Dynamic Load Balancing in Software-Defined Networks

Networks and Distributed System
Group 1025 - 10th semester

Aalborg University
Department of Electronic Systems
Fredrik Bajers Vej 7B
DK-9220 Aalborg

Department of Electronic Systems
Fredrik Bajers Vej 7
DK-9220 Aalborg Ø
http://es.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Dynamic Load Balancing in Software-Defined Networks
-

Theme:
Master's thesis

Project Period:
10^{th} semester
Networks and Distributed System
February – June, 2014

Project Group:
1025

Participants:
Martí Boada Navarro

Supervisors:
Jimmy Jessen Nielsen
Andrea Fabio Cattoni

Copies: 3

Page Numbers: 58

Date of Completion: June 4, 2014

Abstract:

In the last 20 years networks requirements have been changing constantly, the amount of traffic have been increasing exponentially and more demanding end-to-end goals are enforced to be accomplish. However, the networks architectures have been unchanged, increasing the complexity and hindering its configuration. Generally, the guarantee for such demands leads to an over dimensioning of the resources, which, at the end, it is translated to a poor utilisation of resources.

In the last years, a new networking approach called Software-Defined Networking (SDN) is emerging fast. This approach is based on the separation of data and control planes. Such approach allows the network administrator to have a more dynamic control of the network behaviour.

The purpose of this project is to analyse the possibilities that SDN provides to develop a more efficient resources allocation along the network when more than one possible route between source and destination exists.

Preface

This is a project report of an exchange student coming from *Universitat Politècnica de Catalunya* (UPC) enrolled to the study programme *Networks and Distributed Systems* at *Aalborg Universitet* within the Eeasmus programme. This project serves as a Master Thesis as the requirements of the master's programme *Master of Science in Telecommunications Engineering and Management* and has been conducted from February to June of 2014.

This project proposes a dynamic load balancing algorithm applied to Software-Defined Networks to achieve the best possible resource utilisation of each of the links present in a network. The open-source Floodlight project is used as an SDN controller, and the network is emulated using Mininet software.

Aalborg University, June 4, 2014

Martí Boada Navarro
<mboada14@student.aau.dk>

Contents

List of Figures

Acronyms

API	Application Programmers Interface
CB	Contributed Bandwidth
CLI	Command-Line Interface
DFS	Depth-First Search
DLB	Dynamic Load Balancing
ER-LSP	Explicitly Routed Label Switched Path
FC	Free Capacity
ISP	Internet Service Provider
LFC	Link Flows Collection
MPLS	Multi Protocol Label Switching
NLC	Negotiated ER-LSP Collection
OFA	OpenFlow Agent
OFC	OpenFlow Controller
OSPF	Open Shortest Path First
PC	Paths Collection
QoS	Quality of Service
REST	REpresentational State Transfer
SDN	Software Defined Networking
SRC	Selectable Route Collection
STP	Spanning Tree Protocol
TCP	Transmission Control Protocol
TE	Traffic Engineering
UB	Used Bandwidth
UDP	User Datagram Protocol

1 Introduction

This introduction chapter tries to give a general view of the existing problems in current networks, and how new emergent technologies can help to improve the network possibilities and capabilities in order to achieve a more flexible approach and adapt to todays needs.

1.1 Motivation

In the last 20 years networks requirements have been changing constantly, the amount of traffic has been increasing exponentially and more demanding end-to-end goals are needed. However, the networks architectures have been unchanged, increasing the complexity and hindering its configuration. In order to adapt to the new needs, new network paradigms such as Software-Defined Networking (SDN), cognitive networks or automatic networks are emerging fast due the interests of carriers and ISPs.

The first things to do is to analyse which are the problems of the existing networks since they have become a barrier to creating new, innovative services, and an even larger barrier to the continued growth of the Internet.

Analysing the data plane in the ongoing networks architecture we find well defined layers for different purposes, making them autonomous. For instance the physical layer (fibre, copper, radio...) is independent of the link layer (ATM, X.25, PPP...), the transport layer (TCP, UDP...) or the application layer (HTTP, FTP, telnet...), what allows the evolution of each of them independently. Thanks to the fact of dividing the problem in tractable pieces, networks have been able to evolve, increasing many magnitude changes in terms of speed, scale or diversity of uses.

On the other hand, there is the control plane which, unlike the data plane, has no abstraction layers at all. Within the control plane there are several protocols to configure the network elements, such as Multiprotocol Label Switching (MPLS)[1] or NETCONF[2]. There are also a bunch of routing protocols e.g. Routing Information Protocol (RIP)[3], Enhanced Interior Gateway Routing Protocol (EIGRP)[4] or Shortest Path Bridging (SPB)[5].

[1] MPLS architecture RFC 3031
[2] Network Configuration Protocol (NETCONF) RFC 6241
[3] RIP Version 2 RFC 2453
[4] CISCO Enhanced Interior Gateway Routing Protocol (EIGRP)
[5] SPB IEEE Std 802.1aq

Protocols tend to be defined in isolation, however, with each solving a specific problem and without the benefit of any fundamental abstractions. This has resulted in one of the primary limitations of today's networks: complexity. For example, to add or move any device, IT must touch multiple switches, routers, firewalls, Web authentication portals, etc. and update ACLs, VLANs, Quality of Services (QoS), and other protocol-based mechanisms using device-level management tools. In addition, network topology, vendor switch model, and software version all must be taken into account. Due to this complexity, today's networks are relatively static as IT seeks to minimise the risk of service disruption.

In order to deal with all that mess, hundreds of network monitoring and management tools[6] have appeared trying to help the networks managers to have a control of their networks.

At the same time that technology advances, the communications requirements also evolve. The claims that nowadays are needed in the different services that users demand, such as VoIP or streaming video in High Quality, where inconceivable when the architecture was designed.

All those factors have lead to an over-provisioning of the networks, increasing the cost, and wasting resources due the difficulty to optimise the control and adapt the physical resources available to the requirements of every instant. So it is time to take a step forward and evolve to a more optimal architecture, easy to manage, evolve and understand.

Here is where the SDN approach (explained in section 2.1) comes to play. Having a clear abstraction layers and a centralised control plane, it's much easier to make a more efficient and dynamic management and control of the network, since we have a global overview of the network, and control over its entirety.

The motivation of this Master's Thesis is to take advantage of the new centralised networking approach of SDN to develop a load balancing algorithm that adapt the route of each flow depending on the current state of the network in order to achieve a better resource allocation, reducing the overall cost and adapting its behaviour to the traffic growth.

1.2 Related Work

There is plenty of papers about traffic engineering [3] 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 readjust the networks is using OSPF, as described in [12], by adjusting the state of the links thus the traffic can be adjusted to the current conditions .In [9] they compare OSPF against MPLS. trying several ways to adjust 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 in front of MPLS, since it is much simpler due that the routing is completely determined by one weight of each link, what avoids making decision

[6] Standford Network Monitoring and Management Tools list.

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, multiple paths can be used simultaneously 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 [18] [11] [10] [7]. In [13] 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 of them try to analyse the use cases [4] [16], others 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 [5]. 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 [15].

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 [8] [6].

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.

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, and tries to answer the following questions:

How can Software-Defined Networking improve the total throughput of the network when there is more than one path between the source and destination?

Which is the best way to decide which flows must be rerouted?

Which are the effects of rerouting a flow in terms of throughput, delay, jitter and packet losses?

It is possible to maintain the quality parameters within a certain margin?

2 Pre-analysis

In order to investigate how to answer the initial problem stated in section 1.3, some background information is needed. This chapters gives a general overview about the concept of Software-Defined Networking, OpenFlow, providing the information that must be taken into account in order to develop and understand this project.

2.1 Software-Defined Networking

As it is explained in section 1.1, the main problem of the current networks is that the control plane has no abstractions. That means that there is no modularity, which leads to limited functionality, since protocols with different proposals (such as routing, isolation or traffic engineering) coexist in the same layer.

One of the tendencies that seems to have more power for the next generation networking is Software-Defined Networking (SDN). One of the reasons to believe that SDN will be *the one* is that some big companies, such as Google, are already using it.[17][2].

SDN is a new network architecture that allows be programmed as if it were a computer. It provides an abstraction of the forwarding function decoupling the data plane from control plane, which gives freedom to manage different topologies, protocols without many restrictions from the physical layer.

SDN was designed to change that paradigm, dividing the control plane in three main layers: the forwarding model, the Network Operating System (NOS) and the control program.

Forwarding model: composed by the Network Elements (i.e. switches) and a dedicated communicated channel from each NE with the NOS which uses a standard way of defining the forwarding state.

Network Operation System: piece of software running in servers (controllers) which provides information about the current state of the network such as the topology or the state of each port.

Control program: express the operator goals, and compute the forwarding state of each NE to accomplish those goals.

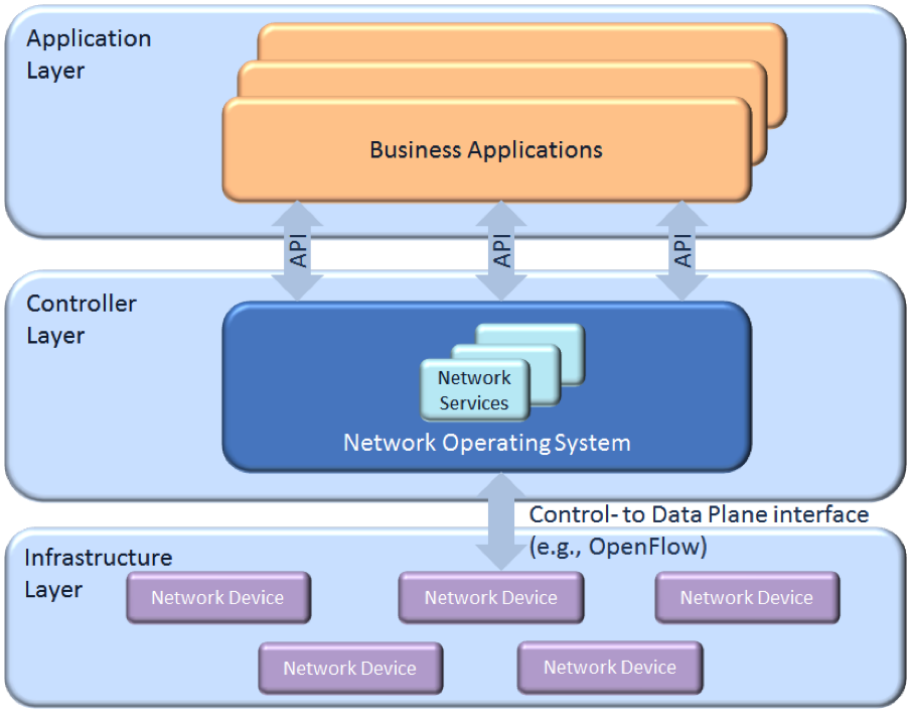

Figure 2.1: SDN architecture (image from Open Networking Foundation)

2.1.1 Advantages

SDN tries to improve the current networks. Bellow there is a list of the main advantages of the SDN paradigm.

OPEX reduction: centralised control helps to eliminate manual interaction with the hardware, improving the uptime of the network.

CAPEX reduction: separating the data plane of the control plane brings to a more simple hardware and increases the possibility of more competence between hardware manufacturers, since the devices don't depends on the proprietary software.

Agility: since the control layer can interact constantly with the infrastructure layer, the behaviour of the network can adapt fast to changes like failures or new traffic patterns.

Flexibility: having a separated abstraction for the control program allows to express different operator goals, adapting to a specific objectives. Operators can implement features in software they control, rather than having to wait for a vendor to add it in their proprietary products.

2.1.2 Applications

To give an idea of how huge SDN is, the list below mentioned some of the applications which is related to.

1. **Appliance Virtualization**
   - Firewalls, Load balancers, Content distribution, Gateways
2. **Service Assurance**
   - Content-specific traffic routing for optimal QoE, Congestion control based on network conditions, Dynamic policy-based traffic engineering
3. **Service Differentiation**
   - Value-add service features, Bandwidth-on-demand features, BYOD across multiple networks, Service insertion/changing
4. **Service Velocity**
   - Virtual edge, Distributed app testing environments, Application development workflows
5. **'Traditional' Control Plane**
   - Network discovery, Path computation, Optimisation & maintenance, Protection & restoration
6. **Network Virtualization**
   - Virtual network control on shared infrastructure, Multi-tenant network automation & API
7. **Application Enhancement**
   - Specific SDN application, Reserved bandwidth for application needs, Geo-distributed applications, Intelligent network responses to app needs.

2.1.3 SDN Controllers

Nowadays there are several projects working up in SDN controllers, and it is important to analyse the choice before starting a project. The selection of the controller can be based in different aspects, such as the programming language, the current activity on its development, the amount of documentation or the set of built-in components.

Table 2.1 shows a summary of features of the most known controllers, specifying the language in which they are written, the supported languages, and other factors.

Table 2.1: SDN controllers comparison

Feature	Ryu	POX	Beacon	Floodlight	Daylight
Written in	Python	Python	Java	Java	Java
Supported languages	Python	Python	Java	Java, Python	Java
Actively developed?	Yes	Yes	No	Yes	Yes
Active community?	Yes	Yes	Yes	Yes	Yes
Well documented?	Yes	No	Yes	Yes	Yes
REST APi?	Yes	Yes (limited)	Yes	Yes	Yes
Utility functions?	Yes	No	No	Yes	No
Has a UI?	web	Py+QT4	web	Java, Web	Web

In a recent paper [14] a deep analysis of the most common controllers is presented. Figure 2.2 shows their analysis of the interfaces used in the northboud (NB) and southbound (SB) APIs. The study, based on ten selection criteria, (Interfaces, Platform support, GUI, REST API, Productivity, Documentation, Modularity, Age, OpenFlow support, and Quantum support), concludes that the best one is Ryu Controller[1] followed by Floodlight Controller.

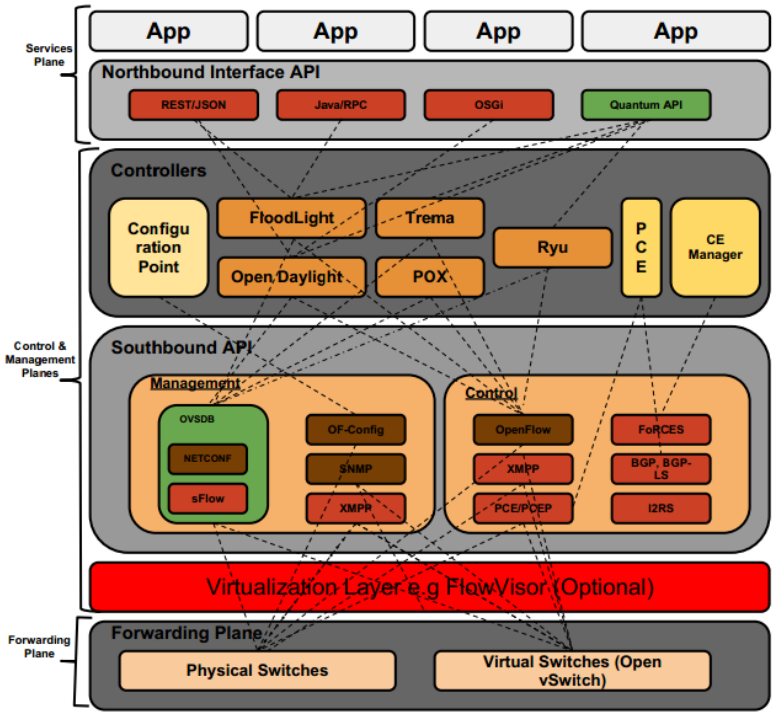

Figure 2.2: Interfaces of SDN controllers (image from [14])

[1]Ryu Controller

2.2 OpenFlow

In the current networks (i.e. non SDN), the routers and switches makes the forwarding (data path) and routing (control path) decisions by itself. In an OpenFlow switch, those decisions are divided between the network element (i.e. the switch) and the controller. Such division keeps the data path on the network element, while high-level routing are moved to the NOS.

