VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

LOAD BALANCING IN OPENFLOW NETWORKS

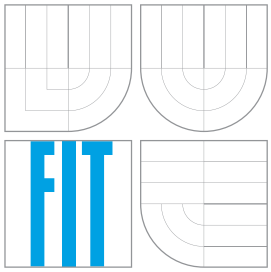
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR MARCINIAK

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VYVAŽOVÁNÍ ZÁTĚŽE V SÍTÍCH OPENFLOW

LOAD BALANCING IN OPENFLOW NETWORKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR MARCINIAK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MATĚJ GRÉGR

BRNO 2013

Abstrakt

Cílem této diplomové práce je vypracování nástroje pro vyvažování zátěže v sítích OpenFlow. Na příkladu protokolu OpenFlow jsou popsány základní principy software-defined networking (SDN) včetně srovnání s tradičním přepínáním a směrováním. OpenFlow je první protokol/API umožňující komunikaci mezi zařízeními na kontrolní a infrastrukturní vrstvě software-defined networking modelu. Jsou popsány klíčové funkce tohoto protokolu a je provedeno porovnání některých aktuálně dostupných OpenFlow kontrolérů. Dále jsou uvedeny techniky vyvažování zátěže v počítačových sítích používané v současné době. Následuje postup vývoje aplikace pro vyvažování zátěže včetně uvedení použitých testovacích prostředí - Mininet (SW) a OFELIA (HW). Je provedeno vyhodnocení výsledků vyvinuté aplikace včetně úvahy nad možnými rozšířeními programu.

Abstract

The aim of this thesis is to develop a load balancing tool for OpenFlow networks. Software-defined networking (SDN) principles are introduced (OpenFlow protocol used as an example) and compared to the legacy routing and switching technology. Openflow is the first protocol/API enabling communication between the control and infrastructure planes of the software-defined networking model. Key features of the protocol are described and several OpenFlow controllers are introduced. Current best practices in computer networks load balancing are discussed as well. The load balancing application development process is described including the test laboratory setups - Mininet (SW) and OFELIA (HW). The application test results are evaluated and possible further enhancements to the program are discussed.

Klíčová slova

OpenFlow, SDN, software-defined networking, vyvažování zátěže

Keywords

OpenFlow, SDN, software-defined networking, load balancing

Citace

Petr Marciniak: Load Balancing in OpenFlow Networks, diplomová práce, Brno, FIT VUT v Brně, 2013

Load Balancing in OpenFlow Networks

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Matěje Grégra

.....
Petr Marciniak
May 21, 2013

Poděkování

Touto cestou bych velmi rád poděkoval vedoucímu práce Ing. Matějovi Grégrovi za vstřícnost, ochotu mi pomoci, podněty a rady v průběhu řešení této diplomové práce.

Contents

Chapter 1

Introduction

1.1 Description of the problem

Load balancing is a matter of great concern in the computer networks field. Load balancing in computer networks may refer to both a) distributing the server load by directing the traffic to different servers providing redundant service, or b) directing the traffic across different paths within e.g. the backbone network in order to effectively utilize the provisioned bandwidth. Our task was to develop a load balancing solution using the OpenFlow protocol.

1.2 Motivation

The challenging topic of load balancing (especially within the backbone networks) seemed very interesting to me. Although, various techniques such as MPLS-TE (Multi-protocol Label Switching - Traffic Engineering) or route metrics manipulation already exist, it is the SDN (software-defined networking) technology, that is expected to revolutionize the way load balancing is performed in computer networks. Hence, working on this sort of next-generation solution using OpenFlow protocol, appeared to be an interesting project as well as a great opportunity for me to gain valuable hands-on experience of working with technology, which is said to become the next „Big Thing“ in computer networking.

1.3 Analysis of the problem

Firstly, the resulting application will be tested mostly in the virtual environment, because the OpenFlow is a relatively young protocol, it is not yet widely supported by the currently available networking devices (switches, etc.). Various OpenFlow controllers should be investigated, and the most suitable one for the application should be included in the virtual test network. The metrics to be taken into account in the load balancing scheme should be identified and the load balancing software should be developed accordingly. The application should then be tested in both virtual environment as well as on the currently available hardware.

Chapter 2

Current networking technology

In current „non-software-defined“ networking technology, it is the routing protocols (e.g. OSPF, BGP, EIGRP) or the Spanning Tree Protocol (when considering just Layer 2 switching) themselves, which are responsible for maintaining the network device's forwarding table. In such networks, networking devices are communicating with each other using one of these protocols mentioned earlier in order to share the information they „know“ about the network (e.g. reachability of network prefixes, link costs). Each device is than responsible for maintenance of its forwarding table (or FIB - Forwarding Information Base) according to the information stored in RIB (Routing Information Base), etc. Thus, it is ensured that no unwanted behavior such as routing loops occurs in the managed network.

2.1 Device complexity

Such an approach implies high complexity of the network devices used. Firstly, the devices have to maintain information about the complete network topology, when link-state routing protocols (e.g. OSPF, IS-IS) are used. Secondly, the devices have to be able to perform all the best path computations by themselves. The computations have to be performed in real time (in order to minimize the packet loss if e.g. one of the links goes down). What's more, it is usually more then just one instance of a routing/switching protocol, that is running on a device in production. Consider for example a router, that is connecting a company to its Internet Service Provider (ISP) - it would use BGP (Border Gateway Protocol) to communicate with the ISP and one (or more) of the interior gateway protocols (e.g. RIP, OSPF) for redistribution of received routes further into the company's network. Similar situation may be observed also in the Layer 2 switching. The switches may be running for example a Per-VLAN Spanning Tree Protocol (PVST) in order to maintain multiple instances of spanning tree for each VLAN (Virtual LAN).

The complexity of network devices is also one of the factors, which are driving the networking hardware prices up. The scalability may also become an issue, as it may happen, that the networking technology architects will not be able to design a reasonably powerful hardware for an acceptable price.

Bad utilization of provisioned network bandwidth may be seen as another downside of the current networking technology. This is mostly due to the fact, that the devices have to exchange network information back and forth with each other. However, that is probably not as big problem now as it might have been few years ago, as the current networking hardware offers much higher bandwidth ports, etc. for a reasonable price.

2.2 Management complexity

The device complexity described in the previous Section 2.1 causes not only high hardware requirements, but also increases the management complexity. In the „non-software-defined“ networking model, the managers of large networks running an extensive range of services have to be familiar with all the technology concepts being used. This is due to the fact, that this model offers almost no level of abstraction when it comes to management techniques. Thus, they have to be not only able to pick the right protocol for their task, but also must have a strong understanding of each device's operating system and architecture in order to configure the entire system correctly.

2.3 Advantages of the current networking technology

The „non-software-defined“ networking technology of course has its advantages as well. Most of the protocols have been in use for a long time now (for example first version of RIP was specified more than 20 years ago). Thanks to this fact, their behavior is well known, their implementation in the device's hardware and software has been enhanced and „cleaned“ over the years. Hence, such network's behavior and performance are highly predictable. When considering the network management, the problem of a single point of failure (as discussed further in Section 3.2) is eliminated, as every device is responsible for running it's own instance of the routing protocol and corresponding algorithm.

Chapter 3

Software-defined networking

In a nutshell: „Software Defined Networking (SDN) is an emerging network architecture where network control is decoupled from forwarding and is directly programmable. This migration of control, formerly tightly bound in individual network devices, into accessible computing devices enables the underlying infrastructure to be abstracted for applications and network services, which can treat the network as a logical or virtual entity. As a result, enterprises and carriers gain unprecedented programmability, automation, and network control, enabling them to build highly scalable, flexible networks that readily adapt to changing business needs.“ taken from [11]. Figure 3.1 illustrates the basic concept of Software-defined networking architecture.

