are looking for the dexterity to access applications, infrastructure, and other IT resources on demand and à la carte. As far it concerns the complexity, IT's planning for cloud services have to be done in an environment of increased security, conformity, and auditing requirements, including business reorganizations, consolidations, and mergers that can change common beliefs unexpectedly.

Providing self-service provisioning, whether in a private or public cloud, calls for elastic scaling of computing, storage, and network resources, preferably from a common viewpoint and with a common suite of tools.

“Big data” equals to additional bandwidth: Administrating today’s “big data” or huge datasets requires big parallel processing on thousands of servers, all of which need direct connections to every other. The rise of mega datasets is fueling a continuous demand for extra network capacity in the data center. Operators of hyperscale data center networks face the daunting task of scaling the network to previously not possible size, preserving any-to-any connectivity without going broke.

Meeting present day market necessities is truly impossible with traditional network architectures. Faced with flat or decreased budgets, enterprise IT departments are looking to squeeze the most from their networks using device-level management tools and manual techniques. Carriers face comparable challenges as call for mobility and bandwidth explode, profits are being eroded by escalating capital equipment costs and flat or declining profits. Present network architectures were not designed to meet the requirements of today’s users, organizations, and companies, rather network designers are limited by the limitations of today’s networks, which include:

Complexity that results in stasis: Networking technology up to now has consisted largely of discrete sets of protocols designed to connect hosts reliably over arbitrary distances, link speeds, and topologies. To fulfill business and technical requirements over the last few decades, the industry has extended the capabilities of networking protocols to provide higher performance and reliability, broader connectivity, and more strict security.

Protocols have a tendency to be defined in isolation, however, with each solving an exact problem and without the advantage of any essential abstractions. This has led to one of the primary limitations of these day’s networks, such as complexity. For example, to add or move any device, IT need to deal with more than one switches, routers, firewalls, web authentication portals, etc. and update VLANs, ACLs, Quality of Services (QoS), and other protocol based mechanisms using device-level management tools. Furthermore, network topology, vendor switch model and software version all need to be taken into consideration. Because of this complexity, these days’ s networks are relatively static as IT tries to minimize the risk of service disruption. The static nature of networks is in full collation to the dynamic nature of these days’ s server environment, where server virtualization has greatly increased the variety of hosts requiring network connectivity and completely modified assumptions about the physical location of hosts. When virtualization was not used, applications were placed on a single server and primarily exchanged traffic with selected clients. Nowadays, applications are distributed with multiple virtual machines (VMs), which exchange traffic flows with each other. VMs migrate to optimize and rebalance server workloads, making the physical end points of current flows to switch (sometimes really fast) over the period of time. VM migration challenges many factors of traditional networking, from addressing schemes and namespaces to the basic notion of a segmented, routing-based design.

In addition to adopting virtualization technologies, many enterprises nowadays perform an IP converged network for voice, data, and video traffic. As far as existing networks can offer differentiated QoS levels for different applications, the provisioning of these resources is exceptionally manual. IT must configure each supplier’s equipment seperately, and regulate parameters including

network bandwidth and QoS on a per-session,per-application basis. Because of its static nature, the network cannot dynamically conform to changing traffic, application, and user needs.

**Inconsistent policies:** To put in force a network-wide policy, IT should have to configure thousands of devices and mechanisms. As an example, whenever a new virtual machine is geminated, it can take hours or days in some cases, for IT to reconfigure ACLs across the entire network. The complexity of nowadays' s networks increases the difficulty for IT to apply a consistent set of access, security, QoS, and many other policies to an increasing number of mobile users, which leaves the enterprise at risk of security breaches, non-compliance with policies, and other negative results.

**Inability to scale:** Because of the rapidly grow of the data center, the network must grow respectively. However, the network becomes increasingly more complicated with the addition of hundreds or thousands of network devices that need to be configured and managed. IT has also based on link oversubscription to scale the network, relied on predictable traffic patterns; but, in these day's virtualized data centers, traffic patterns are incredibly dynamic and consequently unpredictable.

Mega-operators, like Google, Facebook, and Yahoo!, face even more daunting scalability demanding situations. These service providers use large-scale parallel processing algorithms and relevant datasets throughout their entire computing pool. As the scope of end-user applications increases (for instance, crawling and indexing the entire world wide web to directly return search results to users), the quantity of computing elements increases dramatically and data-set exchanges among compute nodes can reach petabytes. These companies need as-called hyperscale networks that can offer high-performance, low-cost connectivity amongst hundreds of thousands (possibly millions) of physical servers. Manual configuration cannot support such scaling.

In order to stay within the context of competition, carriers must deliver ever-higher value, better-differentiated services to their potential customers. Multi-tenancy in addition complicates their task, as the network has to serve groups of users with different applications and different performance needs. Key operations that seem rather straightforward, such as managing a customer's traffic flows to serve customized performance control or on-demand delivery, are very complicated to implement with present networks, especially at carrier scale. They require specialized devices at the network edge, which increase capital and operational expenditure as well as time-to-market to provide new services.

**Vendor dependence:** Carriers and enterprises seek to develop new competencies and services in fast response to changing business needs or user demands. However, their skill to respond is prevented by vendors' equipment product cycles, which can range to three years or greater. Lack of standard, open interfaces limits the ability of network operators to adapt the network to their individual environments. This mismatch among market necessities and network competencies has brought the industry to a tipping point. In response, the industry has discovered the Software-Defined Networking (SDN) architecture and is developing related standards.

## 2.1. Software-Defined Networking

Software Defined Networking (SDN) is a rising network architecture where network control is decoupled from forwarding and is directly programmable. This migration of control, previously tightly pinioned in individual network devices, into accessible computing devices allows the underlying infrastructure to be summarized for applications and network services, which can handle the network as a logical or virtual entity.

Figure 1 reproduces a logical view of the SDN architecture. Network intelligence is (logically) centralized in software-based SDN controllers, which preserve a global view of the network. In conclusion, the network shows up to the applications and policy engines as a single, logical switch. With SDN, enterprises and carriers achieve vendor-independent control over the entire network from a single logical point, which extremely simplifies the network design and operation. SDN also makes much simpler the network devices themselves, since they no longer need to comprehend and process thousands of protocol standards but simply accept instructions from the SDN controllers.

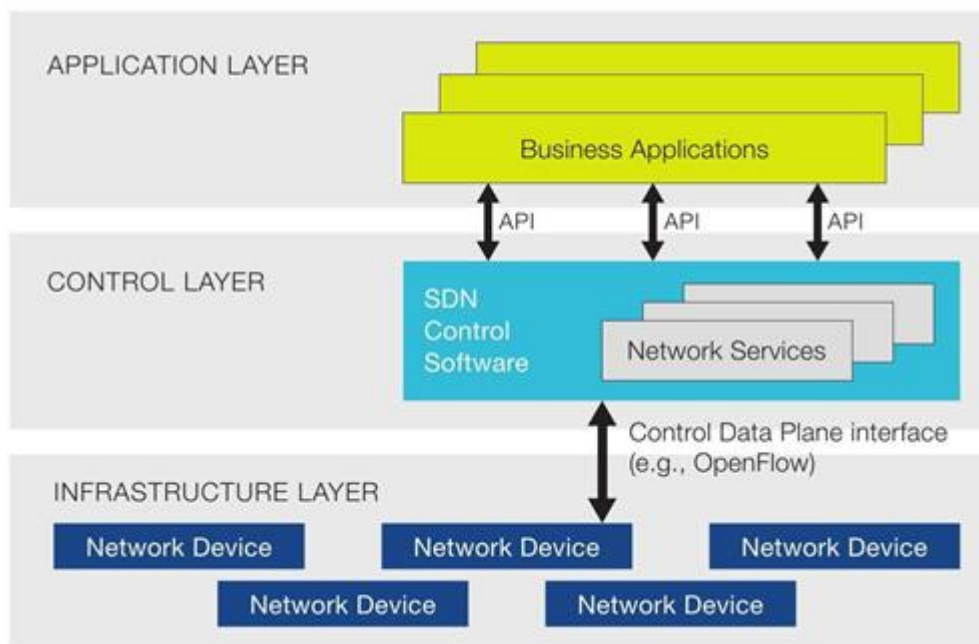


FIGURE 1: Software-Defined Network Architecture

Perhaps most meaningful, network operators and administrators can programmatically configure this simplified network abstraction rather instead of having to hand-code tens of thousands of lines of configuration disunited among thousands of devices. Also, reclaiming the SDN controller's centralized intelligence, IT can change network behavior in real-time and develop new applications and network services in a matter of hours or days instead of the weeks or months required today. By centralizing network state in the control layer, SDN confers network managers the flexibility to configure, manage, secure, and optimize network assets through dynamic, automated SDN programs. Furthermore, they can write these programs themselves and not wait for features to be inserted in vendors' proprietary and closed software environments in the middle of the network. In addition to abstracting the network, SDN architectures contend a set of APIs that make it possible to implement common network services, together with routing, multicast, security, access control, and bandwidth

management, Quality of Service, traffic engineering, processor and storage optimization and energy use, and all kinds of policy management, custom modified to meet business targets. For instance, an SDN architecture makes it easy to determine and impose consistent policies across both wired and wireless connections as well, SDN makes it possible to manage the entire network with intelligent orchestration and provisioning systems. The Open Networking Foundation is perusing open APIs to promote multi-vendor management that shows the door for on-demand resource allocation, self-service provisioning, verily virtualized networking, and secure cloud services. So, using open APIs among the SDN control and applications layers, business applications can operate in an absence of the network, leveraging network services and capabilities without having to be tied to the details of their implementation. SDN makes the network not enough “application-aware” like “application-customized” and applications not enough “network-aware” like “network-capability-aware”. In conclusion, computing, storage, and network resources can be optimized.

Below is a description of some of the key concepts which are part of the SDN system architecture that is shown in Figure 1.

#### Business Applications

This refers to applications that are directly accesible by end users. Possibilities consist of video conferencing, supply chain management and customer relationship management.

#### Network & Security Services

This refers to the effective operation that enables business applications to perform efficiently and securely. Possibilities include a wide range of L4 – L7 functionality including ADCs, WOCs and security capabilities like IDS/IPS, firewalls and DDoS protection.

#### Pure SDN Switch

In a pure SDN switch, all of the managed functions of a conventional switch (i.e., routing protocols that are used to build forwarding information bases) are running in the central controller. The functionality in the switch is confined entirely to the data plane.

#### Hybrid Switch

In a hybrid switch, traditional switching protocols and SDN technologies run at the same time. While a network manager configuring the SDN controller to discover and control certain traffic flows, the traditional, distributed networking protocols will continue to direct the rest of the traffic on the network.

#### Hybrid Network

A hybrid network is a network in which traditional switches operate in the same environment with and SDN, regardless if they are pure SDN switches or hybrid switches.

#### Northbound API

As shown in Figure 1, the northbound API is the API that enables communications between the control layer and the business application layer. It does not exist at present a standards-based northbound API.

## Southbound API

As shown in Figure 1, the southbound API is the API that enables communications between the control layer and the infrastructure layer. The protocols that can enable this communications are the followings: OpenFlow, the extensible messaging and presence protocol (XMPP) and the network configuration protocol.

Part of the complication that surrounds SDN is that many vendors don't buy in completely to the ONF definition of SDN. For example, when some vendors are seeing OpenFlow as a foundational section of their SDN solutions, other vendors are expecting and see approach to OpenFlow. Another source of confusion is dissent comparative to what constitutes the infrastructure layer. To the ONF, the infrastructure layer is a wide range of physical and virtual switches and routers. In the description below, one of the current approaches to performing network virtualization relies on an architecture that seems similar to the one shown in Figure 1, but which only comprises virtual switches and routers.

Software-Defined Networking is an emerging paradigm that enables network novelty based on four basic principles: network control and forwarding planes are plainly decoupled, forwarding decisions are flow-based instead of destination-based.

The network forwarding logic is removed from hardware to a programmable software layer, and an element, called a controller, is imported to coordinate network-wide forwarding decisions.

The SDN architecture, as shown in figure 1, is based on a principle of disjunction of the control plane or the network plane from the forwarding hardware and a logical centralization of a control program or a controller that composes forwarding decisions and installs canons on switches or routers. These canons make the forwarding hardware to switch packets among ports. According to this architecture, the SDN can logically be presented as a three-layered architecture:

**Infrastructure layer:** This layer is often mentioned as a data plane. It involves forwarding hardware, for instance, switches and routers, with the forwarding components, and Application Programming Interfaces (API).

**Control layer:** Its' other name is a control plane. Network intelligence in a type of logically centralized and software-based SDN controller installed using any UNIX based Operating System (OS) run on any hardware. The control layer manages forwarding hardware and installs forwarding rules through APIs.

**Application layer:** Applications and services take control over control and infra-structure layer via Representational state transfer (REST) APIs. The SDN concept enables developers to simply develop applications that implement networking function tasks. Applications are ordinarily deployed to



segregate computers or clouds.

The APIs in the SDN architecture are frequently called northbound and southbound interfaces. Those are used for the communication among hardware, controllers and applications. A southbound interface is declared as the connection among networking devices and controllers, as the northbound interface is the connection among applications and the controllers.

## Data Plane

The data plane infrastructure consists of networking devices. A network device is an occurrence that receives packets on its ports and executes one or more network functions on them. For instance, a received packet can be forwarded, or dropped, or its header can be changed. The packet forwarding function is relied on a Forwarding Information Base (FIB) table or tables, which are placed on a forwarding hardware and include MAC addresses mapped to ports, preprogrammed by the control plane.

In contrast, the SDN technology uses flow tables that are not the same as FIB tables. The FIB table is an ordinary set of instructions based on destination-based switching, while the flow table includes a sequential set of instructions and actions. The flow tables will be covered more comprehensively in another section.

Occasionally, the data plane is referred to as the fast path for packet management because it requires no further search other than an address extraction of the packet destination counting on preprogrammed FIB. However, one exception that exists in the process mentioned previously is if the packet destination is not found from the lookup tables. In order to continue, the detected packet with an unknown address is sent to the control plane where the controller takes a forwarding decision using the Routing Information Base (RIB) table that contains topology, network destinations and metrics associated with routes. With SDN, it resides on a software-based controller.

FIB tables can reside in a worthy number of further targets - software, hardware-accelerated software (Graphics Processing Unit (GPU)/Central Processing Unit (CPU), as exemplified by Intel or ARM), commodity silicon (Network Processing Unit (NPU), as clarified by Broadcom, Intel, or Marvell, in the Ethernet switch market), Field-Programmable Gate Array (FPGA) and specialized silicon (Application-Specific Integrated Circuits (ASIC) like the Juniper Trio), or any combination of them, depending on the network element design.

Moreover, concerning forwarding decisions, the data plane may include other small features and services usually referred as forwarding features. Depending on a system, a separate discrete table may undergo for these features or they can operate as extensions to the forwarding tables.

For instance, the below features can be used in SDN as good as in traditional routing:

**Access Control List (ACL):** It is used to specify drop, change, pass and some other actions for a specific matching flow.

**Quality of Service (QoS):** It is mapping a flow to a queue on an outlet to normalize and provide an incessant service. It may determine packets to be dropped irrespective of the forwarding policy.

On Figure 2 are shown examples of features that are available on a traditional router and which can be used in SDN.

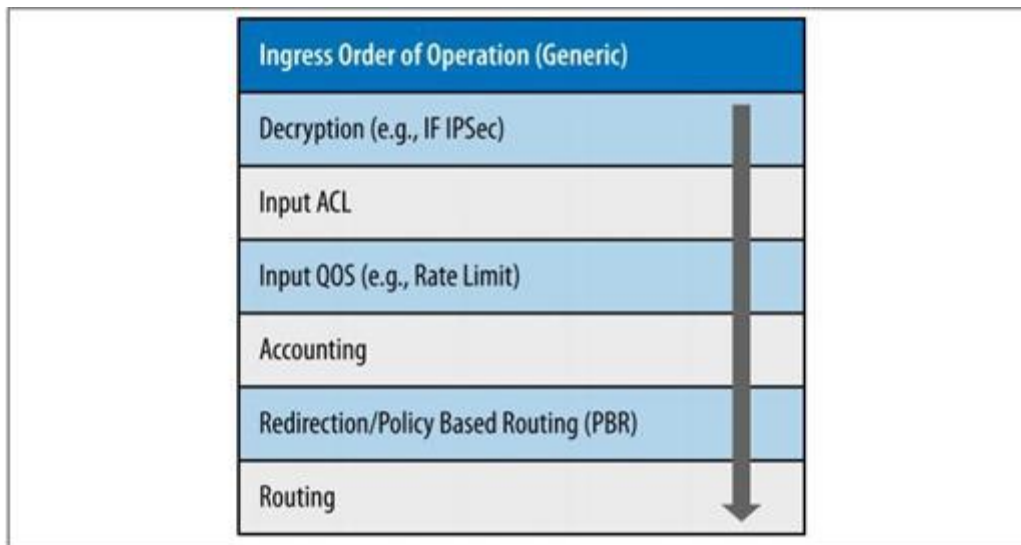


Figure 2: Generic example of ingress feature application on a traditional router.

Also, there is one technology that provides networking functions in SDN, which were removed from the data plane, and is called Networking Functions Virtualization (NFV). It is a networking concept allowing to virtualize entire classes of networking functions into building blocks. Those can be connected or chained in order to create networking services. Examples of the services are Network Address Translation (NAT), Authentication, Authorization and Accounting (AAA) and Secure Sockets Layer (SSL) protocol. A system example of SDN, NFV and clouds can be OpenDaylight controller with OpenStack cloud technology. This combination provides fast provisioning, advanced NFV features and scalability.

Referring to hardware, there are white box switches which exist for SDN purposes. They represent foundational elements of the networking infrastructure that enable organizations to choose and use only those features they need to realize as their SDN objectives.

The white box switches can come with pre-installed OS with minimal software installed or be sold as a bare metal device. This allows users and businesses to customize switches according to their needs.

### Southbound Interfaces

The APIs of the Southbound interface enable an SDN controller to efficiently control a networking infrastructure and make dynamic changes in accordance with real-time traffic, its demands and needs.

Several examples of southbound APIs exist:

OpenFlow: An SDN southbound protocol proposed to be a standard protocol in industry and used to transport a control logic function from a switch to a controller.

Open vSwitch DataBase Protocol (OVSDB): Management protocol used by Open vSwitch open source software switch.

Locator ID Separation Protocol (LISP): It provides a flexible map-and-encap framework that can be used for overlay network applications, such as data centre network virtualization, and Network Function Virtualization (NFV).

Network Configuration Protocol (NETCONF): The protocol is used for networking devices configuration.

## 2.2. Network Virtualization

Network virtualization is not a new topic since network organizations have a long history on implementing techniques such as virtual LANs (VLANs), Virtual Private Networks (VPNs) and Virtual Routing and Forwarding (VRF). However, throughout this thesis, the phrase *network virtualization* refers to the capability shown in the right half of Figure 3. Specifically, network virtualization refers to the ability to supply end-to-end networking that are abstracted away from the details of the underlying physical network which is similar with the way that server virtualization applies compute resources that are abstracted away from the details of the underlying x86 based servers.

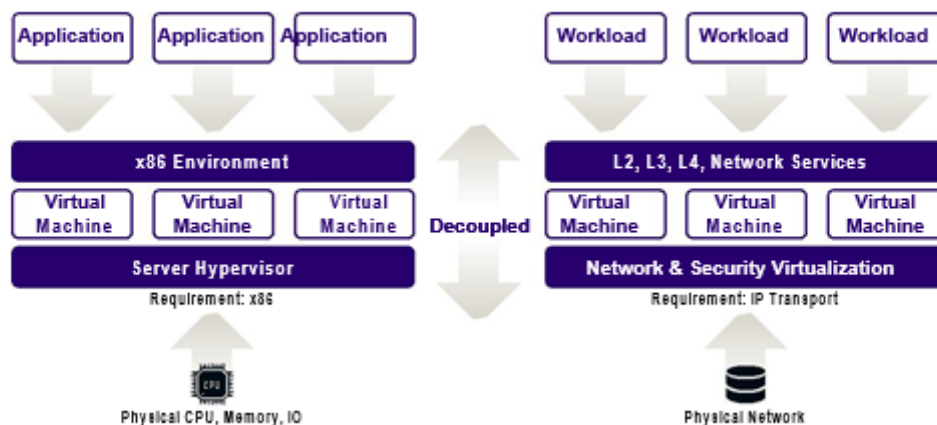


FIGURE 3: Network Virtualization Source: VMware

One way to implement network virtualization is like an application that runs on a SDN controller, uses the OpenFlow protocol and defines virtual networks which are based on policies that map flows to the appropriate virtual network using the L1-L4 portions of the header. This way of implementation is often referred to as fabric-based network virtualization.

One more way to implement network virtualization is to use encapsulation and tunneling to create multiple virtual network topologies overlaid on a common physical network. This method is usually

mentioned as overlay-based network virtualization. IT foundations have been implementing network virtualization via overlays for the last few years based on protocols such as VXLAN. However, the initial wave of these solutions didn't feature a controller. Since these deficient solutions typically used flooding as a way to share information about the end systems, these solutions didn't scale well.

Figure 4 reenact a more recent approach to implementing network virtualization. This approach uses a controller which has an architecture similar to the one shown in Figure 1 except that the network topology consists of vRouters or vSwitches. One of the primary roles of the controller in Figure 4 is to provide tunnel control plane functionality. With this functionality is allowed to the ingress device to apply a mapping operation that determines where the encapsulated packet should be sent to reach its intended destination VM.

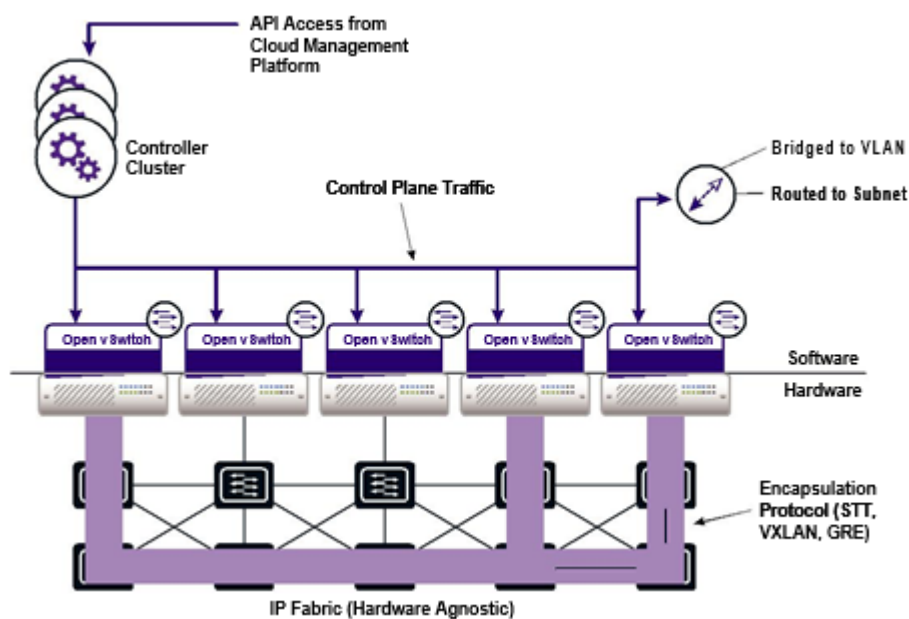


FIGURE 4: Overlay-Based Network Virtualization **Source:** VMware

Regarding the approach to network virtualization that is shown in Figure 3, a virtual network can be a Layer 2 network or a Layer 3 network, while the physical network can be Layer 2, Layer 3 or a combination, depending on the overlay technology. With overlays, inside the outer header is included a field that is generally 24 bits in length, which can be used to identify roughly 16 million virtual networks. However, practical limits are often between 16,000 and 32,000 virtual networks. According to the approach shown in Figure 3, virtualization is performed at the network edge, while the remainder of the physical L2/L3 network remains the same and does not need any configuration modifications in order to support the virtualization of the network.

The primary advantage of an overlay-based network virtualization solution is that it provides support for virtual machine mobility without the dependence of the physical network. So, if a VM is placed to a different location, even to a new subnet, the switches at the edge of the overlay simply have the

ability to update their mapping tables to reflect the new location of the VM.

As a conclusion, we can mention that SDN is comprised of many enabling technologies, which respectively makes SDN not just a technology, but an architecture. Either it is overlay-based or fabric, network virtualization can be viewed as a SDN application. The primary benefit of a network virtualization solution is that it provides support for virtual machine mobility independent of the physical network. SDN, however, has other positive benefits including simplifying the administrative burden of provisioning functionality such as QoS and security.

At the same time that some of the characteristics of a SDN, such as the increased reliance on software, are already widely adopted in the marketplace, vendors have only recently begun to suggest and offer SDN solutions. The sovereignty and consolidation of SDN are just beginning.

Given all of the promising benefits that SDN is likely to provide, IT organizations need to develop a plan for how they will develop their networks to incorporate SDN.

### 2.3. OpenFlow Protocol

OpenFlow is the first standard communications interface, which defined between the control and forwarding layers of an SDN architecture. OpenFlow allows direct access to and management of the forwarding plane of network devices, such as switches and routers, regardless if they are physical or virtual (hypervisor-based). The absence of an open interface to the forwarding plane has led to the characterization of today's networking devices as monolithic, closed, and mainframe-like.

There is no other standard protocol to do what OpenFlow does. A protocol like OpenFlow is needed to transfer network control out of the networking switches to logically centralized control software. OpenFlow can be compared to the instruction set of a CPU. As shown in Figure 5, the protocol specifies basic primitives which can be used by an external software application to program the forwarding plane of network devices, just like a CPU's instruction set would program a computer system.

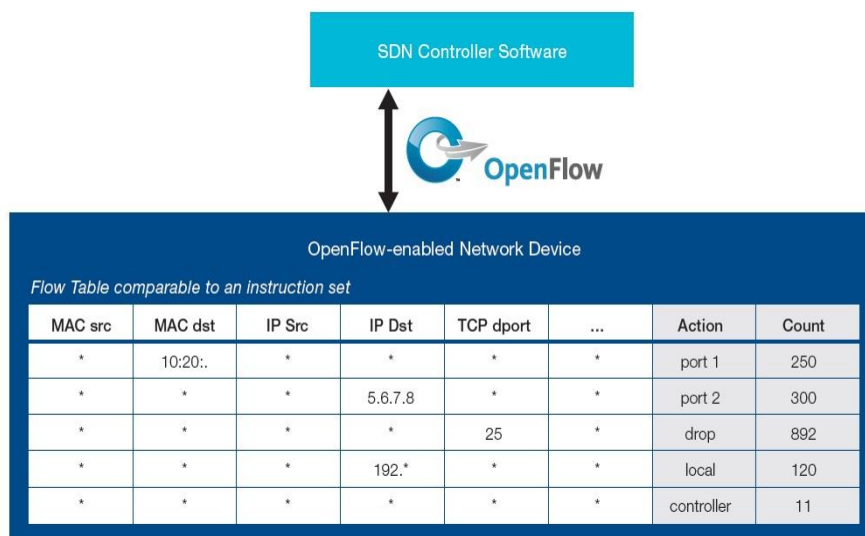


FIGURE 5: Example of OpenFlow Instruction Set

The OpenFlow protocol is implemented on each of the two 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 rules which can be statically or dynamically programmed by the SDN control software. It also enables IT to define how traffic will have to flow through network devices based on parameters such as usage patterns, applications, and cloud resources. Because OpenFlow allows the network to be programmed on a per-flow basis, an OpenFlow-based SDN architecture provides very satisfactorily granular control, giving the ability to the network to respond to real-time changes at the application, user, and session levels. Current IP-based routing does not provide this level of control, since all flows between two endpoints must follow the same path through the network, although they have different requirements.

The OpenFlow protocol is a key enabler for software-defined networks and until now is the only standardized SDN protocol that allows direct manipulation of the forwarding plane of network devices. OpenFlow switching can extend to a much broader set of use cases, although initially was applied to Ethernet-based networks. OpenFlow-based SDNs can be deployed on existing networks, both physical and virtual as well. Network devices can support OpenFlow-based forwarding as equally as traditional forwarding, which makes it very feasible for enterprises and carriers to progressively start the peration of OpenFlow-based SDN technologies, even in multi-vendor network environments.

The Open Networking Foundation is on track to standardize OpenFlow and does so through technical working groups responsible for the protocol, configuration, testing, interoperability and other activities, helping to ensure the best functionality between network devices and control software from different vendors. OpenFlow is being widely recognized by infrastructure vendors, who typically have implemented it using a simple firmware or software upgrade. The OpenFlow-based SDN architecture can integrate seamlessly with an enterprise or carrier's existing infrastructure and provide an ordinary migration path for those segments of the network where need to strengthen SDN's functionality.