OpenFlow is the protocol used to communicate the NOS (controller) with all the Network Elements (NE). Is an open standard that provides a standardised hook to allow researchers to run experiments, without requiring vendors to expose the internal workings of their network devices. OpenFlow is currently being implemented by major vendors, with OpenFlow-enabled switches now commercially available.

OpenFlow is the most common protocol used in SDN networks, and is often confused with the SDN concept itself, but they are different things. While SDN is the architecture dividing the layers, OpenFlow is just a protocol proposed to convey the messages from the control layer to the network elements. There is a bunch of OpenFlow based projects[2], including several controllers, virtualised switches and testing applications.

2.2.1 Flow Tables

Each NE has a flow table which contains a set of flow entries. Each flow entry is written by the controller, which computes the routing decisions sending the corresponding rules to each switch. Those flow entries consist of a determinate actions (e.g forward packet through specified port) to do with packets which headers contains a particular values (e.g. packets with a specific destination).

If no match is found for an incoming packet, this is forwarded to the controller over a secure channel, and it is the controller who manage the flow entries of the NEs adding or removing flow entries in order to accomplish the stablished rules.

Each flow entry consists of three fields: header fields, counters and actions. The process by which each incoming packet is matched against is explained in the next points.

Header Fields

Aiming to determine each flow, there are several fields that can be matched. The header fields that incoming packets are compared against are the following ones:

[2]List of OpenFlow Software Projects.

- Ingress Port
- Ethernet Source
- Ethernet Destination
- Ethernet Type
- VLAN ID
- VLAN Priority
- IP Source
- IP Destination
- IP Protocol
- IP ToS bits
- TCP/UDP Source Port
- TCP/UDP Destination Port

Counters

Each switch maintains per-table, per-flow, per-port and per-queue. The counters for use are listed in Table 2.2. Those values can be obtained through polling requests to the network elements.

Table 2.2: Openflow Counters

Per Table	**Per Flow**	**Per Port**	**Per Queue**
Active Entries	Received Packets	Received Packets	Transmit Packets
Packet Lookups	Received Bytes	Transmitted Packets	Transmit Bytes
Packet Matches	Duration (sec)	Received Bytes	Transmit Overrun Errors
	Duration (nano)	Transmitted Bytes	
		Receive Drops	
		Transmit Drops	
		Receive Errors	
		Transmit Errors	
		Receive Frame Alignment Errors	
		Receive Overrun Errors	
		Receive CRC Errors	
		Collisions	

Actions

Each flow entry is associated with zero or more actions that determine the behaviour of the network element when there is a matching in a incoming packet. There are several types of actions. However, just some of them are not supported in all the OpenFlow switches. When the NE connects to the controller (see Figure fig:OFHandshake), it indicates which of the optional actions it supports.

The required actions are:

Forward: forwarding the packet to physical ports.

- ALL: send the packet to all the interfaces less the incoming one.
- CONTROLLER: send the packet to the controller.
- LOCAL: send the packet to the switches networking stack.
- TABLE: perform actions in flow table.
- IN_PORT: send the packet through the incoming port.

Drop: if there is no specified action, the flow entry indicates that the matching packets should be dropped.

And the optionals:

Forward: forwarding the packet to physical ports.

- NORMAL: Process the packet using the traditional forwarding path supported by the switch (i.e. traditional L2, VLAN, and L3 processing.)
- FLOOD: Flood the packet along the minimum spanning tree, not including the incoming interface.

Enqueue: forwards a packet by some specific queue of the port. Used to provide basic QoS

Modify-Field: this action increases the usefulness of an OpenFlow implementation, allowing to modify the following fields of an incoming packet:

- VLAN ID: adds, modifies, strips or replace the VLAN header.
- Ethernet Source MAC: modifies source MAC address.
- Ethernet Destination MAC: modifies destination MAC address.
- IPv4 source: modifies source IPv4 address.
- IPv4 destination: modifies destination IPv4 address.
- IPv4 ToS bits: replaces the existing IP Type of Service bits.
- Transport source port: replaces the existing source port.

- Transport destination port: replaces the existing destination port.

2.2.2 Messages

The protocol, described in [1], defines three sort of messages, *controller-to-switch*, *asynchronous* and *symmetric*. For instance, when the controller detects a new switch connected, it starts a handshake process (illustrated at Figure 2.3) starting with a symmetric *Hello* message, and followed by two controller-to-switch messages: *Features* and *Configuration*.

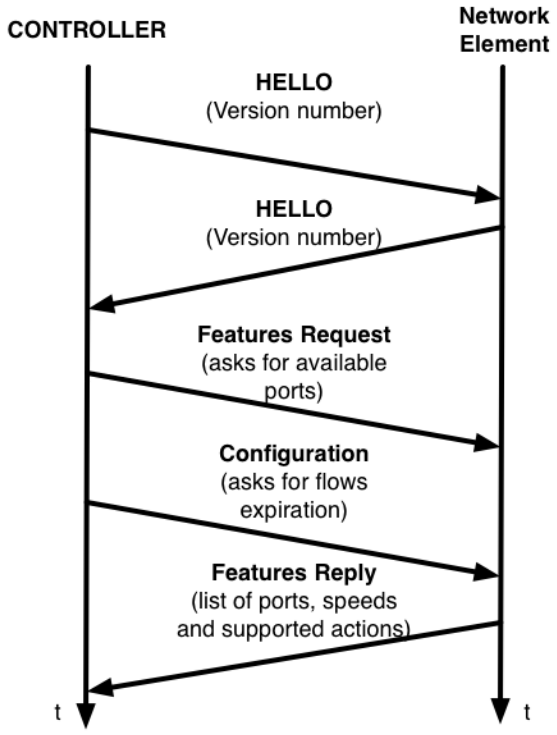

Figure 2.3: OpenFlow messages for Controller-Switch Handshake

The different types of messages are described briefly in the next points.

Controller-to-switch

Controller-to-switch messages are initiated by the controller. Their finality is to get parameters from the NEs or manage them. The different kinds are listed in the next points.

Features: Requesting the specific capabilities of the switch.

Configuration: Set and query configuration parameters in the switch.

Modify-State: Their primary purpose is to add or delete flows in the flow tables and to set switch port properties.

Read-State: Used to collect statistics from every switch.

Send-Packet: Used to send packets out of a specified port on the switch.

Barrier: Used to ensure message dependencies have been met or to receive notifications for completed operations.

Asynchronous

Asynchronous messages are sent from the network elements to the controller when some event occurs. The four different types of asynchronous messages are described beneath.

Packet-in: When a packet arrives to the network element and this doesn't match with any entry of the flow table, a message is send to the controller asking what to do with. By default it includes the first 128 bytes of the packet header.

Flow-Removed: Message sent when a flow entry expires or it is removed from a network element.

Port-Status: Sent by the network element when state configuration state (such as port brought down directly by a user) changes.

Error: Messages indicating some errors in the network element.

Symmetric

Symmetric messages are sent without solicitation, in either direction. The 3 different sort of messages are the following ones:

Hello: Sent when connection startup as a handshake.

Echo: Used to indicate latency, bandwidth or liveness of a controller-NE connection. It must return an echo reply.

Vendor: Pretends to offer additional functionality. Meant for future OpenFlow revisions.

3 Algorithm Description

This section describes the algorithm designed to accomplish a dynamic load balancing system. In the first point, a previous proposed algorithm for MPLS networks is explained, since the proposed algorithm is based on it. Afterwards an explanation about the different data structures used to accomplish the algorithm goal is explained, followed with a detailed description of how has been adapted to a Software-Defined Network.

3.1 MPLS Dynamic Load Balancing Algorithm

The first step before designing the algorithm was to look on the state of the art in current networks. In [13] they propose three algorithms based on MPLS networks which can be adapted to SDN. After analysing their results with the three algorithms, the most interesting one, and more suitable is the one they call Dynamic Load Balancing (DLB).

The algorithm is executed every time that a new traffic flow arrives. First a structure called SRC (Selectable Route Collection) is constructed, formed by all the *route* elements, which define each of the possible routes between the ingress and egress points of the flow.

$$route = \{<RouteID>, <HOP>, <Length>, <Free\_Capacity>\}$$

$$SRC = \{< route_1 >, < route_2 > ... < route_n >\}$$

After that they construct a structure named $ER-LSP$ (Explicitly Routed Label Switched Path), which describes the resource utilisation of each flow. Each $ER-LSP$ has the ID of the *route* which is using, the Used Bandwidth of this flow (UB), the Free Capacity of the path (FC) and the Contributed Bandwidth (CB), which corresponds to: $CB = UB+FC$.

$$ER-LSP = \{<TrafficID>, <LSPID>, <RouteID>, <UB>, <FC>, <CB>\}$$

They also use a variable Pending-Traffic (PT) to keep the TrafficID that is to select route or to reroute, and, for each traffic flow on PT a Negotiate ER-LSP Collection (NLC) with all parallel $ER-LSP$s whose UB is smaller than bandwidth request of this traffic flow

and whose route is in SRC

$$NLC = \{< ER - LSP_1 >, < ER - LSP_2 > ... < ER - LSP_n >\}$$

DLB (represented in 3.1) can take into account network topology character and traffic bandwidth request simultaneously. Under light load condition, traffic flows can be mapped on the short and high- capacity route; when load changes to heavy, small traffic flow can reroute to another appropriate route so as to reserve high- capacity route for large traffic flow. DLB algorithm can greatly enhance network throughput and resource utilisation.

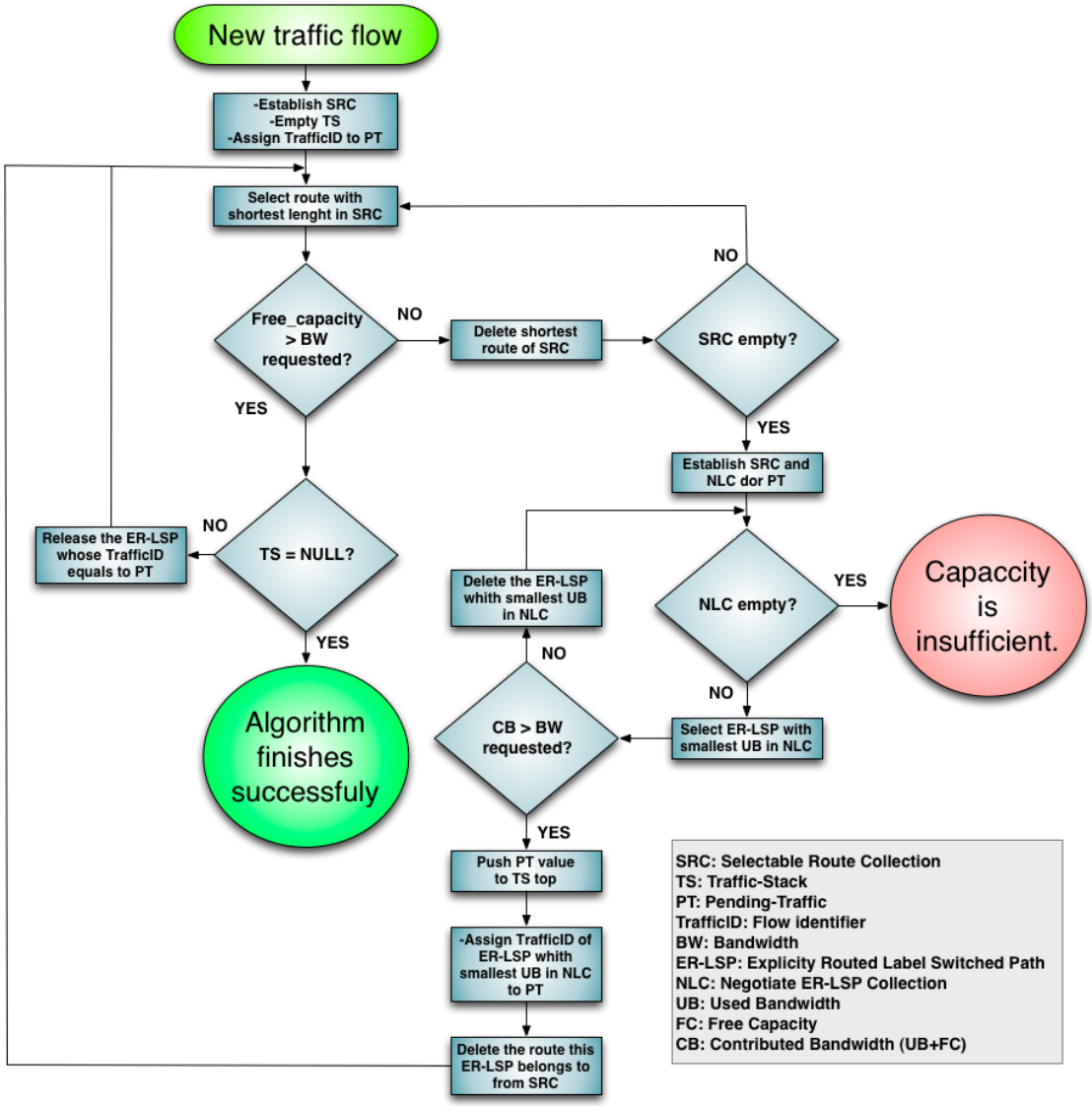

Figure 3.1: MPLS Dynamic Load Balancing algorithm.

The main aim of the algorithm is to move the traffic flows with lower used bandwidth to the paths with lowest capacity. Thus, assuring that the heaviest flows are allocated to the highest-capacity paths.

However, this algorithm has some shortcomings. The first one, is that they assume that an $ER-LSP$ carries just 1 flow which leads to a non-realistic scenario. Another point to be considered is that they assume the bandwidth utilisation of each flow is known beforehand. Even though some systems work under this principle due service learning agreements, this leads to a inefficient resource utilisation in the case that the traffic flow uses less than the bandwidth established.

Furthermore, the results are based on simulations, and the algorithm has not been applied to real condicions.

3.2 Dynamic Load Balancing Algorithm applied to SDN

On the basis of the algorithm commented above, a new algorithm has been designed. This algorithm takes into account the advantages and features of a Software-Defined Network, which can sense the stat of each of the elements on the network in order to act consequently.

3.2.1 Data Structures

In order to describe the algorithm, first it is needed to disclose the different data structures involved on it. Such structures characterise the different elements that have been taken in account in order to achieve an efficient load balancing, at the same time that to reduce as much as possible the computational cost and time.