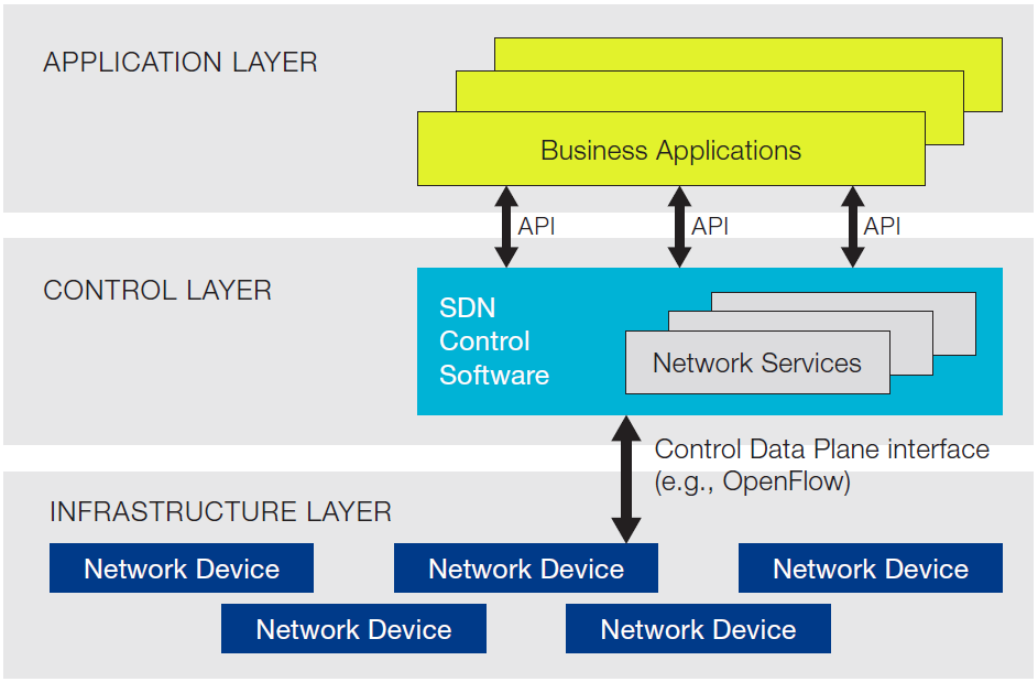


Figure 3.1: Software-defined network architecture [11].

In other words (to compare with Chapter 2) the SDN approach allows for much simpler network devices, as the networking intelligence is separated from the actual networking

hardware, whose main task now is just „simple“ forwarding. Hence, the devices are not responsible for updating of their forwarding tables/FIBs any more. It is the SDN controller's task instead.

The software-defined networking approach also offers a certain level of abstraction for management procedures thanks to use of the SDN controller. Therefore, it opens a way for much easier management of the network infrastructure, as all the important decisions/configurations are made from one place (the SDN controller) instead of network operators being entirely dependent on configuration of all devices separately using their CLIs (Command Line Interface), SSH (Secure Shell), etc. In SDN, network managers are required to have only basic understanding of network devices architectures and operating systems as it is the SDN protocol (e.g. OpenFlow), which is taking care of the devices direct configuration.

In order to provide better understanding of the software-defined networking concept, I will explain it using the Figure 3.1 going from the bottom up.

3.1 Infrastructure layer

The „simple“ networking devices are laying in the so called infrastructure layer. Their forwarding tables are updated centrally or in a distributed manner from the SDN controller (lying in the control layer). The devices theoretically do not have to maintain any knowledge about the network topology they are in. They might implement some of the routing protocols, but they are not required to. On the other hand, they are required to communicate with the SDN controller. The information sent to the controller may include the directly connected prefixes or values of counters (e.g. number of packets forwarded according to a specific rule in the forwarding table).

Such an approach supports lower pricing of the networking hardware, as all its computing power is needed only for the forwarding task.

3.2 Control layer

The control layer is where the SDN controllers are running. The SDN controller may be perceived as the brain in the software-defined networking concept. It is a software application, which is communicating with all the devices in the network. It collects information about the network (reachability of prefixes, forwarding rules counters, etc.) and creates/updates it's own model of the network based on the knowledge learned. The model is then used, when computing the best paths or ensuring load balancing across the network. The forwarding rules for each network device are then created accordingly and pushed to the devices' forwarding tables.

The SDN controller being „just“ a piece of software running on a server somewhere in the network obviously has its advantages and disadvantages. That being said, the controller implements required routing/switching algorithms to be ran (on top of the network model) in software. Hence, adding new functionality to the controller is just a matter of programming a piece of code, which implements it. Thus, not only well known routing protocols may be used. The network managers can now develop their own custom-tailored solutions - better fitting their needs. They can of course update these algorithms to adapt to changing conditions in their networks. The amount of flexibility, that software-defined networking offers is one of its biggest advantages.

On the other hand, the controller software implementations may have more bugs, than their traditional well established „non-software-defined“ networking counterparts. The controller implementations are not yet so well tested over the course of time. The fact, that everyone (including people with insufficient knowledge from both networking and software development fields) can write their own piece of controller software may be another cause of resulting unpredicted network behavior.

Since a controller is the place, where all the intelligence sits, where all the forwarding rules are calculated and further distributed into network, it can easily become a single point of failure. This means, that once the controller is unavailable, the devices will not be able to update or retrieve new forwarding rules and hence the network behavior may be unpredictable, some networks may become unreachable or packet loss may occur. Therefore it is important to ensure high availability of the SDN controller.

A distributed controller may also be an option, however the idea behind it is usually different. The idea then is not so much to support the high availability, but rather to have different controllers perform different tasks (e.g. one for routing, other for load balancing).

In both cases sufficient amount of network bandwidth between the controller and network devices has to be provisioned. It is though usually far less, than the bandwidth needed for exchange of routing information in the nowadays networks. Furthermore, the bandwidth may (if needed) be preserved by wider application of more generic wildcard flow rules (see Section 5.2.1), which then implies a lower frequency of flow table updates.

The machine, where SDN controller is running, must provide a sufficient amount of computational power. This may sound worse than it actually is. Wide range of high power servers are available on the market today. Their prices are usually much lower than prices of dedicated network hardware (of comparable power). And last but not least, it is just the server running the controller software, that has to provide the computational power for network control - not every switch or router in the network.

3.3 Application layer

The application layer consists of management and business-oriented software. It can also be understood as an user interface for the control layer. From the management point of view, the programs running in the application layer may provide the network engineers with statistics about their networks (e.g. link utilization information, link failure updates) as well as an interface to adjust the network configuration, etc.

Whereas from the business point of view, the application layer software may for example support more customized billing policies for the customers. Last but not least, since the global view of the network is provided, it is easier to determine unused ports (according to interface counter values), ensure better link utilization, etc.

3.4 SDN APIs

The Software Defined Networking has been widely discussed in recent years. Different companies willing to join the innovation in computer networking have seen different business opportunities in the growing SDN market. Therefore, different companies (both well-established industry leaders like Cisco and startups like Big Switch Networks) started talking about a need to develop and further standardize SDN APIs (as seen in Figure 3.2 and described below).