Communication between an SDN controller and networking devices is handled through the southbound APIs. One of the most popular protocols is OpenFlow. It was first pro-posed as a way for researchers to conduct experiments in production networks. How-ever, its advantages led to it being used beyond research. Due to its flexibility, the protocol provides a flexible routing of network flows without interrupting other traffic. This possibility was achieved by separating the data and control plane, where a controller orchestrates network traffic and makes or changes rules in networking devices.

OpenFlow does not require any hardware changes and vendors do not have to open their systems to support it as long as only a few well-known interfaces have to be open for the control plane. The Ethernet switches just need some software changes that can be provided as a firmware extension for the existing hardware. The reason why it is possible is the fact that all the modern switches and routers contain flow tables that are used for firewalls, Quality of Service (QoS), NAT, and for statistics collection. Although, the implementation of the flow tables may vary from one vendor to another, there is a com-mon set of functions that run on many switches and routers. The OpenFlow protocol exploits them.

As mentioned above, using the OpenFlow protocol, a controller is able to not only set forwarding rules by sending flow entries, but learn the network statistics, hardware de-tails, ports, connectivity status and the network topology of Ethernet switches.

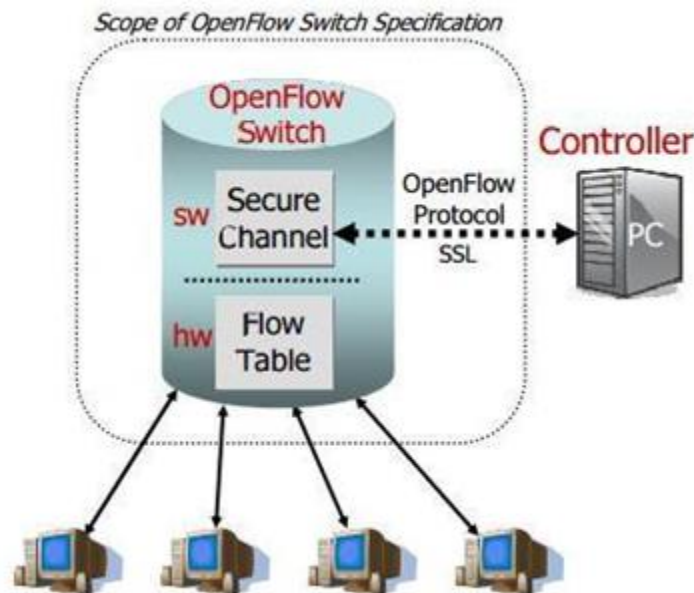


FIGURE 6: OpenFlow switch specs

As shown in figure 6, an OpenFlow Switch consists of at least three parts: (1) A Flow Table, which applies an action associated with each flow entry, to inform the switch how to process the flows, (2) a Secure Channel that connects the switch to a remote control process (called the controller), which allows commands and packets to be sent between the controller and a switch using (3) the OpenFlow Protocol, which provides an open and standard way for the controller to communicate with a switch.

**Flow tables and group tables** consist of flow entries that define how switches should behave with a traffic flow coming from/to different physical and virtual interfaces. The tables of an OpenFlow switch are numbered starting from 0. First, incoming packets are matched against flow entries in table 0. When a flow entry is found, the instructions included in that entry will be executed against the packet. The instructions may explicitly send the packet to another flow table, repeating the process again and again. A flow entry can only direct the packet forward, not backwards (see figure 7), increasing the numbering of the tables. When the packet processing pipeline stops, the packet will be processed in accordance with the instructions set matching this packet.

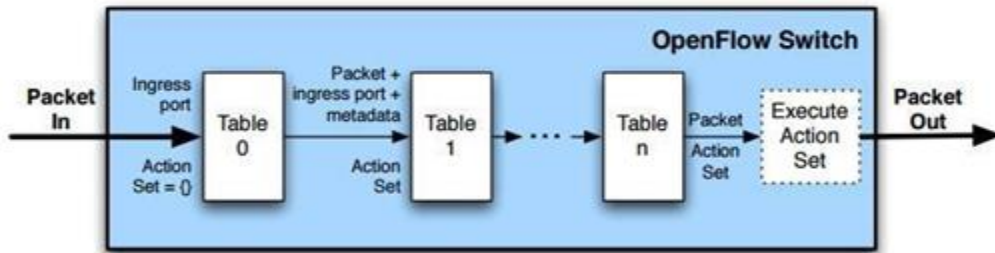


FIGURE 7: Packets match against tables

If a flow table for a packet is missing, it is called a table miss. There are several instructions that can be set for this situation. Options include dropping the packet, passing it to another flow-table or sending it to a controller over the control channel as shown in figure 8.

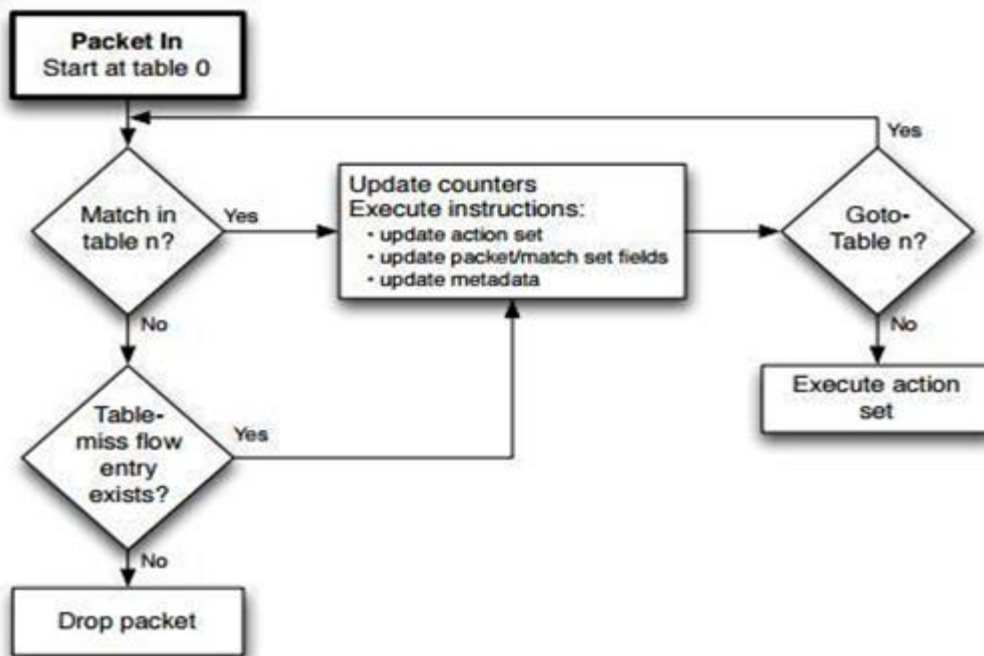


FIGURE 8: Packets Flowchart

The flow entries in the flow tables have the following structure (see figure 9):

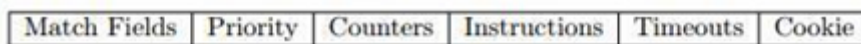


FIGURE 9: Main components of a flow entry in a flow table.

Explanation of the fields is the following:

**Match Fields:** Include ingress port and L2-L4 packet headers. As an option, metadata can be specified by a previous table.

**Priority:** Contains flow entry precedence.



**Counters:** Increased if there is a packet match.

**Instructions:** Used for action set modification

**Timeouts:** Maximum time when flow is expired by the switch

**Cookie:** Opaque data value chosen by the controller, which may be used by the controller to filter flow statistics, flow modification and flow deletion (they are not used when processing packets).

OpenFlow Channel is the interface that is used for a connection between each Open-Flow switch and a controller. Using this interface, the controller can receive switch generated events, send packets out of the switch and, last but not least, configure and manage the switch. The connection between the switch and the controller is encrypted with Transport Layer Security (TLS) protocol. However, it may be run directly over Transmission Control Protocol (TCP).

## Control Plane

The responsibility of the control plane is to configure and manage the data plane devices over southbound interfaces and instruct the networking devices how to handle packets. It is usually distributed and can involve several controllers at the same time, which is called clustering. Examples of the controllers that reside on the control plane are OpenDaylight, Floodlight, Ryu and POX/NOX.

The main functionalities of the control plane are:

- Maintenance and discovery of topology.
- Instantiation and selection of a packet route.
- Mechanisms for path failover.

Moreover, the plane also provides services for applications to use the data plane and data gathered by the control plane to provide other functions within the network.

Examples of the applications include network provisioning, advanced network topology discovery, path reservation and firewall. Communication between the control plane and the applications is established through Northbound APIs.

## Southbound Interfaces

The role of the northbound interface is to provide a high-level API between the controller and applications that can compute network operations based on the events collected by the control plane. Figure 10 represents the existing APIs in the SDN system.

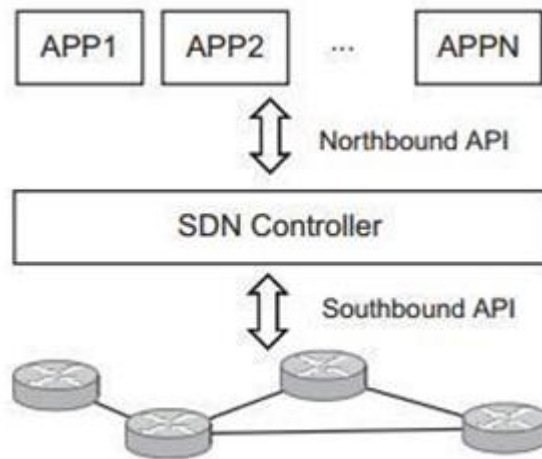


FIGURE 10: The architecture of Software-Defined Network.

Orchestration platforms, such as OpenStack, and automation stack, such as Puppet or CFEngine, also use the SDN northbound APIs to integrate with the system. The main goal of the APIs is to abstract the underlying network infrastructure and enable application developers to have their attention on the application development instead of understanding how their changes can affect the inner-working of the network.

The main architectural style for northbound API is Representational State Transfer (REST). The REST APIs support flexibility and can provide different functions at the same time and make changes to those functions without interrupting clients. This is achieved by the mechanism called hypertext driven navigation of the connected resources.

In particular, a REST API consists of the connected REST resources that provide different services through uniform interfaces. In the case of SDN, it includes services from both data and control planes, such as switches, routers, subnets, networks, NAT devices, and controllers.

The example of the REST GET request looks like:

```
controller> curl -u admin:admin -H 'Accept: application/xml'
'http://<controller-ip>:8080/con-troller/nb/v2/flowprogram-
mer/default'
```

The output of this query shows installed flows in networking devices.

### 2.3.1. Benefits of OpenFlow-Based Software-Defined Networks

For both enterprises and carriers, SDN makes it possible for the network to be a competitive differentiator, not just an unavoidable cost center. OpenFlow-based SDN technologies give the ability to IT to address the high-bandwidth, dynamic nature of today's applications, enable customizing the network depending on ever-changing business needs, and significantly minimize operations and management complexity. As mentioned before, the benefits that enterprises and carriers can achieve through an OpenFlow-based SDN architecture, analytically include the following:

- Centralized control of multi-vendor environments: SDN's control software can manage the configuration of any OpenFlow-enabled network device regardless of vendor, including switches, routers, and virtual switches. Instead of having to manage groups of devices from individual vendors, IT can take advantage of SDN-based orchestration and management tools to rapidly deploy, configure, and update devices across the entire network.
- Reduced complexity through automation: OpenFlow-based SDN offers flexibility on network automation and management framework, which makes it capable to develop tools that automate many management manipulations that are done manually today. These automation tools will reduce operational overhead, minimise network instability introduced by operator error, and support emerging IT-as-a-Service and self-service provisioning models. Additionally, with SDN, cloud-based applications can be managed through intelligent orchestration and provisioning systems, further increasing business agility by reducing operational overhead.
- Higher rate of innovation: SDN adoption accelerates business innovation by giving the ability to IT network operators to completely program (and reprogram) the network in real time in order to cover specific business needs and user requirements according to their will. By virtualizing the network infrastructure and abstracting it from individual network services, for example, SDN and OpenFlow give IT (and possibly even to users) the ability to adapt the behavior of the network and introduce new services and network capabilities in a few hours.
- Increased network reliability and security: Using SDN makes it possible for IT to define high-level configuration and policy statements, which are then translated on the way to the infrastructure via OpenFlow. An OpenFlow-based SDN architecture eliminates the dependence to configure network devices separately each time an end point, service, or application is added or moved, or a policy changes, which reduces the possibilities of network failures due to configuration or policy inconsistencies.

SDN controllers provide complete visibility and control over the network, for this reason they can ensure that access control, traffic engineering, quality of service, security, and other policies are enforced regularly across the wired and wireless network infrastructures, including branch offices, campuses, and data centers. The reduced operational expenses, the more dynamic configuration

capabilities, the fewer errors, and the consistent configuration and policy enforcement, significantly benefit enterprises and carriers.

**More granular network control:** OpenFlow's flow-based control model enables IT to apply policies at a very granular level, including user, device, levels and the session and application layers, in a highly abstracted automated fashion. This kind of control enables cloud operators to support multi-tenancy while maintaining traffic isolation, security, and elastic resource management at the same time that customers use the same infrastructure.

**Better user experience:** An SDN infrastructure can better adapt to dynamic user needs, since it centralizes network control and it makes state information available to higher-level applications. For example, a carrier could introduce a video service that will offer to premium subscribers the maximum possible resolution in an automated and transparent manner. Nowadays, users must explicitly choose a resolution setting, which the network may or may not be able to support, creating delays and interruptions that degrade the user experience. Using OpenFlow-based SDN, the video application would be able to detect the bandwidth, which is available in the network, in real time and automatically adjust the video resolution according to user needs.

As a conclusion, trends such as user mobility, server virtualization, IT-as-a-Service, and the need speedily to respond to changing business conditions place significant demands on the network requests that today's conventional network architectures can't handle. Software-Defined Networking offers a new, dynamic network architecture which transforms traditional network backbones into very functional and flexible service-delivery platforms.

By disengaging the network control and data planes, OpenFlow-based SDN architecture abstracts the basic infrastructure from the applications that use it, by permitting the network to become as programmable and manageable at scale as the computer infrastructure that it increasingly resembles. An SDN approach amplifies network virtualisation, enabling IT to manage servers, applications, storage, and networks using a common approach and tool set. SDN adoption can improve network manageability, scalability and flexibility whether it is applying in a carrier environment or enterprise data center and campus.

The Open Networking Foundation has cultivated a vibrant ecosystem around SDN that includes large and small infrastructure vendors, such as application developers, software companies, systems and semiconductor manufacturers, computer companies and many different kinds of end users. OpenFlow switching has already being embodied into a number of physical and virtual infrastructure designs, as well as SDN controller software. Network services and business applications have already been interfaced with SDN controllers, providing more improved integration and coordination between them. The future of networking tends to be depended more and more on software, which will accelerate the rythm of innovation for networks as it has already happened in the computing and storage domains.

SDN assures to transform today's static networks into flexible, programmable platforms using the intelligence to allocate resources dynamically, the scale to support huge data centers and the virtualization which needed to support dynamic, highly automated, and secure cloud environments.

Because of its many advantages and his amazing industry momentum, SDN is on the way to become the new norm for networks.

## 2.4. Mininet

### An Instant Virtual Network on your Laptop (or other PC)

Mininet is a very useful tool that creates a realistic virtual network, running real kernel, switch and application code, on a single machine (VM, cloud or native), in seconds, with a single command:

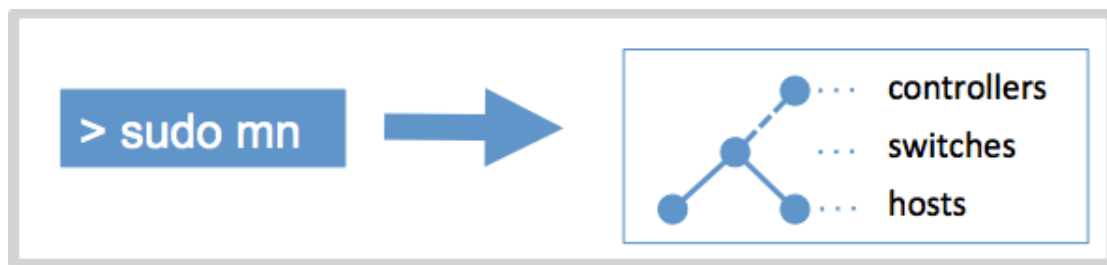


FIGURE 11: Mininet – sudo mn

Using Mininet you can easily interact with your network using the Mininet CLI (and API), customize it, share it with others, or deploy it on a real hardware. These are which make it useful for development, teaching, and research.

Mininet is also a great way to develop, share, and experiment with OpenFlow and Software-Defined Networking systems.

## 2.5. Software Defined Networking (SDN) Controllers

Open standards constitute the basic constituent of SDN architecture. Also, we have to mention that open source SDN controllers have evolved over the years. The most important are the following:

- **NOX**

There are a multitude of open source SDN controllers that have been available for some time. The first highly popular OpenFlow controller was NOX. As with most premature technology implementations, it was the initial spark, but unfortunately it was not heavily implemented or used because of shortages with its implementation and development environment. A lot of this

had to do with NOX being programmed mostly in C++ and because of the absence of good documentation.

- **POX**

POX is NOX's successor. It was built as a friendlier alternative and has been used and implemented by a remarkable number of SDN developers and engineers. Comparing with NOX, POX has an easier development environment to work with and a considerably better well written API and documentation. Moreover it provides a web based GUI and is written in Python, which typically makes shorter its experimental and developmental cycles.

- **Beacon**

Beacon was the next big step in open source controllers. It is a very well written and organized SDN controller written in Java and highly integrated into the Eclipse IDE. Beacon was the first controller which made it possible and easier for beginner programmers to work with and create a working SDN environment. However it was not flexible enough, since it was limited to star topologies (no loops).

- **Floodlight**

Then came the Floodlight, a fork off of Beacon that is managed by Big Switch Networks. Although its beginning was based on Beacon it was built using Apache Ant, a very popular software build tool, which makes the development of Floodlight easier and more agile. Floodlight has a very active community and a large number of features which can be added to create a system that spotlessly meets the requirements of a specific organization. Also, a web based and a Java based GUI are both available and most of Floodlight's functionality is exposed through a REST API.

- **OpenDaylight**

OpenDaylight constitutes a Linux Foundation collaborative project which has been highly supported by Cisco, Big Switch, and many other networking companies. As with Floodlight, OpenDaylight is written in Java and is a well-liked, well-supported SDN controller. Moreover, it includes exposure with a REST API and a Web based GUI. The second release of OpenDaylight (Helium), which is used in this thesis (for the reasons that will be mentioned below), includes support for SDN, NV (Network Virtualization) and NFV (Network Functions Virtualization) and is intended to be scaled to very large sizes. Furthermore, it has a number of pluggable modules (interfaces, protocols, and applications) which can be used to adapt it to the needs of an organization (like Floodlight does). OpenDaylight is a bit different from the other offerings because it permits for other non-OpenFlow southbound protocols.

- **OpenContrail**

OpenContrail is a Juniper project (Apache 2.0 licensed) that is aimed at providing only NV and NFV. It has an extensive REST API which can be used to configure and display information from the system. It is the same exact product that Juniper uses with its own commercial Contrail offering (with the addition of support). Although OpenContrail is a well written product, it does not have the following or support contrasted with OpenDaylight but it is targeted on a smaller overall picture.

- **Ryu**

Ryu in Japanese means “flow”. It is an open, software-defined networking (SDN) Controller designed to increase the flexibility of the network by making it easy to manage and adjust the way that traffic is handled. Generally, the SDN Controller is the “brain” of the SDN environment, communicating information “down” to the switches and routers using the southbound APIs, and “up” to the applications and business logic using the northbound APIs. Also, the Ryu Controller is supported by NTT and is deployed in NTT cloud data centers as well.

The Ryu Controller provides software components, using well-defined application program interfaces (APIs) makes it easy for developers to create new network management and control applications. This component approach helps organizations to customize deployments in order to meet their specific needs. Developers can rapidly and easily modify existing components or implement their own to ensure that the underlying network can meet the changing demands of their applications.

The Ryu Controller source code can easily be found on GitHub and managed and maintained by the open Ryu community. OpenStack, which is running an open collaboration focused on developing a cloud operating system that can control the compute, storage and networking resources of an organization, supports deployments of Ryu as the network Controller.

It is written entirely in Python and all of Ryu’s code is available under the Apache 2.0 license and open for anyone to use. The Ryu Controller supports NETCONF and OF-config network management protocols, and OpenFlow, which is the first and most widely deployed SDN communications standard.

Then we will discuss, compare and contrast the analysis of the controllers. But first it is important to mention the following points regarding the use cases, which highlight the challenges, from the point of view of comparative analysis:

1. The use cases covered here are the important and popular ones, and should be considered a complete list. Therefore, we may see some insignificant use cases here that some controllers support and some don’t.
2. The majority of the use cases (legacy network support, campus-networks, multi-layer network optimization, etc.) do not have a clear specification and scope-definition. It would be very difficult to say “YES” or “NO” if a controller supports it or not.

3. Alternatively, there are use cases (edge-overlays, OpenStack Neutron support, hop-by-hop network virtualization, etc.) that have clear specifications, and comparative analysis of those is relatively straightforward as a “YES” or “NO.”

4. There are use cases, such as load-balancing, monitoring, policy enforcement, etc that have a lot of overlapping in their functionalities. Some controllers are possible to support such use cases partially. Also the comparison in those cases becomes challenging.

Figure 12 (below) summarizes the comparison of open source controllers considering different use cases.

Use-Cases	Controllers					
	Trema	Nox/Pox	RYU	Floodlight	ODL	ONOS***
Network Virtualization by Virtual Overlays	YES	YES	YES	PARTIAL	YES	NO
Hop-by-hop Network Virtualization	NO	NO	NO	YES	YES	YES
OpenStack Neutron Support	NO	NO	YES	YES	YES	NO
Legacy Network Interoperability	NO	NO	NO	NO	YES	PARTIAL
Service Insertion and Chaining	NO	NO	PARTIAL	NO	YES	PARTIAL
Network Monitoring	PARTIAL	PARTIAL	YES	YES	YES	YES
Policy Enforcement	NO	NO	NO	PARTIAL	YES	PARTIAL
Load Balancing	NO	NO	NO	NO	YES	NO
Traffic Engineering	PARTIAL	PARTIAL	PARTIAL	PARTIAL	YES	PARTIAL
Dynamic Network Taps	NO	NO	YES	YES	YES	NO
Multi-Layer Network Optimization	NO	NO	NO	NO	PARTIAL	PARTIAL
Transport Networks - NV, Traffic-Rerouting, Interconnecting DCs, etc.	NO	NO	PARTIAL	NO	PARTIAL	PARTIAL
Campus Networks	PARTIAL	PARTIAL	PARTIAL	PARTIAL	PARTIAL	NO
Routing	YES	NO	YES	YES	YES	YES

Figure 12: Comparison of open source controllers

1. YES - Supports completely.

- The majority of the elements exists and can support with minor enhancements.
- There exists a commercial/proprietary solution(s) developed on top of the open source controller helping support the use case.
- Supports a remarkable number of use cases within an application network domain.

2. NO - Does not support.

- Only foundational elements exist and they need critical enhancements.
- No commercial/proprietary solution exists on top of the open source controller.



3. PARTIAL - Supports the use case partially.

- Considerable number of services/applications exists in the controller to realize the support for the use case.
- Supports partial set of use cases using an application network domain.
- The work is in progress.

From Figure 12 we can see that ODL's features make it more suitable to support the majority of use cases. This could be one of the reasons for its enormous popularity. Ryu and Floodlight have a similar number of use case supports, with Ryu having support for two of the most significant use cases: service insertion and chaining, and transport networks. Furthermore, NOX/POX and Trema have similar use case supports.

As a result of the above reasons and because of the better compatibility for load balancing and packet flow implementations the chosen controllers for this thesis is ODL and Floodlight.

### 3. OpenDaylight Helium Application Developers' Tutorial

#### Setup

To get started, download and set up the SDN Hub Tutorial VM in Virtualbox or VMware. We can update the tutorial code on the VM before we start:

```
$ cd SDNHub_Openaylight_Tutorial && git pull --rebase
```

#### Quickstart

- Boot up our tutorial VM with OpenDaylight already installed. The complete source code for the Hydrogen Release is located in /home/ubuntu/opendaylight, and the tutorial application that we will first use is located in /home/ubuntu/SDNHub\_OpenDaylight\_tutorial directory.
- After that, we have to create a simple (virtual) network topology using Mininet and Open vSwitch. In a Terminal window, run the following command, that starts a network emulation environment to emulate 1 switch, which attempts to connect to a remote controller, with 3 hosts.

```
$ sudo mn --topo single,3 --mac --switch ovsk,protocols=OpenFlow13 --controller remote
```

- We will now see a mininet terminal. Begin to ping between two of the hosts. The ping will fail because there is not enough information in the switch to learn the MAC addresses of each host and forward traffic to the correct switch ports.

```
mininet> h1 ping h2
```

```
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
```

```
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
```

```
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
```

```
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
```

- We have to add that intelligence. Open a Terminal tab (Ctrl-Shift-T) and go into the SDNHub\_Openaylight\_Tutorial folder and run the following commands to build the example application that we will walk through for the tutorial (a Hub / L2 learning switch):

```
ubuntu@ubuntu:~$ cd SDNHub_Openaylight_Tutorial
```

```
ubuntu@ubuntu:~/SDNHub_Openaylight_Tutorial$ mvn install -nsu
```

... lots of text ...

```
[INFO] -----  
[INFO] Reactor Summary:  
[INFO]  
[INFO] SDN Hub OpenDaylight tutorial POM ..... SUCCESS [ 0.495 s]  
[INFO] L2 forwarding tutorial with AD-SAL OpenFlow plugins SUCCESS [ 11.837 s]  
[INFO] L2 forwarding tutorial with MD-SAL OpenFlow plugins SUCCESS [ 4.118 s]  
[INFO] OpenDaylight OSGi AD-SAL tutorial Distribution .... SUCCESS [ 32.992 s]  
[INFO] OpenDaylight OSGi MD-SAL tutorial Distribution .... SUCCESS [ 43.150 s]  
[INFO] tutorial ..... SUCCESS [ 0.184 s]  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 01:33 min  
[INFO] Finished at: 2014-09-30T09:48:27-08:00  
[INFO] Final Memory: 60M/371M  
[INFO] -----
```

This indicates that the build completed successfully.

- This command uses Apache Maven to build the tutorial code. “mvn” is the maven command that compiles code based on the pom.xml file in that directory. “clean” command is not essential if we do not want to clean the temporary build files. The “install” command is essential for compilation.
- The way Maven operates is by resolving dependencies between packages. The tutorial\_L2\_forwarding package (hub/switch code which we will edit) depends on the Opendaylight controller package, for this reason Maven acts by downloading the pre-compiled jar files from Opendaylight.org, resolving dependencies, and so on.
  - Maven has a module name and Java code has a package name, which they do not need to match. The module adsal\_L2\_forwarding has the following features. The Maven group-id and artifact-id are used for the name of the Jar file generated, while package name is only recognized into the source code, which contains:
    - pom.xml group-id: org.sdnhub.odl.tutorial

- pom.xml artifact-id: adsal\_L2\_forwarding
  - Source code package name: org.opendaylight.tutorial.tutorial\_L2\_forwarding
  - Creating our own installation by this way is much more faster than compiling the tutorial code within the full source of the OpenDaylight controller. This can be compared with compiling an application instead of compiling the full operating system.
  - **Note:** In order to speed up subsequent compilations, you can run “mvn install -DskipTests -DskipIT -nsu”. The “nsu” command is a abbreviation for no-snapshot-updates. It ensures that the compilation will not download definitions from nexus.opendaylight.org
- Next, we can run the controller itself (preferably using another terminal).

```
ubuntu@ubuntu:~/SDNHub_OpenDaylight_Tutorial$cd distribution/opendaylight- osgi
adsal/target/distribution-osgi-adsal-1.1.0-SNAPSHOT-osgipackage/opendaylight
ubuntu@ubuntu:~/SDNHub_OpenDaylight_Tutorial/distribution/opendaylight- osgi-
adsal/target/distribution-osgi-adsal-1.1.0-SNAPSHOT-osgipackage/opendaylight$ ./run.sh
```

- The above command starts all the OSGi bundles installed as jar files into the plugins directory. When all bundles are started, we will be able to check theirs status using the “ss” command.

```
osgi> ss tutorial
```

```
"Framework is launched."
```

```
id State Bundle
```

```
36 ACTIVE org.sdnhub.odl.tutorial.adsal_L2_forwarding-0.5.0-SNAPSHOT
```

In the above we can see the OSGi output. The word “ACTIVE” means that the module/bundle is running, while “RESOLVED” if it appears mean that it has been stopped, either explicitly by other modules or implicitly through a missing dependent module.