The three main data structures are explained bellow.

Flow

Since the algorithm provides load balancing based on flows, it is necessary to define a structure to describe each of the different flows with distinctive parameters.

Notice that for the goal of this project have been taking in account the IPs and Ports, but using OpenFlow it is possible to make a much more accurate identification of each flow with any of the header fields enumerated in section 2.2.1.

A *Flow* structure specifies a specific traffic flow from one host to another one. The structure is as follows:

$$Flow = \{<FlowID>, <SrcIP>, <DstIP>, <SrcPort>, <DstPort>, <UsedBandwidth>\}$$

FlowID: identify each flow with an unique ID.

SrcIP: this field contains the IPv4 of the source host who initialised a flow.

DstIP: IPv4 of the destination host.

SrcPort: port number of the source.

DstPort: port number of the destination.

UsedBandwidth: transmission speed of a specific flow, in Mbps.

Flows Collection

Groups of flows are put together in collections, which contain a number of flows with common characteristic (e.g flows that goes through a same link).

$$FlowsCollection = \{< Flow_1 >, < Flow_2 > ... < Flow_n >\}$$

Path

Keeping information about each parallel route between each pair of hosts it is crucial to be able to redirect the flows according to the current network conditions. To accomplish that tracking, a data structure representing each of the possibles paths has been designed.

A *Path* structure contains the information about a precise path between two hosts, and it is composed as shown bellow:

$$Path = \{<PathID>, <Hops>, <Links>, <Ingress>, <Egress>, <Capacity>, <Flows>, <UsedBandwidth>, <FreeCapacity>\}$$

PathID: identify each single possible path with a unique ID.

Hops: contains a identifier of each of the switches within the path.

Links: list of all the links involved in the path. Each of the links is composed by a pair of Switch-Port identifiers.

Ingress: identifier of the switch with the source host is connected to.

Egress: identifier of the switch with the destination host is connected to.

Capacity: specify the maximum capacity of the path. Which corresponds to the capacity of the link with smallest capacity along the path.

Flows: list of flows which are routed through this path.

UsedBandwidth: sum of the traffic of all the flows that are using this path, in Mbps.

FreeCapacity: capacity available in this path. Is the minimum capacity free of the *Links* that shape the path.

Paths Collection

PathCollection contains a list of all the possible paths of the hosts that have initiated a communication between them, and information about each of the paths.

$$PathsCollection = \{< Path_1 >, < Path_2 > ... < Path_n >\}$$

3.2.2 Algorithm Description

As explained formerly, the aim of the algorithm is to balance dynamically the loads depending on traffic conditions in order to achieve the best resource profit possible. In pursuance of such goal, is essential to keep track of the current state of the network in terms of traffic.

New flow detected

The first step is detecting a new flow event. This is done when a packet arrives to a switch in the network, and the header of that packet doesn't matches with any of the rules that the switch has, which will trigger the Algorithm 1.

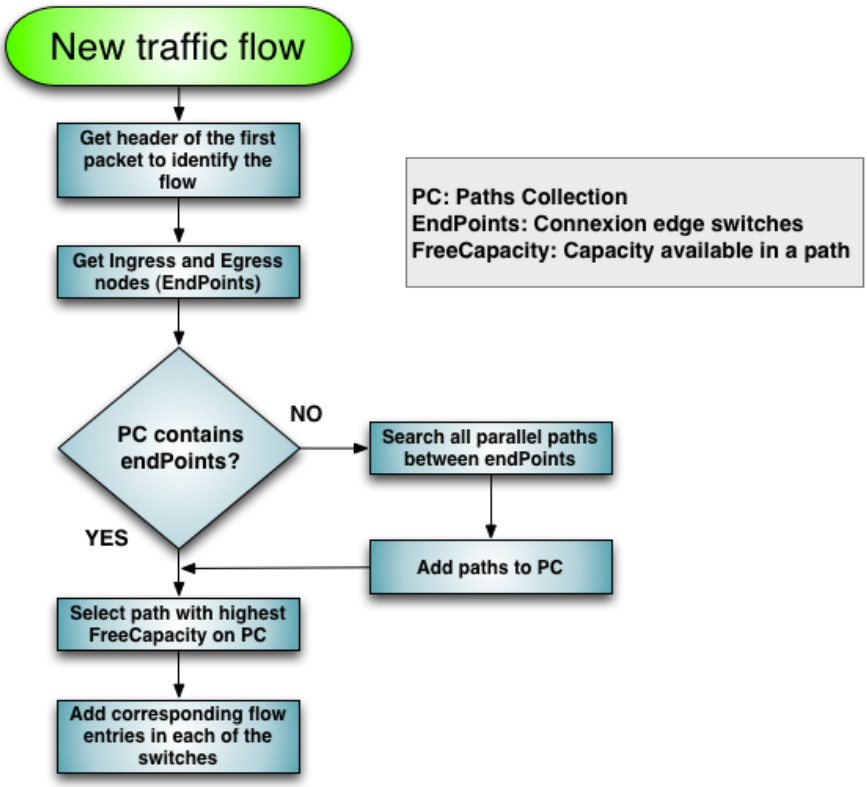

Figure 3.2: New flow routing.

The next step is to find if all the possible paths between the to points have been computed already, thus avoid to recalculate the possible paths in case that have been computed already. This is useful due the possibility that different flows can have the same ingress and egress switches (e.g flows from the same source host to the same destination but with different ports, or hosts attached to the same switch). In the case that the paths between the ingress and egress switches have not been computed yet, a function to calculated it is triggered. The way to find all the possible paths is by using a Depth-First Search (DFS) algorithm. For each new path discovered, a new *Path* (section 3.2.1) is stored into the *PathCollection*, having as a unique identifier a string with all the nodes which the path goes through.

Once all the possible paths are found, it is needed to, first, select a route, and second, write the rules in the flow tables of each switch within the selected route. To select the route, the algorithm looks for the *Path* with bigger *FreeCapacity* value. Since when the flow starts there is no information about the traffic that it will bear and this way we guarantee that the flow will use the route where there is more capacity available.

After the route is selected, and using the *Hops* and *Links* of the specific *Path*, it is needed to send the appropriated messages to the switches to forward the packets through that

path.

With that, every switch within the selected route will have the necessary flow entries to carry out the communication between the two end points.

Rerouting flows

After the flow has initiated its communication between the two hosts, and starts to transmit traffic, the algorithm starts to capture information about the network state in order to adapt to the conditions of every single moment. Once congestion is detected at any link, appears the need of rerouting some of the current flows. To accomplish the best accommodation of resources possible, this algorithm is focused on reroute the flows with lowest traffic.

The process conducted is explained in two parts. The first one, illustrated in Figure 3.3, explains the first step, where the link which is overcrowded is detected and, afterwards, a *FlowsCollection* is established (LFC), containing all the flows that are using that link. Subsequently it selects the flow with lowest bandwidth usage (assigning it to the variable *PendingFlow*), and checks out if there is any other parallel route for this flow with enough free capacity to carry its traffic. In the case that there is another possible path with sufficient capacity, the flow will be routed through that route, sending the corresponding flow entries to each of the OpenFlow switches.

Once the flow with the lowest bandwidth usage have been routed across another path, the process starts again, and, in the case that the congestion still exists, the same procedure will be followed, moving the lightest flows along another routes.

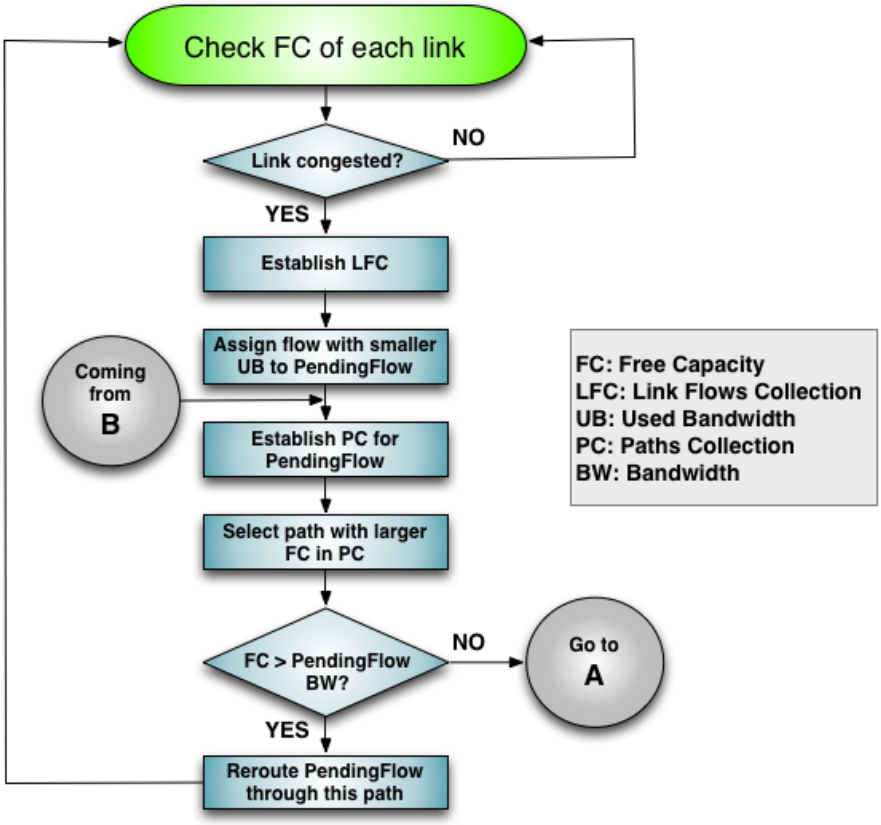

Figure 3.3: Rerouting flow first part.

The second part (shown in Figure 3.4) is launched when, after congestion is detected, no route with enough free capacity have been found to reroute any of the flows. In this case, we need to analyse the different flows to figure out which is the best way to allocate them.

As mentioned before, the first flows to be targeted will be the smallest ones, so, following the process explained in the first part of the rerouting, and having established the *PathsCollection* with all the parallel routes of the faintest flow, the next step is to execute the main loop of the algorithm. This loop consist on revise, each of the *Paths* within that collection, starting from the one with highest capacity.

From that *Path*, we need to get all the flows which are carrying a smaller amount of traffic that the flow that we are trying to reroute. This step is very important, to assure that we are not rerouting a bigger flow than the one that we are trying to allocate, thus guaranteeing that after all the iterations in the algorithm, we will have the maxim profit of resources along the network.

Afterwards, and inside the same loop, the algorithm iterate for every flow, starting with lightest one again, and analysing if the CB (Contributed Bandwidth) of the selected flow is higher than the bandwidth than the flow that we are trying to allocate, i.e. we are inquiring the free resources that will be available in this *Path* if the selected flow is moved

along another route, and see if that will be enough to carry the bandwidth the flow needs. If so, the *PendingFlow* will be routed through that route, and the route that we had to move to free the space will be now the new *PendingFlow*, and going to the first part of the algorithm, and trying to find a smaller that could be moved away to leave resources for it.

In the case that after iterate after every single flow, of every single path there is no possibility to reroute any of the flows (*PathCollection* empty), the algorithm finishes, without being able to reroute the flow, which means that there are not enough resources to satisfy the needs of all the flows.

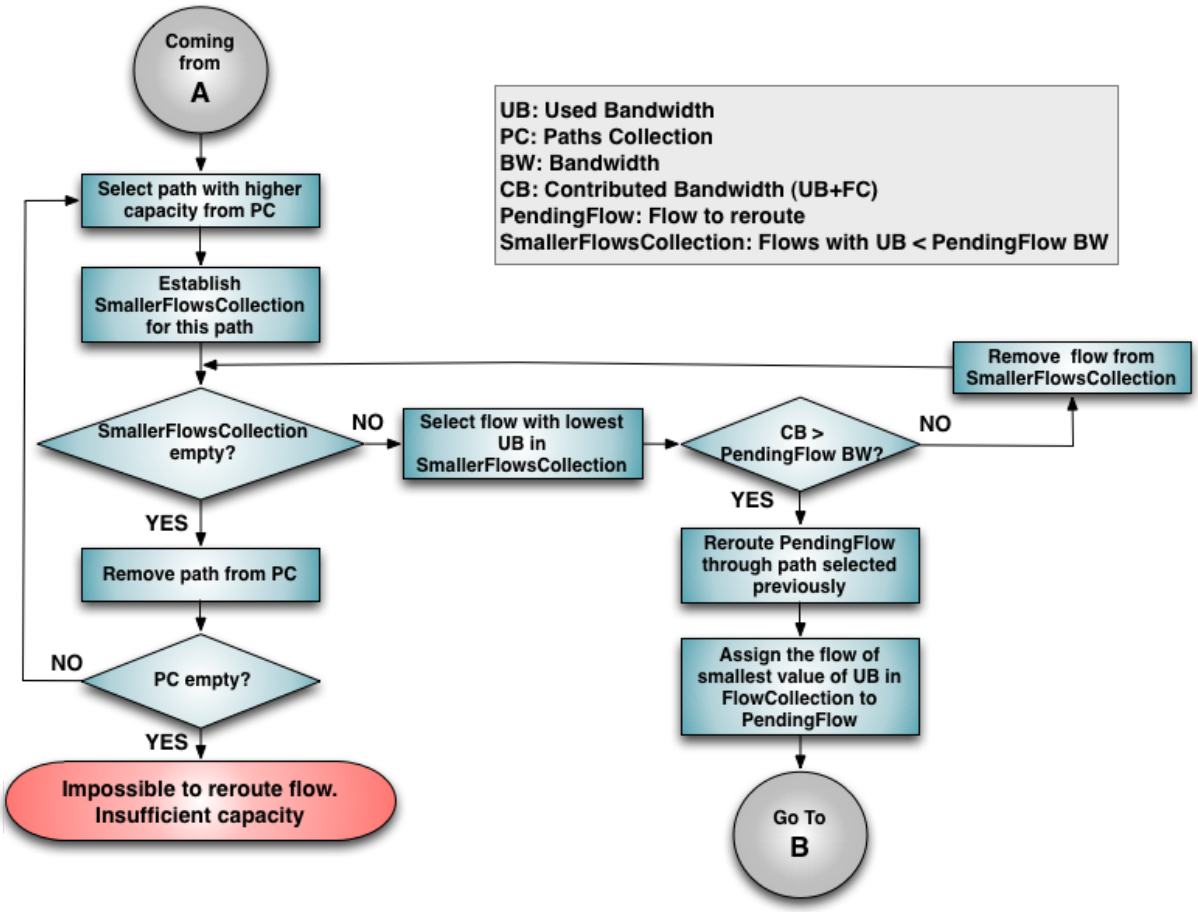

Figure 3.4: Rerouting flow second part.

4 Scenario Description

This chapter explains the software tools used in this project in order to set the testbed, giving a justification of the selected tools. It also shows the topology used for the evaluation of the algorithm.

4.1 Mininet Network Emulator

Mininet is an open source software that allows to emulate an entire network. It is the main tool for Software-Defined Networking testbed environments to design, undergo and verify OpenFlow projects.