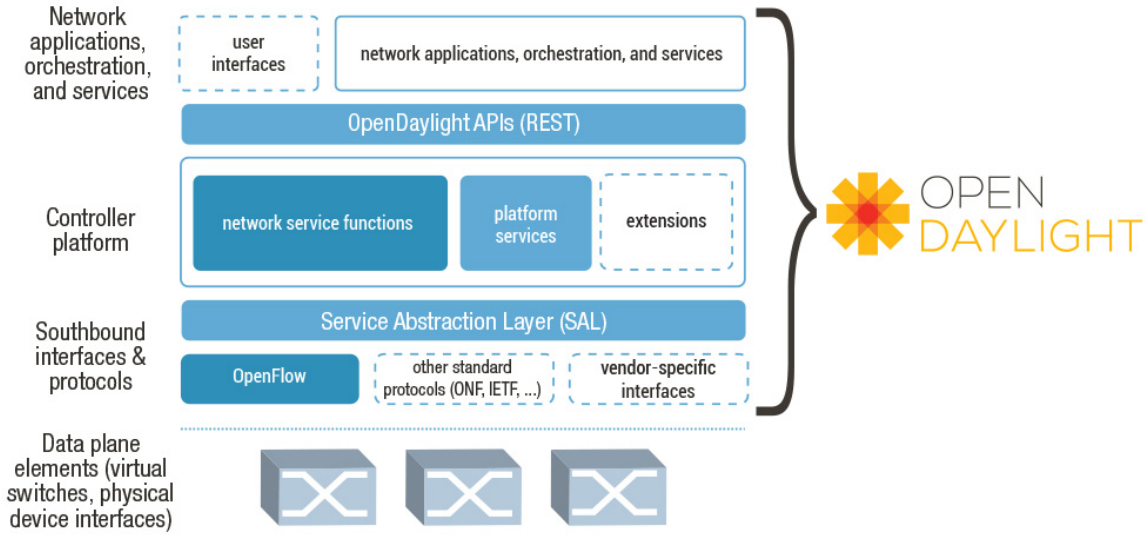


Figure 3.2: Software Defined Networking APIs [29].

3.4.1 Southbound API

The Southbound API specifies communication between the forwarding hardware infrastructure and the controller laying in the controller layer. This API was the first to be standardized and the topic is currently covered most extensively. The standards assure, that controllers from the controller plane can communicate with a wide range of both physical and virtual switches in the data/infrastructure layer.

OpenFlow protocol is one example of Southbound API. Vendor specific protocols for networking hardware-controller communication are available as well.

3.4.2 Northbound API

The Northbound API is currently (as of May 2013) being discussed the most. This is mainly due to the fact, that during the third Annual Open Networking Summit [33], big players like Cisco suggested they would be moving in this direction. Announcement of the OpenDaylight project is another clue since it takes the Northbound API into account as well (see Section 3.5.1). The Northbound API aims to specify a communication interface between SDN controllers and applications running on top of them (in the application layer).

The motivation behind is to help SDN applications developers to make their code compatible with different SDN controllers. However, it is not only focused on networking applications. For example business applications such as Oracle server should be able to tell controller what they need from the network, as well as retrieve information about the network's state in order to make informed decisions about their operation.

3.4.3 Eastbound/Westbound API

Complex networks may include more then one SDN controller, therefore there is a need for an interface to provide communication channels between the controllers. Currently there are two approaches to do so - first being arrangement of controllers into hierarchy and

second being use of BGP. These techniques seem to be efficient enough for the time being, hence the main focus is on Southbound and Northbound APIs.

3.5 Transition from „non-software-defined“ to software-defined networking

No matter how tempting and promising the concept of software-defined networking may sound, it is clear that the transition of current „non-software-defined“ networks towards SDN will take a lot of time and probably it will never be a full transition. As is with every technology, there certainly will be fields, where the SDN just will not be the best solution. Say, in a small office network with a printer, web server and a few computers, it would probably be easier to just deploy a traditional routing protocol instead of setting up a dedicated SDN controller. Although, I can imagine that with nowadays push to providing „everything-as-a-service“ - why not providing SDN controllers as a service for this kind of businesses?

One of the selling points of SDN is its lower price. For as much as this may be true for building a complete new network from the ground up (where the architects really may go with the „simple“ forwarding devices in the infrastructure layer), this may not entirely be the case when upgrading an existing network. In such case, it is important to maintain backward compatibility with the existing hardware in the transition phase. That calls for network devices supporting both communication with the SDN controller as well as communication with other networking hardware using the standard currently used routing protocols. This may then increase the price of the new network devices and hence make the whole investment less profitable.

Additional training of company's staff may be required in order to successfully transition to and further maintain the software-defined networking technology. On the other hand, once the network operators learn how to manage SDN network (as in to learn how to work with the controller and its APIs), they will be able to work with networking hardware of any kind, regardless the manufacturer. It will be the SDN protocol (e.g. OpenFlow), that „configures“ the hardware and thus the network managers will not have to be so familiar with all the different routers/switches operating systems (like Cisco IOS). This may also result into easier deployment of multi-vendor hardware within the managed network (e.g. in order to lower the cost).

3.5.1 OpenDaylight

OpenDaylight project is yet another new development in the SDN field, announced during the the third Annual Open Networking Summit [33]. It is an open source project under Linux Foundation, by Big Switch Networks, Cisco, RedHat, HP just to list a few. The project's mission is to help the industry with adaptation of SDN technology into their infrastructure, through mitigation of risks of adopting early stage technologies. Its main part is a controller running within its own Java Virtual Machine, which can therefore work on any system with Java support.

The controller exposes northbound APIs for applications running both within and outside of the controller's address space. Applications running above these APIs will implement the business logic and algorithms and will use the controller to gather data about the managed network for their computations.

The controller's southbound API supports different protocols like OpenFlow v1.3 or BGP-LS as separate plug-ins.

Additionally, the controller also carries a collection of pluggable modules to perform basic network tasks [29].

That being said, OpenDaylight can be the core of any SDN system. Support of wide range of biggest industry vendors should bring the confidence, that the controller will work properly with their hardware, that e.g. OpenFlow implementation in vendors' networking hardware will truly be compatible with the OpenDaylight's southbound APIs OpenFlow plug-ins. This is currently considered to be one of the biggest issues, as different vendors support different parts of different versions of the OpenFlow protocol.

Chapter 4

OpenFlow protocol

OpenFlow is the first standardized protocol for Software-defined networking. Its development is mainly supported by Open Networking Foundation (ONF). The Open Networking Foundation is a non-profit organization gathering companies and organizations from the computer networking itself as well as other related fields. Among its members are well established networking hardware manufacturers like Cisco Systems, Juniper Networks, Hewlett-Packard, Dell or Brocade, which suggests that we should see an increasing amount of networking hardware with OpenFlow support built in. Although as of today, it is rather the software switches like Open vSwitch, that support the OpenFlow protocol. On the other hand, the fact that Open Networking Foundation also has members like NTT Communications, Level 3 Communications or Verizon Communications suggests, that the protocol will be developed to fit the practical needs of network operators. There are also members from the virtualization field (e.g VMware, Citrix Systems) or companies operating large data centers (e.g. Google, Facebook). All of this makes it look like the OpenFlow protocol is set to become the next „Big Thing“ (not only) in the computer networking field [13].

OpenFlow is a protocol that controller (lying in the control layer - see Section 3.2) uses to communicate with the managed network device (lying in the infrastructure layer - see Section 3.1) in order to push new, update or delete forwarding rules in device's flow tables.