- Since OpenDaylight today does not have a way to specify which modules get precedence over other modules, we require that any conflict modules (such as SimpleForwarding and ARPHandler) have to be stopped using the “stop” OSGi command. As a result, all packet forwarding decisions will be left to the tutorial applications.
    - **Note:** In order to make it easier, the adsal\_L2\_forwarding package has some lines of code in its init(), which used to automatically “uninstall” these conflicting modules.
- When the controller loads completely, we should be able to see the list of switches connected using the command “printnodes”. In this environment, we have a topology with 1 OpenFlow switch connected with Datapath-id “00:00:00:00:00:00:01”.

```
osgi> printnodes
```

```
Nodes connected to this controller :
```

```
[OF]00:00:00:00:00:00:01]
```

- Now when we go back to the mininet terminal, we can confirm that the pings succeed.

```
From 10.0.0.1 icmp_seq=17 Destination Host Unreachable
```

```
From 10.0.0.1 icmp_seq=18 Destination Host Unreachable
64 bytes from 10.0.0.2: icmp_req=19 ttl=64 time=94.1 ms
64 bytes from 10.0.0.2: icmp_req=20 ttl=64 time=21.2 ms
64 bytes from 10.0.0.2: icmp_req=21 ttl=64 time=21.9 ms
```

- We will have noticed that the pings are actually taking longer than they should for a learning switch (sub-1ms) for such a simple topology. This is because the controller is initially functioning in “Hub” mode.
- Let’s do a few actions to start appreciating OSGi. As the controller is running, open a new terminal, and edit TutorialL2Forwarding.java in order to change the function variable from “hub” to “switch”. This must be done as follows:

```
ubuntu@ubuntu:~/SDNHub_Opendaylight_Tutorial$ sed -i -e "s/function = \"hub\"/function = \"switch\"/g" `find . -name TutorialL2Forwarding.java`
```

```
ubuntu@ubuntu:~/SDNHub_Opendaylight_Tutorial$ cd adsal_L2_forwarding
```

```
ubuntu@ubuntu:~/SDNHub_Opendaylight_Tutorial/adsal_L2_forwarding$ mvn install -DskipTests -DskipIT -nsu
```

- This will recompile the change we made before and generate new Jar files that will be placed in the correct location so that OSGi will be able to pick up the changes at run-time. On the OSGi console, we will be able to see the automatic reloading of the bundle:

```
MDT [fileinstall-./plugins] INFO o.o.t.t.i.TutorialL2Forwarding - Stopped
MDT [fileinstall-./plugins] INFO o.o.t.t.i.TutorialL2Forwarding - Initialized
MDT [fileinstall-./plugins] INFO o.o.t.t.i.TutorialL2Forwarding - Started
```

- If everything went right, we will also see the ping RTT times drop to sub-millisecond. Later we will describe how we can turn the hub into a switch.

### 3.1. Introduction to OpenDaylight

Before we start to walk through into the code, a high-level overview of the OpenDaylight controller is in order. OpenDaylight is a modular platform with most modules reusing some common services and interfaces. Every module is developed under a multi-vendor sub-project.

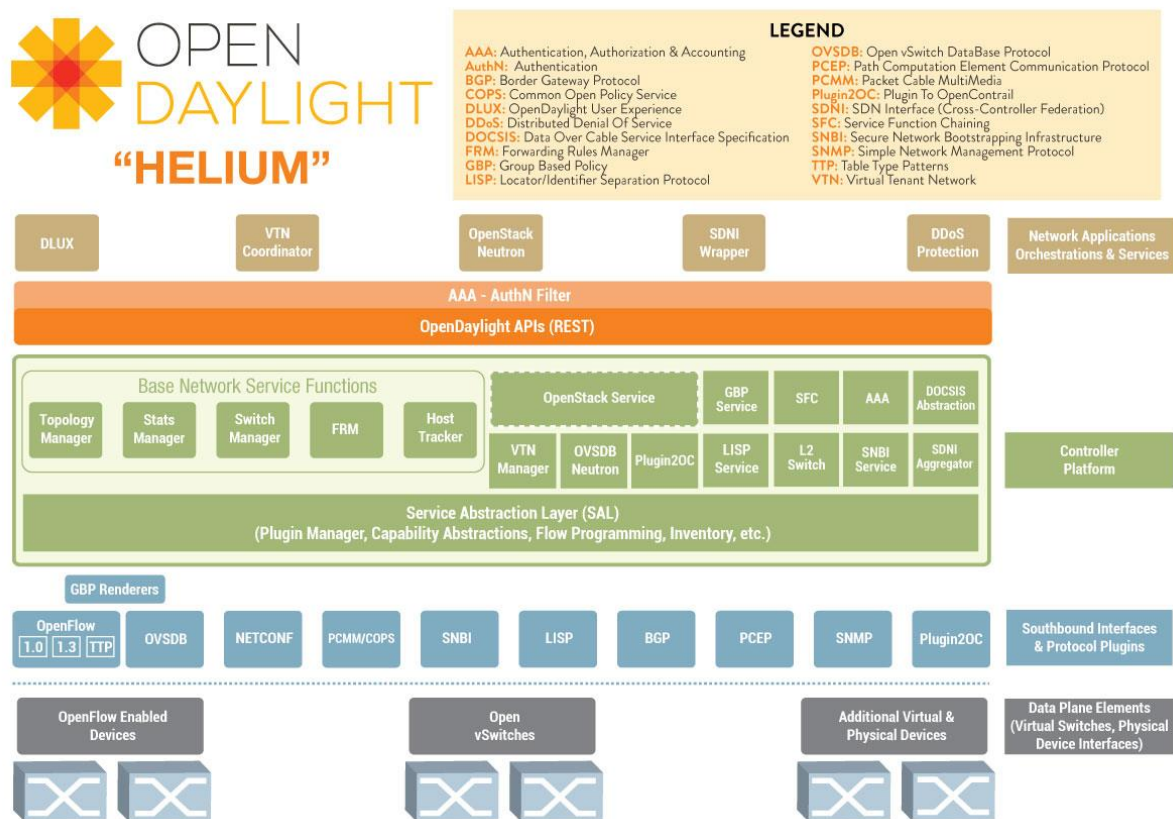


FIGURE 13: OpenDaylight architecture

OpenDayLight uses the following software tools/paradigms. The Service Abstraction Layer (SAL) is our “ally” for most development aspects.

- **Java interfaces:** Java Interfaces are used for event listening, specifications and forming patterns. This is the main way with whom specific bundles implement call-back functions for events and also to indicate awareness of specific state.
- **Maven:** OpenDayLight uses Maven to make build automation easier. Maven uses pom.xml (Project Object Model for this bundle) to script the dependencies between bundles and also to describe what bundles must load on start.
- **OSGi:** This framework in the backend of OpenDayLight permits dynamically loading bundles and packaged Jar files, and binding bundles together for information exchange.
- **Karaf:** Karaf is a small OSGi based runtime which supplies a lightweight container for loading different modules.

The controller project offers certain core bundles, each of which export important services through Java interfaces. Here is a short list of the most important bundles that come in handy when developing network services.

Bundle	Exported interface	Description
arphandler	IHostFinder	Component responsible for learning about host location by handling ARP.
hosttracker	IfIptHost	Track the location of the host relatively to the SDN network.
switchmanager	ISwitchManager	Component holding the inventory information for all the known nodes (i.e., switches) in the controller.
topologymanager	ITopologyManager	Component holding the whole network graph.
usermanager	IUserManager	Component taking care of user management.
statisticsmanager	IStatisticsManager	Component in charge of using the SAL ReadService to collect several statistics from the SDN network.
sal	IReadService	Interface for retrieving the network node's flow/port/queue hardware view
sal	ITopologyService	Topology methods provided by SAL toward the applications
sal	IFlowProgrammerService	Interface for installing/modifying/removing flows on a network node
sal	IDataPacketService	Data Packet Services SAL provides to the applications
web	IDaylightWeb	Component tracking the several pieces of the UI depending on bundles installed on the system.

FIGURE 14: OpenDaylight - Core bundles

### 3.2. Basic steps to writing an OpenDayLight Application

For beginners, it is recommend using Eclipse for viewing and editing the source code, and command-line for compiling and running the controller. Open Eclipse using the desktop shortcut.

The Eclipse environment is already setup, but if you wish to do it from scratch, to setup Maven-Eclipse integration.

After that you can and go to File->Import. Choose Maven > Existing Maven Projects. Continue on the next screen and choose as the Root Directory ~/ubuntu/SDNHub\_Opendaylight\_Tutorial. Select 'adsal\_L2\_forwarding' and 'mdsal\_L2\_forwarding' pom.xml that it finds, click on 'Add project(s) to working set' at the bottom of the window, and click 'Finish'.

When Eclipse is setup, we should see 'adsal\_L2\_forwarding' and 'mdsal\_L2\_forwarding' on our Project explorer pane. Select on one of the two, double-click on it and do the following: src/main/java > org.opendaylight.tutorial.tutorial\_L2\_forwarding. Then we can walk-through through the code.

Most controller platforms expose some native features which are allowing these key features:

- A. Ability to listen to asynchronous events (e.g., PACKET\_IN, FLOW\_REMOVED) and to register call back functions (e.g., receiveDataPacket)
- B. Ability to parse incoming packets (e.g., ARP, ICMP, TCP) and fabricate packets to send out into the network
- C. Ability to create and send an OpenFlow/SDN message (e.g., PACKET\_OUT, FLOW\_MOD, STATS\_REQUEST) to the programmable dataplane.
- D. Managing decision making modules on the same controller

With OpenDayLight we can achieve all of the above using a combination of SAL implementation classes and listener interfaces.

It is very important to mention that there are two types of SAL:

1. API-driven SAL (AD-SAL), where the API is statically defined at compile/build time, and the “provider” and “consumer” of events/data are directly plumbed to one another in code.
2. Model-driven (MD-SAL), where the API code is generated from models during compile time and loaded into the controller when the plugin OSGI bundle is loaded into the controller. All event calls and data go from “provider” to a “consumer” through a central datastore with MD-SAL.

The app development is based on which type of SAL you’re building it with, which it makes it different. Each type of SAL has different tradeoffs like ease of use, scalability, debug ability, and feature set. Once you get the hang of both, you can choose which is best for you, but you can use both of them in an application you build. In this thesis, we will take a look at both types.

### 3.3. Writing an AD-SAL OpenDayLight Application

#### Theory

##### 3.3.1. Receiving events

There are listener interfaces for several types of events including events that are thrown by modules like TopologyManager. As for example, in order to allow a module to receive packets sent by the switch to the controller, the class needs to implement `IListenDataPacket`:

```
public class TutorialL2Forwarding implements IListenDataPacket
```

When the class implements this `IListenDataPacket` interface, any packet that arrives at a node without a pre-cached rule will result in a callback of `receiveDataPacket`. So overriding using this method is allowed to the application to get a copy of the packet.

```
public PacketResult receiveDataPacket(RawPacket inPkt) { ... }
```

One more feature that was useful was generating the list of ports in a node/switch. Assuming that the `incoming_node` is the switch whose ports I want to list, we can use the `SwitchManager` class as follows:

```
Set<NodeConnector> nodeConnectors =  
  
this.switchManager.getUpNodeConnectors(incoming_node);
```



### 3.3.2. Parsing packets

After the packet is received, we are able to decode the packet using the *dataPacketService*. Usage of command is shown below. Today, according to the *org.opendaylight.controller.sal.packet* package, we can inspect the packet headers and the following packet types: ARP, Ethernet, ICMP, IPv4, LLDP, TCP, UDP.

```
Packet formattedPak = this.dataPacketService.decodeDataPacket(inPkt);
```

Along with the packet header fields, you can extract the incoming node (switch) and node-connector (switchport) using the following:

```
NodeConnector incoming_connector = inPkt.getIncomingNodeConnector();
```

We use the following two useful commands to extract header details and use them in the MAC learning switch:

```
byte[] srcMAC = ((Ethernet)formattedPak).getSourceMACAddress();
```

```
long srcMAC_val = BitBufferHelper.toNumber(srcMAC);
```

### 3.3.3. Sending messages to switch

After the decisions are taken and the controller decides that a packet needs to be forwarded, we can use the *dataPacketService* to send a packet out of a node-connector (switchport). In the following example, we are attempting to resend the incoming packet out of node-connector 'p':

```
RawPacket destPkt = new RawPacket(inPkt);
```

```
destPkt.setOutgoingNodeConnector(p);
```

```
this.dataPacketService.transmitDataPacket(destPkt);
```

Except of the `PACKET_OUT`, we can also perform a `FLOW_MOD` rule, which can be inserted into a switch. For that, we use the following classes: `Match`, `Action` and `Flow`. Here is given an example of how to create a match rule where the `IN_PORT` matches the value extracted from the `PACKET_IN`:

```
Match match = new Match();
```

```
match.setField(MatchType.IN_PORT, incoming_connector);
```

There are many other fields you can match on. The followings are the existing fields that we see in the *org.opendaylight.controller.sal.match*: “inPort”, “dlSrc”, “dlDst”, “dlVlan”, “dlVlanPriority”, “dlOuterVlan”, “dlOuterVlanPriority”, “dlType”, “nwTOS”, “nwProto”, “nwSrc”, “nwDst”, “tpSrc”, “tpDst”.

In the package *org.opendaylight.controller.sal.action*, we can see the following supported actions: “drop”, “loopback”, “flood”, “floodAll”, “controller”, “interface”, “software path”, “hardware path”, “output”, “enqueue”, “setDlSrc”, “setDlDst”, “setVlan”, “setVlanPcp”, “setVlanCif”, “stripVlan”, “pushVlan”, “setDlType”, “setNwSrc”, “setNwDst”, “setNwTos”, “setTpSrc”, “setTpDst”, “setNextHop”. Here is an example which creates an action list for the flow rule, for OUTPUT to *anoutgoing\_connector* that is pre-calculated.

```
List<Action> actions = new ArrayList<Action>();
```

```
actions.add(new Output(outgoing_connector));
```

After the match rule is formed and action list is created, we can continue by creating a Flow and add it to a particular switch. In this case, the rule is inserted into the switch where the original packet arrived:

```
Flow f = new Flow(match, actions);
```

```
Status status = programmer.addFlow(incoming_node, f);
```

```
if (status.isSuccess())
```

```
    logger.trace("Installed flow {} in node {}", f, incoming_node);
```

```
else
```

```
    //What to do in case of flow insertion failure?
```

### 3.3.4. Managing multiple modules

As mentioned previously, it is possible that there are multiple modules contesting for the same events and want to act upon it. At the moment OpenDaylight does not provide a precedence order among modules. This happens because of the requirement of some coordination among modules to ensure there is no conflict.

Because of that, it is possible for a module to signal to the platform after it has finished processing a packet. When the packet is appropriately handled, the controller application has three choices on returning status back to the SAL, which are the following:

- return PacketResult.CONSUME: The packet has been processed and none in the chain after us should handle it.
- return PacketResult.KEEP\_PROCESSING: The packet has been processed, but still pass onto other bundles.
- return PacketResult.IGNORED: The packet has been ignored and gets ready to pass to other bundles

Since we are updated on the theory, we are now ready to create a new module.

### 3.4. Sample AD-SAL application: Learning Switch

For the purposes of this thesis, we should attempt to convert the hub application to a MAC learning switch by writing code from scratch, using the above code snippets.

A skeleton file is provided to write our own code. Overwrite the TutorialL2Forwarding.skeleton file over the current TutorialL2Forwarding as follows:

```
ubuntu@ubuntu:~/SDNHub_Openaylight_Tutorial$ cd
adsal_L2_forwarding/src/main/java/org/.opendaylight/tutorial/tutorial_L2_forwarding/internal && cp
TutorialL2Forwarding.skeleton TutorialL2Forwarding.java
```

Modify the implementation of the application by editing this new TutorialL2Forwarding.java. We will find provided hints through Java comments.

Once we make the necessary changes, successfully compile and run, we will be able to see that mininet ping succeeds (NRTT time is less than 1ms once flow rules are inserted, unlike the hub case):

```
mininet> h1 ping h2
```

```
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
```

```
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
```

```
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
```

```
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
```

```
From 10.0.0.1 icmp_seq=4 Destination Host Unreachable
```

```
64 bytes from 10.0.0.2: icmp_req=5 ttl=64 time=0.529 ms
```

```
64 bytes from 10.0.0.2: icmp_req=6 ttl=64 time=0.133 ms
```

```
64 bytes from 10.0.0.2: icmp_req=7 ttl=64 time=0.047 ms
```

In the OSGi console we can see the following sample log:

```
2013-06-13 21:00:31.844 PDT [SwitchEvent Thread] INFO o.o.t.t.i.TutorialL2Forwarding
```

```
- Installed flow Flow[match = Match[[DL_DST(00:00:00:00:00:01,null),
```

```
IN_PORT(OF|2@OF|00:00:00:00:00:00:00:01,null)],
```

```
actions = [OUTPUT[OF|1@OF|00:00:00:00:00:00:00:01]], priority = 0, id = 0,
```

```
idleTimeout = 0, hardTimeout = 0] in node OF|00:00:00:00:00:00:00:01
```

### **Solutions**

In the tutorial\_L2\_forwarding/internal directory, we will see three solutions files:

- *TutorialL2Forwarding\_singleswitch.solution*: L2 MAC is a learning switch that pushes flow\_mod rules to the switch based on the packet\_in. It only tracks MAC for a single switch.
- *TutorialL2Forwarding\_multiswitch.solution*: L2 MAC is a learning switch that pushes flow\_mod rules to the switch based on the packet\_in. It tracks MAC for multiple switches independently.
- *TutorialL2Forwarding\_statelesslb.solution*: Stateless load-balancer code where we can statically set the list of servers and their IP addresses. After traffic arrives for the virtual IP of the load-balancer, it selects the server in a round-robin basis and sends request to a different server after implementing IP header rewriting in both directions.

## **3.5. Workflow, dependencies and information sharing**

### **3.5.1. Maven and OSGI**

- **pom.xml**: It is used more often to indicate the inter-bundle dependencies. The syntax for this must be enough apparent from the format of the pom.xml in the arphandler directory.
- **Activator.java**: It handles the configuration of the dependencies during run-time. The function of configureInstance in the bottom of Activator.java of each bundle has two main tasks:
  - Displays the list of Java interfaces implemented by a bundle 'X' using *setInterface* call
  - Registers other bundles that the bundle 'X' uses using *add* call
- **set/unset binding**: The bundle 'X' receives a reference to the instantiated object of other needed bundle 'Y' through the *setY* and *unsetY* calls in the main class. For instance, TutorialL2Forwarding needs data from SwitchManager. This is accomplished by defining two methods *setSwitchManager* and *unsetSwitchManager*. OSGi calls these two methods and object references are appropriately passed.

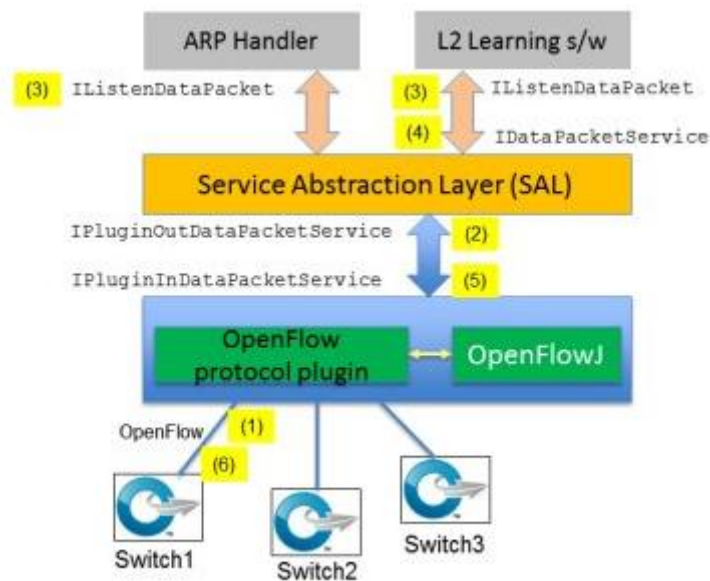


FIGURE 15: Life of a packet – OpenFlow protocol

### 3.5.2. Life of a packet

In OpenDaylight, the SAL is in charge of all plumbing between the applications and the underlying plugins. Here is a detailed discription of the life of a packet.

1. A packet that arrives at Switch1 will be sent to the appropriate plugin managing the switch.
2. The plugin will parse the packet and generate an event for SAL.
3. SAL will dispatch the packet to the modules listening for DataPacket
4. Module handles packet and sends packet\_out through IDataPacketService
5. SAL dispatches the packet to the modules listening for DataPacket
6. OpenFlow message sent to the appropriate switch

### 3.6. AD-SAL OpenDaylight Web UI

One of the benefits of using OpenDaylight as our controller platform is that it offers hooks for providing a Web UI with many pre-existing features. The GUI can also be extended for user-defined applications. To access the GUI, open a browser and enter at <http://localhost:8080>. Now we should be able to see the login page.

- Login with username and password, which is the same for both: “admin”

We can see the different tabs on top which offer different features. Each feature has a “web” module (to render the web page), a “northbound” module (to provide web API for the web module to call), and a backend module which does all the relevant functionality in the state management. For example, here are the bundles loaded in order to rend the topology in the UI:

```
osgi> ss topology
```

```
"Framework is launched."
```

id	State	Bundle
81	ACTIVE	org.opendaylight.controller.topology.northbound_0.4.2.SNAPSHOT
98	ACTIVE	org.opendaylight.controller.topology.web_0.4.2.SNAPSHOT
181	ACTIVE	org.opendaylight.controller.topologymanager_0.4.2.SNAPSHOT

The TopologyManager module tracks the topology of the underlying network, while the “web” and “northbound” modules are responsible for the UI portion.

### 3.7. Writing an MD-SAL OpenDayLight Application

In the environment of the VM, is included the sample learning-switch tutorial application which uses the MD-SAL OpenFlow 1.3 plugin. Here is how we can start the MD-SAL application, after we stop the AD-SAL controller application. We use the maven option “pl” to limit building only the MD-SAL relevant packages:

```
ubuntu@ubuntu:~/SDNHub_Openaylight_Tutorial$ mvn install -pl distribution/opendaylight-osgi-  
mdsal --also-make -DskipTests -DskipIT -nsu
```

```
....
```

```
[INFO] -----
```

```
[INFO] Reactor Summary:
```

```
[INFO]
```

```
[INFO] SDN Hub OpenDaylight tutorial POM ..... SUCCESS [ 0.157 s]
```

```
[INFO] L2 forwarding tutorial with MD-SAL OpenFlow plugins SUCCESS [ 3.965 s]
```

```
[INFO] OpenDaylight OSGi MD-SAL tutorial Distribution .... SUCCESS [ 32.674 s]
```

```
[INFO] -----
```

[INFO] BUILD SUCCESS

[INFO] -----

```
ubuntu@ubuntu:~/SDNHub_Openaylight_Tutorial$ cd distribution/opendaylight-  
mdsal/target/distribution-osgi-mdsal-1.1.0-SNAPSHOT-osgipackage/opendaylight
```

```
ubuntu@ubuntu:~/SDNHub_Openaylight_Tutorial/distribution/opendaylight-  
mdsal/target/distribution-osgi-mdsal-1.1.0-SNAPSHOT-osgipackage/opendaylight$ ./run.sh
```

When the controller is started, we can use RestConf API to access all the details in the data store. The information is routed over HTTP and is based on the Yang models for that particular information. For example, we can query the list of nodes and the details stored for them in the data store by typing the following link in a browser: <http://localhost:8080/restconf/operational/opendaylight-inventory:nodes/>. Then we will be able to see data as follows.

```
▼<nodes xmlns="urn:opendaylight:inventory">  
  ▼<node>  
    <id>controller-config</id>  
  </node>  
  ▼<node>  
    <id>openFlow:1</id>  
    ▼<table xmlns="urn:opendaylight:flow:inventory">  
      <id>0</id>  
      ▼<flow>  
        <id>561183150</id>  
        ▼<match>  
          ▼<ethernet-match>  
            ▼<ethernet-destination>  
              <address>00:00:00:00:00:02</address>  
            </ethernet-destination>  
            ▼<ethernet-source>  
              <address>00:00:00:00:00:01</address>  
            </ethernet-source>  
          </ethernet-match>  
        </match>  
        <flags/>  
        <flow-name>mac2mac</flow-name>  
        <priority>512</priority>  
        ▼<instructions>  
          ▼<instruction>  
            <order>0</order>  
            ▼<apply-actions>  
              ▼<action>  
                <order>0</order>  
                ▼<output-action>  
                  <output-node-connector>openFlow:1:1</output-node-conn<  
                    <max-length>65535</max-length>  
                  </output-action>  
                </action>  
              </apply-actions>  
            </instruction>  
          </instructions>  
          <idle-timeout>0</idle-timeout>  
          <hard-timeout>0</hard-timeout>  
          <table_id>0</table_id>  
          <cookie>3026418949592973313</cookie>  
          <buffer_id>0</buffer_id>  
        </flow>  
      </table>  
    </node>  
  </nodes>  
  ▼<flow>  
    <id>0</id>  
    <match/>  
    <flags/>  
    <flow-name>allPacketsToCtrl</flow-name>  
    <priority>0</priority>  
    ▼<instructions>  
      ▼<instruction>  
        <order>0</order>  
        ▼<apply-actions>  
          ▼<action>  
            <order>0</order>  
            ▼<output-action>  
              <output-node-connector>CONTROLLER</output<  
                <max-length>65535</max-length>  
              </output-action>  
            </action>  
          </apply-actions>  
        </instruction>  
      </instructions>  
      <idle-timeout>0</idle-timeout>  
      <hard-timeout>0</hard-timeout>  
      <table_id>0</table_id>  
      <buffer_id>0</buffer_id>  
    </flow>  
  ▼<flow>  
    <id>146983713</id>  
    ▼<match>  
      ▼<ethernet-match>  
        ▼<ethernet-destination>  
          <address>00:00:00:00:00:01</address>  
        </ethernet-destination>  
        ▼<ethernet-source>  
          <address>00:00:00:00:00:02</address>  
        </ethernet-source>  
      </ethernet-match>  
    </match>  
    <flags/>  
    <flow-name>mac2mac</flow-name>  
    <priority>512</priority>  
    ▼<instructions>  
      ▼<instruction>  
        <order>0</order>  
        ▼<apply-actions>  
          ▼<action>  
            <order>0</order>  
            ▼<output-action>  
              <output-node-connector>openFlow:1:2</output-node-connector<  
                <max-length>65535</max-length>  
              </output-action>  
            </action>  
          </apply-actions>  
        </instruction>  
      </instructions>  
      <idle-timeout>0</idle-timeout>  
      <hard-timeout>0</hard-timeout>  
      <table_id>0</table_id>  
      <cookie>3026418949592973312</cookie>  
      <buffer_id>0</buffer_id>  
    </flow>  
  </table>  
</node>  
</nodes>
```

FIGURE 16: RestConf

In order to get list of all APIs exported over RestConf, we can use the API explorer and seeing the list of API by typing the following link in a browser: <http://localhost:8080/apidoc/explorer/>. We can use this data access and possibly create new flows through this REST interface. Once we create a new flow, we can see it in the Open vSwitch table using the command:

```
$ sudo ovs-ofctl dump-flows s1 -O OpenFlow13
```

OFPOST\_FLOW reply (OF1.3) (xid=0x2):

```
cookie=0x0, duration=35.255s, table=0, n_packets=22, n_bytes=1668, priority=0  
actions=CONTROLLER:65535
```

```
cookie=0x2a00000000000000, duration=35.248s, table=0, n_packets=36, n_bytes=3416,  
priority=512,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01 actions=output:2
```

```
cookie=0x2a00000000000001, duration=35.247s, table=0, n_packets=35, n_bytes=3374,  
priority=512,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 actions=output:1
```

## OpenDaylight RestConf API Documentation

[Controller Resources](#) | [Mounted Resources](#)

Below are the list of APIs supported by the Controller.