Mininet provides a high level of flexibility since topologies and new functionalities are programmed using python language. It also provides a scalable prototyping environment, able to manage up to 4000 switches on a regular computer. This is possible thanks to a OS-level virtualization features, including processes and network namespaces, which allows to have different and separate instances of network interfaces and routing tables that operate independent of each other.

Unlike other simulators, like ns-2[1] or Riverbed Modeler[2], which lack realism and the code created in the simulator needs to be changed to be deployed in the real network, Mininet needs no changes on code either configuration when applied to hardware-based networks, offering a realistic behaviour with a high degree of confidence. Another advantage in front of other simulators is that allows real time interaction.

4.1.1 Topology Elements

There are four topology elements that Mininet can create:

Link: emulates a wired connection between two virtual interfaces which act as a fully functional Ethernet ports. Packets sent through one interface are delivered to the other. It is possible to configure Traffic Control for the links importing the TCLink library[3] via the Python API.

[1] The Network Simulator-ns-2
[2] Riverbed Modeler
[3] mininet.link.TCLink Class Reference

Host: emulates a linux computer. Is simply a shell process moved into its own namespace, from where commands can be called. Each host has its own virtual Ethernet interface.

Switch: software OpenFlow switches provide the same packet delivery semantics that would be provided by a hardware switch.

Controller: Mininet allows to create controller within the same emulation or to connect the emulated network to an external controller running anywhere there is IP connectivity with the machine where Mininet is running.

Figure 4.1 shows the elements in a simple network with 2 hosts connected to one switch. The fact that each host and each host are inside a Linux namespace with their own virtual Ethernet interfaces allow to use networks analysers, such as Wireshark[4], snnifing in each of the interfaces within the emulated scenario.

The topology script used for this project is shown in Appendix B

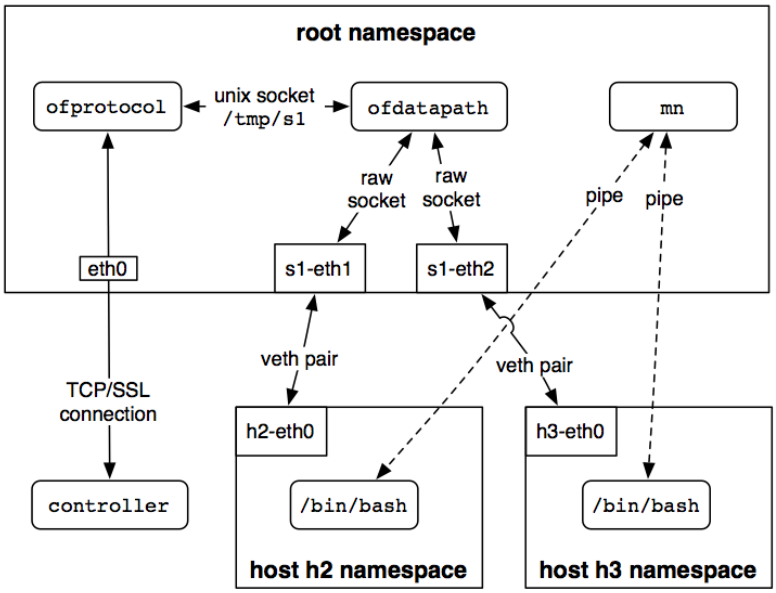

Figure 4.1: Mininet internal architecture with 2 hosts, 2 links, 1 switch and 1 controller (image from openflowswitch.org)

4.2 Floodlight Controller

Even though, as exposed in section 2.1.3, Ryu has an advantage in front of other controllers due its compatibility with higher versions of OpenFlow (up to v1.4), from the point of view of the author of this project, it has some drawbacks. The principal shortcoming is that it is not so well documented as others are, since it is a younger project, and its developer

[4]Wireshark network protocol analyser

community it is not active as others are. For those reasons, altogether with the language in which is written, the author of this project decided to choose **Floodlight Controller**.

Floodlight is an OpenFlow controller altogether with a collection of applications built on its top. Figure 4.2 shows the diagram of the controller. The controller by itself contains a bunch of interfaces which keep track of the state of the network. It also provides a set of services which can be used from the extensible modules.

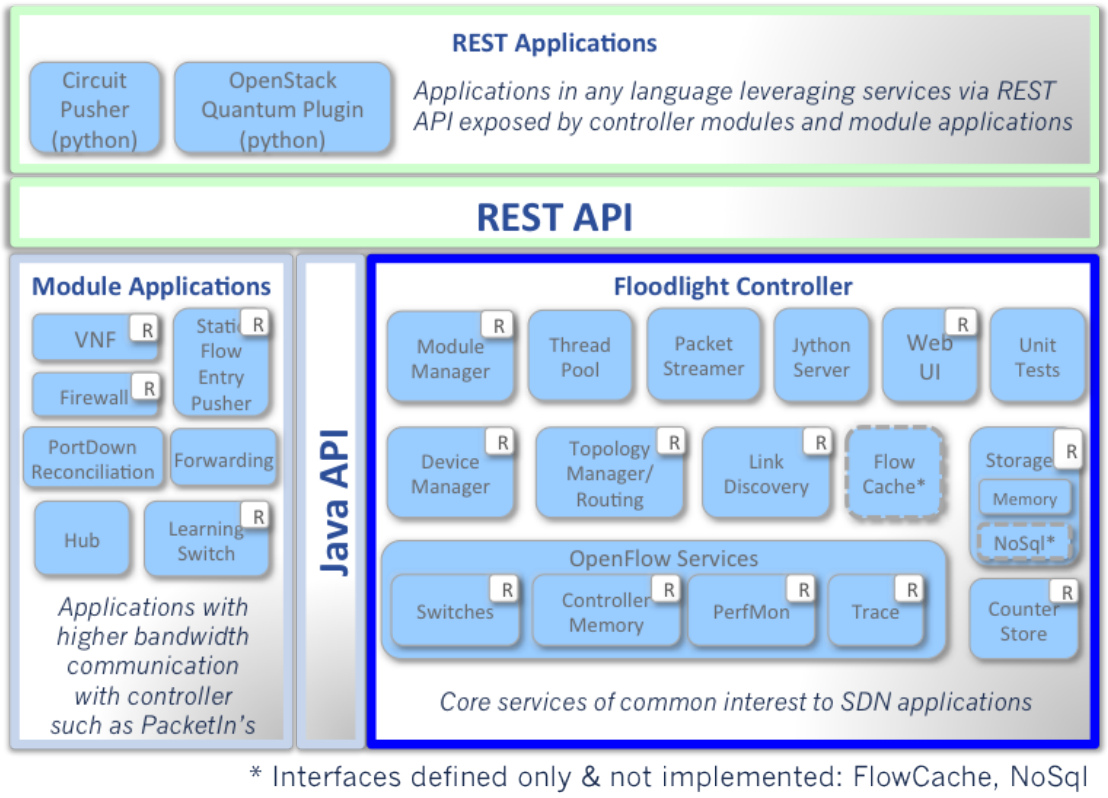

Figure 4.2: Floodlight diagram (image from www.openflowhub.org/)

The controller core has a Java Application Programming Interface (API) to expose the controller functionalities, consenting the expansion of the controller functionalities. Thus, the network manager can adapt the behaviour of the network by writing a custom module. Furthermore the Java API, Floodlight also provides a REST API, therefore any REST applications, written in any language, can retrieve information and invoke services by sending http REST commands to the controller REST port. This decouples services from the controller, enabling to write software applications which can interact with the network, without affect the controller core.

The main modules concerning to this project are explained bellow.

- Controller modules

  Link Discovery: this service is used internally to discover all the connections (links)

in the network using Link Layer Discovery Protocol (LLDP) , and keep the status of each of them. The update operations it handle are: link update/removed, switch updated/removed, port up/down and Tunnel port added/removed.

Topology Manager: module responsible of maintaining topology information. All the information about the current topology is stored in an immutable data structure called the topology instance. If there is any change in the topology, a new instance is created and the topology changed notification message is called. This module is also responsible of finding routing in the network using Dijkstra algorithm to find the shortest path between source and destination. It has an interface that can be implemented by other modules in order to listen for changes in topology.

Device Manager: track the information about the devices (hosts) which are connected to the switches that are under the controller domain. The way how devices are learned is through PacketIn (messages that arrive to the controller), storing their MAC address, VLAN and the switch to who is connected (attachment point).

- Application modules

  Static Flow Pusher: this module allows to insert flows entries into an OpenFlow switch. OpenFlow supports two methods of flow insertion: proactive and reactive. Reactive flow insertion occurs when a packet reaches an OpenFlow switch without a matching flow. The packet is sent to the controller, which evaluates it, adds the appropriate flows, and lets the switch continue its forwarding. Alternatively, flows can be inserted proactively by the controller in switches before packets arrive. Static Flow Pusher is generally useful for proactive flow insertion.

In order to achieve the goal of this project, a new application module have been created, which makes use of the modules explained above. Thus, it is able to analyse the traffic of the network altogether with different events that occur (such as new flows or failures of links or switches), and adapt the behaviour of the network (such as reroute a specific flow through a different route).

4.2.1 Implementation

In order to implement the algorithm into Floodlight controller, a new module has been designed.

This module is developed in Java, within the code of the controller and added to the load system in the startup of Floodlight, as explained in Appendix A.2.1. One of the purpose of the implementation is to provide scalability, for this reason, the data structures are defined as hash maps, what means that, if there are not collisions, the search, insertion and delete times will be constant ($O(1)$).

A listener for incoming $Packet-in$ messages have been deployed in the controller to detect

new packets which are type IP (0x800) has the property NO_MATCH, which will trigger the Algorithm 1. The packet header is saved as an OpenFlow match, to characterise the flow. At the same time, the IPs of the source and destination are saved. Each flow is saved following the structure described on 3.2.1.

Floodlight collects the information about the links using the Link Layer Discovery Protocol (LLDP)[5], and exposes the service through the Java API. A listener using this API has been implemented to be aware of the topology changes that occurs on the network. When an event involving the topology happens, this listener will be triggered, and it will update the links and switches that have suffer any changes. Thus, the system adapts to the conditions of every instant.

After detecting the new flow event, and using the information gotten through the topology listener, a graph is built to find all the possible paths between the source and the destination. The search has been implemented using a Depth-first search (DFS) algorithm, which, for each possible path found, checks if already exists on the paths collection, if it doesn't it inserts it.

Afterwards, the route with highest free capacity is selected to route the flow.

Floodlight provides an API called *Static Flow Pusher* to generate the messages to send to the switches to add static flow entries. To use it is needed to define a name of the flow per each switch (that must be unique), a flow entry (i.e. matching header fields and action for this flow), and the identifier of the switch. So, what the module does is to iterate per each element in *Hops*, gets the switch ports connections used from *Links*, and sets the action as a output according to that data. Furthermore, it uses the same data, but turning the input and output switch ports, to write another flow entry with the opposite direction, ensuring that the flow traffic will be carried through the same path.

Algorithm 1 shows the algorithm behaviour when a new flow is detected.

```
if new flow detected then
    OpenFlowMatch ← header fields that identify the flow;
    endPoints ← source and destination of the new flow;
    if endPoints is not in PathsCollection then
        search all the paths between endPoints;
        add each possible path to PathsCollection;
    end
    routeSelected ← path with highest capacity available;
    for each hop in routeSelected do
        add OpenFlowMatch to forward two-way the flow;
    end
end
```

Algorithm 1: New flow procedure

In order to get the information about capacity used the developed module has a separated

[5]IEEE 802.1AB (LLDP) Specification

thread which, sends a *controller-to-switch* message to each of the switches, to get the counters (see Table 2.2) that provide the data necessary to calculate de bandwidth usage, and update the $UsedCapacity$ of every $Flow$ and the $FreeCapacity$ of every $Path$.

Since the counters provides the total amount of bytes transmitted, is needed to compute that data to get the bandwidth used per each of the flows and links. To do so, and to avoid a large amount of traffic between the switches and the controller, the function is called every second, updating the use of every flow and every switch port in the network. Another reason to don't do the stats request more often is because the values obtained are not always as accurate as they should be, so doing the average of shortest periods of time provokes imprecise measurements that cause the rerouting of the flows when they should not.

Those measurements, and the fact that are done every second, generate the principal shortcoming of the implementation, which is that the rerouting process is done just once per second, causing delays since congestion is detected until the moment that the controller sends the new flow entries to the switches.

The available capacity of each path is computed taking the information of all the switch ports involved on it, and taking the minimum value (counting received and transmitted bytes).

When the value of free capacity of any of the ports reach 0, Algorithm 2 is executed .

```
if congestion detected then
    CongestedPort ← switch port which reach capacity threshold;
    Flows ← HashMap of all the active flows;
    FlowToReroute ← null;
    for each flow in Flows do
        if flow is using CongestedPort & flow used bandwidth < FlowToReroute
        used bandwidth then
            FlowToReroute ← flow;
        end
    end
    PC ← all possible path for FlowToReroute;
    Path ← null;
    PendingFlow ← null;
    for each path in PC do
        if path free capacity > FlowToReroute used bandwidth then
            reroute FlowToReroute through this path;
            Algorithm ends;
        end
    end
    if path capacity > Path capacity then
        Path ← path;
        SFC ← flows using this path with used bandwidth < FlowToReroute used
        bandwidth;
        for each flow in SFC do
            if flow used bandwidth < PendingFlow used bandwidth then
                PendingFlow ← flow;
            end
            else
                remove flow from SFC
            end
        end
        if PendingFlow used bandwidth + Path free capacity > FlowToReroute
        used bandwidth then
            reroute FlowToReroute to Path;
            Algorithm ends;
        end
    end
    else
        remove path from PC
    end
end
```

Algorithm 2: Congestion detection procedure

4.3 Topology

In order to test the behaviour of the algorithm on the network with real traffic, the topology shown in Figure 4.3 has been implemented on Mininet. The aim of this topology is to have three different paths from source to destination, with different number of hops and capacities.

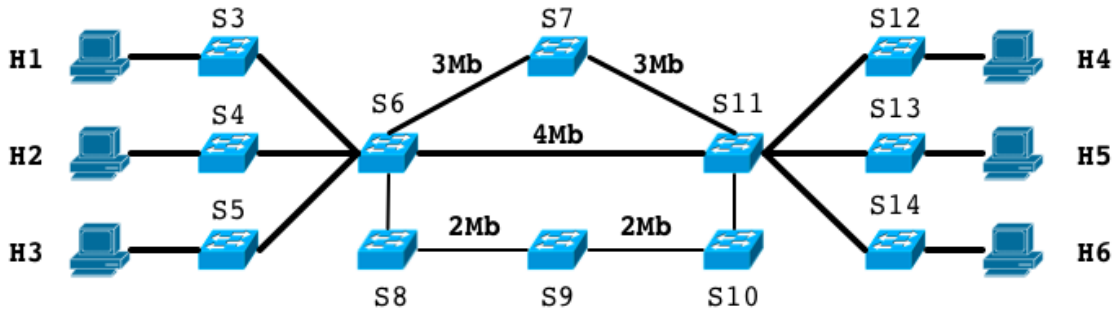

Figure 4.3: Topology used.