4.1 OpenFlow versions

The key OpenFlow protocol versions are the following (version's key features are mentioned as well) [12]:

v1.0 Released on December 31, 2009. Currently supported on Hewlett-Packard switches (see Section 4.3). Support for multiple queues per output port - each queue can be assigned a slice of the available bandwidth. Matching on IP ToS/DSCP bits as well as on IP fields in ARP packets. Retrieving port stats for individual port.

v1.1 Released on February 28, 2011. OpenFlow can now expose multiple tables to the controller, hence hardware device's internal tables can be exposed more flexibly. Multiple tables may simplify maintenance of different packet processing methods (e.g. ACLs, QoS), by deploying dedicated tables accordingly. Ports may be assigned to a group, which acts as a single entity for forwarding. Newly, multiple VLAN tagging and a similar functionality for MPLS is supported. Support for virtual ports, that can e.g. represent tunnels, was added.

v1.2 Released on December 5, 2011. A TLV (Type-length-value) structure instead of a fixed length structure was introduced for matching, which dramatically increases matching flexibility. Basic IPv6 support was added. Controller role change mechanism was introduced, thus multiple controllers are supported for fail-over. Although, a switch has to remember the controller's role, in order to elect new one in case of failure.

v1.3 Released on April 13, 2012. Support for the most common IPv6 extension headers added. Per-flow meters enable e.g. rate limiting of packets sent to the controller. In case multiple controllers are connected to the switch, each controller can filter some of the switch's events, that it does not want. MPLS BoS (Bottom of Stack) matching was added.

4.2 OpenFlow applications

There is a wide range of applications, where use of OpenFlow can improve overall system performance. The following list brings some of the experiments (performed at the Stanford University) [16].

Slicing the network The network infrastructure can be divided into logical slices (using e.g FlowVisor software). Hence, different services can be mapped to different network slices, and the traffic could then be treated accordingly.

Load balancing The whole network can be viewed as one big software load balancing switch instead of deploying expensive load balancing hardware switches.

Packet and circuit network convergence OpenFlow provides a solution for merging packet and circuit networks into one, thus reducing CAPEX/OPEX spendings of telecommunications companies.

Reduction of energy consumption The unused links can be switched off, so that less energy is needed to run e.g. a data center network.

Dynamic flow aggregation OpenFlow can help saving the resources (CPU, routing tables) or further ease management of the network.

Providing MPLS services OpenFlow can simplify deployment of new MPLS services (e.g. new tunnels including adjustments of their bandwidth reservation).

4.3 OpenFlow support on Hewlett-Packard switches

According to the latest press release by Hewlett-Packard, the OpenFlow is currently available on 16 models from its 3500, 5400 and 8200 series switches. Furthermore, Hewlett-Packard plans to deploy OpenFlow support on its FlexNetwork architecture switches this year as well, thus expanding its OpenFlow networking solutions range even more [18].

Currently OpenFlow up to version 1.0 is supported on HP ProCurve switches allowing one OpenFlow instance per VLAN. The firmware versions with OpenFlow support include the following [15][14]:

K14.830 The latest version by HP Lab. Full support of OpenFlow v1.0.

K15.05.5001 The latest version by HP Networking division (for OpenFlow researchers only). Support for new chassis blades, new commands (e.g. „no openflow“).

In May 2013 Hewlett-Packard announced to add support for OpenFlow v1.3 on over forty more switches through 2013 [19].

4.4 OpenFlow support (other vendors)

A significant number of vendors announced support for OpenFlow protocol in their networking hardware during the third Annual Open Networking Summit [33] and in the following weeks. Here is a list of the major ones.

Arista added support for OpenFlow v1.0 to their Extensible Operating System (EOS). The feature is available on their 7050 series hardware. In addition, Arista's DirectFlow is also supported which enables it to run in a controller-less mode [1].

Brocade joined the OpendDaylight initiative (see Section 3.5.1). OpenFlow is supported on their MLX series routers [3].

Cisco joined the OpenDaylight initiative as well (see Section 3.5.1). Their proprietary Open Network Environment (ONE) controller supports OpenStack, OpenFlow and Network Functions Virtualization (NFV). Cisco also introduced a number of APIs providing interface to communicate with their hardware in form of onePK (ONE Platform Kit). So far, the technology is supported in ISR G2, ASR 1000, and Nexus 3000 hardware [31].

NEC supports OpenFlow on their ProgrammableFlow family of products [26].

Chapter 5

OpenFlow architecture

This chapter briefly describes the basic concept of the OpenFlow architecture as specified in the OpenFlow Switch Specification v1.3.0 white paper [12]. Communication between the OpenFlow controller and OpenFlow-enabled switch using the OpenFlow protocol usually takes place over a secure channel (e.g. TLS). It is established by the symmetric Hello message and its state is further verified by exchange of symmetric Echo request/reply messages. The OpenFlow controller then uses Controller-to-Switch messages to determine OpenFlow device's functionality, state and to adjust its configuration (e.g. add/modify/delete flow table entries). Likewise, the OpenFlow switch uses asynchronous messages to inform controller about its status changes or to forward a packet to controller in order to determine new forwarding rule for the packet/flow.

Essentially there are two types of OpenFlow-enabled switches. The OpenFlow-hybrid switches support both the traditional way of packet processing (e.g. layer 2 switching, layer 3 routing, VLANs, ACLs), as well as packet processing strictly based on the information stated in the switch's flow tables. The OpenFlow-only switches on the other hand support only packet processing by pipelining of the flow tables.

5.1 OpenFlow ports

First of all, not all physical ports of the networking device have to be OpenFlow-enabled. There are three OpenFlow port types: physical (correspond to hardware interfaces of the device), logical (offer a certain level of abstraction above the physical hardware interface) and reserved ports.

There are several types of OpenFlow reserved ports such as `CONTROLLER` (channel to the OpenFlow controller), `TABLE` (starts a pipeline of flow tables), `LOCAL` (in-band OpenFlow controller connection), `NORMAL` (legacy packet processing/non-OpenFlow pipeline of the switch), `ALL` or `FLOOD` (packet flooding).

All the different ports may be used within the instructions field of flow entries as actions to be taken when packet matching the entry is received.

5.2 OpenFlow tables

Every OpenFlow-enabled switch must contain at least one flow table, a group table and a meter table.

5.2.1 Flow tables

Flow table consists of flow entries, which may be pushed to the OpenFlow-enabled switch by a controller both proactively (in advance) and reactively (once a packet from unknown flow occurs). Each flow entry consists of following:

- **Match fields** to match the packet against (e.g. source/destination ports/IP/MAC addresses).
- **Priority** of the entry determines, whether to apply the rule or not in case when the packet fits match fields of multiple entries. The entry with the higher priority is chosen.
- **Instructions** to modify the set of actions to be performed on the packet. They may introduce a meter, add/clean actions from the action set, point to a next table, apply changes to the packet itself (headers) or write its metadata.
- **Timeouts** to define automatic flow expiration.
- **Counters** of packets.
- **Cookie** specified and further used by controller to filter flow statistics, etc.

The flow entry is uniquely determined by its match fields and priority. It is desirable to introduce so called miss-entry, which describes what happens if a packet does not match any flow entry in the current flow table. Such packet may be then forwarded e.g. to the OpenFlow controller. In case no miss-entry is provided, the packet will be dropped.

Each packet has an action set associated with it, which is modified by instructions from a corresponding flow entry. The actions are performed once the last flow entry of the packet processing is matched and its instructions are executed. Possible packet actions include sending packet to a specified output port or group, changing its TTL, applying/modifying its VLAN or MPLS tags, etc.

Incoming packet may be matched against flow entries of one flow table or its processing may be pipelined through multiple flow tables (in case they are present on the device). Multiple flow tables are sequentially numbered. Flow table with ID of 0 is understood to be the default/first table to look up the packet match. Pipelining between tables may occur only in a „from the lower ID to a higher ID table“ manner. Some of the packet fields may be changed and additional metadata may accompany the packet as it „travels“ between the tables.

5.2.2 Group table

Group table provides additional forwarding scenarios for example for multicasts or broadcasts. The table contains group entries defining so called action buckets. The action buckets are lists of actions to be performed on a packet in the listed order.