<a href="#">config(2013-04-05)</a>	<a href="#">Show/Hide</a>	<a href="#">List Operations</a>	<a href="#">Expand Operations</a>	<a href="#">Raw</a>
<a href="#">config-logging(2013-07-16)</a>	<a href="#">Show/Hide</a>	<a href="#">List Operations</a>	<a href="#">Expand Operations</a>	<a href="#">Raw</a>
<a href="#">flow-capable-transaction(2013-11-03)</a>	<a href="#">Show/Hide</a>	<a href="#">List Operations</a>	<a href="#">Expand Operations</a>	<a href="#">Raw</a>
<a href="#">flow-topology-discovery(2013-08-19)</a>	<a href="#">Show/Hide</a>	<a href="#">List Operations</a>	<a href="#">Expand Operations</a>	<a href="#">Raw</a>
<a href="#">ietf-netconf-monitoring(2010-10-04)</a>	<a href="#">Show/Hide</a>	<a href="#">List Operations</a>	<a href="#">Expand Operations</a>	<a href="#">Raw</a>
<a href="#">kitchen-service-impl(2014-01-31)</a>	<a href="#">Show/Hide</a>	<a href="#">List Operations</a>	<a href="#">Expand Operations</a>	<a href="#">Raw</a>
<a href="#">network-topology(2013-07-12)</a>	<a href="#">Show/Hide</a>	<a href="#">List Operations</a>	<a href="#">Expand Operations</a>	<a href="#">Raw</a>
<a href="#">network-topology(2013-10-21)</a>	<a href="#">Show/Hide</a>	<a href="#">List Operations</a>	<a href="#">Expand Operations</a>	<a href="#">Raw</a>
<a href="#">opendaylight-action-types(2013-11-12)</a>	<a href="#">Show/Hide</a>	<a href="#">List Operations</a>	<a href="#">Expand Operations</a>	<a href="#">Raw</a>
<a href="#">opendaylight-flow-statistics(2013-08-19)</a>	<a href="#">Show/Hide</a>	<a href="#">List Operations</a>	<a href="#">Expand Operations</a>	<a href="#">Raw</a>
<a href="#">opendaylight-flow-table-statistics(2013-12-15)</a>	<a href="#">Show/Hide</a>	<a href="#">List Operations</a>	<a href="#">Expand Operations</a>	<a href="#">Raw</a>
<a href="#">opendaylight-group-statistics(2013-11-11)</a>	<a href="#">Show/Hide</a>	<a href="#">List Operations</a>	<a href="#">Expand Operations</a>	<a href="#">Raw</a>
<a href="#">opendaylight-inventory(2013-08-19)</a>	<a href="#">Show/Hide</a>	<a href="#">List Operations</a>	<a href="#">Expand Operations</a>	<a href="#">Raw</a>
<a href="#">POST</a> /config/				
<a href="#">GET</a> /config/opendaylight-inventory:nodes/				
<a href="#">PUT</a> /config/opendaylight-inventory:nodes/				
<a href="#">DELETE</a> /config/opendaylight-inventory:nodes/				
<a href="#">POST</a> /config/opendaylight-inventory:nodes/				
<a href="#">GET</a> /config/opendaylight-inventory:nodes/node/{id}/				
<a href="#">PUT</a> /config/opendaylight-inventory:nodes/node/{id}/				

FIGURE 17: Opendaylight RestConf API Documentation



### 3.8. Karaf

Based on all future releases that OpenDaylight will be made available as a Karaf-based distribution, is recommended to understand how to use Karaf. As mentioned above it is an OSGi-based runtime. This serves all the benefits of OSGi, such as hot deployment of modules, and the Karaf-specific features such as the ability to prescribe a streamlined set of bundles to load for an application.

opendaylight-tutorial-karaf-adsal

In the SDNHub\_OpenDaylight\_tutorial project, we can also see the Karaf configurations which allow us to experiment with Karaf. Here are the two main configurations that we had to make:

**feature files:** Two folders are existed: 'features-tutorial/adsal' and 'features-tutorial/mdsal' that defines two features 'sdnhub-tutorial-adsal' and 'sdnhub-tutorial-mdsal' in the respective pom.xml, as well as describes the exact version and set of bundles that are required by each feature in 'src/main/resources/features.xml' file.

**distribution POM:** The directory SDNHub\_OpenDaylight\_tutorial/opendaylight/distribution-karaf is specially crafted to generate the Karaf-based distribution that includes both sample applications.

For example, the Figure 18 shows how the Karaf dependency chart looks like for the feature 'sdnhub-tutorial-adsal'. The green boxes represent bundles and the blue boxes represent Karaf features.

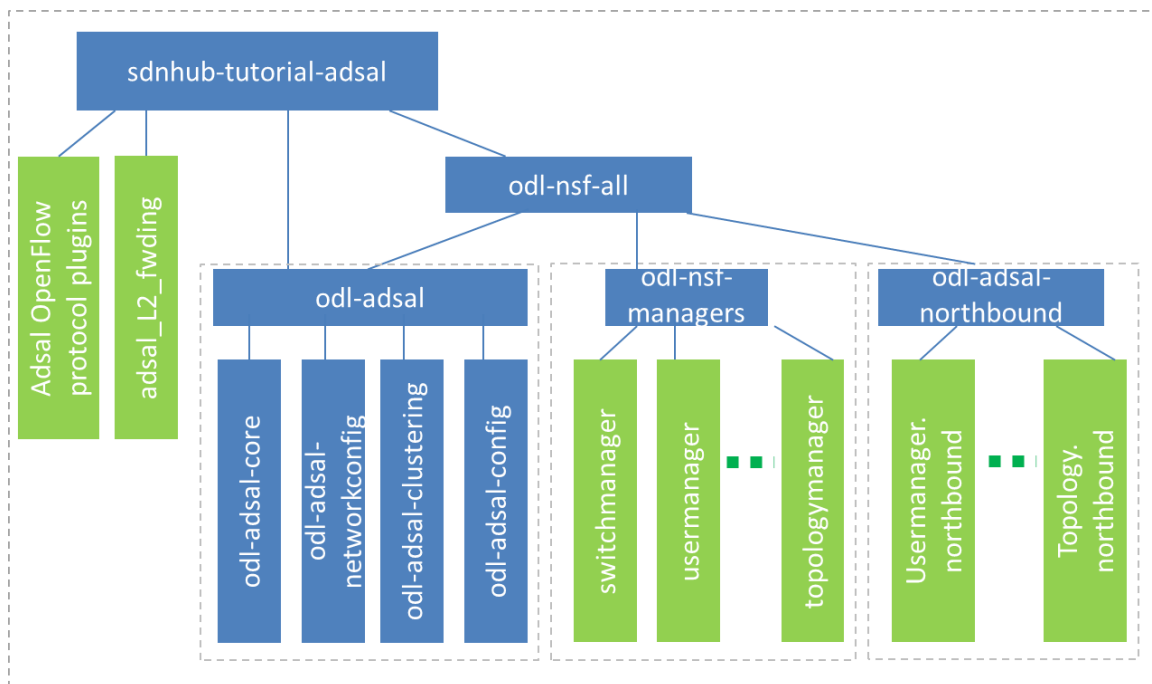


FIGURE 18: Karaf - sdnhub-tutorial-adsal – Bundles and features

After compiling all features and the Karaf distribution, we can run Karaf as follows:



SLF4J: Found binding in  
[bundleresource://126.fwk126488680:1/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: Found binding in  
[bundleresource://126.fwk126488680:2/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: Found binding in  
[bundleresource://126.fwk126488680:3/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: See [http://www.slf4j.org/codes.html#multiple\\_bindings](http://www.slf4j.org/codes.html#multiple_bindings) for an explanation.

SLF4J: Actual binding is of type [org.slf4j.impl.JDK14LoggerFactory]

```
opendaylight-user@root>bundle:list -s | grep sdnhub
```

```
230 | Active | 80 | 0.5.0.SNAPSHOT | org.sdnhub.odl.tutorial.adsal_L2_forwarding
```

```
231 | Active | 80 | 0.1.0.SNAPSHOT | org.sdnhub.odl.ofbroker
```

```
232 | Active | 80 | 0.1.0.SNAPSHOT | org.sdnhub.odl.protocol_plugins.openflow10
```

```
233 | Active | 80 | 0.1.0.SNAPSHOT | org.sdnhub.odl.protocol_plugins.openflow13
```

```
opendaylight-user@root>feature:uninstall sdnhub-tutorial-adsal
```

```
opendaylight-user@root>bundle:list -s | grep sdnhub
```

```
opendaylight-user@root>feature:install sdnhub-tutorial-mdsal
```

Refreshing bundles osgi.cmpn (75), org.ops4j.pax.web.pax-web-api (110), org.apache.xbean.finder-shaded (109), org.ops4j.pax.web.pax-web-runtime (112), org.apache.xbean.bundleutils (106), org.ops4j.pax.web.pax-web-spi (111), org.apache.aries.util (13), javax.persistence (90)

```
opendaylight-user@root>bundle:list -s | grep sdnhub
```

```
329 | Active | 80 | 0.1.0.SNAPSHOT | org.sdnhub.odl.tutorial.mdsal_L2_forwarding
```

```
opendaylight-user@root>feature:list | grep sdnhub
```

```
sdnhub-tutorial-mdsal | 1.0.0-SNAPSHOT | x | tutorial-mdsal-1.0.0-SNAPSHOT  
| SDN Hub Tutorial :: OpenDaylight :: MD-SAL
```

```
sdnhub-tutorial-adsal | 1.0.0-SNAPSHOT | | tutorial-adsal-1.0.0-SNAPSHOT  
| SDN Hub Tutorial :: OpenDaylight :: AD-SAL
```

```
opendaylight-user@root>feature:install odl-restconf-all
```

```
opendaylight-user@root>feature:install odl-mdsal-apidocs
```

```
opendaylight-user@root>^D (to quit)
```

```
ubuntu@sdnhubvm:~/SDNHub_Openaylight_Tutorial/distribution/opendaylight-  
karaf/target/distribution-karaf-0.5.0-SNAPSHOT$ rm -rf data
```

- In the above output, “x” means the feature is loaded.
- In the Karaf shell, we can get a list of all the bundles and features, as well as check on what interfaces are imported or exported (by using commands “imports” and “exports”).
- If we had mininet running, we will notice that the ping will succeed after the “feature:install” is performed for both of the two features.
- Using Karaf, the RESTConf and REST API doc explorer are available on TCP port 8181 (not 8080). For example:  
http://localhost:8181/restconf/operational/opendaylightinventory:nodes and  
http://localhost:8181/apidoc/explorer/.
- Finally, we delete the “data” directory to clear all the cache and configs from the previously run.

### 3.9. Debugging

Debugging is an significant part of app development. Once we get the Eclipse environment fine-tuned to an advanced level, we will be able to Run and Debug within that IDE. The debugging is depicted with the AD-SAL app, but the concepts are same for the MD-SAL app too.

To debug any code, we need to start OpenDaylight including the parameter “-debug”. As shown above:

```
ubuntu@ubuntu:~/SDNHub_Openaylight_Tutorial$ cd distribution/opendaylight-osgi-  
adsal/target/distribution-osgi-adsal-1.1.0-SNAPSHOT-osgipackage/opendaylight
```

```
ubuntu@ubuntu:~/SDNHub_Openaylight_Tutorial/distribution/opendaylight-osgi-  
adsal/target/distribution-osgi-adsal-1.1.0-SNAPSHOT-osgipackage/opendaylight$ ./run.sh -debug
```

“-debug” is allowing us to connect to OSGI (listening at port 8000) for debugging purposes.

The first step in order to start the debugger is to click “Connect to ODL” as shown below (Figure 19). This allows as to debug our code using all the Eclipse benefits.

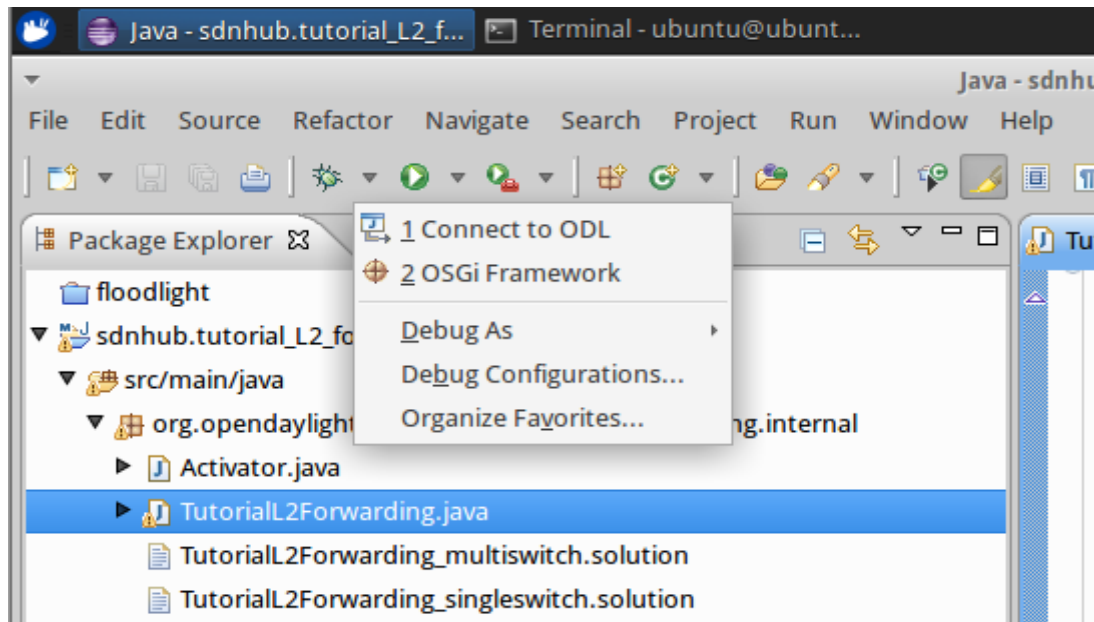


Figure 19:how to debug code using Eclipse

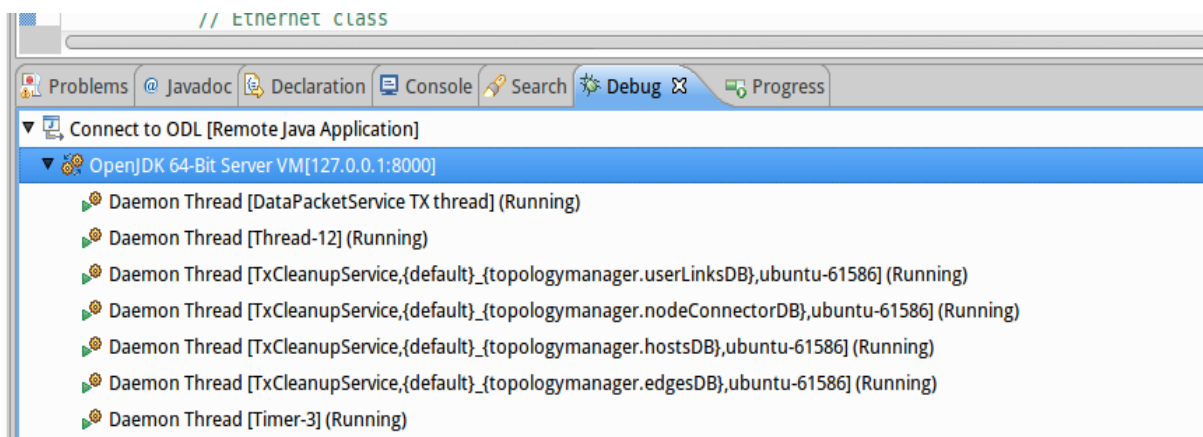


Figure 20: Debug code using Eclipse

## 4. Floodlight

### 4.1. Introduction to Floodlight

#### Why Use Floodlight?

- OpenFlow: works with physical and virtual switches that speak the OpenFlow protocol.
- It is licensed by apache, which allows you to use Floodlight for almost any purpose.
- Open community: An open community of developers is responsible for Floodlight's development.
- Easy to Use: Floodlight is really simple to build and run, which makes it accessible to users.
- Tested and Supported: A community of professional developers (floodlight-dev@openflowhub.org) is actively keeping Floodlight tested and improved.

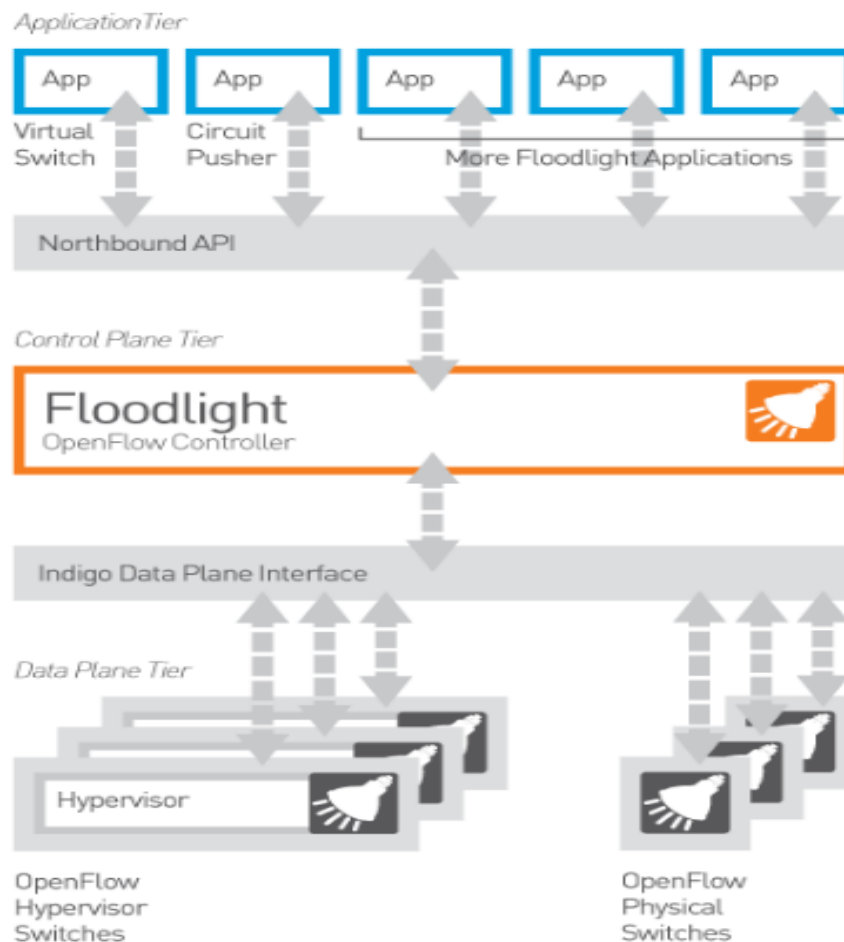


Figure 21: Floodlight OpenFlow Controller

The Floodlight Open SDN Controller as mentioned before is an enterprise-class, Apache-licensed, Java-based OpenFlow Controller. It is supported by a community of developers consisting of a number of engineers from Big Switch Networks.

OpenFlow is an open standard managed by Open Networking Foundation. It specifies a protocol through which a remote controller can modify the behavior of networking devices through a well-defined “forwarding instruction set”. Floodlight is configured to work with the growing number of routers, switches, virtual switches, and access points that support the OpenFlow standard.

Feature Highlights

Gives to users a module loading system, which it is simple to extend and enhance.

Minimal dependencies are required because of it’s easy to set up process.

Supports a broad range of virtual and physical OpenFlow switches

Can support mixed OpenFlow and non-OpenFlow networks and it can manage multiple “islands” of OpenFlow hardware switches.

Designed to be high-performance since it is multithreaded from the ground up.

Support for OpenStack (link) cloud orchestration platform.

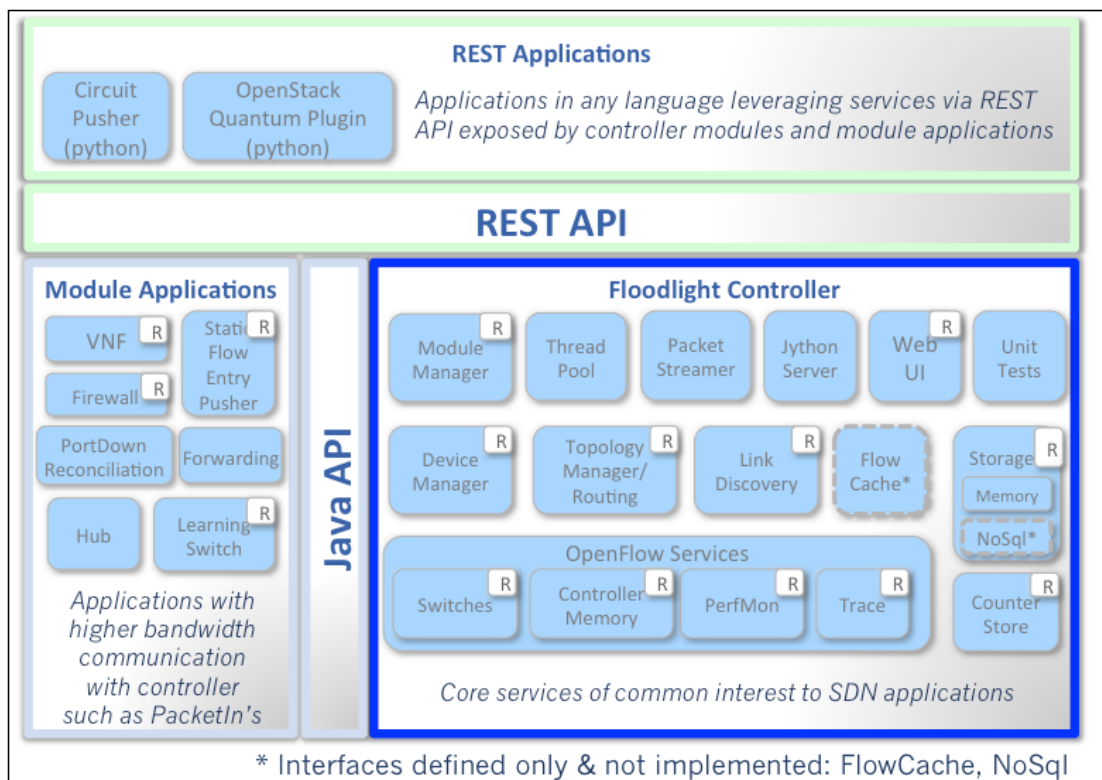


FIGURE 22: Floodlight – Architecture Diagram

Floodlight is not just an OpenFlow controller (the "Floodlight Controller") but also it is a collection of applications built on top the Floodlight Controller (Figure 22).

The Floodlight Controller realizes a set of common functionalities to control and inspect an OpenFlow network, but applications on top of it realize different features to solve different user needs over the

network. The figures above show the relationship between the Floodlight Controller, the applications built as Java modules compiled with Floodlight, and the applications built over the Floodlight REST API.

Just after you run Floodlight, the controller and the set of Java module applications (those loaded in the floodlight properties file) start running. The REST APIs exposed by all running modules are available via the specified REST port, which by default is 8080. All REST applications can be written in any language and it is easy to retrieve information and invoke services by sending http REST commands to the controller REST port.



## 5. Reactive Flow Programming with OpenDaylight

To make things more specifically, we consider a simple primary scenario in this thesis: load balancing of a TCP service (e.g., a web service using HTTP over TCP). The basic idea is that TCP connections to a service addressed with a public IP address and port number are distributed between two physical server instances using IP address re-writing performed by an OpenFlow switch. Every time a client starts a TCP connection to the service, one of the server instances is chosen randomly, and a forwarding rule is installed by the network controller on the ingress switch to forward all incoming packets of this TCP connection to the chosen server instance. In order to make sure that the server instance accepts the packets of the TCP connection, the destination IP address is re-written to the IP address of the chosen server instance, and the destination MAC address is set to the MAC address of the server instance. In the reverse direction from server to client, the switch re-writes the source IP address of the server to the public IP address of the service. Therefore, to the client it looks like the response is coming from the public IP address. Thus in this way, load balancing is transparent to the client.

In order to keep things simple, we do not study the routing of packets. Rather, it is assuming that the clients and the two server instances are connected to the same switch on different ports (see figure 23). Moreover, we have also simplify the MAC address resolution by setting a static ARP table entry at the client host for the public IP address. Since there is no physical server assigned to the public IP address, we just set a fake MAC address (in a real setup, the gateway of the data center would receive the client request, so we would not need an extra MAC address assigned to the public IP address).

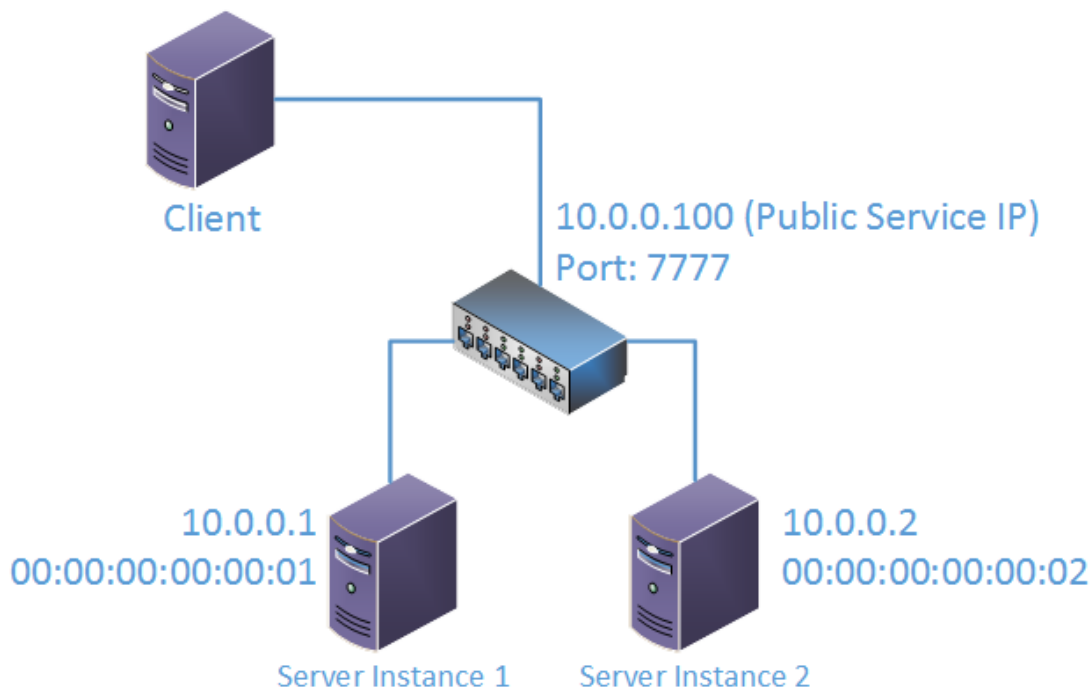


Figure 23: Topology – 1 Switch with 3 servers

The folder `myctrapp` contains the Maven project of the OSGi module. We can compile and create the OSGi bundle with the following command.

```
user@host:$ tar xzf myctrlapp.tar.gz
```

```
user@host:$ cd ~/myctrlapp
```

```
user@host:$ mvn package
```

The corresponding Eclipse project can be created using:

```
user@host:$ cd ~/myctrlapp
```

```
user@host:$ mvn eclipse:eclipse
```

## 5.1. Registering Required Services and Subscribing to Packet-in Events

For our load balancer implementation, are required the following OpenDaylight services:

- Data Packet Service for decoding incoming packets and encoding and sending outgoing packets.
- Flow Programmer Service for setting flow table entries on the switch.
- Switch Manager Service to determine the output of packets forwarded to the server instances.  
Implementing the `configureInstance(...)` method of the Activator class:

```
public void configureInstance(Component c, Object imp, String containerName) {  
  
    log.trace("Configuring instance");  
  
    if (imp.equals(PacketHandler.class)) {  
  
        // Define exported and used services for PacketHandler component.  
  
        Dictionary<String, Object> props = new Hashtable<String, Object>();  
  
        props.put("salListenerName", "mypackethandler");  
  
        // Export IListenDataPacket interface to receive packet-in events.  
  
        c.setInterface(new String[] {IListenDataPacket.class.getName()}, props);  
    }  
}
```

```

// Need the DataPacketService for encoding, decoding, sending data packets

c.add(createContainerServiceDependency(containerName).setService(IDataPacketService.class).setCa
llbacks(

    "setDataPacketService", "unsetDataPacketService").setRequired(true));

// Need FlowProgrammerService for programming flows

c.add(createContainerServiceDependency(containerName).setService(IFlowProgrammerService.class)
.setCallbacks(

    "setFlowProgrammerService", "unsetFlowProgrammerService").setRequired(true));

// Need SwitchManager service for enumerating ports of switch

c.add(createContainerServiceDependency(containerName).setService(ISwitchManager.class).setCallb
acks(

    "setSwitchManagerService", "unsetSwitchManagerService").setRequired(true));

```

set... and unset... define names of callback methods. These callback methods are implemented in our PacketHandler class to receive service proxy objects, which can be used to call the services:

```

/**
 * Sets a reference to the requested DataPacketService
 */
void setDataPacketService(IDataPacketService s) {
    log.trace("Set DataPacketService.");

    dataPacketService = s;
}

```

```
/**
 * Unsets DataPacketService
 */
void unsetDataPacketService(IDataPacketService s) {
    log.trace("Removed DataPacketService.");

    if (dataPacketService == s) {
        dataPacketService = null;
    }
}

/**
 * Sets a reference to the requested FlowProgrammerService
 */
void setFlowProgrammerService(IFlowProgrammerService s) {
    log.trace("Set FlowProgrammerService.");

    flowProgrammerService = s;
}

/**
 * Unsets FlowProgrammerService
 */
void unsetFlowProgrammerService(IFlowProgrammerService s) {
    log.trace("Removed FlowProgrammerService.");
```

```
    if (flowProgrammerService == s) {  
        flowProgrammerService = null;  
    }  
}  
  
/**  
 * Sets a reference to the requested SwitchManagerService  
 */  
void setSwitchManagerService(ISwitchManager s) {  
    log.trace("Set SwitchManagerService.");  
  
    switchManager = s;  
}  
  
/**  
 * Unsets SwitchManagerService  
 */  
void unsetSwitchManagerService(ISwitchManager s) {  
    log.trace("Removed SwitchManagerService.");  
  
    if (switchManager == s) {  
        switchManager = null;  
    }  
}
```

Further more, register for packet-in events in the Activator class. Also, we have to mention that we implement the `IListenDataPacket` interface. This interface basically consists of one callback method `receiveDataPacket(...)` for receiving packet-in events as described next.