5 Emulation

This chapter shows and explains the results obtained with the scenario proposed in the previous chapter. The analysis consist of end-to-end measurements of throughput, latency, jitter and losses for UDP and TCP connections with different traffic patterns. A first point with analysis of the default behaviour of the controller is also included. All the measurements have been done generating real traffic.

5.1 Floodlight default behaviour

The first thing to be evaluated is the default behaviour provided by the Floodlight controller. When a packet arrives to a switch, and the header of this doesn't matches with any of the flow entries that the switch has, the packet is sent to the controller, this analyse the header, taking the source and destination fields of the Ethernet header (MAC addresses) of the incoming message. These addresses are used by the forwarding module, which, using the topology information learned by the controller, computes the Dijkstra algorithm to select the shortest path between them, and routes the flow through that route. As a result, even if we have parallel routes, all the flows will go through the same path.

To analyse the consequences of routing based on shortest path, three different flows (of 4, 3 and 2 Mbps, at constant rate) are generated from one side (Hosts 1, 2 and 2) and received by the other side (Hosts 4, 5 and 6). Figure 5.1 shows the received rate of each of the flows, and Figure 5.2 the total throughput.

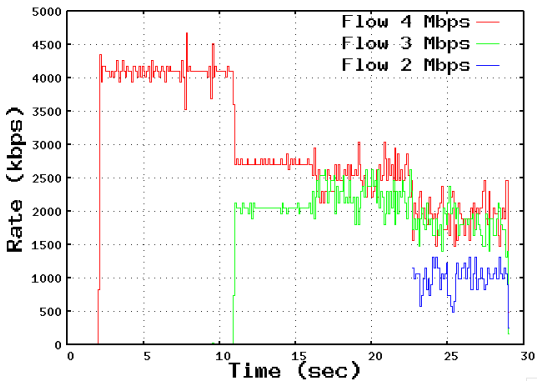

Figure 5.1: Throughput per flow

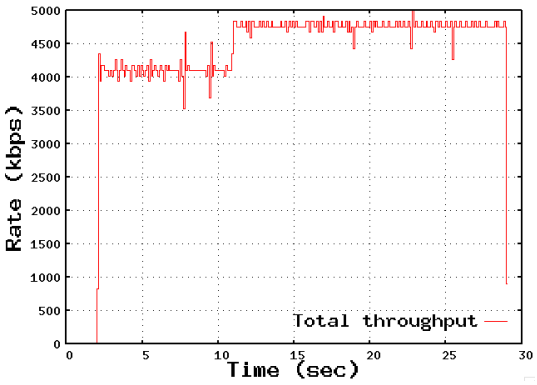

Figure 5.2: Total throughput

As shown in the previous graphics, the performance achieved from the point of view of throughput is poor, due to a bad resource utilisation. As the link gets congested, the delay increases significantly, arriving to almost 1.8 seconds delay, as shown in Figure 5.3. Additionally, the packet losses in this case is about the 50% of the paquets (Figure 5.4).

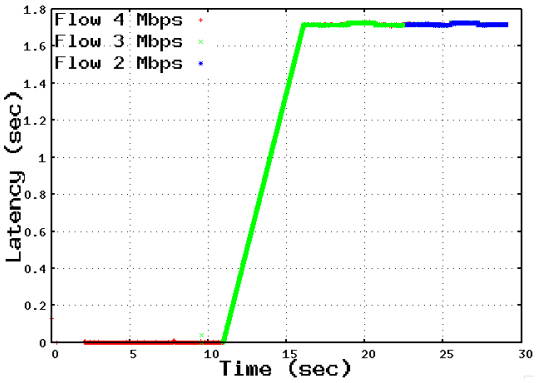

Figure 5.3: Latency

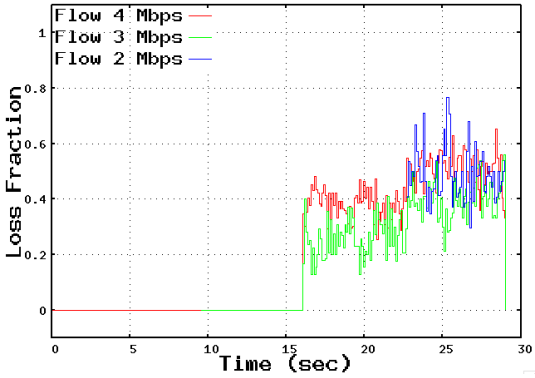

Figure 5.4: Packet losses fraction

In addition to these results about the problems regarding the carried traffic, there is also a deficient resource utilisation, as there are other possible routes that are not being exploited, which means that some resources are wasted.

The results presented in this section shown the problem that the proposed algorithm faces. In the next points the performance of the implementation are explained altogether with the drawbacks and advantages.

5.2 New flow routing

As explained in Chapter 3, the algorithm can be divided in two parts. This section analyse the effects of the first one, that is the establishment of the route for a new flow. Figures 5.5 and 5.6 shown the throughput per flow and total respectively.

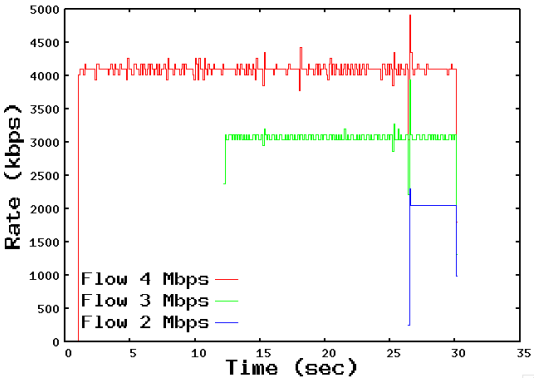

Figure 5.5: Throughput per flow

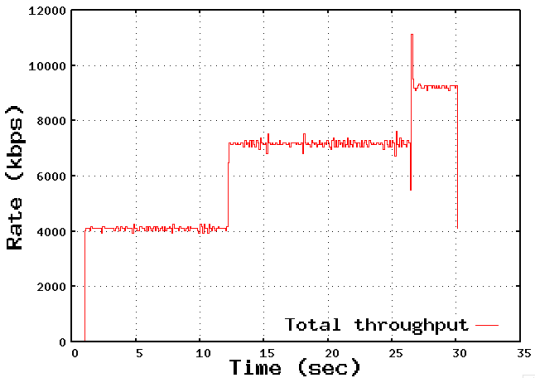

Figure 5.6: Total throughput

In this case the flows have been set to arrive in descendant order (referring to their rate), thus the first flow (represented in red colour) is routed through the highest capacity path, using all its capacity. At $t = 12$ the second flows arrives (green), and, after analysing the free capacity of each of the possible paths to reach its destination, is routed to another path. The same happens when the third flow (blue) arrives ($t = 26$).

The illustrations shown how using this routing decision has improved the throughput, as it was expected. However, what must be evaluated is the effect of that decision. Figures 5.8 and 5.8 shown the latency and the jitter respectively, and we can see how with the arrival of the third flow, there is peak of latency, up to $26ms$ and a couple of packets with an inter-arrival time of $20ms$. these values are within an acceptable boundaries for all kind of connexion (i.e. VoIP or web browser). There are no packet losses.

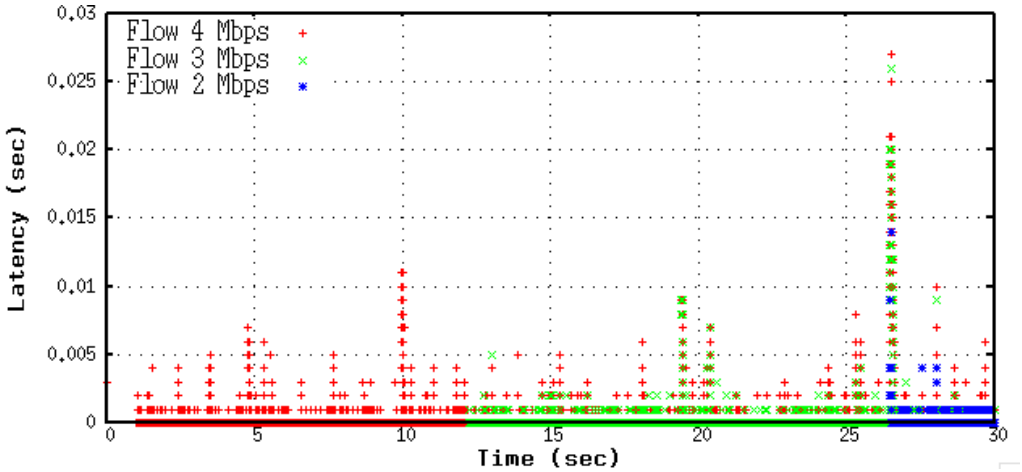

Figure 5.7: Latency of new flow process

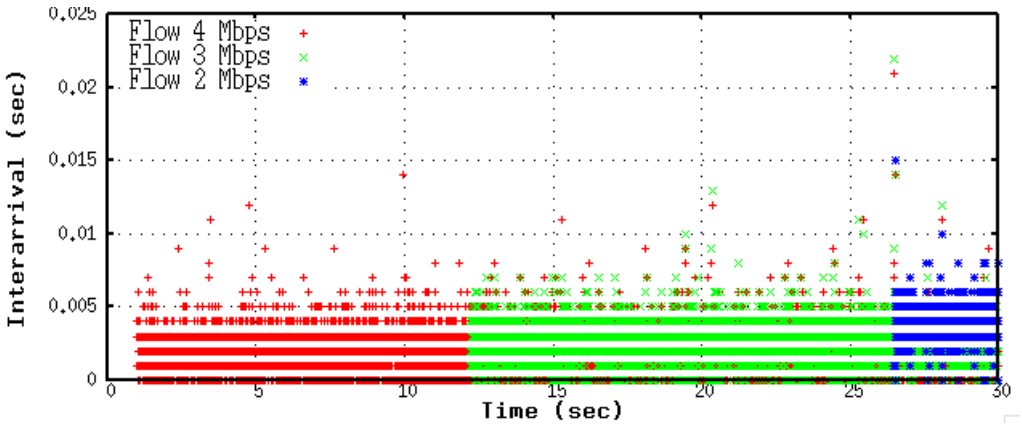

Figure 5.8: Jitter of new flow process

The displayed graphics have been measured using UDP, although the same test has been done using TCP and the results are practically the same.

5.3 UDP Traffic

In this section some tests using User Datagram Protocol are presented.

5.3.1 Simple rerouting

The first thing to be evaluated is what happens during the rerouting process. To do so, two flows are used, one with 2 Mbps bandwidth and the other with 4 Mbps. The smallest one is launched first, and, following the behaviour of the algorithm, it is routed through the path with higher free capacity, which, at initial conditions, is the central link of the topology (see Figure 4.3) which has 4 Mbps capacity. When the second flow arrives, follows the same procedure, but at that time ($t = 8$) the path with bigger value of free capacity is the path of three Mbps, therefore, this flow is initially routed through this path. Figure 5.9 shows the throughput of both flows described and Figure 5.10 the total throughput.

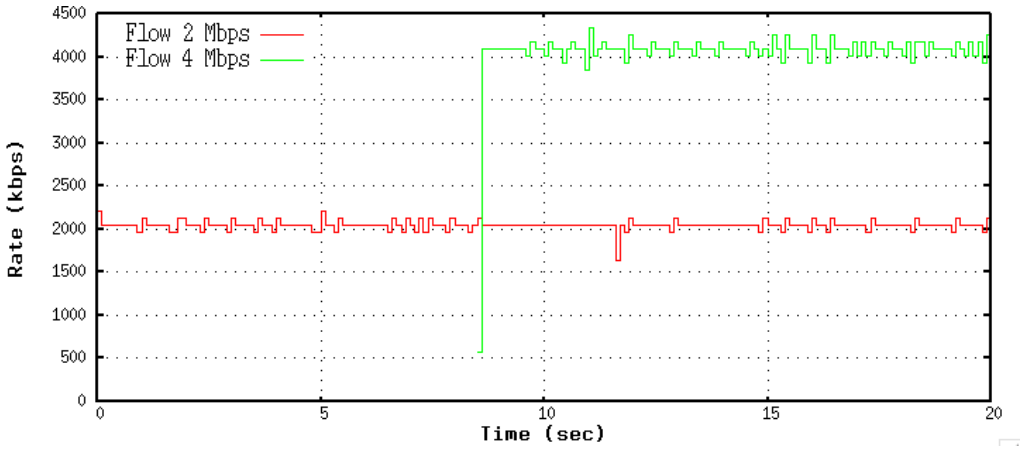

Figure 5.9: Throughput per flow in UDP rerouting

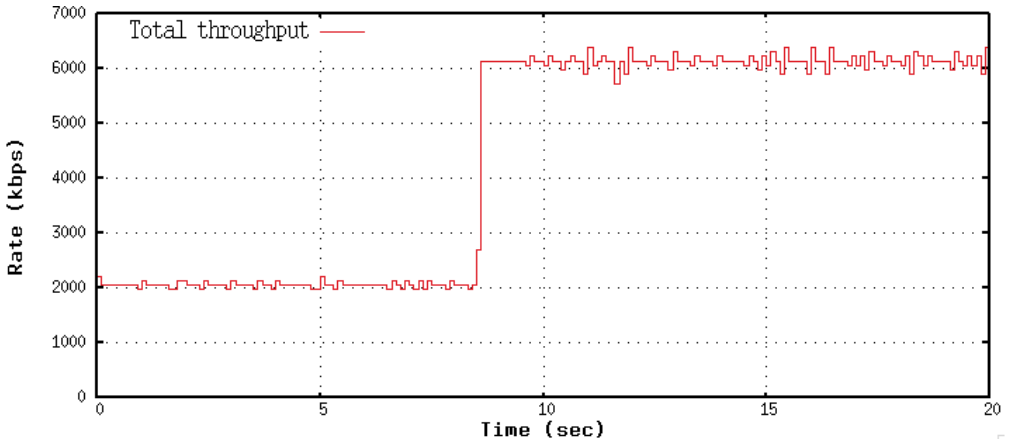

Figure 5.10: Total throughput of UDP rerouting

As explained in Section 4.1, Mininet emulator allows us to capture the traffic that is going through every interface of the emulated network. This feature has been used in order to analyse the resource utilisation along the network. Figure 5.11 shows the traffic carried by each of the parallel routes that the used topology has. Notice that the time axis is slightly different from the other graphics (about 2 seconds), since this measurement has been conducted using another software (Wireshark). This graphic shows the behaviour explained before, and we can see how the first flow is routed through the route with highest capacity (green line), and, at $t = 10.5$ the second flow arrives and it is carried by the 3 Mbps route (red line). As explained in the Floodlight implementation, the measurement of the bandwidth is done every second. For that reason, there is a delay since the congestion is produced and the rerouting is done. So, at $t = 12.5$ we can see how the traffic going to this route is redirected to the highest capacity path, and, one second later, the 2 Mbps flow is rerouted to the 3 Mbps path, since it corresponds to the path with biggest value of free capacity at that time.