5.2.3 Meter table

Meter table consists of per-flow meter entries. The meters are triggered by instructions in flow entries in the flow tables and are used to perform basic Quality of Service tasks such as rate limiting according to defined meter bands, etc.

Chapter 6

OpenFlow controllers

There are currently multiple both open and closed source OpenFlow controllers available. The controllers support various programming languages e.g. Java, C++ or Python. Some of the controllers include graphical user interface or web management interface. Following is the list describing some of the most well known OpenFlow controllers.

NOX/POX NOX is said to be the first OpenFlow controller, with C++ support, initially developed at Nicira. NOX is now divided into NOX (new architecture, faster) and NOX Classic (with Python support, but no substantial further development expected). POX is NOX's Python-oriented sister project. Both are open source [25].

Beacon is an open source, cross-platform, modular, Java-based OpenFlow controller. It optionally offers a Web UI [7].

Simple Network Access Control (SNAC) uses a web-based policy manager to manage the network [32].

Maestro is a modular, cross-platform OpenFlow controller with Java support [21].

Floodlight is a Java-based OpenFlow controller distributed under the Apache license. It newly offers OpenStack support and is for example used as a core of the commercial controller by Big Switch Networks [8].

Trema is a full-stack Openflow framework. It supports programming in both Ruby and C. Additionally, it allows users to set up a virtual test network within the Trema framework thanks to its full-stack property (Trema is further described in Section 8.3.1).

Last but not least, the FlowVisor software allows network managers to divide their physical network into logical slices. The FlowVisor acts as a proxy between the network physical devices and multiple controllers, that can be used for each slice [9].

Chapter 7

Current best practices of load balancing in computer networks

Load balancing has been becoming more and more important for the network managers looking for a better bandwidth utilization across their networks. However, unlike in the ATM or Frame Relay technologies (mostly) in the pre-IP days, load balancing is much harder to achieve in nowadays full-IP networks as the IP itself is connection-less and thus does not offer virtual circuits or reliable end-to-end connectivity.

7.1 MPLS Traffic Engineering

Multiprotocol Label Switching Traffic Engineering (MPLS-TE) is probably the most effective of currently used load balancing techniques. MPLS allows establishment of so called LSPs (Label Switched Paths) and thus allows routing of traffic for a specific prefix over a specific path consisting of LSRs (Label Switch Routers), who will have learned how to forward a packet with this specific label.

As long as MPLS TE is concerned, the routers on the edges of the network are grouped into tunnels specified by head-end and tail-end routers. Link state protocol has to be enabled on the managed network in order to provide each router with a full knowledge of the topology. The head-end router then performs Constraint Based Routing (CBR) calculation and the least cost path to the tail-end router is calculated. The CBR is chosen for the calculation because it can take multiple paths into account. Resource Reservation Protocol (RSVP) with traffic engineering extension is used to allocate needed bandwidth on the LSRs of the calculated least cost LSP [20][30].

7.2 Routing and switching adjustments

Certain degree of load balancing may also be obtained by introducing various routing adjustments to the router's/switch's configuration.

Static routes may be manually specified and hence traffic for different prefixes will be sent out specific interfaces. It is also possible to manipulate the route metrics of routes learned from different routing protocols. Furthermore, a variance command may be introduced when using Enhanced Interior Gateway Routing Protocol (EIGRP) protocol, thus enabling unequal-cost load balancing. Cisco supports on its routers per-destination or per-packet load balancing. All packets for one destination are sent over one link, whereas packets

for other destination are sent across a parallel link in the per-destination load balancing. On the other hand in the per-packet load balancing, each consecutive packet is sent over a different parallel link. This ensures better load distribution, although the in-order delivery of the packets may be broken. Cisco Express Forwarding (CEF) may provide further load balancing improvements [4].

Large networks may be running multiple Virtual LANs (VLANs). Load balancing between switches with redundant interconnecting links may be obtained for example by allowing different VLANs on different interconnecting trunk links.

7.3 Gateway Load Balancing Protocol (GLBP) and load balancers

Gateway Load Balancing Protocol is a proprietary high availability solution by Cisco. High availability is usually configured to avoid loss of connectivity to the gateway in case the edge router connecting to the gateway fails. Hence multiple edge routers are configured with a virtual IP address, an active router is then elected and used for the connection. Backup router immediately takes over and starts forwarding the packets to the gateway in case the active router fails. Even though GLBP is similar to Hot Standby Router Protocol (HSRP) or Virtual Router Redundancy Protocol (VRRP), unlike the others it allows more then one router to be operate as active, hence allowing load balancing between routers connecting the network to gateway.

Load balancers or load balancing switches on the other hand help to distribute the server load. Sample implementation would be for example a server farm consisting of redundant servers sitting behind a load balancer. Incoming traffic for the servers (logically acting as one), would then be distributed between the redundant servers by the load balancer (e.g. in round-robin fashion). This is however server load balancing and hence it is not the main focus of this project.

Chapter 8

OpenFlow testbed choice

Aim of the thesis was to test the resulting OpenFlow load balancing application both on a virtual network (to be used mainly for development), as well as on physical networking hardware currently available for purchase (namely Hewlett-Packard). Thus, choice of an appropriate testbed was crucial and a decent amount of time was dedicated to the task. Choice of the testbed also influenced the selection of OpenFlow controller, which was further used for the load balancing application development.

8.1 OpenFlow load balancing tool draft

OpenFlow offers great range of features, that can be used to implement load balancing in an Openflow-enabled network. A network model is build and maintained on the OpenFlow controller and thus the applications built on top of the controller have full knowledge about the current network infrastructure. At the same time, the applications are not required to have any knowledge about the devices used in the infrastructure layer (their configuration methods, etc.) since it is the controller's work to configure those devices using OpenFlow protocol.

The programming language to build the application will depend on the chosen OpenFlow controller. The application will be periodically polling the network devices directly (e.g. via SNMP) or checking the OpenFlow entries counters in order to obtain statistics regarding link utilization, latencies, etc. An algorithm will be developed to analyze the statistics and update or delete existing or introduce new flow entries to the network devices aiming to balance the links' load as equally as possible (or within a given threshold). The algorithm will be ran also every time a table-miss occurs in order to determine appropriate flow entries for the new flow.

The efficiency of such solution will depend on the latency of controller and the resulting application running on top of it when dealing with packets from an unknown flow or when dealing with excessive load on some of the links. Hence, it will be crucial to determine the right trade off between the use of proactive wildcard flow entries to minimize the polling of controller with new table-miss messages on one hand, while at the same time keeping the flow entries as specific as possible in order to obtain fine-grained load distribution across the network. Similarly, suitable flow entries timeout values will have to be determined to keep the balance between an extensive flow table size and extensive rate of table-miss events. These issues will be subject to testing and the trade off will have to be found empirically.

8.2 Initial laboratory setup

A virtual network was set up in the lab environment in order to perform basic OpenFlow support test and to provide a starting point for design and development of a load balancing tool for OpenFlow network. The network was set on a VMware server provided by the Faculty of Information Technology of Brno University of Technology.

The network consisted of four virtual machines interconnected in a mesh topology. Each virtual machine represented an OpenFlow switch and was running Open vSwitch software. The NOX OpenFlow controller was installed on additional virtual machine, which had a connection to all four virtual switches. All machines were running a basic installation of Debian Linux distribution.

As a result of unsolved problems with communication between Open vSwitch and NOX controller the setup was not used for further development.

8.3 Virtual networking testbed

Virtualized networking testbeds provide significant flexibility as far as networking applications development and testing is concerned. Test infrastructure can be defined in a straightforward programming-like manner, and changed at any time (sometimes even during the infrastructure runtime). Such a testbed gives almost no limits to the number of switches and interconnecting links, and hence different scenarios can be tested (full-mesh, star topology, etc.). The computational power of computers used for such experiments is usually considered to be the only limiting factor.