## 5.2. Handling Packet-in Events

Every time that a packet without matching flow table entry arrives at the switch, it is sent to the controller and the event handler `receiveDataPacket(...)` of our packet handler class is called with the received packet as parameter:

```
@Override
public PacketResult receiveDataPacket(RawPacket inPkt) {
    // The connector, the packet came from ("port")
    NodeConnector ingressConnector = inPkt.getIncomingNodeConnector();

    // The node that received the packet ("switch")
    Node node = ingressConnector.getNode();

    log.trace("Packet from " + node.getNodeIDString() + " " +
    ingressConnector.getNodeConnectorIDString());

    // Use DataPacketService to decode the packet.
    Packet pkt = dataPacketService.decodeDataPacket(inPkt);

    if (pkt instanceof Ethernet) {
        Ethernet ethFrame = (Ethernet) pkt;
        Object l3Pkt = ethFrame.getPayload();

        if (l3Pkt instanceof IPv4) {
            IPv4 ipv4Pkt = (IPv4) l3Pkt;

```

```

InetAddress clientAddr = intToInetAddress(ipv4Pkt.getSourceAddress());

InetAddress dstAddr = intToInetAddress(ipv4Pkt.getDestinationAddress());

Object l4Datagram = ipv4Pkt.getPayload();

if (l4Datagram instanceof TCP) {

    TCP tcpDatagram = (TCP) l4Datagram;

    int clientPort = tcpDatagram.getSourcePort();

    int dstPort = tcpDatagram.getDestinationPort();

    if (publicInetAddress.equals(dstAddr) && dstPort == SERVICE_PORT) {

        log.info("Received packet for load balanced service");

        // Select one of the two servers round robin.

        InetAddress serverInstanceAddr;

        byte[] serverInstanceMAC;

        NodeConnector egressConnector;

        // Synchronize in case there are two incoming requests at the same time.

        synchronized (this) {

            if (serverNumber == 0) {

                log.info("Server 1 is serving the request");

                serverInstanceAddr = server1Address;

                serverInstanceMAC = SERVER1_MAC;

                egressConnector = switchManager.getNodeConnector(node,
SERVER1_CONNECTOR_NAME);

                serverNumber = 1;

```

```

    } else {
        log.info("Server 2 is serving the request");
        serverInstanceAddr = server2Address;
        serverInstanceMAC = SERVER2_MAC;
        egressConnector = switchManager.getNodeConnector(node,
SERVER2_CONNECTOR_NAME);
        serverNumber = 0;
    }
}

```

```

// Create flow table entry for further incoming packets

```

```

// Match incoming packets of this TCP connection

```

```

// (4 tuple source IP, source port, destination IP, destination port)

```

```

Match match = new Match();

```

```

match.setField(MatchType.DL_TYPE, (short) 0x0800); // IPv4 ethertype

```

```

match.setField(MatchType.NW_PROTO, (byte) 6); // TCP protocol id

```

```

match.setField(MatchType.NW_SRC, clientAddr);

```

```

match.setField(MatchType.NW_DST, dstAddr);

```

```

match.setField(MatchType.TP_SRC, (short) clientPort);

```

```

match.setField(MatchType.TP_DST, (short) dstPort);

```

```

// List of actions applied to the packet

```

```

List actions = new LinkedList();

```

```

// Re-write destination IP to server instance IP

```

```

actions.add(new SetNwDst(serverInstanceAddr));

```



```

// Re-write destination MAC to server instance MAC
actions.add(new SetDIDst(serverInstanceMAC));

// Output packet on port to server instance
actions.add(new Output(egressConnector));

// Create the flow
Flow flow = new Flow(match, actions);

// Use FlowProgrammerService to program flow.
Status status = flowProgrammerService.addFlow(node, flow);
if (!status.isSuccess()) {
    log.error("Could not program flow: " + status.getDescription());
    return PacketResult.CONSUME;
}

// Create flow table entry for response packets from server to client

// Match outgoing packets of this TCP connection
match = new Match();
match.setField(MatchType.DL_TYPE, (short) 0x0800);
match.setField(MatchType.NW_PROTO, (byte) 6);
match.setField(MatchType.NW_SRC, serverInstanceAddr);
match.setField(MatchType.NW_DST, clientAddr);
match.setField(MatchType.TP_SRC, (short) dstPort);

```

```

match.setField(MatchType.TP_DST, (short) clientPort);

// Re-write the server instance IP address to the public IP address
actions = new LinkedList();
actions.add(new SetNwSrc(publicInetAddress));
actions.add(new SetDlSrc(SERVICE_MAC));

// Output to client port from which packet was received
actions.add(new Output(ingressConnector));

flow = new Flow(match, actions);
status = flowProgrammerService.addFlow(node, flow);
if (!status.isSuccess()) {
    log.error("Could not program flow: " + status.getDescription());
    return PacketResult.CONSUME;
}

// Forward initial packet to selected server

log.trace("Forwarding packet to " + serverInstanceAddr.toString() + " through port " +
egressConnector.getNodeConnectorIDString());

ethFrame.setDestinationMACAddress(serverInstanceMAC);
ipv4Pkt.setDestinationAddress(serverInstanceAddr);
inPkt.setOutgoingNodeConnector(egressConnector);
dataPacketService.transmitDataPacket(inPkt);

return PacketResult.CONSUME;

```

```

    }
}
}
}

// We did not process the packet -> let someone else do the job.

return PacketResult.IGNORED;

```

Our load balancer reacts as follows to packet-in events. At first, it uses the Data Packet Service to decode the incoming packet using method `decodeDataPacket(inPkt)`. We are only interested in packets addressed to the public IP address and port number of our load-balanced service. For this reason, we have to check the destination IP address and port number of the received packet. After this, we iteratively decode the packet layer by layer. Firstly, we check whether we received an Ethernet frame, and get the payload of the frame, which should be an IP packet for a TCP connection. If the payload of the frame is indeed an IPv4 packet, we typecast it to the corresponding IPv4 packet class and use the methods `getSourceAddress(...)` and `getDestinationAddress(...)` to retrieve the IP addresses of the client (source) and service (destination). After, we have to go up one layer and check for a TCP payload to retrieve the port information with a similar way.

When we have retrieved the IP address and port information from the packet, we check whether it is targeted at our load-balanced service. If it is not addressed at our service, we ignore the packet and let another handler process it (if any) by returning `packetResult.IGNORED` as a result of the packet handler.

In case that the packet is addressed at our service, we choose one of the two physical service instances in a round-robin fashion. The idea is to send the first request to server 1, second request to server 2, third to server 1 again, etc. Note that it is possible to have multiple packet handlers for different packets executed in parallel (at least, we should not rely on a sequential execution as long as we do not know how OpenDaylight handles requests). For this reason, we synchronize this part of the packet handler, to make sure that only one thread is in this code section at a time.

### 5.3. Programming Flows

In order to forward a packet to the selected server instance, we select its IP address and MAC address as target addresses for each packet of this TCP connection from the client. At this point, we use IP address and MAC address re-writing to re-write the IP destination address and MAC destination address of each incoming packet of this connection to the selected server address. We have also to mention that a TCP connection is identified by the 4-tuple [source IP, source MAC, destination IP, destination MAC]. For this reason, we use this information as match criteria for the flows that perform address re-writing and packet forwarding.

A flow table entry consists of a match rule and list of actions. As previously said, the match rule should be able to identify packets of a certain TCP connection. At this point, we create a new Match object, and set the required fields. And since we are matching on a TCP/IPv4 datagram, we have to make sure to identify this packet type by setting the ethertype (0x0800 meaning IPv4) and protocol id (6 meaning TCP). In addition, we set the source and destination IP address and port information of the client and service which is used to identify the individual TCP connection.

Then, we have to define the actions to be applied to a matched packet of the TCP connection. We define an action for re-writing the IP destination address to the IP address of the selected server instance, as well as the destination MAC address. Also, we define an output action to forward packets over the switch port of the server instance. We utilize the Switch Manager Service to retrieve the corresponding connector of the switch with its name. Retain that these names are not just the port numbers but s1-eth1 and s1-eth1 in the setup using Mininet. If we want to find out the name of a port, we can use the web GUI of the OpenDaylight controller (<http://controllerhost:8080/>) and see over the port names of the switch.

Sometimes, it might also be useful to enumerate all connectors of a switch (node). As for example, to flood a packet, using this method:

```
Set ports = switchManager.getUpNodeConnectors(node)
```

At this end, we create the flow with match criteria and actions, and program the switch using the Flow Programmer service.

In the reverse direction from server to client, we also install a flow that re-writes the source IP address and MAC address of outgoing packets to the address information of the public service.

## 5.4. Forwarding Packets

However, we have not finished yet. Although we have succeed that every new packet of the connection will be forwarded to the right server instance, we also have to forward the received initial packet (TCP SYN request) to the correct server. Then, we modify the destination address information of this packet. Finally, we use the Data Packet Service to forward the packet by using method `transmitDataPacket(...)`.

In this example, we simply re-used the received packet. However, sometimes we might want to create and send a new packet. We create the payloads of the packets on the different layers and encode them as a raw packet using the Data Packet Service:

```
TCP tcp = new TCP();  
  
tcp.setDestinationPort(tcpDestinationPort);  
  
tcp.setSourcePort(tcpSourcePort);  
  
IPv4 ipv4 = new IPv4();  
  
ipv4.setPayload(tcp);
```

```

ipv4.setSourceAddress(ipSourceAddress);

ipv4.setDestinationAddress(ipDestinationAddress);

ipv4.setProtocol((byte) 6);

Ethernet ethernet = new Ethernet();

ethernet.setSourceMACAddress(sourceMAC);

ethernet.setDestinationMACAddress(targetMAC);

ethernet.setEtherType(EtherTypes.IPv4.shortValue());

ethernet.setPayload(ipv4);

RawPacket destPkt = dataPacketService.encodeDataPacket(ethernet);

```

Instead of implementing a load balance using the Rest Appi, which imports the flows of packets we use a simpler example, the test application. It is used a text counter for the servers's requests.

```

public void run() {

    String f="/home/ubuntu/Documents/stats";

    FileInputStream in=null;

    String text = null;

    while(true){

        try{

            Thread.sleep(1000);

            try (BufferedReader br = new BufferedReader(new FileReader(f)) {

                String line;

                int counter1 = 0, counter2 = 0;

                while ((line = br.readLine()) != null) {

                    if(line.equals("1")){

                        counter1++;

                    }

                    else{

```

```

        counter2++;
    }
}
System.out.println(counter1 + " " + counter2);
gPanel.update(counter1, counter2);

}
} catch(Exception ex) {
}
}
}
}

```

## 5.5. Implementation

Jumpstart your SDN development through our all-in-one pre-built tutorial VM, built for you by SDN Hub. This is a 64-bit Ubuntu 14.04 image (3GB) that has a number of SDN software and tools installed.

- SDN Controllers: OpenDaylight, Floodlight, ONOS, POX, RYU, Floodlight-OF1.3 and Trema
- Example code for a hub, traffic tap, L2 learning switch and other applications
- Open vSwitch 2.3.0 with support for Openflow 1.2, 1.3 and 1.4, and LINC switch
- Mininet: in order to create and run example topologies
- Pyretic
- Wireshark 1.12.1 with native support for OpenFlow parsing
- JDK 1.8, Eclipse Luna, and Maven 3.3.3

### Download

You can directly download the OVA file from the SDNhub's file server:  
<http://sdnhub.org/tutorials/sdn-tutorial-vm/>

Instructions to use the VM

Import the OVA (SDN Hub Tutorial VM) into Virtualbox or VMware Player and boot it. We can change anything of the VM attributes, but it is highly recommend to allocate at least 2 vCPUs and 2GB memory.

In case that the OVA (version 1.0) does not work on our VirtualBox or VMware player, we have to unzip the OVA and extract the VMDK file. Then this file can be used to create a VM in our environment.

Also we have to make sure the connectivity to the Internet from the VM is working. If not, we have to ensure that our Virtualbox/VMware network settings are correct for the VM's network adapter (should be in NAT mode).

In order to login: Username and passwd are both "ubuntu"

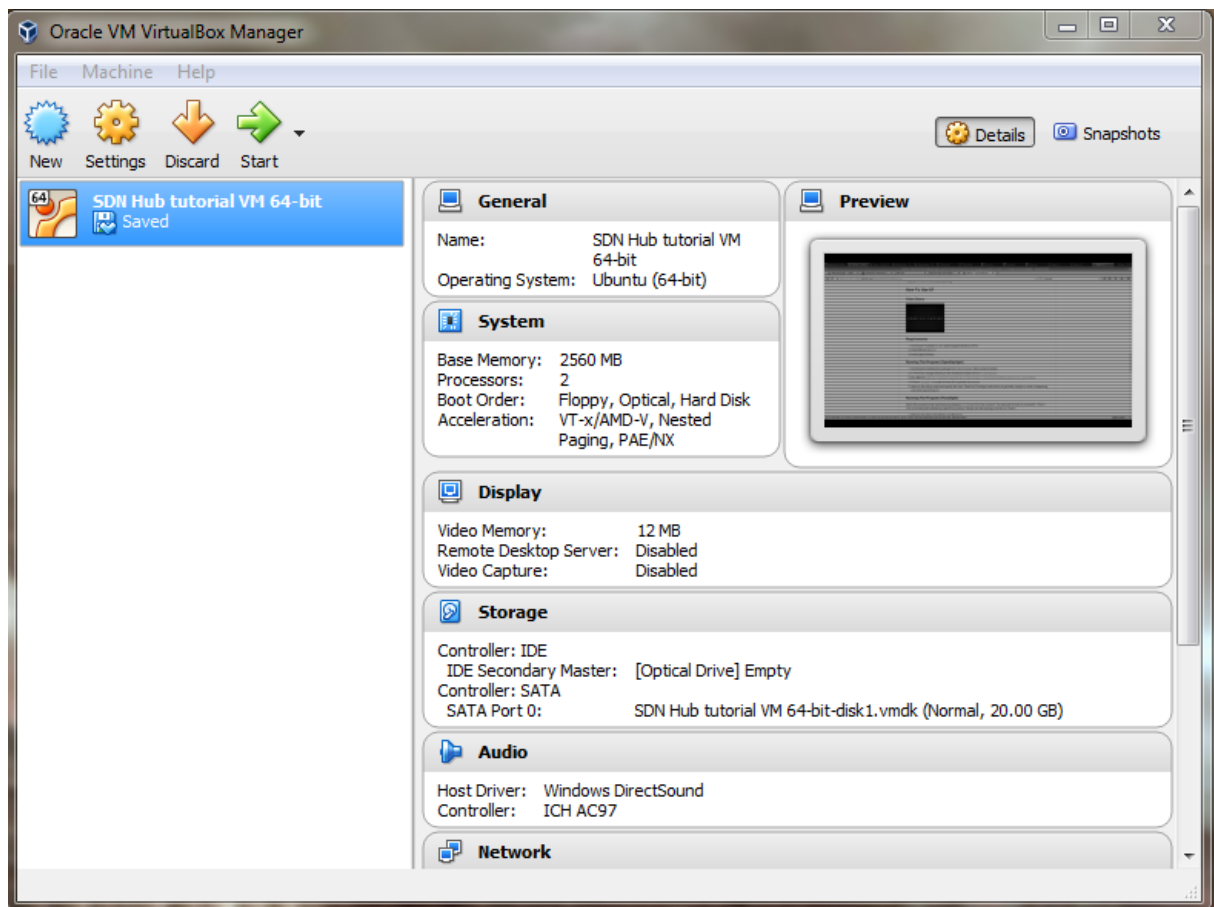


Figure 24: Import the OVA (SDN Hub Tutorial VM) into Virtualbox

First, we download the files `test.tar.gz`, `myctrlapp.tar.gz` and `loadbalancer.tar.gz` to Downloads. Then we open a terminal and execute the followings, in order to unzip the files:

```
cd Downloads
```

```
tar xvf loadbalancer.tar.gz
```

```
tar xvf test.tar.gz
```

```
tar xvf myctrlapp.gz
```

Then we open the Eclipse and go to File -> Import -> General -> Existing Projects into Workspace.

From the Browse go to Downloads and find the folder test and we import it in Eclipse.

To avoid some errors using an external library (JFreeChart).

Go to page <http://sourceforge.net/projects/jfreechart/files/1.%20JFreeChart/1.0.19/>

and download the file jfreechart-1.0.19.zip on Downloads. After that we open a terminal and unzip the file as previously.

```
cd Downloads
```

```
unzip jfreechart-1.0.19.zip
```

Open the eclipse and do right click on the project test and choose Properties. Then we go to Java Build Path and then on Libraries. Select the jcommon and jfreechart and do remove. Then press the Add External Jars, go to Downloads / jfreechart / lib and select the new jcommon-1.0.23.jar and jfreechart-1.0.19.jar files.

We can compile the OSGi bundle using Maven as follows:

```
user@host:$ cd ~/myctrlapp
```

```
user@host:$ mvn package
```

```
user@host:$ cd ~/test
```

```
user@host:$ mvn package
```

Then we start the OpenDaylight controller. Open another terminal and type:

```
cd /home/ubuntu/SDNHub_OpenDaylight_Tutorial/distribution/opendaylight-osgi-  
mdsal/target/distribution-osgi-mdsal-1.1.0-SNAPSHOT-osgipackage/opendaylight/
```

```
sudo ./run.sh
```

We open a browser and type the following link 127.0.0.1:8080 to see if it is properly load the environment.



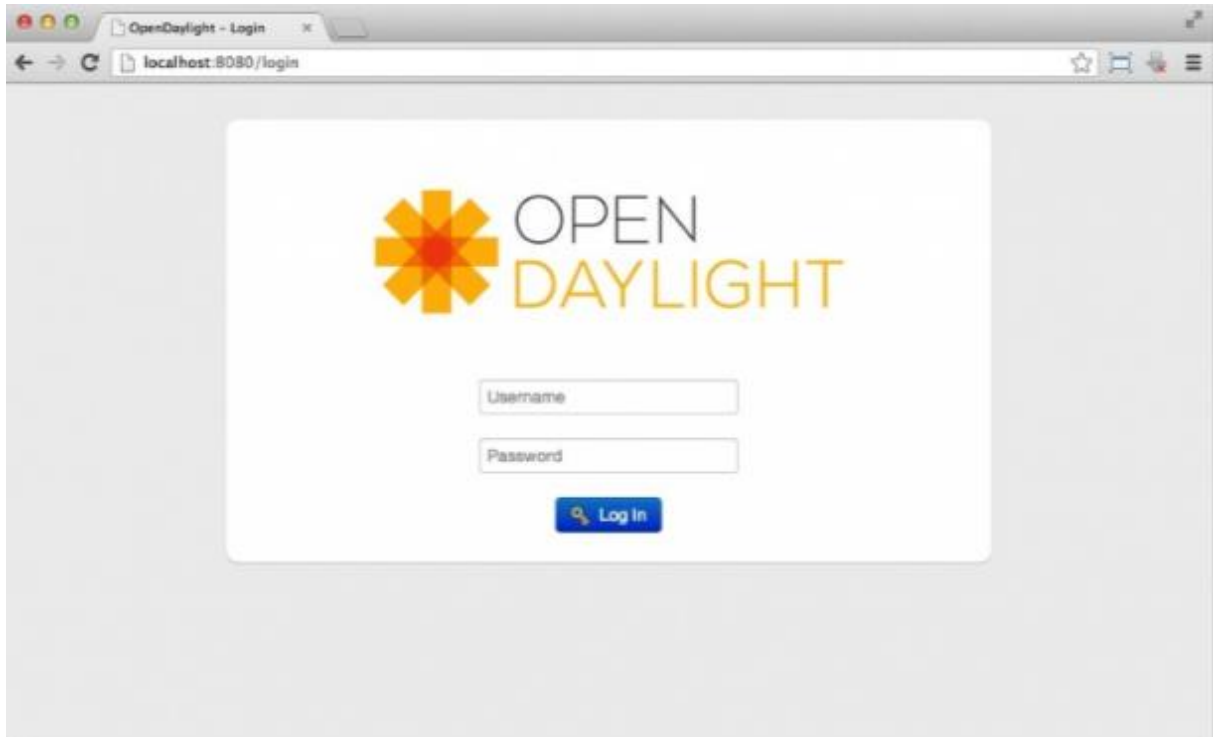


FIGURE 25: OpendayDaylight GUI - login

To get list of all APIs exported over RestConf, we can use the API explorer and seeing the list of API by accessing <http://localhost:8080/apidoc/explorer/>.

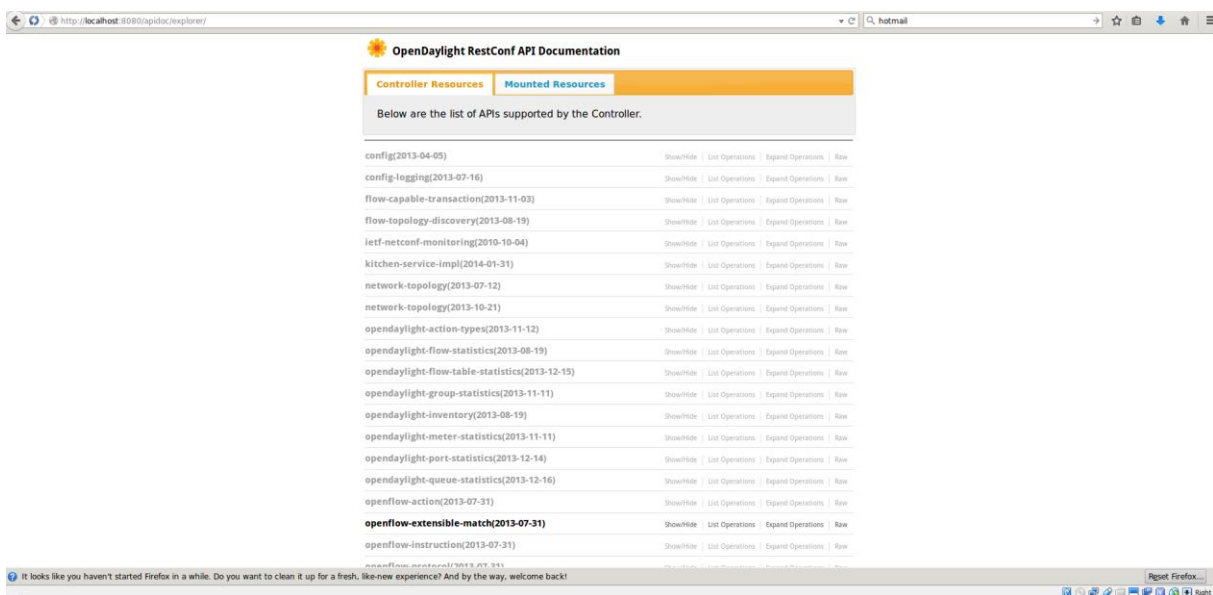


FIGURE 26: OpendayDaylight RestConf API (1)

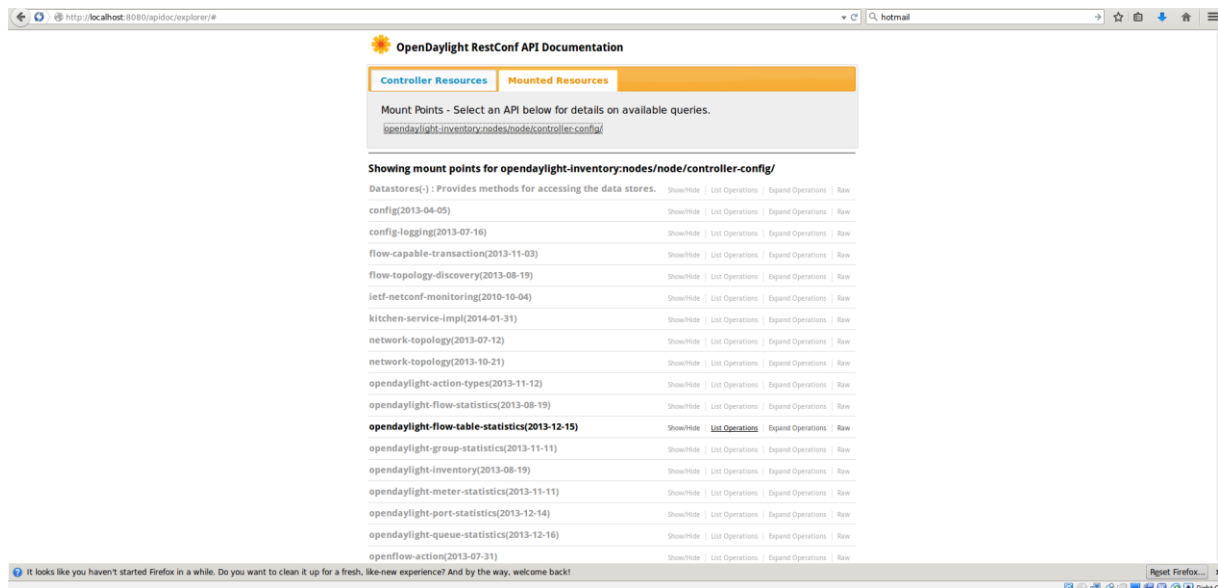


FIGURE 27: OpendayDaylight RestConf API (2)

Afterwards, to avoid conflicts with our service, we should first stop OpenDaylight's simple forwarding service and any other service that is running, and has nothing to do with our load balancing service, from the OSGi console:

On the same terminal type:

```
osgi> ss | grep simple
```

```
171 ACTIVE org.opendaylight.controller.samples.simpleforwarding_0.4.1.SNAPSHOT
```

```
true
```

```
osgi> stop 171
```

If another service is running, we have to stop it.

```
osgi> ss | grep loadbalancer
```

```
osgi> 150 ACTIVE org.opendaylight.controller.samples.loadbalancer.northbound_0.4.1.SNAPSHOT
```

```
187 ACTIVE org.opendaylight.controller.samples.loadbalancer_0.5.1.SNAPSHOT
```

```
true
```

```
osgi> stop 187
```

The only service we need to be active is the following:

```
org.opendaylight.controller.samples.loadbalancer.northbound_0.4.1.SNAPSHOT
```

Since, both of these services implement packet handlers, now we want to make sure that they do not interfere with our handler.

Then, we can install our compiled OSGi bundle.