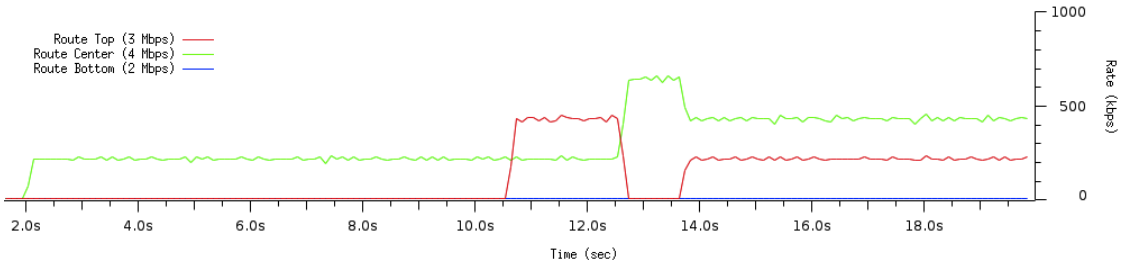

Figure 5.11: Resources utilisation in UDP rerouting

Beyond the throughput, there are other parameters which are important to decide if this algorithm meets the requirements for all kind of traffic demands. These parameters are the packet losses, the latency and the Jitter (delay variance). First we analyse the packet losses, shown in Figure 5.12.

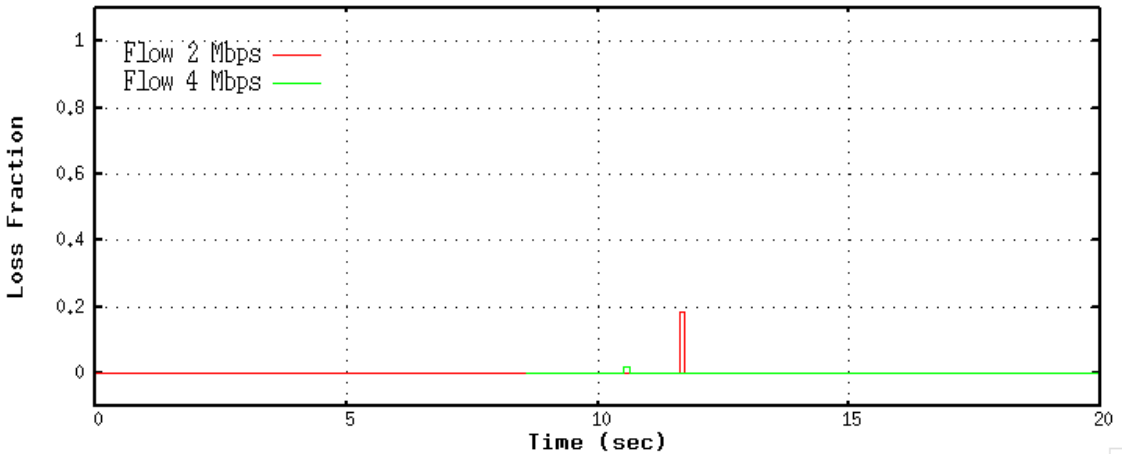

Figure 5.12: Packet losses in UDP rerouting

The previous figure shows the percentage of packet losses, calculated in fraction of 100 ms. The results shown how the losses are pretty low. For the biggest flow there is about a 2% for 2 samples (i.e. 200 ms), and for the smallest one are 18.5% (also 2 samples). Taking in account the speed of each flow, and knowing that the packets transmitted have a length of 1024 bytes, the losses per each flow are about 2 and 9 packets respectively. The reason of those losses are due the time that it takes to write the new flow entries to the switches. The reasoning behind the highest losses in the second route is because the number of switches involve in the new path is bigger, therefore the time that takes to rewrite the flow entries it is longer.

Regarding the latency and the jitter introduced by the process of rerouting, it can be seen that the effects are almost priceless, and these values are within an acceptable boundaries, as shown in Figure 5.13 and Figure 5.14.

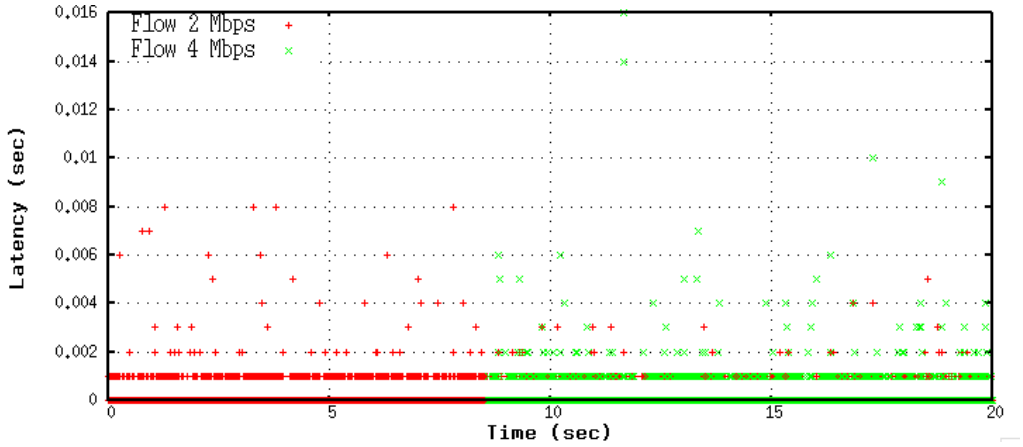

Figure 5.13: Latency in UDP rerouting

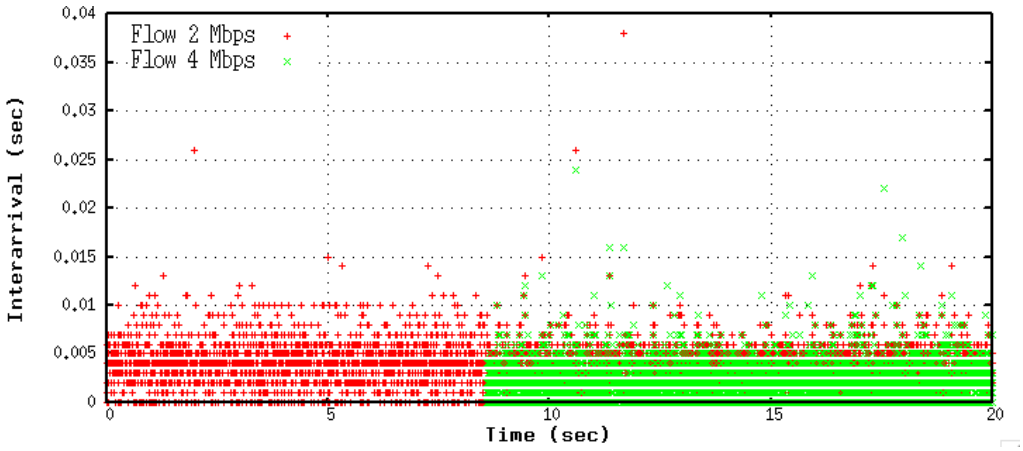

Figure 5.14: Jitter in UDP rerouting

5.3.2 Multiple routing

With the purpose of testing the performance of the applied algorithm in a more realistic way, three different flows with changing bandwidth utilisation have been set up. These flows follow the pattern shown in 5.1.

Table 5.1: UDP Flows pattern

		Mbps							
		0	**5**	**10**	**15**	**25**	**35**	**45**	**t**
FLOW #	**1**	2	2	2	4	3	2	3	
	2	0	3	3	2	4	4	2	
	3	0	0	4	3	2	3	4	

Figure 5.15 shows the throughput of each of the flows. From $t = 15$ to the end, the total throughput is is equal at the maximum capacity of the network (9 *Mbps*), alternating the capacity of each of the flows between 2, 3 and 4 *Mbps* at $t = 25$, $t = 35$ and $t = 45$. From this graph, altogether with Figure 5.16 which shows the total throughput, it can be seen that during the bandwidth transitions of the flows, the system is working perfectly.

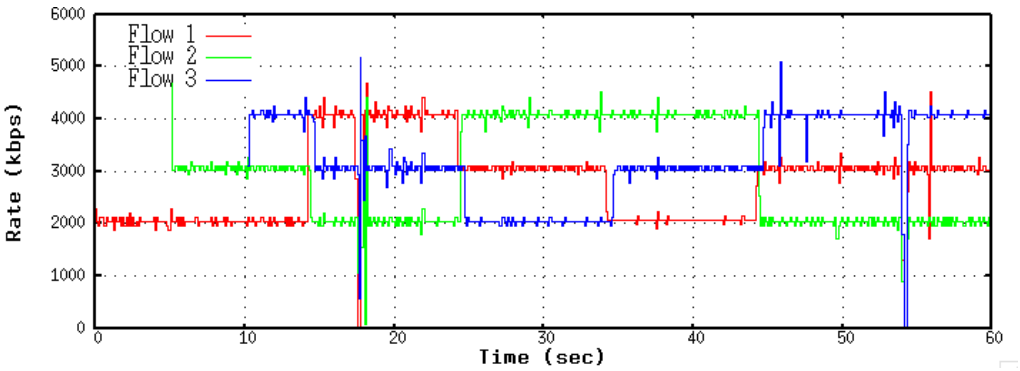

Figure 5.15: Throughput per flow in UDP multiple rerouting

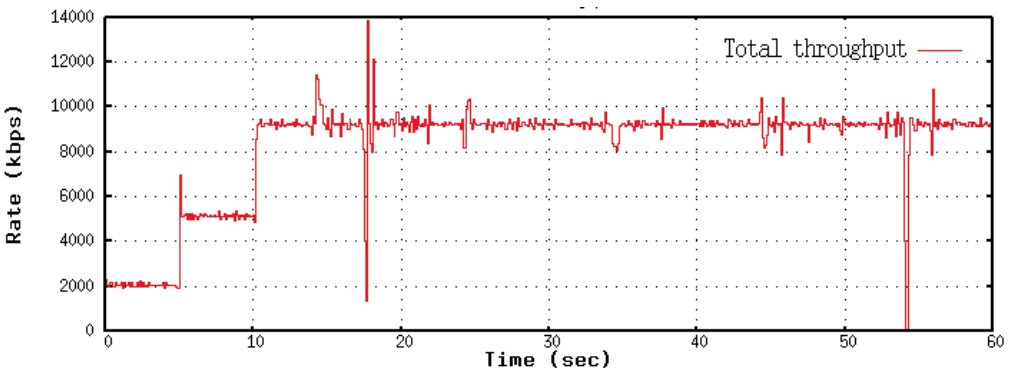

Figure 5.16: Total throughput of UDP multiple rerouting

However, the performance presents some fluctuations at instants where the system is supposed to be in a steady state ($t = 19$ and $t = 55$). Analysing the resource utilisation (shown in Figure 5.17) it can be seen that those throughput fluctuations are owing that the flows have been rerouted. Such event is due an inaccurate measurement from the statistics thread that has been implemented. The fact of reading the stats every second, and calculate the average of that period, is not enough precise to achieve a good performance when the bandwidth used is so close to the threshold that the congestion requires, and reducing the time between stats requests will be translated in a high computational cost altogether with a large amount of traffic between the controller and the switches, which will lead to a scalability problems.

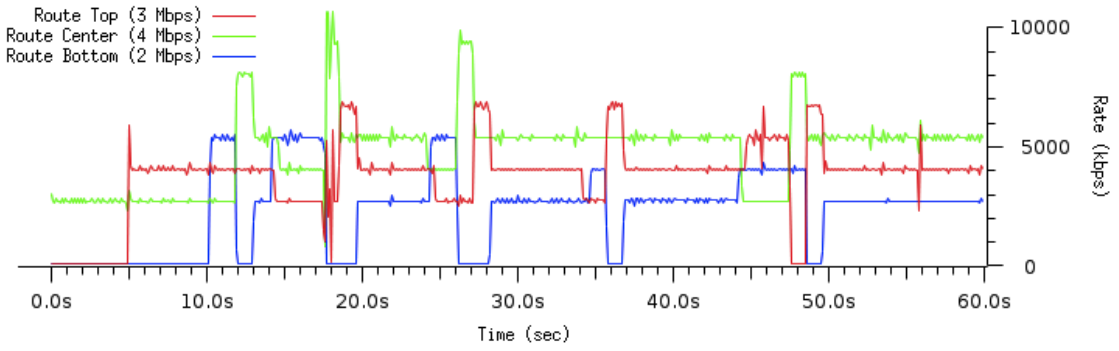

Figure 5.17: Resources utilisation in UDP multiple rerouting

Considering the results obtained, those undesired fluctuations that produce the unnecessary rerouting of the flows, are causing a peak of losses, as it is shown in Figure 5.18, where it can be seen how the two highest peaks of losses are placed at the times that those variations happens. In the first fluctuation ($t = 19$) there is a peak with almost 100% of losses of $flow$ 1, and in the second ($t = 55$) $flow$ 3 has a 40%. In both cases the flow which is most affected is the one which is carrying less traffic. That is due the behaviour of the algorithm, that always tries to move the lightest flow.

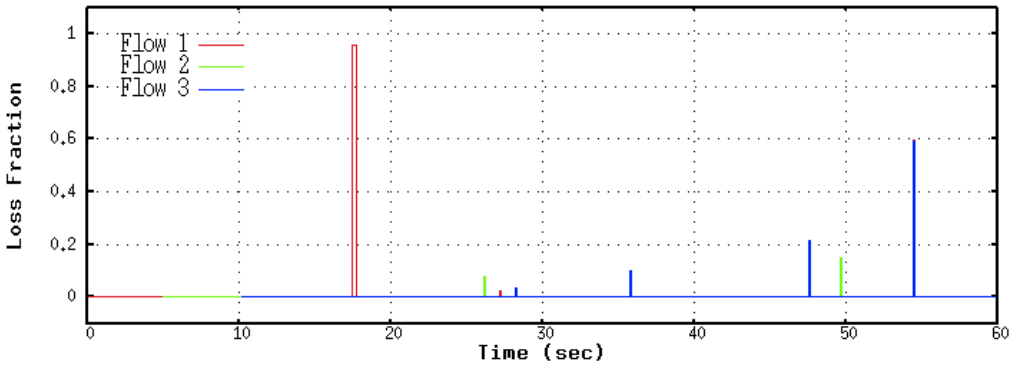

Figure 5.18: Packet losses in UDP multiple rerouting

As regards the delay introduced by the routing process (Figure 5.19), it can be said that is almost negligible, and, as same that for the packet losses commented before, and the jitter (showed in Figure 5.20), it just presents some problems when there are inaccurate measurements that cause the rerouting of the flows when it is no needed. Nonetheless, the values during the others instants are good enough.