8.3.1 Trema

Trema package provides developers with both OpenFlow controller and network emulator. Trema aims to be a simple and fast OpenFlow development framework. The development is supported in Ruby and C. Infrastructure configuration seems very simple, although it is not as flexible during runtime as Mininet (see Section 8.3.2. This was considered the main disadvantage of the testbed.

8.3.2 Mininet

Mininet is basically a network emulator. It runs virtual switches, hosts and links between them in a single Linux kernel. Mininet provides a Python API for better infrastructure configuration. Attributes like link bandwidth or delay are supported can be specified as well. Emulated switches may also be ran as Open vSwitch instances. There are no restrictions regarding OpenFlow controller, thus any controller software may be chosen for development and will work with the mininet infrastructure.

The infrastructure itself may be altered from the Mininet command-line interface (CLI). Thanks to the CLI, actions such as link shutdown may be performed even during the Mininet runtime. Emulated hosts act as real computers, hence they are reachable by SSH (once SSH daemon is started) or an their xTerms may be started from the Mininet CLI. This allows for extensive networking applications testing.

Mininet was chosen as the virtual networking testbed, thanks to all the mentioned features. For more information please see [2].

8.4 Physical networking (hardware) testbed

There is a justified need for a physical networking testbed, even though a virtual networking testbed (see Section 8.3) is very flexible and helpful for the developer. This is mostly due to the fact, that behavior of software-emulated devices is usually not the same as of its hardware counterparts, as different manufacturers may support just some functions from the OpenFlow specification, etc. Therefore, the resulting application should be tested on a physical infrastructure in addition to the virtual one, in order to prove its proper functionality.

8.4.1 OFELIA project

OFELIA stands for OpenFlow in Europe: Linking Infrastructure and Applications. „It is a collaborative project within the European Commission's FP7 ICT Work Programme. The project creates a unique experimental facility that allows researchers to not only experiment on a test network but to control and extend the network itself precisely and dynamically. The OFELIA facility is based on OpenFlow, a currently emerging networking technology that allows virtualization and control of the network environment through secure and standardized interfaces.“ taken from [27].

OFELIA's resources are divided into islands - geographically distributed and interconnected OpenFlow networks running physical networking hardware by different vendors. Access to island's resources is granted to researchers after a simple registration, no admission costs apply. Researchers/developers are then allowed to specify their projects and may later request networking resources once their projects are approved by the corresponding island's manager. Provided resources include switches, interconnecting links and Virtual Machine (VM) servers which may be used for running test hosts or OpenFlow controllers. The research project may consists of several experiments. Separate slice of island's infrastructure may be requested for every experiment. Projects and corresponding experiments may also be ran across several OFELIA islands, although an additional approval is required.

Available OFELIA islands were reviewed and the CREATE-NET testbed in Trento (Italy) was chosen as the physical networking testbed. The main reason for this choice was that Hewlett-Packard ProCurve 3500 switches are present at this island (one of this thesis tasks was to test support for OpenFlow on Hewlett-Packard hardware).

8.5 OpenFlow controller choice

The main criterion for Openflow controller choice in this particular case was, that it was required to work both with the chosen virtual networking testbed (Mininet) and the physical networking testbed (OFELIA). Thus, different controllers from the list in Section 6 were reviewed in order to meet criteria.

8.5.1 POX

Initially, NOX controller seemed to be the best fit. However, its sister project POX seems to be better supported and evolving quicker. Therefore the POX controller was chosen. Although, the current version supports only OpenFlow v1.0, it is not considered to be a drawback as the resulting application should be compatible with the hardware switches as well, and such devices as of May 2013 support usually at most OpenFlow v1.0.

POX controller may also be ran together with POXDesk component, which is a simple graphical user interface for POX (e.g. topology maps or flow tables may be displayed) [23]. The web interface was found useful during applications testing. Sample output may be seen in Figure 8.1.

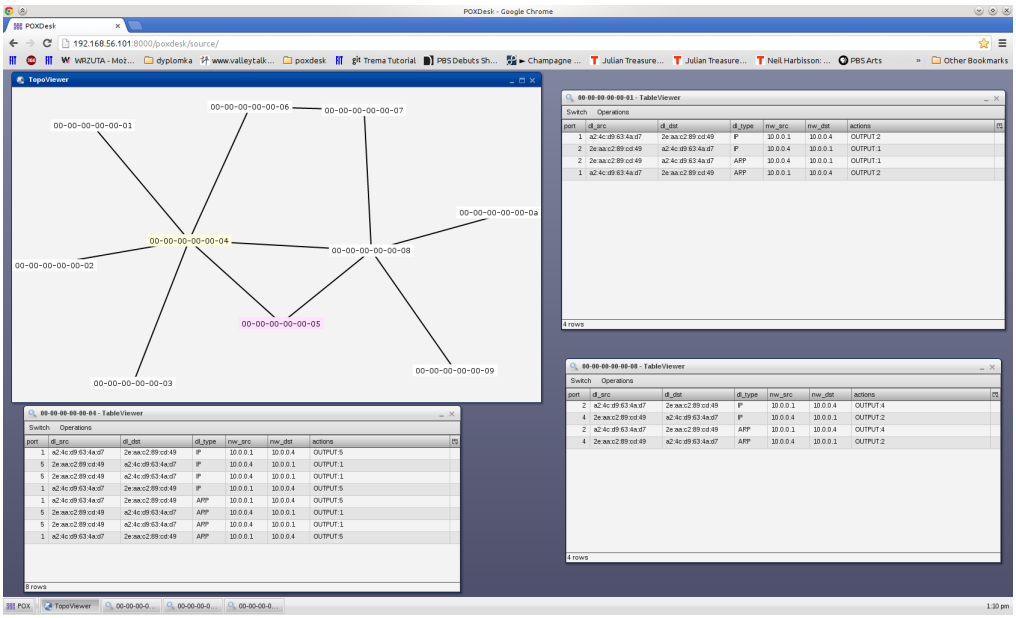


Figure 8.1: Sample POXDesk web interface output.

Last but not least, there are some ready to use OpenFlow implementations of spanning-tree protocol or ARP proxy coming together with POX controller. Some of the components were later used as a base of the resulting application (see Chapter 10).

Chapter 9

Application development process

The application was developed to utilize OpenFlow v1.0. Higher versions were not taken into consideration during the development process, as neither the POX controller chosen for development or the majority of currently available networking hardware support newer versions of OpenFlow than v1.0 (see Section 8.5.1).

9.1 Test topologies

Two different test topologies were designed - one for the physical and one for the virtual networking testbed (see Figures 9.1 and 9.2). The topologies were designed with a thought of load balancing testing. Therefore, they include loops, which effectively mean that there are multiple paths between switches (and thus source and destination) allowing load balancing across the paths.