On the same terminal type:

```
install file:/home/ubuntu/Downloads/loadbalancer/target/loadbalancer-1.0.jar
```

We can also change the log level of our bundle to see log output down to the trace level:

```
setLogLevel gr.opendaylight.loadbalancer.PacketHandler trace
```

We do the same for the myctrlapp experiment:

```
osgi> install file:/home/user/myctrlapp/target/myctrlapp-0.1.jar
```

Bundle id is 256 and start it:

```
osgi> start 256
```

Next, we create a simple Mininet topology with one switch and three hosts. We open another terminal and type:

```
user@host:$ sudo mn --controller=remote,ip=127.0.0.1 --topo single,3 --mac --arp
```

```
Terminal
File Edit View Terminal Tabs Help
ubuntu@sdnhubvm:~[09:06]$ sudo mn --controller=remote,ip=127.0.0.1 --topo single
,3 --mac --arp
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet> xtern h3
*** Unknown command: xtern h3
mininet> xtern h1
*** Unknown command: xtern h1
mininet> xterm h1
mininet> xterm h2
mininet> xterm h3
mininet> ping all
*** Unknown command: ping all
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
mininet>
```

FIGURE 28: Mininet default topology

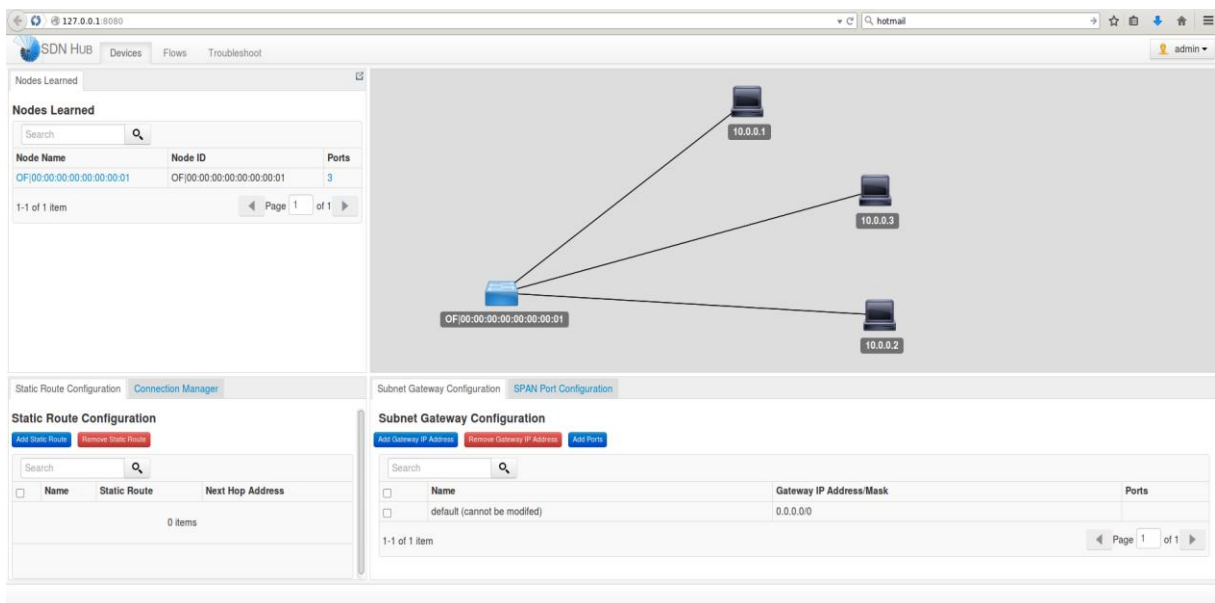


FIGURE 29: Topology inside OpenDaylight

Be sure to use the IP address of your OpenDaylight controller host (or local host). The option `--mac` assigns a MAC address according to the host number to each host (e.g., `00:00:00:00:00:01` for the first host).

Option `--arp` pre-populates the ARP cache of the hosts. We use host 1 and 2 as the server hosts with the IP addresses `10.0.0.1` and `10.0.0.2` respectively. Host 3 runs the client. Moreover, we also set a static ARP table entry on host 3 for the public IP address of the service (`10.0.0.100`):

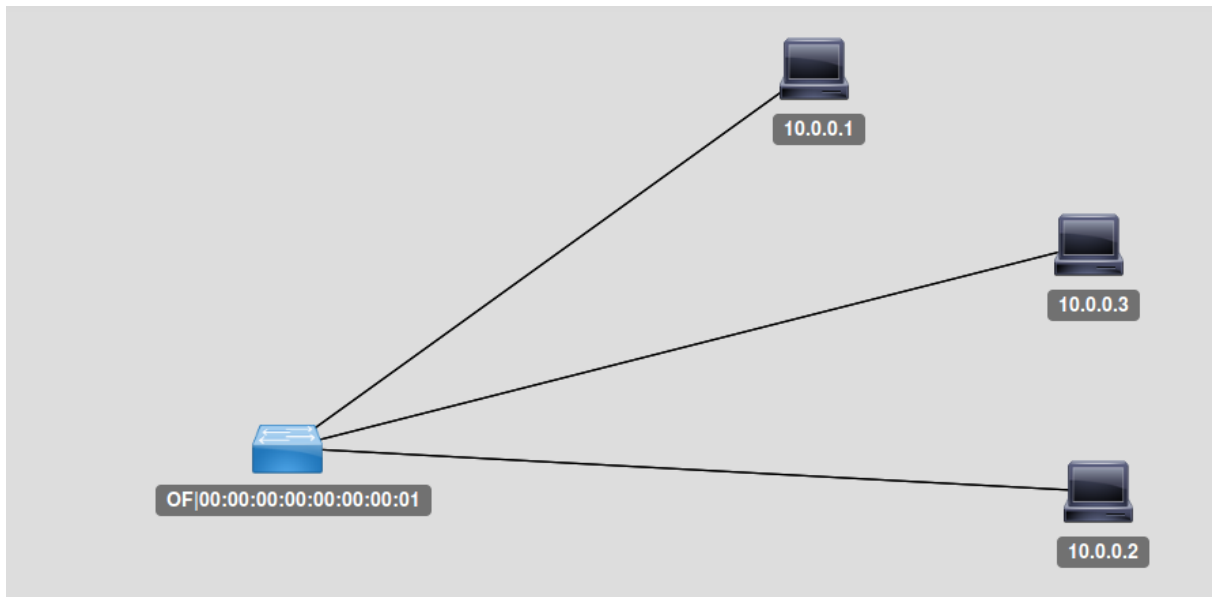


FIGURE 30: Topology from Opendaylight

```
mininet> xterm h3
```

```
mininet h3> arp -s 10.0.0.100 00:00:00:00:00:64
```

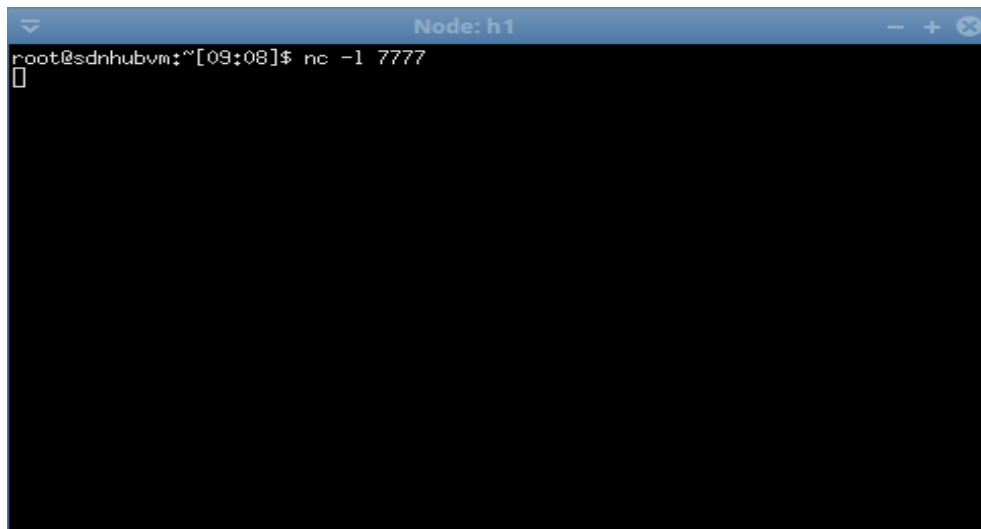
We start two simple servers using netcat listening on port `7777`, on hosts 1 and 2 respectively:

```
mininet> xterm h1
```

```
mininet> xterm h2
```

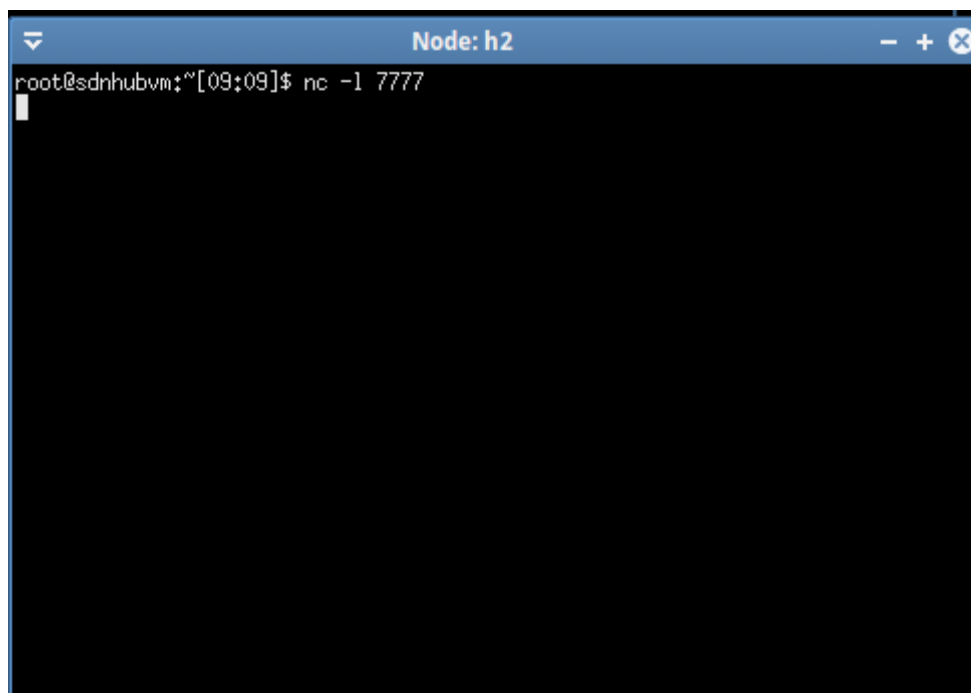
```
mininet h1> nc -l 7777
```

```
mininet h2> nc -l 7777
```



```
Node: h1
root@sdnhubvm:~[09:08]$ nc -l 7777
[]
```

FIGURE 31: Node h1 - netcat listening on port 7777



```
Node: h2
root@sdnhubvm:~[09:09]$ nc -l 7777
[]
```

FIGURE 32: Node h2 - netcat listening on port 7777

Then, using netcat again, we send a message to our service from the client (in our case host 3):

```
mininet h3> echo "Hello" | nc 10.0.0.100 7777
```

```
Node: h3
root@sdnhubvm:~[09:09]$ arp -s 10.0.0.100 00:00:00:00:00:64
root@sdnhubvm:~[09:10]$ echo "Hello" | nc 10.0.0.100 7777
root@sdnhubvm:~[09:16]$ echo "Hello" | nc 10.0.0.100 7777
root@sdnhubvm:~[20:13]$ echo "Hello" | nc 10.0.0.100 7777
root@sdnhubvm:~[20:18]$ echo "Hello" | nc 10.0.0.100 7777

root@sdnhubvm:~[20:20]$
root@sdnhubvm:~[20:20]$
root@sdnhubvm:~[20:20]$ echo "Hello" | nc 10.0.0.100 7777

root@sdnhubvm:~[20:38]$
root@sdnhubvm:~[20:38]$ echo "Hello" | nc 10.0.0.100 7777
root@sdnhubvm:~[20:43]$ echo "Hello" | nc 10.0.0.100 7777
root@sdnhubvm:~[20:46]$ □
```

FIGURE 33: Node h3 - netcat sen message on port 7777

After that we go to eclipse and run the test application.

Now, we should see the output “Hello” in the xterm of host 1. If we try to execute the same command again, the output will appear in the xterm of host 2. This shows that requests (TCP connections) are correctly distributed between the two servers.

If we go to eclipse that runs the application will see the requests to climb in the GUI.

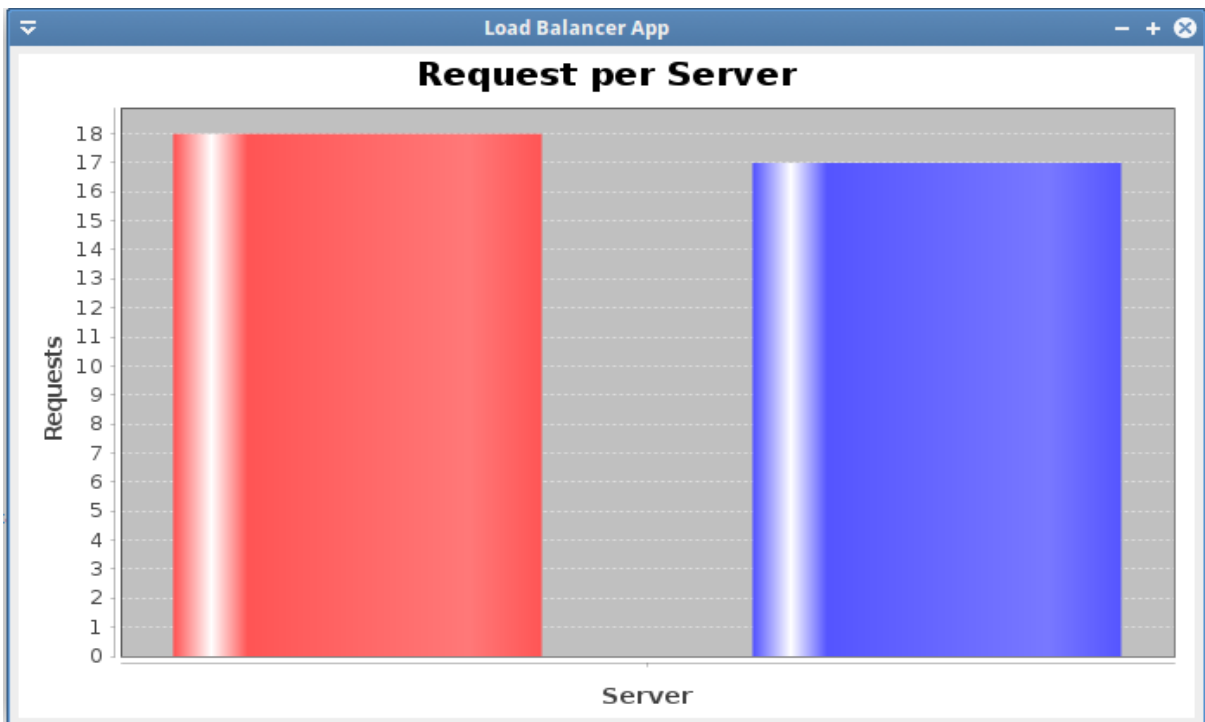


FIGURE 34: Load Balancer App GUI

## 6. Load Balancing

The goal is to perform load balancing on any fat tree topology using SDN Controllers (Floodlight and OpenDaylight) but at the same time ensure that the latency is set to minimum. Dijkstra's algorithm is implemented in order to find multiple paths of same length. This enables us to reduce the search to a small region in the fat tree topology. It is also important to mention that OpenDaylight by default forwards traffic to all ports. For this reason, specific rules might need to be pushed to get a proper load balancing output. Currently the program simply finds the path with least load and forwards traffic on that path.

### 6.1. Implementation Approach

At first we can use the REST APIs to collect operational information of the topology and its devices.

- Enable statistics collection in case of Floodlight (TX, Transmission Rate, RX, Receiving Rate, etc). This step is not available for OpenDaylight.
- Find information about the connected hosts such as their IP, the Switches to which they are connected, MAC Addresses, Port mapping, etc
- Acquire path/route information (using Dijkstra becomes feasible to limit search to shortest paths and only one segment of fat tree topology) between the hosts (Host 1 to Host 2 i.e.). Load balancing has to be performed.
- Find total link cost for all these paths between Host 1 and Host 2. OpenDaylight gives only transmitted data. For that reason subsequent REST requests are made to compute this, which adds latency to the application (when using OpenDaylight).
- The flows are created depending on the minimum transmission cost of the links for each given time.
- Based on the cost, the best path is decided and static flows are pushed into each switch in the current best path. Information such as In-Port, Out-Port, Source IP, Destination IP, Source MAC, Destination MAC is used to feed the flows.
- The program continues to update this information every minute, which is making it dynamic.

### 6.2. Running The Program (OpenDaylight)

- Download the distribution package from <https://www.opendaylight.org/downloads>. In this case we use a newer version OpenDaylight (Beryllium SR1). Next unzip the folder.
- In Terminal, change directory to the distribution folder and run `./bin/karaf`.

Install the following features:

```
feature:install odl-restconf-all
```



```
feature:install odl-flow-model odl-flow-services
```

API explorer is accesible at: <http://localhost:8181/apidoc/explorer/index.html>

To list all available optional features, run the command:

```
opendaylight-user@root> feature:list
```

To list all installed features, run the command:

```
opendaylight-user@root> feature:list --installed
```

Navigate to <http://127.0.0.1:8181/index.html#/login> to access the DLUX UI. The default username and password are both "admin".

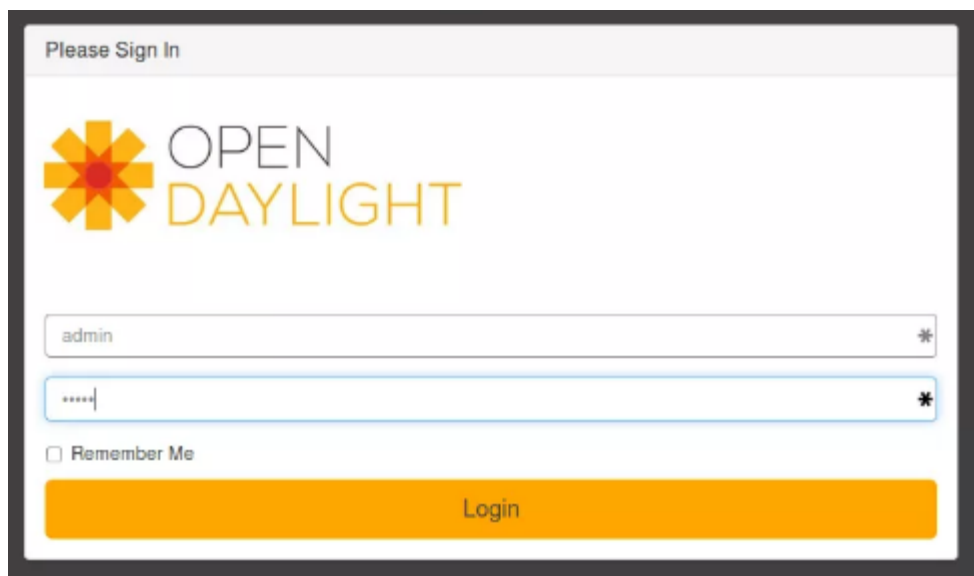


FIGURE 35: OpenDaylight Dlux UI login

- Run Mininet

```
sudo mn --custom topology.py --topo mytopo -controller=remote,ip=127.0.0.1,port=6633
```

- Perform pingall a couple of times till no packet loss occurs.

We check the controller: <http://127.0.0.1:8181/index.html#/topology>

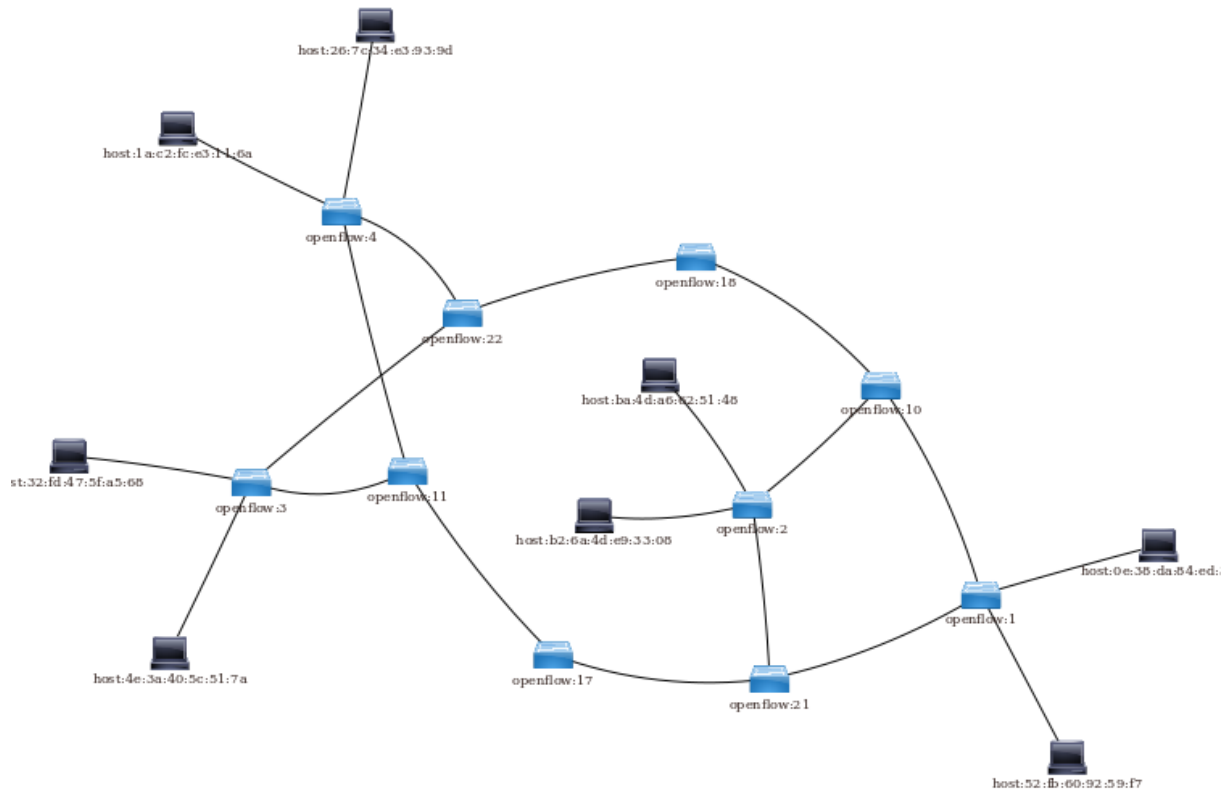


FIGURE 36: Topology from OpenDaylight Dlux UI

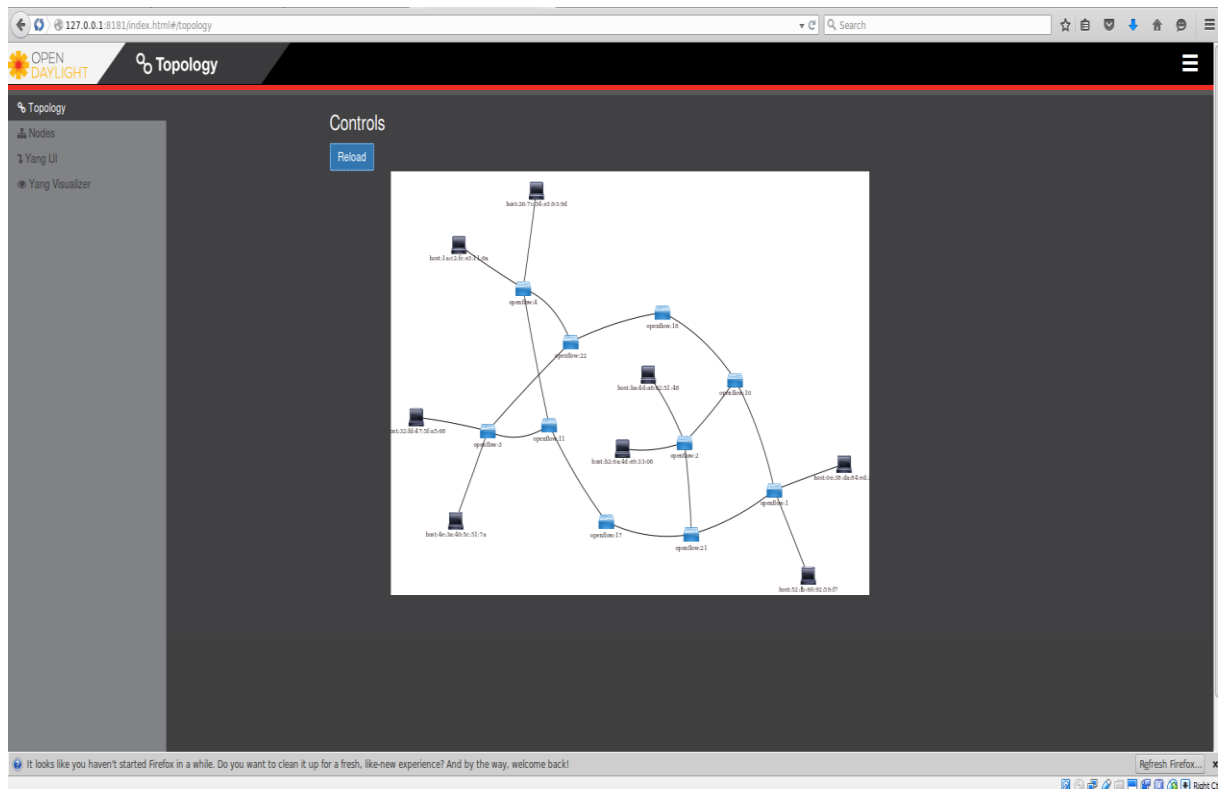


FIGURE 37: Topology inside OpenDaylight Dlux UI

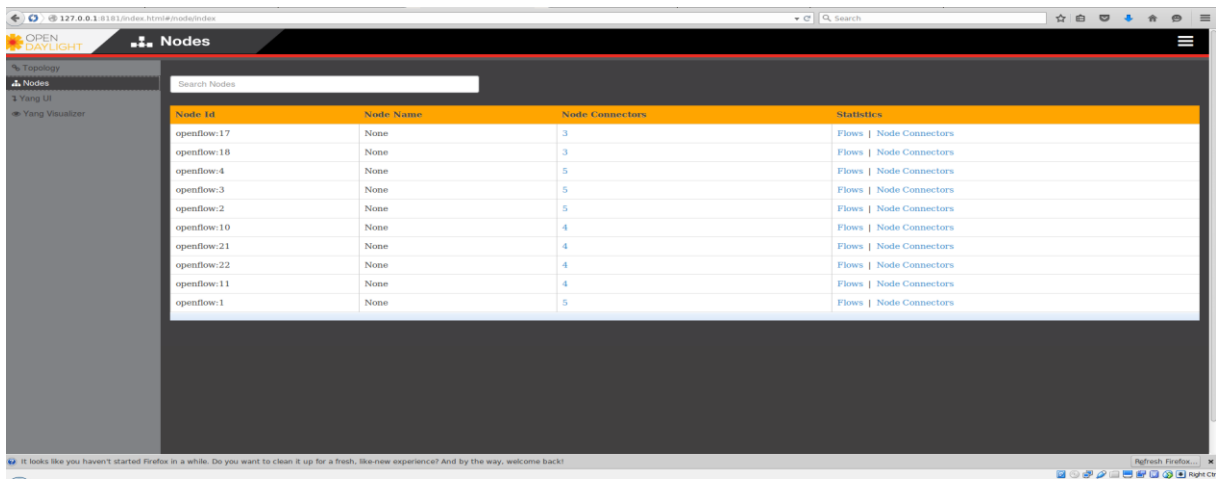


FIGURE 38: Nodes OpenDaylight Dlux UI

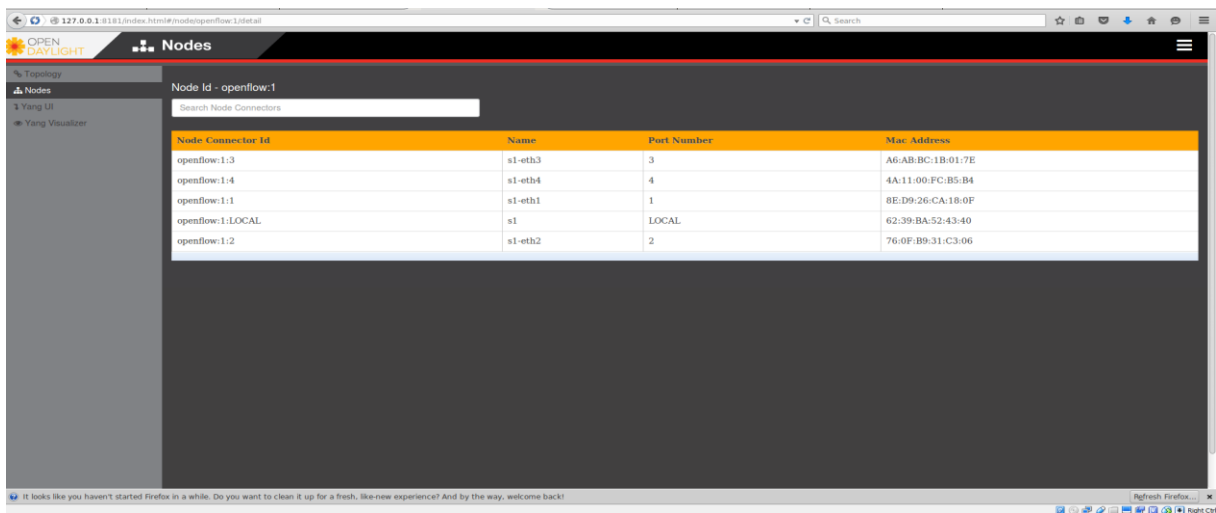


FIGURE 39: Node Connector (1) OpenDaylight Dlux UI

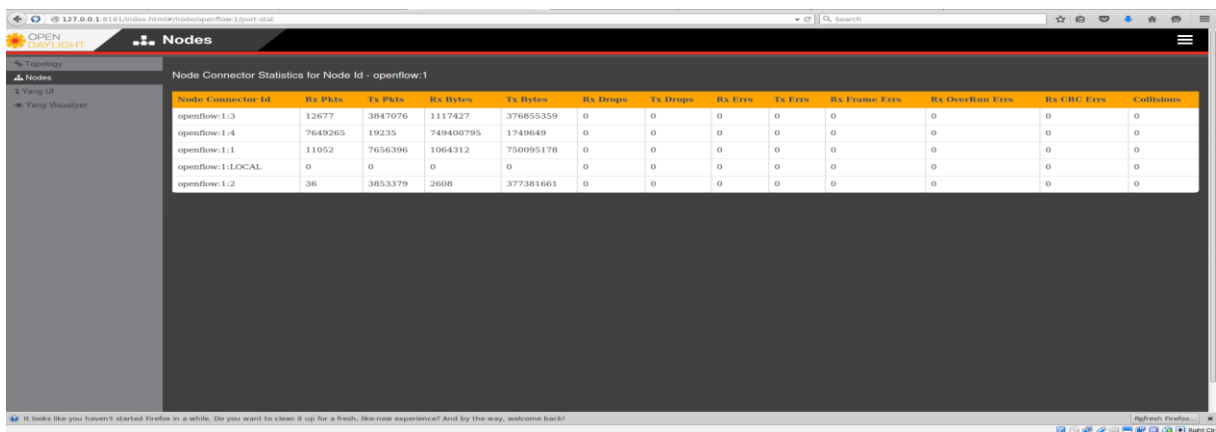


FIGURE 40: Node Connector (2) OpenDaylight Dlux UI

### 6.3. Running The Program (Floodlight)

Download Floodlight from here: <http://www.projectfloodlight.org/download/>

V1.2 is chosen for our implementation.