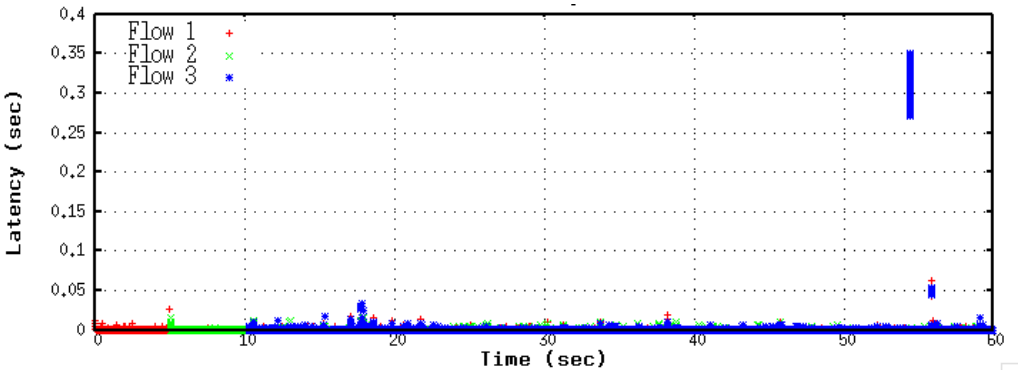

Figure 5.19: Latency in UDP multiple rerouting

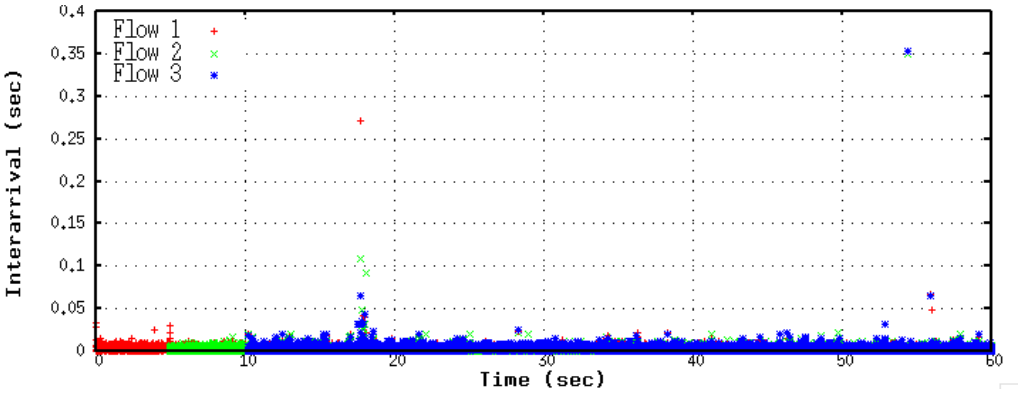

Figure 5.20: Jitter in UDP multiple rerouting

Besides the discussed fluctuations, the system presents a good behaviour with UDP traffic to achieve a high-performance utilisation of the resources, while maintaining the quality of the communication. The packet losses in the rerouting process are below the 20% and for a short time (between 100 and 200 ms), the latency is maintained behind the 50 ms, as same as the jitter.

5.4 TCP Traffic

In this section some tests using Transmission ControlProtocol are presented. Unlike UDP, TCP provides reliable, ordered and error-checked delivery of packets, which means that for each packet sent, the receiver will send an acknowledgment (ACK) message to the transmitter, to let it known that it has received the packet correctly. If the ACK is not received, the packet is sent again. That is why is interesting to analyse the behaviour of the implemented algorithm with this protocol.

The analysis conducted is the same as exposed for UDP.

5.4.1 Simple rerouting

as it was done with the UDP analysis, the first test consists on generate a flow of 2 $Mbps$, which is rerouted through the highest capacity route, and once the flow is already in steady state, a second flow at 4 $Mbps$ is generated. Figure 5.21 shows the throughput of both flows, where can be seen that the reroute process (at $t = 16$ and $t = 17$) doesn't affect the throughput of the flows, keeping a continuous during all the time.

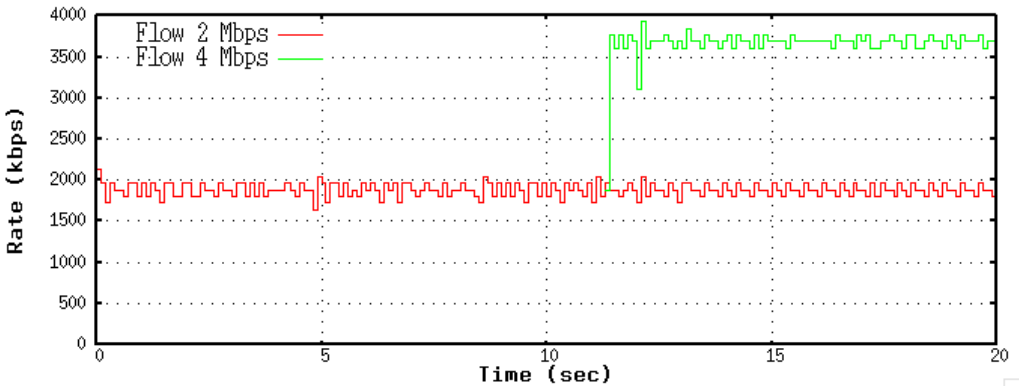

Figure 5.21: Throughput per flow in TCP rerouting

However, taking a look en the resources utilisation showed in Figure 5.22, the second flow, which, as predicted, at the beginning is routed through the path with 3 $Mbps$ capacity ($t = 11$), it took 5 seconds to the algorithm to detect the congestion (traffic above 3 $Mbps$). This show again the inaccuracy of the measurements with the method implemented.

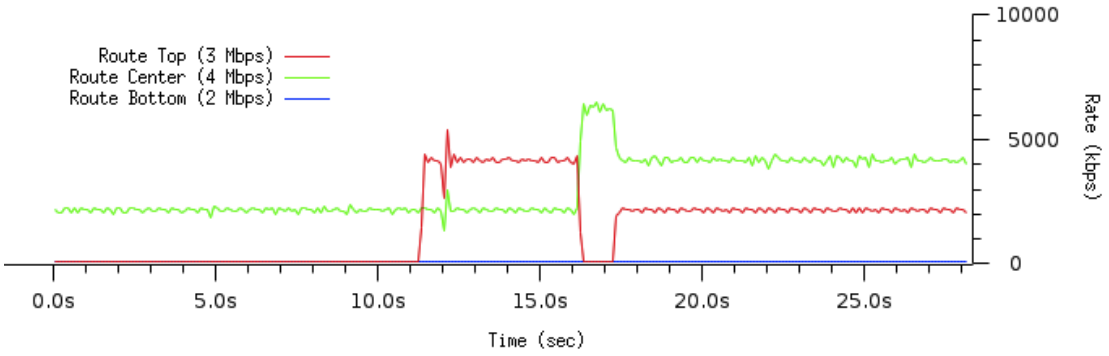

Figure 5.22: Resources utilisation in TCP rerouting

The result about packet losses after running this test is 0. Even though some retransmissions of sequences can be detected during the process of rerouting the flow, those does not affect the other quality parameters. In terms of latency, the delay introduced by changing the path of the routes is negligible, obtaining values below the 15 *ms*. The jitter is also maintained pretty stable in low values, below the 25 *ms* (Figure 5.24).

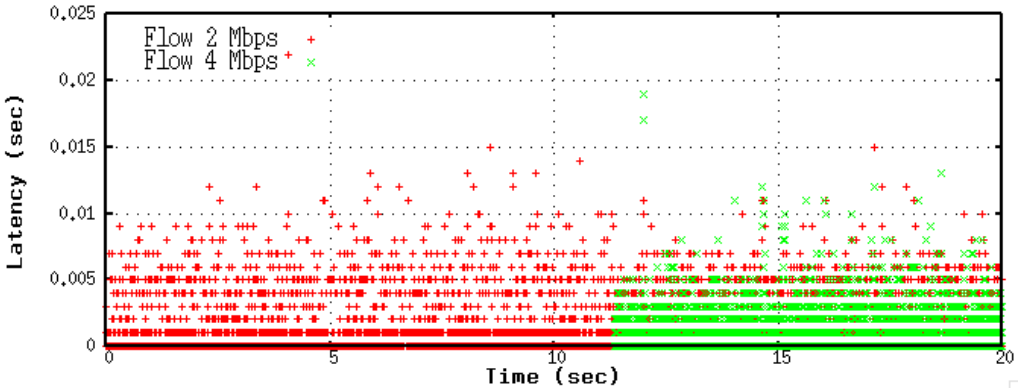

Figure 5.23: Latency in TCP rerouting

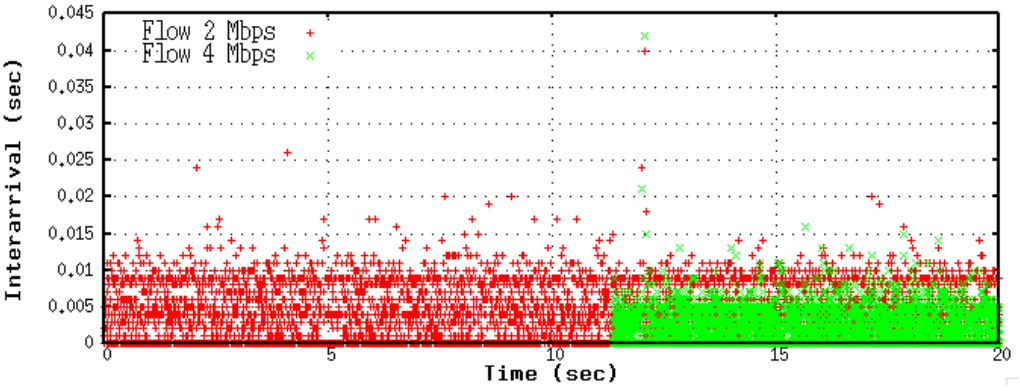

Figure 5.24: Jitter in TCP rerouting

5.4.2 Multiple routing

In order to analyse the behaviour with flows changing their bandwidth utilisation at the same time to compare the results with the ones presented in section 5.3.2 with UDP traffic, the same flow pattern has been set, an it is shown in Table 5.2.

Table 5.2: TCP Flows pattern

		Mbps							
		0	**3**	**6**	**11**	**21**	**31**	**41**	**t**
FLOW #	**1**	2	2	2	4	3	2	3	
	2	0	3	3	2	4	4	2	
	3	0	0	4	3	2	3	4	

The purpose of this pattern is to see the effects in two situations: when the 3 flows change their bandwidth ($t = 11$, $t = 21$ and $t = 41$) and when just two of them change and the third one keeps its value ($t = 31$). Figure 5.25 presents the throughput of each of the flows, and Figure 5.26 the total throughput.

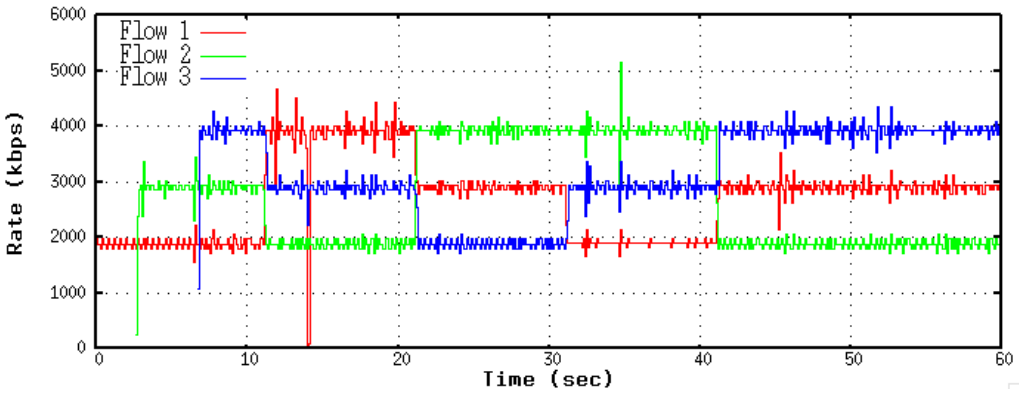

Figure 5.25: Throughput per flow in TCP multiple rerouting

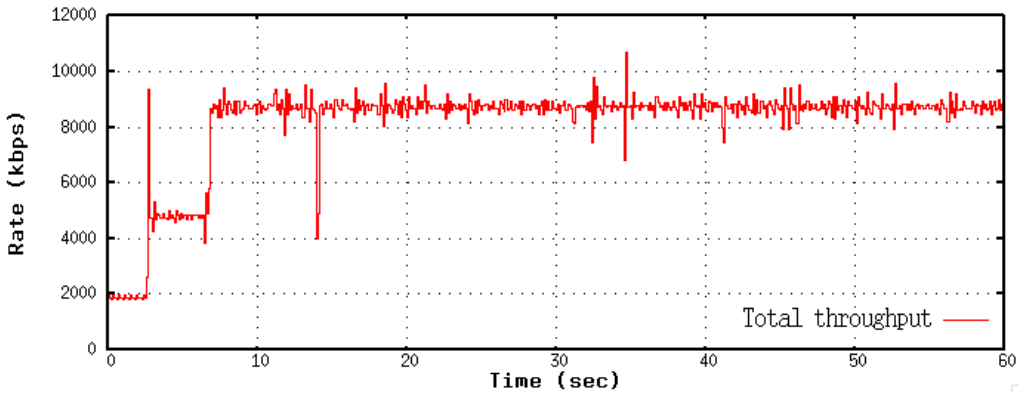

Figure 5.26: Total throughput of TCP multiple rerouting

The previous figures illustrate that the throughput is not affected by the changes of route of the flows, and even when the 3 flows are rerouted at the same time, the total throughput is maintained practically constant. However, analysing what is going on on each of the paths, it can be seen that there are some issues regarding the rerouting process. Since the measurements are done with one second granularity, when the flows change their bandwidth it took a while to measure it correctly again, what some errors in path selection. In those cases, the tendency is to reroute the flows through the highest capacity path, as can be seen in Figure 5.27. Nonetheless, after having the correct readings about the throughput of each flow (which take between 1 and 2 seconds), the flows are routed properly through the corresponding paths. Therefore, it is demonstrated that the algorithm is deciding the path correctly, and the problem in those fluctuations are due an not enough accurate measurements altogether with the fact that the rerouting event is executed in discrete time each second, what is translated in a longer time from the congestion detection to the rerouting process.

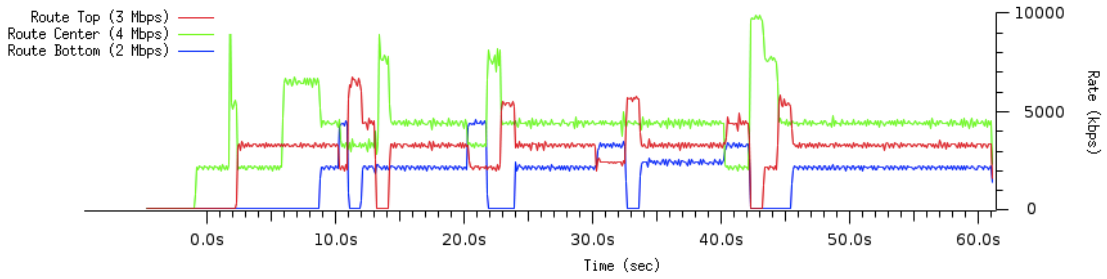

Figure 5.27: Resources utilisation in TCP multiple rerouting

The mentioned issues provoke some effects that can be detected on the latency and jitter analysis., as is shown in Figures 5.28 and 5.29 respectively. However, besides those uncertainty moment, the parameters are below the 50 ms, which is considered a good performance. Regarding the losses, once again are zero, due the TCP protocol is controlling it, and asking for the missed packets.