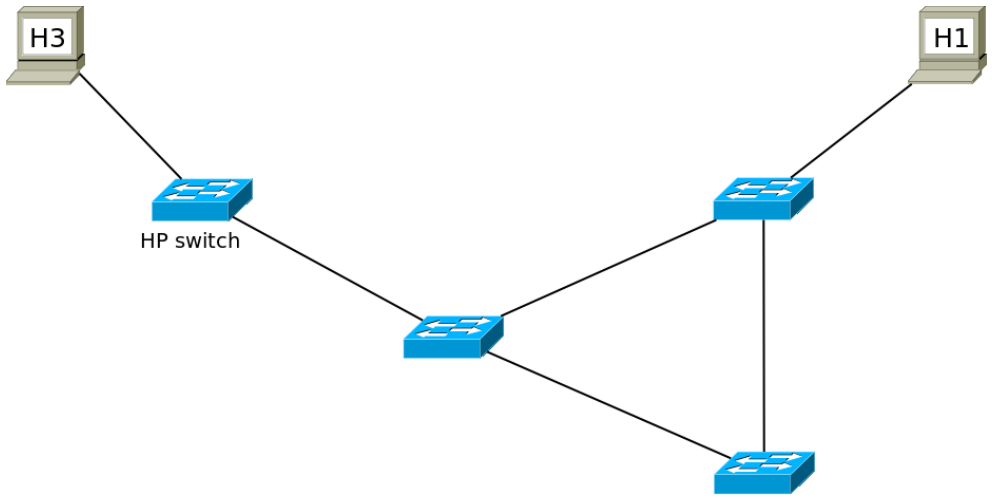


Figure 9.1: Physical networking testbed ran in OFELIA (see Section 8.4.1) with the Hewlett-Packard switch depicted.

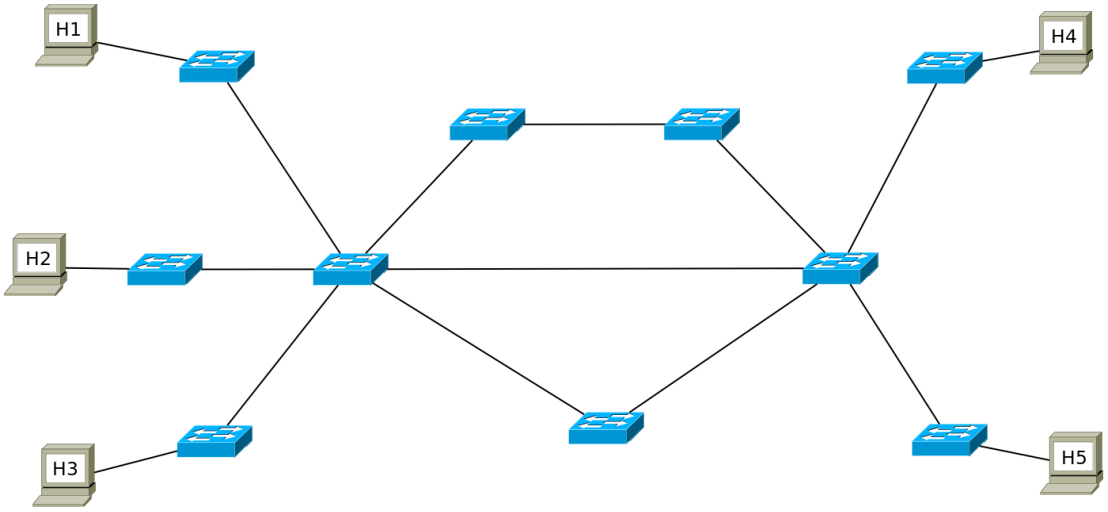


Figure 9.2: Virtual networking testbed ran in a Mininet virtual machine (see Section 8.3.2).

9.2 Spanning Tree

It is crucial to take care of the topology loops in the first place, otherwise flooding packets out of switches' interfaces would cause network's overload and eventually switches failure. Disabling flooding functionality may seem like a sufficient solution in the beginning. However, initially the controller does not have any clue about hosts (and their IP or MAC addresses) connected to a particular switch. Hence it is important to enable flooding in order to let the Address Resolution Protocol (ARP) to work properly, so that different hosts would be able to reach each other across the network.

Calculating a spanning tree of some sort to represent the network was considered a better solution than blindly disabling all the flooding. Exact implementation of the Spanning Tree Protocol (SPT) was however not important. The aim was to obtain a logical model of the network, where all switches (reachable from one another) in the physical/virtual topology can still reach each other, even though no loops occur in the topology.

9.3 Flow entries timeouts

Each flow entry in the switch's flow table may be set with a hard and idle timeout (see Section 5.2.1). These timeouts prevent the flow table to be running out of space, which could eventually lead to switch failure and connectivity issues. Flow entry is removed from the flow table in case one of the timeouts expires.

- **Idle timeout** (in seconds) represents the amount of time after which a particular flow entry is removed in case no packet matched the entry.
- **Hard timeout** (in seconds) represents the amount of time after which a particular flow entry is removed, even if the flow is still active (there are still packets forwarded according to the entry's match field).

Initially, it was decided that the idle and hard timeouts will be set to 20 and 30 seconds accordingly. These values were chosen for a couple of reasons. Firstly, keeping a significant

amount of flow entries on virtual switches may lead to degradation of the virtual networking testbed performance (because of high memory usage). Secondly, expiry of hard timeout if the flow is still alive leads to the `OFPT_PACKET_IN` message being sent from a switch to the controller, because the switch does not have any forwarding rule for such a packet. Such state is not considered to be a disadvantage since it allows for path recalculation based on current network links utilization.

On the other hand, `OFPT_PACKET_IN` events introduce some latency into the network operation as the first packet has to travel from a switch to controller, path calculation is performed, flow entries are installed and only then the packet goes back to the switch and is forwarded according to the newly-installed flow entry. All the following packets are then forwarded right away with no controller's interaction, thus experiencing lower latency than the first packet. Timeout values would naturally need to be adjusted according to network characteristics, where the load balancing tool would be deployed. For example Content Delivery Networks (CDNs) are dealing with much longer lasting flows (e.g. Video on Demand or software updates downloads), and hence the timeouts could be set to higher values so that the stream of packets is not interrupted and a smaller amount of `OFPT_PACKET_IN` events occurs. Another scenario could be e.g. in office environment with IP telephony deployed - all VoIP traffic would be set with only idle and no hard timeout in order to minimize the latency and thus maintain high phone call quality.

9.4 Floyd-Warshall algorithm with adjusted weights

The initial prototype deployed the Floyd-Warshall algorithm with some minor tweaks to it. In standard setup, Floyd-Warshall algorithm calculates shortest paths between all vertices in a given graph with weighted (positive and negative) edges. POX's `forwarding.l2_multi` component was used as a base for this prototype and further extended into the resulting application (see Section 10.2.3).

Initially, a graph representation of the managed network is constructed - vertices represent switches, edges represent interconnecting links. Shortest paths between the switches are then calculated using the modified version of Floyd-Warshall algorithm. The edge weights/distances never change during the original algorithm's runtime (values set to 1 by default). This often leads to a situation, where some links or subpaths are used more often than other parallel but slightly longer (with more hops) paths. Load balancing's role here is then to utilize the longer paths as well in order to unburden part of the shortest subpath's load.

Hence an enhancement to the original algorithm was developed. Initially all the weights are set to default value of 1 and the calculation starts. Every time a new shortest path between two nodes is found: the path is stored and weights of edges along the path are incremented by 1. Thus it is ensured, that this path is not likely to be chosen again as a sub-path of another shortest path between next pair of vertices. The graph is updated and new round of computation takes place every time a change in topology (link goes down, new switch connects, etc.) occurs.

Results were as expected, thus also paths with a larger number of hops (physical switches on the way) were chosen. However, such a solution was not considered to be sufficient, as the paths are calculated without any knowledge about the traffic which is/will be flowing among them. Furthermore, the paths are calculated only between the switches and for example the number of hosts on each of them is not taken into account. Only the one calculated path between two switches will be overloaded once hosts on these nodes start

to exchange large amounts of data. No route recalculation occurs even though there might exist another route between the two switches. Hence, better solution needed to be found.