- How to Build

Installing Floodlight from Scratch

```
$ sudo rm -rf /var/lib/floodlight
```

```
$ sudo mkdir /var/lib/floodlight
```

```
$ sudo chmod 777 /var/lib/floodlight
```

- Running Floodlight in the Terminal

Make sure that java is in your path, and after this we can directly run the floodlight.jar file, which has been produced by ant from within the floodlight directory:

```
$ java -jar target/floodlight.jar
```

Floodlight will start running and print log and debug output to our console. Also we can save logs redirecting it to a file.

Open a browser a type: <http://127.0.0.1:8080/topology>

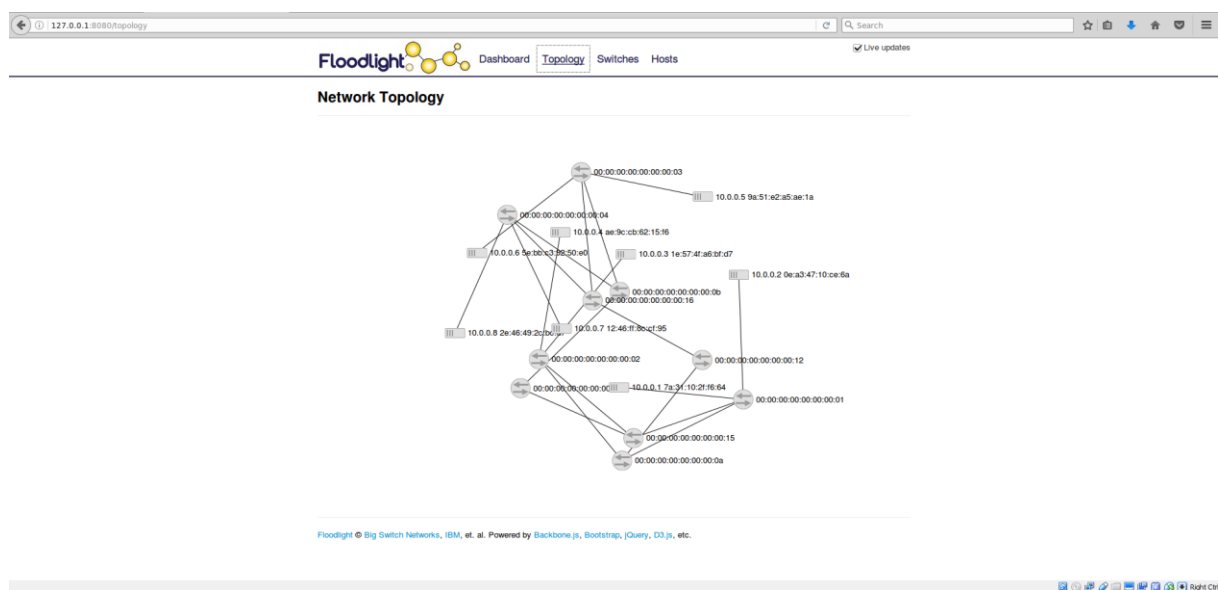


FIGURE 41: Topology inside Floodlight UI

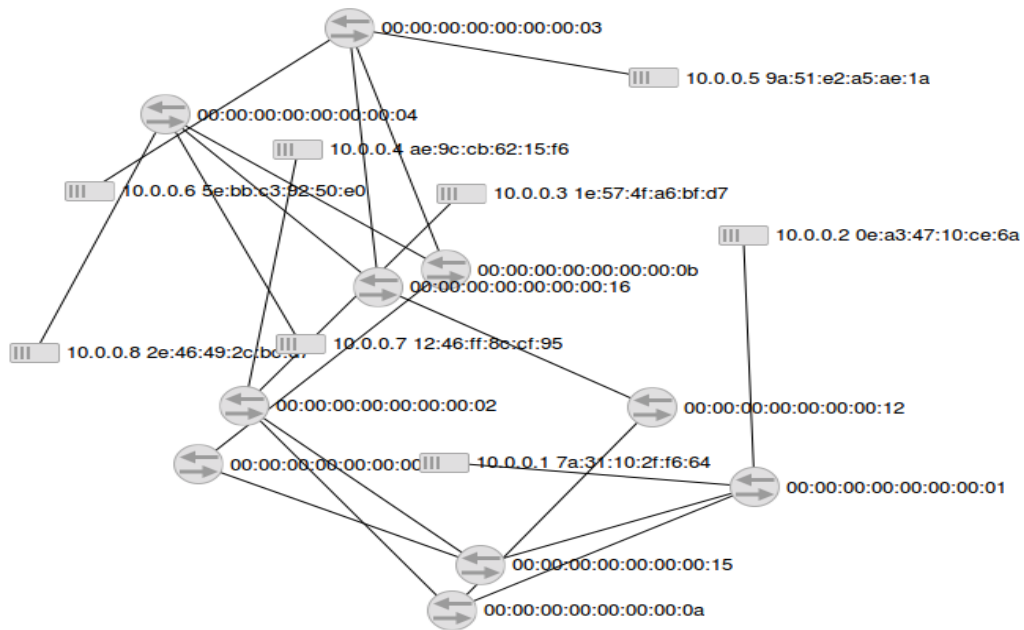


FIGURE 42: Topology from Floodlight UI

- Run the fat tree topology, topology.py using Mininet

`sudo mn --custom topology.py --topo mytopo --controller=remote,ip=127.0.0.1,port=6653`

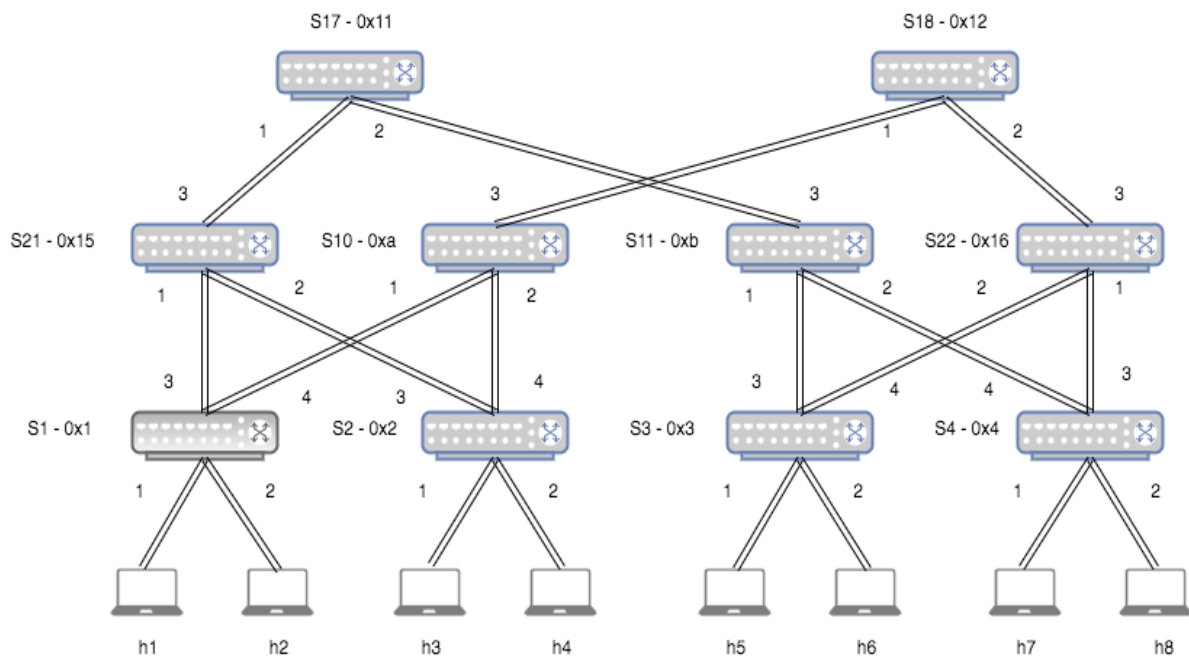


FIGURE 43: Fat tree topology – topology.py

- In order to open two different consoles of h1, type the following command in Mininet:

```
xterm h1 h1
```

- In the first console of h1 type: ping 10.0.0.3
- And in the second console of h1 type: ping 10.0.0.4
- We now have to use Wireshark, which will help us to check the traffic. Open another terminal and type: sudo wireshark.
- In wireshark, go to Capture->Interfaces and select s1-eth4 and then start the capture.
- In filters section in wireshark type ip.addr==10.0.0.3 and check if we are receiving packets for h1 -> h3. Do same thing for h1->h4. Once we see packets, we find out that this is the best path.
- But, in order to confirm it is, repeat the above two steps for s1-eth3 and you will find that no packets are transmitted to this port. The only packets it will receive will be broadcast and multicast. Ignore them.
- Now in the other console of xterm of h1, stop pinging h4. Our goal is to create congestion on the best path of h1->h3, h1->h4 and vice versa (from h1 ping h3).
- Go to your Terminal and open a new tab and run the **odl.py** script (for OpenDaylight) and **loadbalancer.py** script (for Floodlight).
- Provide input arguments such as host 1, host 2 and host 2's neighbor in integer format. For example select-type 1 (as host 1), 4 (as host 2) and 3 (as host 2's neighbor). Looking at the topology above and we will find that these hosts are nothing but h1, h4 and h3 respectively.

```
ubuntu@sdnhubvm:~[15:01]$ cd Downloads/
ubuntu@sdnhubvm:~/Downloads[15:01]$ cd sdn
\bash: cd: sdn: No such file or directory
ubuntu@sdnhubvm:~/Downloads[15:05]$ cd sdn-loadbalancing-master
ubuntu@sdnhubvm:~/Downloads/sdn-loadbalancing-master[15:05]$ ls
assets/ Floodlight.sh loadbalancer.py* odl.py* README.md topology.py*
ubuntu@sdnhubvm:~/Downloads/sdn-loadbalancing-master[15:05]$ ./odl.py
Enter Host 1
1

Enter Host 2
3

Enter Host 3 (H2's Neighbour)
4
```

FIGURE 44: Provide input arguments

- The loadbalancer.py and odl.py perform REST requests, so initially the link costs will be 0. Because statistics need to be enabled we have to re-run the script few times, which may range from 1-10 times. After enabling statistics, it takes some time to gather information. When it starts updating the transmission rates, we will get the best path and the flows for best path will be statically pushed to all the switches in the new best route.

```

Device IP & MAC
{'10.0.0.8': '1a:c2:fc:e3:11:6a', '10.0.0.5': '4e:3a:40:5c:51:7a', '10.0.0.4': '
ba:4d:a6:62:51:48', '10.0.0.7': '26:7c:34:e3:93:9d', '10.0.0.6': '32:fd:47:5f:a5
:68', '10.0.0.1': '52:fb:60:92:59:f7', '10.0.0.3': 'b2:6a:4d:e9:33:08', '10.0.0.
2': '0e:38:da:84:ed:35'}

Switch:Device Mapping
{'10.0.0.8': 'openflow:4', '10.0.0.5': 'openflow:3', '10.0.0.4': 'openflow:2', '
10.0.0.7': 'openflow:4', '10.0.0.6': 'openflow:3', '10.0.0.1': 'openflow:1', '10
.0.0.3': 'openflow:2', '10.0.0.2': 'openflow:1'}

Host:Port Mapping To Switch
{'10.0.0.8': '2', '10.0.0.5': '1', '10.0.0.4': '2', '10.0.0.7': '1', '10.0.0.6':
'2', '10.0.0.1': '1', '10.0.0.3': '1', '10.0.0.2': '2'}

Switch:Switch Port:Port Mapping
{'11::17': '3::2', '10::18': '3::1', '18::10': '1::3', '4::22': '3::1', '22::4':
'1::3', '22::3': '2::4', '1::10': '4::1', '2::21': '3::2', '22::18': '3::2', '1
7::21': '1::3', '18::22': '2::3', '4::11': '4::2', '2::10': '4::2', '3::22': '4:
:2', '3::11': '3::1', '1::21': '3::1', '17::11': '2::3', '21::17': '3::1', '21::
1': '1::3', '21::2': '2::3', '11::4': '2::4', '11::3': '1::3', '10::1': '1::4',
'10::2': '2::4'}

All Paths
[2, 10, 1]
[2, 21, 1]

Cost Computation....

Cost Computation....

Cost Computation....

Cost Computation....

Final Link Cost
{'2::10::1': 2, '2::21::1': 0}

Shortest Path: 2::21::1

```

FIGURE 45: Cost computation for best path

- To check the flows, perform a REST GET request to

OpenDaylight:

<http://localhost:8181/restconf/operational/opendaylight-inventory:nodes/node/openflow>

Floodlight:

<http://127.0.0.1:8080/wm/core/switch/all/flow/json>

```

[{"version": "08_13", "cookie": "08", "tableid": "080", "packetCount": "327", "byteCount": "22930", "durationSeconds": "1192", "durationMicroseconds": "215000000", "priority": "0", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "0", "match": {}, "instructions": [{"instruction_apply_actions": [{"actions": "output-controller"}]}]}, {"version": "08_13", "cookie": "458359840453774", "tableid": "080", "packetCount": "31", "byteCount": "3038", "durationSeconds": "524", "durationMicroseconds": "799000000", "priority": "32768", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "1", "match": {"in_port": "2", "eth_dst": "7a:31:10:2f:16:64", "eth_src": "ae:8c:62:15:16", "eth_type": "0x0800", "ip4_src": "10.0.0.4", "ip4_dst": "10.0.0.3"}, "instructions": [{"instruction_apply_actions": [{"actions": "output-1"}]}]}, {"version": "08_13", "cookie": "458359840453774", "tableid": "080", "packetCount": "31", "byteCount": "3038", "durationSeconds": "524", "durationMicroseconds": "799000000", "priority": "32768", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "1", "match": {"in_port": "1", "eth_dst": "ae:8c:62:15:16", "eth_src": "7a:31:10:2f:16:64", "eth_type": "0x0800", "ip4_src": "10.0.0.1", "ip4_dst": "10.0.0.4"}, "instructions": [{"instruction_apply_actions": [{"actions": "output-2"}]}]}, {"version": "08_13", "cookie": "08", "tableid": "080", "packetCount": "422", "byteCount": "20915", "durationSeconds": "1192", "durationMicroseconds": "180000000", "priority": "0", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "0", "match": {}, "instructions": [{"instruction_apply_actions": [{"actions": "output-controller"}]}]}, {"version": "08_13", "cookie": "08", "tableid": "080", "packetCount": "395", "byteCount": "27850", "durationSeconds": "1192", "durationMicroseconds": "219000000", "priority": "0", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "0", "match": {}, "instructions": [{"instruction_apply_actions": [{"actions": "output-controller"}]}]}, {"version": "08_13", "cookie": "080719925474992", "tableid": "080", "packetCount": "922", "byteCount": "90356", "durationSeconds": "921", "durationMicroseconds": "648000000", "priority": "1", "idleTimeoutSec": "5", "hardTimeoutSec": "0", "flags": "0", "match": {"in_port": "4", "eth_dst": "1e:57:4f:a6:b1:d7", "eth_src": "7a:31:10:2f:16:64", "eth_type": "0x0800", "ip4_src": "10.0.0.1", "ip4_dst": "10.0.0.3"}, "instructions": [{"instruction_apply_actions": [{"actions": "output-1"}]}]}, {"version": "08_13", "cookie": "980719925474992", "tableid": "080", "packetCount": "921", "byteCount": "90356", "durationSeconds": "921", "durationMicroseconds": "648000000", "priority": "1", "idleTimeoutSec": "5", "hardTimeoutSec": "0", "flags": "0", "match": {"in_port": "1", "eth_dst": "7a:31:10:2f:16:64", "eth_src": "1e:57:4f:a6:b1:d7", "eth_type": "0x0800", "ip4_src": "10.0.0.1", "ip4_dst": "10.0.0.3"}, "instructions": [{"instruction_apply_actions": [{"actions": "output-1"}]}]}, {"version": "08_13", "cookie": "458359840453774", "tableid": "080", "packetCount": "31", "byteCount": "3038", "durationSeconds": "524", "durationMicroseconds": "799000000", "priority": "32768", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "1", "match": {"in_port": "2", "eth_dst": "7a:31:10:2f:16:64", "eth_src": "ae:8c:62:15:16", "eth_type": "0x0800", "ip4_src": "10.0.0.4", "ip4_dst": "10.0.0.3"}, "instructions": [{"instruction_apply_actions": [{"actions": "output-1"}]}]}, {"version": "08_13", "cookie": "458359840453774", "tableid": "080", "packetCount": "31", "byteCount": "3038", "durationSeconds": "524", "durationMicroseconds": "799000000", "priority": "32768", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "1", "match": {"in_port": "1", "eth_dst": "ae:8c:62:15:16", "eth_src": "7a:31:10:2f:16:64", "eth_type": "0x0800", "ip4_src": "10.0.0.1", "ip4_dst": "10.0.0.4"}, "instructions": [{"instruction_apply_actions": [{"actions": "output-2"}]}]}, {"version": "08_13", "cookie": "08", "tableid": "080", "packetCount": "382", "byteCount": "22784", "durationSeconds": "1192", "durationMicroseconds": "230000000", "priority": "0", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "0", "match": {}, "instructions": [{"instruction_apply_actions": [{"actions": "output-controller"}]}]}, {"version": "08_13", "cookie": "08", "tableid": "080", "packetCount": "382", "byteCount": "22784", "durationSeconds": "1192", "durationMicroseconds": "230000000", "priority": "0", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "0", "match": {}, "instructions": [{"instruction_apply_actions": [{"actions": "output-controller"}]}]}, {"version": "08_13", "cookie": "08", "tableid": "080", "packetCount": "313", "byteCount": "22086", "durationSeconds": "1192", "durationMicroseconds": "180000000", "priority": "0", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "0", "match": {}, "instructions": [{"instruction_apply_actions": [{"actions": "output-controller"}]}]}, {"version": "08_13", "cookie": "08", "tableid": "080", "packetCount": "278", "byteCount": "18664", "durationSeconds": "1192", "durationMicroseconds": "193000000", "priority": "0", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "0", "match": {}, "instructions": [{"instruction_apply_actions": [{"actions": "output-controller"}]}]}, {"version": "08_13", "cookie": "980719925474992", "tableid": "080", "packetCount": "921", "byteCount": "90356", "durationSeconds": "921", "durationMicroseconds": "648000000", "priority": "1", "idleTimeoutSec": "5", "hardTimeoutSec": "0", "flags": "0", "match": {"in_port": "1", "eth_dst": "1e:57:4f:a6:b1:d7", "eth_src": "7a:31:10:2f:16:64", "eth_type": "0x0800", "ip4_src": "10.0.0.1", "ip4_dst": "10.0.0.3"}, "instructions": [{"instruction_apply_actions": [{"actions": "output-1"}]}]}, {"version": "08_13", "cookie": "980719925474992", "tableid": "080", "packetCount": "922", "byteCount": "90356", "durationSeconds": "921", "durationMicroseconds": "650000000", "priority": "1", "idleTimeoutSec": "5", "hardTimeoutSec": "0", "flags": "0", "match": {"in_port": "4", "eth_dst": "7a:31:10:2f:16:64", "eth_src": "1e:57:4f:a6:b1:d7", "eth_type": "0x0800", "ip4_src": "10.0.0.1", "ip4_dst": "10.0.0.3"}, "instructions": [{"instruction_apply_actions": [{"actions": "output-1"}]}]}, {"version": "08_13", "cookie": "458359840453774", "tableid": "080", "packetCount": "31", "byteCount": "3038", "durationSeconds": "524", "durationMicroseconds": "799000000", "priority": "32768", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "1", "match": {"in_port": "2", "eth_dst": "7a:31:10:2f:16:64", "eth_src": "ae:8c:62:15:16", "eth_type": "0x0800", "ip4_src": "10.0.0.4", "ip4_dst": "10.0.0.3"}, "instructions": [{"instruction_apply_actions": [{"actions": "output-1"}]}]}, {"version": "08_13", "cookie": "458359840453774", "tableid": "080", "packetCount": "31", "byteCount": "3038", "durationSeconds": "524", "durationMicroseconds": "799000000", "priority": "32768", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "1", "match": {"in_port": "1", "eth_dst": "ae:8c:62:15:16", "eth_src": "7a:31:10:2f:16:64", "eth_type": "0x0800", "ip4_src": "10.0.0.1", "ip4_dst": "10.0.0.4"}, "instructions": [{"instruction_apply_actions": [{"actions": "output-2"}]}]}, {"version": "08_13", "cookie": "08", "tableid": "080", "packetCount": "360", "byteCount": "25274", "durationSeconds": "1192", "durationMicroseconds": "190000000", "priority": "0", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "0", "match": {}, "instructions": [{"instruction_apply_actions": [{"actions": "output-controller"}]}]}, {"version": "08_13", "cookie": "08", "tableid": "080", "packetCount": "360", "byteCount": "25274", "durationSeconds": "1192", "durationMicroseconds": "190000000", "priority": "0", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "0", "match": {}, "instructions": [{"instruction_apply_actions": [{"actions": "output-controller"}]}]}, {"version": "08_13", "cookie": "980719925474992", "tableid": "080", "packetCount": "921", "byteCount": "90356", "durationSeconds": "921", "durationMicroseconds": "650000000", "priority": "1", "idleTimeoutSec": "5", "hardTimeoutSec": "0", "flags": "0", "match": {"in_port": "1", "eth_dst": "1e:57:4f:a6:b1:d7", "eth_src": "7a:31:10:2f:16:64", "eth_type": "0x0800", "ip4_src": "10.0.0.1", "ip4_dst": "10.0.0.3"}, "instructions": [{"instruction_apply_actions": [{"actions": "output-1"}]}]}, {"version": "08_13", "cookie": "980719925474992", "tableid": "080", "packetCount": "922", "byteCount": "90356", "durationSeconds": "921", "durationMicroseconds": "650000000", "priority": "1", "idleTimeoutSec": "5", "hardTimeoutSec": "0", "flags": "0", "match": {"in_port": "4", "eth_dst": "7a:31:10:2f:16:64", "eth_src": "1e:57:4f:a6:b1:d7", "eth_type": "0x0800", "ip4_src": "10.0.0.1", "ip4_dst": "10.0.0.3"}, "instructions": [{"instruction_apply_actions": [{"actions": "output-1"}]}]}, {"version": "08_13", "cookie": "08", "tableid": "080", "packetCount": "392", "byteCount": "27968", "durationSeconds": "1192", "durationMicroseconds": "230000000", "priority": "0", "idleTimeoutSec": "0", "hardTimeoutSec": "0", "flags": "0", "match": {}, "instructions": [{"instruction_apply_actions": [{"actions": "output-controller"}]}]}]}]}

```

FIGURE 46: Floodlight Rest Get

- Now on the other console of h1 type ping 10.0.0.4
- Go again to wireshark and monitor interface s1-eth4 with the filter ip.addr = 10.0.0.x where x is 3 and 4. You will find 10.0.0.3 packets but no 10.0.0.4 packets
- Stop the above capture and now make the capture with s1-eth3, s21-eth1, s21-eth2, s2-eth3 and type the filter ip.addr = 10.0.0.x where x is 3 and 4. We will successfully find 10.0.0.4 packets but no 10.0.0.3 packets.

## 6.4. iPerf

iPerf is a tool, which helps as to measure actively the maximum achievable bandwidth on IP networks. It supports tuning of many different parameters related to timing, buffers and protocols (UDP, TCP, SCTP with IPv4 and IPv6). It reports the bandwidth, loss, and other parameters for each test.

iPerf was originally developed by NLNR/DAST and it is released under a three-clause BSD license.

Download iPerf from the following link: <https://iperf.fr/iperf-download.php>

In our case the chosen version is iPerf 2.0.5 - DEB package.

iPerf installation:

```
sudo dpkg -i iPerf 2.0.5.deb
```

The iPerf client connects by default to the iPerf server on the TCP port 5001 and the bandwidth displayed by iPerf is the bandwidth from the client to the server.

Type the following into the server's command line window:

```
xterm h4
```

```
iperf -s
```



The Iperf server connects back to the client allowing the bi-directional bandwidth measurement. By default, only the bandwidth from the client to the server can be measured.

Type the following into the client's command line window:

```
xterm h1
```

```
iperf -c 10.0.0.4
```

## 6.5. Results We Achieved

iPerf enables us to check the amount of bytes transferred and the rate i.e. bandwidth. Below are 4 tables. It shows how the average transferred data increases after load balancing.

Transfer (Gbytes) - BLB	B/W(Gbits) - BLB	Transfer (Gbytes) - ALB	B/W(Gbits) - ALB
15.7	13.5	38.2	32.8
21.9	18.8	27.6	32.3
24.6	21.1	40.5	34.8
22.3	19.1	40.8	35.1
39.8	34.2	16.5	14.2
<b>Average = 24.86</b>	<b>Average = 21.34</b>	<b>Average = 32.72</b>	<b>Average = 29.84</b>

iPerf H1 to H3 Before Load Balancing (BLB) and After Load Balancing (ALB)

Transfer (Gbytes) - BLB	B/W(Gbits) - BLB	Transfer (Gbytes) - ALB	B/W(Gbits) - ALB
18.5	15.9	37.2	31.9
18.1	15.5	39.9	34.3
23.8	20.2	40.2	34.5
17.8	15.3	40.3	34.6
38.4	32.9	18.4	15.8
<b>Average = 23.32</b>	<b>Average = 19.96</b>	<b>Average = 35.2</b>	<b>Average = 30.22</b>

iPerf H1 to H4 Before Load Balancing (BLB) and After Load Balancing (ALB)

Ping test has been performed on 30 packets. A total of 5 such ping tests have been performed before

and after load balancing. And the results after load balancing show the ping latency to decrease.

<b>Min</b>	<b>Avg</b>	<b>Max</b>	<b>Mdev</b>
0.049	0.245	4.407	0.807
0.050	0.155	4.523	0.575
0.041	0.068	0.112	0.019
0.041	0.086	0.416	0.066
0.018	0.231	4.093	0.759
<b>Avg: 0.0398</b>	<b>Avg: 0.157</b>	<b>Avg: 2.7102</b>	<b>Avg: 0.4452</b>

Ping from H1 to H4 Before Load Balancing

<b>Min</b>	<b>Avg</b>	<b>Max</b>	<b>Mdev</b>
0.039	0.075	0.407	0.068
0.048	0.155	0.471	0.091
0.040	0.072	0.064	0.199
0.038	0.074	0.283	0.039
0.048	0.099	0.509	0.108
<b>Avg: 0.0426</b>	<b>Avg: 0.0796</b>	<b>Avg: 0.3468</b>	<b>Avg: 0.101</b>

Ping from H1 to H4 After Load Balancing

## Bibliography

- [1] Open Networking Foundation, "Software-defined networking: the new norm for networks," ONF Paper, Open Networking Foundation, Apr. 2012.