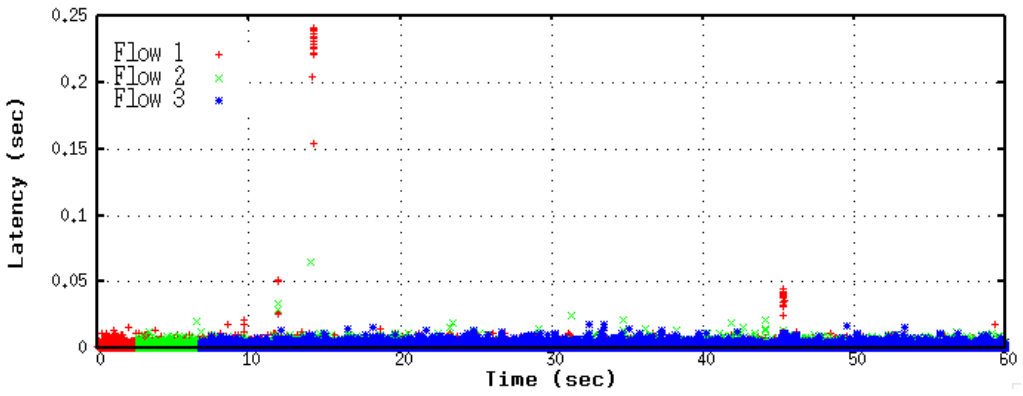

Figure 5.28: Latency in TCP multiple rerouting

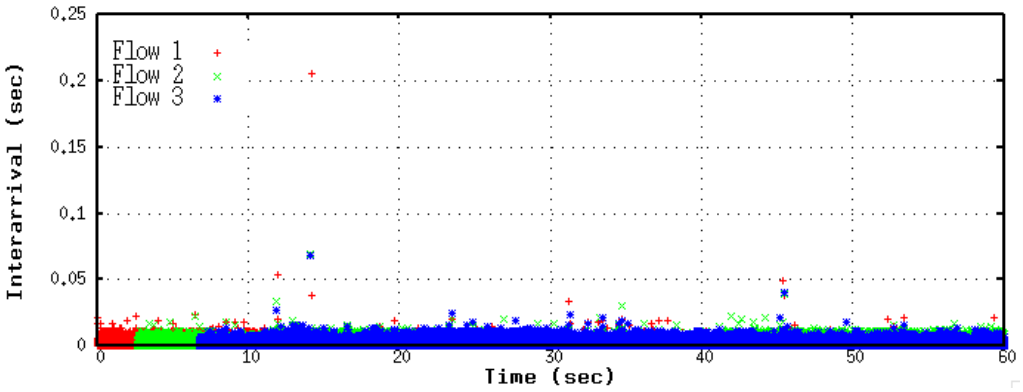

Figure 5.29: Jitter in TCP multiple rerouting

As this section shows, the behaviour of the presented implementation is practically the same with bot, UDP and TCP. The constant throughput during rerouting times is maintained, and the latency and jitter, except on a short time during routing conflict, below the 50 ms, which is below the threshold of almost any sort of traffic. Additionally, TCP has 0 losses.

6 Conclusions and Future Work

6.1 Conclusions

This project describe the implementation of a dynamic load balancing algorithm to distribute the different traffic flows carried by a network through the different parallel paths between source and destination. The implementation has been conducted over a Software-Defined Network, trying to explore the capabilities that this new paradigm of networking brings to us.

SDN provides a view of each of the elements of the network as well as control over them. Thus, allows a dynamic control of the actions to be done in each possible situation. Regarding the optimisation of resources, such control over the network adds a bunch of new functionalities, such as the dynamic routing proposed in this project.

Nowadays, OpenFlow is the most popular protocol for the southbound interface (i.e. to communicate the controller with the network elements), nonetheless, it presents some limitations. One of them is the monitoring of the network, which is involved in this project. OpenFlow, being a control-plane protocol has great difficult accurately determining data-plane measurements. The way to do this with OpenFlow entails making periodic flow stats requests to the switches on the network. The problem is that stats are never truly accurate. By the time they have been processed on the switch, sent over the network, and then processed by the controller they will be out of date.This is one of the problems showed in the emulation section. In addition to that, the stats requests generate a large amount of traffic (even though the traffic between controller and the switches is carried by a dedicated channel, it should be reduces as much as possible), and increase the computational cost in the controller to process these messages. It is because of that that in the implementation the stats are updated once per second, which leads to a delay in the rerouting process. As a conclusion about this point, data-plane measurements are best handled by third party applications, thus improving the results of the implementation.

In relation to the load balancing, the proposed algorithm works well for non-priority traffic, with the objective of reroute the traffic flows which are carrying less traffic. Based on that principle, it has been shown how it is possible to reduce congestion and simultaneously enhance network resource utilisation.

One of the most important points of the present project, was to evaluate the effect of redirecting the traffic flows while these keep transmitting information. As a conclusion about that, the results prove that the effects in terms of latency and jitter, as same as for packet losses, are almost negligible, even though they exist, but are maintained within an acceptable measure. The biggest problems are due fluctuations between the routes, but that is because of inaccuracy of the measures rather than a malfunction in the algorithm, and it could be avoided using a third-party application .

This project is an example of the unlimited possibilities that SDN introduces. Although SDN is still in a premature state, it is showing us the way how networking will be in a near future. Maybe it will be with another name, perhaps instead of the popular OpenFlow will be another protocol. What is sure is that SDN is doing a step-forward into networking, with a clear purpose: achieve a more simple, efficient, and flexible way to manage networks, and adapt the functionalities of several possible configurations in order to satisfy future needs

6.2 Future work

Currently the flows are determine by IPs and ports (source and destination). In order to carry out a more exhaustive classification, a flow could be determined by any of the header fields (i.e. VLAN ID, ToS bits or the size of the IP mask). Thus allowing a more specific treatment per each of them. In the same way, the ToS bits or other fields, could be used to determine the priority or needs of the flow, providing a guarantee for its QoS requirements (i.e. if one flow has as a destination a VoIP server or has a ToS of "low delay", use the shortest path or avoid rerouting to avert jitter or delay effects).

Even though the implementation has been designed to be adapted to any possible topology, a further analysis with different topologies of different sizes should be made so could be guaranteed that it has no limitations on that aspect. A deepest analysis with a larger number of flows, with different kinds of traffic patterns it is also needed.

The main problem of the implementation is due the inexactitude in the measurements of the traffic. Therefore, the next step will be to use a third-party application to carry out the measurements (such as NetFlow[1] or SFlow[2]). In order to use the external measures inside of the controller, an extension of the implementation via REST API will be the most suitable way. Furthermore, with an appropriate REST extension, part of the logic of the implementation could be done outside the controller, at application level. Therefore, the computational load on the controller will be reduced.

[1] Cisco Systems NetFlow Services Export Version 9 RFC 3954
[2] SFlow website

Moreover, the application of an API REST will provide an external way to interact with the functionality, for instance allowing to determine the maximum bandwidth usage per flow or per link from a web page. Offering such feature will raise the usability of the module, and make it more dynamic and more interesting for ISPs or other network operators.

Bibliography

[1] (2009). Openflow switch specification. Technical report, Open Networking Foundation.

[2] (2012). Inter-datacenter wan with centralized te using sdn and openflow. White paper, Google, Inc.

[3] Awduche, D., Chiu, A., Elwalid, A., Widjaja, I., and Xiao, X. (2002). Overview and principles of inter- net traffic engineering. RFC 3272 (Informational).

[4] Bakshi, K. (2013). Considerations for software defined networking (sdn): Approaches and use cases. IEEE.

[5] Curtis, A. R., Mogul, J. C., Tourrilhes, J., Yalagandula, P., Sharma, P., and Banerjee, S. (2011). Devoflow: scaling flow management for high-performance networks. ACM.

[6] Egilmez, H. E., Dane, S. T., Bagci, K. T., and Tekalp, A. M. (2012). Openqos: An openflow controller design for multimedia delivery with end-to-end quality of service over software-defined networks.

[7] Elwalid, A., Jin, C., Low, S., and Widjaja, I. (2001). Mate: Mpls adaptive traffic engineering. IEEE.

[8] Fang, S., Yu, Y., Foh, C. H., and Aung, K. M. M. (2012). A loss-free multipathing solution for data center network using software-defined networking approach. IEEE.

[9] Fortz, B. and Thorup, M. (2000). Internet traffic engineering by optimizing ospf weights. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 519–528.

[10] Jorge Crichigno, Joud Khoury, N. G. (2013). Routing in mpls networks with probabilistic failures. Technical report.

[11] Jose Marzo, Eusebi Calle, C. S. T. A. (2003). Qos online routing and mpls multilevel protection: A survey. Technical report.

[12] Katz, D. Kompella, K. Y. D. (2003). Traffic engineering (te) extensions to ospf version 2.

[13] Keping Long, Zhongshan Zhang, S. C. (2001). Load balancing algorithms in mpls traffic engineering.

[14] Khondoker, R., Zaalouk, A., Marx, R., and Bayarou, K. (2014). Feature-based comparison and selection of software defined networking (sdn) controllers. In *International Conference on Computer Software and Applications, 2014 Proceedings of.*

[15] Lantz, B., Heller, B., and McKeown, N. (2010). A network in a laptop: rapid prototyping for software-defined networks. ACM.

[16] Reitblatt, M., Foster, N., Rexford, J., and Walker, D. (2011). Consistent updates for software-defined networks. ACM.

[17] Sushant Jain, Alok Kumar, S. M. (2013). B4: Experience with a globally-deployed software defined wan. Technical report, Google Inc.

[18] Xiao, X., Hannan, A., Bailey, B., and Ni, L. M. (2000). Traffic engineering with mpls in the internet. *Network, IEEE*.

A Setting up Mininet and Floodlight

In order to simulate the scenario with SDN, this project uses Mininet[1] software, which is developed by Stanford University and released under a permissive BSD Open Source license, to simulate the network. And Floodlight[2] as a controller of the network. Others software tools used are Virtual Box[3] to run the Mininet virtual network and Eclipse[4] as a IDE. This appendix explains the steps needed in order to setting up the environment needed for the study.

A.1 Mininet

1. The easiest and recommended way to setting up mininet is downloading a Virtual Machine (VM) image, which is provided in the following link: *Mininet 2.1.0*.
2. Open VirtualBox and create a new VM Ubuntu type and use the download file in the previous step as a virtual hard disk for the new VM.
3. Select "*settings*", and add an additional host-only network adapter so that you can log in to the VM image.
4. Run the new VM.
5. **Login:** mininet

   Password: mininet
6. Start Mininet with the command "*sudo mn*". In the case we want to load a specific topology and connect Mininet to the controller, we have to add some parameters to that command. To load the topology specify in the file *topo.py* and connect to the controller which is running in the host with the IP: *172.26.20.23* the command will be the following:

[1]Mininet website: http://mininet.org
[2]Floodlight website: www.projectfloodlight.org/floodlight
[3]VirtualBox website: www.virtualbox.org
[4]Eclipse website: www.eclipse.org

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

7. For further information about Mininet and how to use it go to: Mininet Walkthrough page.

A.2 Floodlight

Floodlight is an open-source software which source code is published on GitHub: github.com/floodlight/floodlight. However, in order to add new modules it is easier to use a Integrated Development Environment (IDE) such as Eclipse. The following steps are needed in order to download Floodlight and integrate it with Eclipse.

1. Type the following commands:

```
sudo apt-get install build-essential default-jdk ant python-dev eclipse
git clone git://github.com/floodlight/floodlight.git
cd floodlight
ant eclipse
```

2. Launch Eclipse
3. "File" → "Import" → "General" → "Existing Projects into Workspace" → "Next"
4. From "Select root directory" click "Browse" and select the parent directory where you placed floodlight. → click "Finish"
5. Create the FloodlightLaunch target:
   (a) Click "Run" → "Run Configurations"
   (b) Right Click on "Java Application" → "New"
      i. For "Name" use *"FloodlightLauncher"*
      ii. For "Project" use *"Floodlight*
      iii. For "Main" use *"net.floodlightcontroller.core.Main"*
   (c) Click "Apply"

A.2.1 Adding a module to Floodlight

In order to add a module with new functionalities, it is needed to change the default startup modules:

`src/main/resources/floodlightdefault.properties`
`src/main/resources/META-INF/services/net.floodlight.core.module.IFloodlightModule`

Adding the new class created.

B Mininet Topology Script

```
#!/usr/bin/python

#Allow to pass arguments
import sys

from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.topo import Topo
from mininet.node import CPULimitedHost
from mininet.util import dumpNodeConnections
from mininet.log import setLogLevel

from mininet.link import Link, TCLink

def emptyNet():

    "Create an empty network and add nodes to it."
    net = Mininet( controller=RemoteController, link=TCLink, autoSetMacs =
        True )
    info( '*** Adding controller\n' )
    net.addController( 'c0' , controller=RemoteController,ip=sys.argv[1],
        port=6633 )
    print 'connecting to:', sys.argv[1]

    info( '*** Adding hosts\n' )
    srcHost1 = net.addHost( 'h1' )
    srcHost2 = net.addHost( 'h2' )
    srcHost3 = net.addHost( 'h3' )
    dstHost1 = net.addHost( 'h4' )
    dstHost2 = net.addHost( 'h5' )
    dstHost3 = net.addHost( 'h6' )

    info( '*** Adding switch\n' )
    switch3 = net.addSwitch( 's3' )
    switch4 = net.addSwitch( 's4' )
    switch5 = net.addSwitch( 's5' )
    switch6 = net.addSwitch( 's6' )
    switch7 = net.addSwitch( 's7' )
    switch8 = net.addSwitch( 's8' )
    switch9 = net.addSwitch( 's9' )
    switch10 = net.addSwitch( 's10' )
    switch11 = net.addSwitch( 's11' )
    switch12 = net.addSwitch( 's12' )
```

```
44      switch13 = net.addSwitch( 's13' )
45      switch14 = net.addSwitch( 's14' )
46
47      info( '*** Creating links\n' )
48      srcHost1.linkTo( switch3 )
49      srcHost2.linkTo( switch4 )
50      srcHost3.linkTo( switch5 )
51      switch3.linkTo( switch6 )
52      switch4.linkTo( switch6 )
53      switch5.linkTo( switch6 )
54      net.addLink( switch6, switch7, bw=3 )
55      net.addLink( switch7, switch11, bw=3 )
56      net.addLink( switch6, switch11, bw=4  )
57      switch6.linkTo( switch8 )
58      net.addLink( switch8, switch9, bw=2 )
59      net.addLink( switch9, switch10, bw=2 )
60      switch10.linkTo( switch11 )
61      switch11.linkTo( switch12 )
62      switch11.linkTo( switch13 )
63      switch11.linkTo( switch14 )
64      switch12.linkTo( dstHost1 )
65      switch13.linkTo( dstHost2 )
66      switch14.linkTo( dstHost3 )
67
68      info( '*** Starting network\n')
69      net.start()
70
71      info( '*** Running CLI\n' )
72      CLI( net )
73
74      info( '*** Stopping network' )
75      net.stop()
76
77  if __name__ == '__main__':
78      setLogLevel( 'info' )
79      emptyNet()
```

Listing B.1: Topology Script