Available:

<https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf>

- [2] OpenDaylight Application Developer's tutorial

Available: <http://sdnhub.org/tutorials/opendaylight/>

- [3] Floodlight Documentation

Available:

<https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Floodlight+Documentation#suk=>

- [4] Floodlight Controller - Installation Guide

Available:

<https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Installation+Guide#suk=>

- [5] OpenDaylight

Available: [https://wiki.opendaylight.org/view/Main\\_Page](https://wiki.opendaylight.org/view/Main_Page)

- [6] The OpenDaylight platform

Available: <https://www.opendaylight.org/>

- [7] OpenFlow a valid technology

Available: <http://docplayer.net/4847096-Sdn-openflow-outline-performance-u-winterschool-zurich-www-openflow-org-sdn-to-openflow-openflow-a-valid-technology.html>

- [8] M. Gorius, G. Petrovic, and T. Herfet, "Predictably reliable real-time transport over large bandwidth-delay product networks," in NEM Summit - Towards Future Media Internet, 2011

Available: <http://www.cercs.gatech.edu/tech-reports/tr2003/git-cercs-03-02.pdf>

- [9] "Equal Cost Multipath Routing in IP Networks" Ari Lappetelainen, Aalto University School of Science and Technology, 17.3.2011.  
Available: <http://lib.tkk.fi/Dipl/2011/urn100416.pdf>
- [10] "Openflow: enabling innovation in campus networks" N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, Mar. 2008.  
Available: <http://ccr.sigcomm.org/online/files/p69-v38n2n-mckeown.pdf>
- [11] Mininet  
Available: <http://mininet.org>
- [12] Software Defined Networking  
Available: [https://en.wikipedia.org/wiki/Software-defined\\_networking](https://en.wikipedia.org/wiki/Software-defined_networking)
- [13] Network Functions Virtualisation – Introductory White Paper  
Available: [https://portal.etsi.org/nfv/nfv\\_white\\_paper.pdf](https://portal.etsi.org/nfv/nfv_white_paper.pdf)
- [14] "The Extended Dijkstra's-based Load Balancing for OpenFlow Network", Widhi Yahya1, Achmad Basuki2, Jehn-Ruey Jiang, International Journal of Electrical and Computer Engineering (IJECE) Vol. 5, No. 2, April 2015, pp. 289~296  
Available: <http://staff.csie.ncu.edu.tw/jrjiang/publication/7113-16315-1-PB.pdf>

## Appendices

### Codes

#### Code for mininet

##### topology.py

```
#!/usr/bin/python
```

```
from mininet.node import CPULimitedHost, Host, Node
```

```
from mininet.node import OVSKernelSwitch
```

```
from mininet.topo import Topo
```

```
class fatTreeTopo(Topo):
```

```
    "Fat Tree Topology"
```

```
    def __init__(self):
```

```
        "Create Fat tree Topology"
```

```
        Topo.__init__(self)
```

```
        #Add hosts
```

```
        h7 = self.addHost('h7', cls=Host, ip='10.0.0.7', defaultRoute=None)
```

```
        h8 = self.addHost('h8', cls=Host, ip='10.0.0.8', defaultRoute=None)
```

```
        h1 = self.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)
```

```
        h2 = self.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)
```

```
        h4 = self.addHost('h4', cls=Host, ip='10.0.0.4', defaultRoute=None)
```

```
h3 = self.addHost('h3', cls=Host, ip='10.0.0.3', defaultRoute=None)
```

```
h5 = self.addHost('h5', cls=Host, ip='10.0.0.5', defaultRoute=None)
```

```
h6 = self.addHost('h6', cls=Host, ip='10.0.0.6', defaultRoute=None)
```

```
#Add switches
```

```
s10 = self.addSwitch('s10', cls=OVSKernelSwitch)
```

```
s3 = self.addSwitch('s3', cls=OVSKernelSwitch)
```

```
s17 = self.addSwitch('s17', cls=OVSKernelSwitch)
```

```
s4 = self.addSwitch('s4', cls=OVSKernelSwitch)
```

```
s18 = self.addSwitch('s18', cls=OVSKernelSwitch)
```

```
s1 = self.addSwitch('s1', cls=OVSKernelSwitch)
```

```
s11 = self.addSwitch('s11', cls=OVSKernelSwitch)
```

```
s21 = self.addSwitch('s21', cls=OVSKernelSwitch)
```

```
s22 = self.addSwitch('s22', cls=OVSKernelSwitch)
```

```
s2 = self.addSwitch('s2', cls=OVSKernelSwitch)
```

```
#Add links
```

```
self.addLink(h1, s1)
```

```
self.addLink(h2, s1)
```

```
self.addLink(h3, s2)
```

```
self.addLink(h4, s2)
```

```
self.addLink(h5, s3)
```

```
self.addLink(h6, s3)
```

```
self.addLink(h7, s4)
```

```
self.addLink(h8, s4)
```

```
self.addLink(s1, s21)
```

```
self.addLink(s21, s2)
```

```
self.addLink(s1, s10)
self.addLink(s2, s10)
self.addLink(s3, s11)
self.addLink(s4, s22)
self.addLink(s11, s4)
self.addLink(s3, s22)
self.addLink(s21, s17)
self.addLink(s11, s17)
self.addLink(s10, s18)
self.addLink(s22, s18)
```

```
topos = { 'mytopo': (lambda: fatTreeTopo() ) }
```

## **Code for OpenDaylight**

### **odl.py**

```
#!/usr/bin/env python

import requests

from requests.auth import HTTPBasicAuth

import json

import unicodedata

from subprocess import Popen, PIPE

import time

import networkx as nx
```

```

from sys import exit

# Method To Get REST Data In JSON Format

def getResponse(url,choice):

    response = requests.get(url, auth=HTTPBasicAuth('admin', 'admin'))

    if(response.ok):

        jData = json.loads(response.content)

        if(choice=="topology"):

            topologyInformation(jData)

        elif(choice=="statistics"):

            getStats(jData)

    else:

        response.raise_for_status()

def topologyInformation(data):

    global switch

    global deviceMAC

    global deviceIP

    global hostPorts

    global linkPorts

    global G

    global cost

    for i in data["network-topology"]["topology"]:

        for j in i["node"]:

```



```
# Device MAC and IP
```

```
if "host-tracker-service:addresses" in j:
```

```
    for k in j["host-tracker-service:addresses"]:
```

```
        ip = k["ip"].encode('ascii','ignore')
```

```
        mac = k["mac"].encode('ascii','ignore')
```

```
        deviceMAC[ip] = mac
```

```
        deviceIP[mac] = ip
```

```
# Device Switch Connection and Port
```

```
if "host-tracker-service:attachment-points" in j:
```

```
    for k in j["host-tracker-service:attachment-points"]:
```

```
        mac = k["corresponding-tp"].encode('ascii','ignore')
```

```
        mac = mac.split(":",1)[1]
```

```
        ip = deviceIP[mac]
```

```
        temp = k["tp-id"].encode('ascii','ignore')
```

```
        switchID = temp.split(":")
```

```
        port = switchID[2]
```

```
        hostPorts[ip] = port
```

```
        switchID = switchID[0] + ":" + switchID[1]
```

```
        switch[ip] = switchID
```

```
# Link Port Mapping
```

```
for i in data["network-topology"]["topology"]:
```

```
    for j in i["link"]:
```

```

        if "host" not in j['link-id']:

            src = j["link-id"].encode('ascii','ignore').split(":")

            srcPort = src[2]

            dst = j["destination"]["dest-
tp"].encode('ascii','ignore').split(":")

            dstPort = dst[2]

            srcToDst = src[1] + "::" + dst[1]

            linkPorts[srcToDst] = srcPort + "::" + dstPort

            G.add_edge((int)(src[1]),(int)(dst[1]))

def getStats(data):

    print "\nCost Computation....\n"

    global cost

    txRate = 0

    for i in data["node-connector"]:

        tx = int(i["opendaylight-port-statistics:flow-capable-node-connector-
statistics"]["packets"]["transmitted"])

        rx = int(i["opendaylight-port-statistics:flow-capable-node-connector-
statistics"]["packets"]["received"])

        txRate = tx + rx

        #print txRate

    time.sleep(2)

    response = requests.get(stats, auth=HTTPBasicAuth('admin', 'admin'))

    tempJSON = ""

    if(response.ok):

        tempJSON = json.loads(response.content)

```

```

for i in tempJSON["node-connector"]:

    tx    =    int(i["opendaylight-port-statistics:flow-capable-node-connector-
statistics"]["packets"]["transmitted"])

    rx    =    int(i["opendaylight-port-statistics:flow-capable-node-connector-
statistics"]["packets"]["received"])

    cost = cost + tx + rx - txRate

#cost = cost + txRate

#print cost

def systemCommand(cmd):

    terminalProcess = Popen(cmd, stdout=PIPE, stderr=PIPE, shell=True)

    terminalOutput, stderr = terminalProcess.communicate()

    print "\n*** Flow Pushed\n"

def pushFlowRules(bestPath):

    bestPath = bestPath.split("::")

    for currentNode in range(0, len(bestPath)-1):

        if (currentNode==0):

            inport = hostPorts[h2]

            srcNode = bestPath[currentNode]

            dstNode = bestPath[currentNode+1]

            outport = linkPorts[srcNode + "::" + dstNode]

            outport = outport[0]

        else:

            prevNode = bestPath[currentNode-1]

```

```

#print prevNode

srcNode = bestPath[currentNode]

#print srcNode

dstNode = bestPath[currentNode+1]

inport = linkPorts[prevNode + "::" + srcNode]

inport = inport.split("::")[1]

outport = linkPorts[srcNode + "::" + dstNode]

outport = outport.split("::")[0]

```

```

xmlSrcToDst = '\<?xml version="1.0" encoding="UTF-8"
standalone="no"?><flow
xmlns="urn:opendaylight:flow:inventory"><priority>32767</priority><flow-name>Load
Balance 1</flow-name><match><in-port>' + str(inport) + '</in-port><ipv4-
destination>10.0.0.1/32</ipv4-destination><ipv4-source>10.0.0.4/32</ipv4-
source><ethernet-match><ethernet-type><type>2048</type></ethernet-type></ethernet-
match></match><id>1</id><table_id>0</table_id><instructions><instruction><order>0</or-
der><apply-actions><action><order>0</order><output-action><output-node-connector>' +
str(outport) + '</output-node-connector></output-action></action></apply-
actions></instruction></instructions></flow>'

```

```

xmlDstToSrc = '\<?xml version="1.0" encoding="UTF-8"
standalone="no"?><flow
xmlns="urn:opendaylight:flow:inventory"><priority>32767</priority><flow-name>Load
Balance 2</flow-name><match><in-port>' + str(outport) + '</in-port><ipv4-
destination>10.0.0.4/32</ipv4-destination><ipv4-source>10.0.0.1/32</ipv4-
source><ethernet-match><ethernet-type><type>2048</type></ethernet-type></ethernet-
match></match><id>2</id><table_id>0</table_id><instructions><instruction><order>0</or-
der><apply-actions><action><order>0</order><output-action><output-node-connector>' +
str(inport) + '</output-node-connector></output-action></action></apply-
actions></instruction></instructions></flow>'

```

```

flowURL = "http://127.0.0.1:8181/restconf/config/opendaylight-
inventory:nodes/node/openflow:" + bestPath[currentNode] + "/table/0/flow/1"

```

```
command = 'curl --user "admin":"admin" -H "Accept: application/xml" -H
"Content-type: application/xml" -X PUT ' + flowURL + ' -d ' + xmlSrcToDst
```

```
systemCommand(command)
```

```
flowURL = "http://127.0.0.1:8181/restconf/config/opendaylight-
inventory:nodes/node/openflow:" + bestPath[currentNode] + "/table/0/flow/2"
```

```
command = 'curl --user "admin":"admin" -H "Accept: application/xml" -H
"Content-type: application/xml" -X PUT ' + flowURL + ' -d ' + xmlDstToSrc
```

```
systemCommand(command)
```

```
srcNode = bestPath[-1]
```

```
prevNode = bestPath[-2]
```

```
inport = linkPorts[prevNode + "::" + srcNode]
```

```
inport = inport.split("::")[1]
```

```
outport = hostPorts[h1]
```

```
xmlSrcToDst = '\<?xml version="1.0" encoding="UTF-8"
standalone="no"?><flow
xmlns="urn:opendaylight:flow:inventory"><priority>32767</priority><flow-name>Load
Balance 1</flow-name><match><in-port>' + str(inport) + '</in-port><ipv4-
destination>10.0.0.1/32</ipv4-destination><ipv4-source>10.0.0.4/32</ipv4-
source><ethernet-match><ethernet-type><type>2048</type></ethernet-type></ethernet-
match></match><id>1</id><table_id>0</table_id><instructions><instruction><order>0</or-
der><apply-actions><action><order>0</order><output-action><output-node-connector>' +
str(outport) + '</output-node-connector></output-action></action></apply-
actions></instruction></instructions></flow>\'
```

```
xmlDstToSrc = '\<?xml version="1.0" encoding="UTF-8"
standalone="no"?><flow
xmlns="urn:opendaylight:flow:inventory"><priority>32767</priority><flow-name>Load
```

```
Balance 2</flow-name><match><in-port>' + str(outport) + '</in-port><ipv4-destination>10.0.0.4/32</ipv4-destination><ipv4-source>10.0.0.1/32</ipv4-source><ethernet-match><ethernet-type><type>2048</type></ethernet-type></ethernet-match></match><id>2</id><table_id>0</table_id><instructions><instruction><order>0</order><apply-actions><action><order>0</order><output-action><output-node-connector>' + str(inport) + '</output-node-connector></output-action></action></apply-actions></instruction></instructions></flow>\\"
```

```
flowURL = "http://127.0.0.1:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:"+ bestPath[-1] + "/table/0/flow/1"
```

```
command = 'curl --user \"admin\":\"admin\" -H \"Accept: application/xml\" -H \"Content-type: application/xml\" -X PUT ' + flowURL + ' -d ' + xmlSrcToDst
```

```
systemCommand(command)
```

```
flowURL = "http://127.0.0.1:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:"+ bestPath[-1] + "/table/0/flow/2"
```

```
command = 'curl --user "admin":"admin" -H "Accept: application/xml" -H "Content-type: application/xml" -X PUT ' + flowURL + ' -d ' + xmlDstToSrc
```

```
systemCommand(command)
```

```
# Main
```

```
# Stores H1 and H2 from user
```

```
global h1,h2,h3
```

```
h1 = ""
```

```
h2 = ""
```

```
print "Enter Host 1"
h1 = int(input())
print "\nEnter Host 2"
h2 = int(input())
print "\nEnter Host 3 (H2's Neighbour)"
h3 = int(input())

h1 = "10.0.0." + str(h1)
h2 = "10.0.0." + str(h2)
h3 = "10.0.0." + str(h3)

flag = True

while flag:

    #Creating Graph
    G = nx.Graph()

    # Stores Info About H3 And H4's Switch
    switch = {}

    # MAC of Hosts i.e. IP:MAC
    deviceMAC = {}

    # IP of Hosts i.e. MAC:IP
    deviceIP = {}
```

```

# Stores Switch Links To H3 and H4's Switch
switchLinks = {}

# Stores Host Switch Ports
hostPorts = {}

# Stores Switch To Switch Path
path = {}

# Stores Link Ports
linkPorts = {}

# Stores Final Link Rates
finalLinkTX = {}

# Store Port Key For Finding Link Rates
portKey = ""

# Statistics
global stats
stats = ""

# Stores Link Cost
global cost
cost = 0

try:
    # Device Info (Switch To Which The Device Is Connected & The MAC Address
    Of Each Device)
    topology = "http://127.0.0.1:8181/restconf/operational/network-
topology:network-topology"
    getResponse(topology,"topology")

```



```

# Print Device:MAC Info

print "\nDevice IP & MAC\n"

print deviceMAC

# Print Switch:Device Mapping

print "\nSwitch:Device Mapping\n"

print switch

# Print Host:Port Mapping

print "\nHost:Port Mapping To Switch\n"

print hostPorts

# Print Switch:Switch Port:Port Mapping

print "\nSwitch:Switch Port:Port Mapping\n"

print linkPorts

# Paths

print "\nAll Paths\n"

#for path in nx.all_simple_paths(G, source=2, target=1):

    #print(path)

    for path in nx.all_shortest_paths(G, source=int(switch[h2].split(":")[1]),
target=int(switch[h1].split(":")[1]), weight=None):

        print path

# Cost Computation

tmp = ""

```

```

        for currentPath in nx.all_shortest_paths(G,
source=int(switch[h2].split(":",1)[1]), target=int(switch[h1].split(":",1)[1]), weight=None):

            for node in range(0,len(currentPath)-1):

                tmp = tmp + str(currentPath[node]) + "::"

                key = str(currentPath[node])+ "::" + str(currentPath[node+1])

                port = linkPorts[key]

                port = port.split(":",1)[0]

                port = int(port)

                stats =
"http://localhost:8181/restconf/operational/opendaylight-
inventory:nodes/node/openflow:"+str(currentPath[node]+)/node-
connector/openflow:"+str(currentPath[node])+":"+str(port)

                getResponse(stats,"statistics")

                tmp = tmp + str(currentPath[len(currentPath)-1])

                tmp = tmp.strip("::")

                finalLinkTX[tmp] = cost

                cost = 0

                tmp = ""

            print "\nFinal Link Cost\n"

            print finalLinkTX

            shortestPath = min(finalLinkTX, key=finalLinkTX.get)

            print "\n\nShortest Path: ",shortestPath

            pushFlowRules(shortestPath)

            time.sleep(60)

except KeyboardInterrupt:

```

```
break
```

```
exit
```

## **Floodlight Code**

### **loadbalancer.py**

```
#!/usr/bin/env python
```

```
import requests
```

```
import json
```

```
import unicodedata
```

```
from subprocess import Popen, PIPE
```

```
import time
```

```
import networkx as nx
```

```
from sys import exit
```

```
# Method To Get REST Data In JSON Format
```

```
def getResponse(url,choice):
```

```
    response = requests.get(url)
```

```
    if(response.ok):
```

```
        jData = json.loads(response.content)
```

```
        if(choice=="deviceInfo"):
```

```
            deviceInformation(jData)
```

```
        elif(choice=="findSwitchLinks"):
```

```
            findSwitchLinks(jData,switch[h2])
```

```
        elif(choice=="linkTX"):
```

```

linkTX(jData,portKey)

else:
    response.raise_for_status()

# Parses JSON Data To Find Switch Connected To H4
def deviceInformation(data):
    global switch
    global deviceMAC
    global hostPorts
    switchDPID = ""
    for i in data:
        if(i['ipv4']):
            ip = i['ipv4'][0].encode('ascii','ignore')
            mac = i['mac'][0].encode('ascii','ignore')
            deviceMAC[ip] = mac
            for j in i['attachmentPoint']:
                for key in j:
                    temp = key.encode('ascii','ignore')
                    if(temp=="switchDPID"):
                        switchDPID = j[key].encode('ascii','ignore')
                        switch[ip] = switchDPID
                    elif(temp=="port"):
                        portNumber = j[key]
                        switchShort = switchDPID.split(":")[7]
                        hostPorts[ip+ "::" + switchShort] =
str(portNumber)

# Finding Switch Links Of Common Switch Of H3, H4

```

```

def findSwitchLinks(data,s):

    global switchLinks

    global linkPorts

    global G

    links=[]

    for i in data:

        src = i['src-switch'].encode('ascii','ignore')

        dst = i['dst-switch'].encode('ascii','ignore')

        srcPort = str(i['src-port'])

        dstPort = str(i['dst-port'])

        srcTemp = src.split(":")[7]

        dstTemp = dst.split(":")[7]

        G.add_edge(int(srcTemp,16), int(dstTemp,16))

        tempSrcToDst = srcTemp + "::" + dstTemp
        tempDstToSrc = dstTemp + "::" + srcTemp

        portSrcToDst = str(srcPort) + "::" + str(dstPort)
        portDstToSrc = str(dstPort) + "::" + str(srcPort)

        linkPorts[tempSrcToDst] = portSrcToDst

        linkPorts[tempDstToSrc] = portDstToSrc

```

```

        if (src==s):
            links.append(dst)

        elif (dst==s):
            links.append(src)

        else:
            continue

switchID = s.split(":")[7]
switchLinks[switchID]=links

# Finds The Path To A Switch

def findSwitchRoute():
    pathKey = ""
    nodeList = []
    src = int(switch[h2].split(":")[7],16)
    dst = int(switch[h1].split(":")[7],16)
    print src
    print dst
    for currentPath in nx.all_shortest_paths(G, source=src, target=dst, weight=None):
        for node in currentPath:

            tmp = ""
            if node < 17:
                pathKey = pathKey + "0" + str(hex(node)).split("x",1)[1] +
"::"
                tmp = "00:00:00:00:00:00:00:0" +
str(hex(node)).split("x",1)[1]
            else:

```

```

        pathKey = pathKey + str(hex(node)).split("x",1)[1] + "::"
        tmp = "00:00:00:00:00:00:" +
str(hex(node)).split("x",1)[1]
        nodeList.append(tmp)

    pathKey=pathKey.strip(":")
    path[pathKey] = nodeList
    pathKey = ""
    nodeList = []

```

```

print path

```

```

# Computes Link TX

```

```

def linkTX(data,key):

```

```

    global cost

```

```

    port = linkPorts[key]

```

```

    port = port.split(":")[0]

```

```

    for i in data:

```

```

        if i['port']==port:

```

```

            cost = cost + (int)(i['bits-per-second-tx'])

```

```

# Method To Compute Link Cost

```

```

def getLinkCost():

```

```

    global portKey

```

```

    global cost

```

```

for key in path:
    start = switch[h2]
    src = switch[h2]
    srcShortID = src.split(":")[7]
    mid = path[key][1].split(":")[7]
    for link in path[key]:
        temp = link.split(":")[7]

        if srcShortID==temp:
            continue
        else:
            portKey = srcShortID + "::" + temp
            stats = "http://localhost:8080/wm/statistics/bandwidth/" + src
+ "/0/json"

            getResponse(stats,"linkTX")
            srcShortID = temp
            src = link

            portKey = start.split(":")[7] + "::" + mid + "::" + switch[h1].split(":")[7]
            finalLinkTX[portKey] = cost
            cost = 0
            portKey = ""

```

```

def systemCommand(cmd):
    terminalProcess = Popen(cmd, stdout=PIPE, stderr=PIPE, shell=True)
    terminalOutput, stderr = terminalProcess.communicate()
    print "\n***", terminalOutput, "\n"

```

```

def flowRule(currentNode, flowCount, inPort, outPort, staticFlowURL):
    flow = {

```



```

        'switch':"00:00:00:00:00:00:00:" + currentNode,
    "name":"flow" + str(flowCount),
    "cookie":"0",
    "priority":"32768",
    "in_port":inPort,
        "eth_type": "0x0800",
        "ipv4_src": h2,
        "ipv4_dst": h1,
        "eth_src": deviceMAC[h2],
        "eth_dst": deviceMAC[h1],
    "active":"true",
    "actions":"output=" + outPort
}

jsonData = json.dumps(flow)

cmd = "curl -X POST -d \"\" + jsonData + "\" \" + staticFlowURL

systemCommand(cmd)

flowCount = flowCount + 1

flow = {
    'switch':"00:00:00:00:00:00:00:" + currentNode,
    "name":"flow" + str(flowCount),
    "cookie":"0",
    "priority":"32768",
    "in_port":outPort,

```

```

        "eth_type": "0x0800",
        "ipv4_src": h1,
        "ipv4_dst": h2,
        "eth_src": deviceMAC[h1],
        "eth_dst": deviceMAC[h2],
        "active": "true",
        "actions": "output=" + inPort
    }

```

```
jsonData = json.dumps(flow)
```

```
cmd = "curl -X POST -d \" + jsonData + "\" " + staticFlowURL
```

```
systemCommand(cmd)
```

```
def addFlow():
```

```
    # Deleting Flow
```

```
    #cmd      =      "curl      -X      DELETE      -d      \"{"name\":\"flow1\"}\"
http://127.0.0.1:8080/wm/staticflowpusher/json"
```

```
    #systemCommand(cmd)
```

```
    #cmd      =      "curl      -X      DELETE      -d      \"{"name\":\"flow2\"}\"
http://127.0.0.1:8080/wm/staticflowpusher/json"
```

```
    #systemCommand(cmd)
```

```
    flowCount = 1
```

```
    staticFlowURL = "http://127.0.0.1:8080/wm/staticflowpusher/json"
```

```

shortestPath = min(finalLinkTX, key=finalLinkTX.get)
print "\n\nShortest Path: ",shortestPath

currentNode = shortestPath.split(":",2)[0]
nextNode = shortestPath.split(":")[1]

# Port Computation

port = linkPorts[currentNode+"::"+nextNode]
outPort = port.split(":")[0]
inPort = hostPorts[h2+"::"+switch[h2].split(":")[7]]

flowRule(currentNode,flowCount,inPort,outPort,staticFlowURL)

flowCount = flowCount + 2

bestPath = path[shortestPath]
previousNode = currentNode

for currentNode in range(0,len(bestPath)):
    if previousNode == bestPath[currentNode].split(":")[7]:
        continue
    else:
        port
linkPorts[bestPath[currentNode].split(":")[7]+"::"+previousNode]
        inPort = port.split(":")[0]
        outPort = ""

```

```

        if(currentNode+1<len(bestPath)                                and
bestPath[currentNode]==bestPath[currentNode+1]):
            currentNode = currentNode + 1
            continue
        elif(currentNode+1<len(bestPath)):
            port =
linkPorts[bestPath[currentNode].split(":")[7]+":"+bestPath[currentNode+1].split(":")[7]]
            outPort = port.split("::")[0]
        elif(bestPath[currentNode]==bestPath[-1]):
            outPort = str(hostPorts[h1+":"+switch[h1].split(":")[7]])

        flowRule(bestPath[currentNode].split(":")[7],flowCount,str(inPort),str(outPort),staticF
lowURL)

        flowCount = flowCount + 2
        previousNode = bestPath[currentNode].split(":")[7]

# Method To Perform Load Balancing
def loadbalance():
    linkURL = "http://localhost:8080/wm/topology/links/json"
    getResponse(linkURL,"findSwitchLinks")

    findSwitchRoute()
    getLinkCost()
    addFlow()

# Main

# Stores H1 and H2 from user
global h1,h2,h3

```

```
h1 = ""
h2 = ""

print "Enter Host 1"
h1 = int(input())
print "\nEnter Host 2"
h2 = int(input())
print "\nEnter Host 3 (H2's Neighbour)"
h3 = int(input())

h1 = "10.0.0." + str(h1)
h2 = "10.0.0." + str(h2)
h3 = "10.0.0." + str(h3)

while True:

    # Stores Info About H3 And H4's Switch
    switch = {}

    # Mac of H3 And H4
    deviceMAC = {}

    # Stores Host Switch Ports
    hostPorts = {}

    # Stores Switch To Switch Path
```

```
path = {}
```

```
# Switch Links
```

```
switchLinks = {}
```

```
# Stores Link Ports
```

```
linkPorts = {}
```

```
# Stores Final Link Rates
```

```
finalLinkTX = {}
```

```
# Store Port Key For Finding Link Rates
```

```
portKey = ""
```

```
# Stores Link Cost
```

```
cost = 0
```

```
# Graph
```

```
G = nx.Graph()
```

```
try:
```

```
    # Enables Statistics Like B/W, etc
```

```
    enableStats = "http://localhost:8080/wm/statistics/config/enable/json"
```

```
    requests.put(enableStats)
```

```
    # Device Info (Switch To Which The Device Is Connected & The MAC  
    Address Of Each Device)
```

```
    deviceInfo = "http://localhost:8080/wm/device/"
```

```

getResponse(deviceInfo,"deviceInfo")

# Load Balancing

loadbalance()

# ----- PRINT -----

print "\n\n##### RESULT #####\n\n"

# Print Switch To Which H4 is Connected
print "Switch H4: ",switch[h3], "\tSwitchH3: ", switch[h2]

print "\n\nSwitch H1: ", switch[h1]

# IP & MAC
print "\nIP & MAC\n\n", deviceMAC

# Host Switch Ports
print "\nHost::Switch Ports\n\n", hostPorts

# Link Ports
print "\nLink Ports (SRC::DST - SRC PORT::DST PORT)\n\n", linkPorts

# Alternate Paths
print "\nPaths (SRC TO DST)\n\n",path

# Final Link Cost

```

```
print "\nFinal Link Cost (First To Second Switch)\n\n",finalLinkTX
```

```
print "\n\n#####\n\n"
```

```
time.sleep(60)
```

```
except KeyboardInterrupt:
```

```
    break
```

```
    exit()
```