9.5 Dijkstra's shortest path algorithm - variant I.

The application conception was changed for the second prototype. Initially, all edge weights are set to a value of 1 as in the previous version. However, Dijkstra's algorithm was used instead of Floyd-Warshall and hence the paths are calculated on demand, for a new pair of connection endpoints (pair of source and destination IP addresses) [5]. The weights of edges along a new path are incremented by a value of 1 every time such path is determined. The weights are decremented back by a value of 1, once their hard timeout goes off. Deployed paths and timestamps of their installation are stored in order to be able to determine which paths are expired. This helps to determine, which edge weights should be decremented.

The solution appeared to be much more flexible, the routes were calculated according to the number of flows deployed on the links between switches. However, this solution does not make any assumptions about the traffic. This may lead to situations when a certain link is chosen just because only a small number of paths crosses the link. On the other hand, the link-load caused by these few paths (e.g. carrying Video on Demand) may be well higher then load on other links with a large number of flows (like web page requests). Therefore, number of flows was not considered to be the right metric for edge weights adjustments. Another disadvantage of the solution is the fact, that edge weights are being reviewed (to determine, whether to decrement or not - according to the timeouts) every time a new path is going to be calculated.

9.6 Dijkstra's shortest path algorithm - variant II.

Load or remaining capacity of a link was taken for a more suitable metric than the number of flows crossing the link. Bandwidth of each link is determined in the beginning while constructing a graph representation of the managed network. The bandwidth is taken from the port's „current features“ field in the `OFPT_FEATURES_REPLY` message. The message is received by a controller every time a new switch connects.

Initially, port statistics are collected by issuing an `ofp_port_stats_request`. Number of transfered bytes (extracted from `tx_bytes` field in `OFPST_PORT` message) together with timestamps of the moment when collection occurred is stored. New statistics are then obtained every 30 seconds (same as hard timeout of a flow entry) and a speed at which the port is sending data is calculated as stated in Equation 9.1.

$$port\_tx\_speed = \frac{tx\_bytes\_new - tx\_bytes\_old}{30} \tag{9.1}$$

Particular port's (transmitting) load is then calculated as described in Equation 9.2.

$$port\_tx\_load = \frac{port\_tx\_speed}{port\_bandwidth} \tag{9.2}$$

Link weights are updated every 30 seconds (once new statistics are received). The port's transmitting load is used as the weight value of a link.

This solution was very near the results we were looking for. Sometimes however, a longer path (with low load e.g. 0.5%) was chosen instead of adding the flow to a shorter

path, just because shorter path was running at a slightly higher rate (e.g. 1%). Such behavior was to be adjusted in the final application as it is widely accepted, that the lower the number of hops on the path the better (as long as the capacity is still sufficient, which the 1% in this case undoubtedly is). Furthermore, transmitting load of only one port (one link endpoint) is taken into consideration, when updating edge weights. This means, that the graph representation is not exact, as edge weights should represent overall load of the link (sum of transmitting loads of both ports).

9.7 Final application (using Dijkstra's algorithm)

Enhancements solving the drawbacks of previous prototype (see Section 9.6) were implemented.

The graph generation algorithm was updated so that total load on the link (summing up bytes transfered in both directions) is taken into account when determining particular weight of particular edge.

Link-load threshold was introduced in order to cope with the unwanted behavior of choosing a longer path even if the capacity of shorter path is sufficient. The threshold value was set to 0.5 by default. The value means that if a link is utilized up to 50% of its capacity, it is still recognized to provide sufficient capacity to carry another flow. Therefore, the corresponding edge's weight is set to a value of 0 in case link-load is less than 50%. Therefore, the link appears to be carrying no traffic and hence is likely to be preferred when calculating path between the given source and destination. The Dijkstra's algorithm was preserved as a technique for the shortest/best path lookup.

Algorithm 1 describes basic concept of the final application.

Algorithm 1 Final application work flow (using functions from Algorithm 2)

```
1:  Calculate the spanning-tree
2:  global G ← graph representation of the network
3:
4:  Start Wait for switches to connect
5:  Start Request port stats
6:  Start Calculate loads
7:
8:  loop
9:      if OFPT_PACKET_IN is received then
10:         path ← Dijkstra(G, source, destination)
11:         deploy path on corresponding switches
12:     end if
13: end loop
```

Algorithm 2 Functions used in Algorithm 1

```
1: function Wait for switches to connect
2:     if new switch connects then
3:         Determine port/link speeds of the switch
4:     end if
5: end function
6:
7: function Request port stats
8:     loop
9:         for every connected switch do
10:            Send ofp_port_stats_request to the switch
11:        end for
12:        wait(30 seconds)
13:    end loop
14: end function
15:
16: function Calculate loads algorithm
17:    loop
18:        for every connected switch do
19:            for every port on the switch do
20:                Extract tx_bytes from OFPST_PORT message
21:                Calculate and store the port transmitting load for past 30 seconds
22:            end for
23:        end for
24:        Review graph(G)
25:        wait(30 seconds)
26:    end loop
27: end function
28:
29: function Review graph(G)
30:    for every link between connected switches do
31:        if linkload < threshold then
32:            G.link.weight ← 0
33:        else
34:            G.link.weight ← link-load
35:        end if
36:    end for
37: end function
```

Wildcard entries were introduced in the final version of the algorithm as well. A single flow entry was previously installed for every network protocol used even if the source and destination IP addresses were the same. This feature led to installation of usually unnecessary flow entries. For example, when running the ping between two hosts: flow entries for ARP protocol were installed first followed by flow entries for IP protocol. The corresponding entries differed only in the protocol field, while other match fields (like source and destination IP addresses) were the same. Therefore, `OFPFW_NW_PROTO` value is now set in the `nw_proto` field of `ofp_match` structure. With this value being set, only one flow entry per each IP source and destination address pair is installed on switches, thus preserving

their flow table's space.

Additionally, command line arguments were enabled in order to let the user adjust the application's functionality to better suit his needs. Thus values of both hold and idle flow entry timeouts as well as link-load threshold can be specified by user (see Appendix B).

9.7.1 Variant using Floyd-Warshall algorithm

A variant of the final application which would use Floyd-Warshall instead of Dijkstra's algorithm was considered as well.

Dijkstra's algorithm was firstly used in the second prototype (see Section 9.5). In that case it was very desirable as the edge weights in the first prototype were updated every time a new flow was installed. Thus Dijkstra's algorithm was always used to find just one shortest path - between source and destination, using current edge weights. Running Floyd-Warshall algorithm was considered to be insufficient in this case. Namely, it would mean finding shortest path between all possible source and destination pairs and later picking just one of the paths.

However, since the third prototype implemented link load monitoring (see Section 9.6), edge weights are not updated every time a new flow is installed. The weights are reviewed periodically instead and all Dijkstra's algorithm shortest path calculations, which happen within the update period are based on the same weights values.

This fact turns the attention back to the Floyd-Warshall algorithm, because running it every 30 seconds may be more efficient compared to running the Dijkstra's algorithm every time a new flow occurs and a corresponding path needs to be computed. Hence, running Floyd-Warshall algorithm (time complexity $\Theta(n^3)$) every 30 seconds may eventually be a better solution than running Dijkstra's algorithm (time complexity $O(n^2)$) multiple times during a given 30 seconds period (the n being the number of vertices corresponding to the number of switches) [5].

Therefore, a separate variant of the final application was implemented, with Floyd-Warshall algorithm in its core. The algorithm's computation is called every 30 seconds, right after the edge weights are updated. Shortest paths corresponding to emerging flows are then extracted from the predecessor dictionary (returned by Floyd-Warshall algorithm, keyed by source and destination nodes pointing to the destination's predecessor on the shortest path), which itself takes less time then running the Dijkstra's algorithm from scratch.

Chapter 10

Implementation outline

10.1 Programming language and libraries

Python programming language was chosen for the application development. The choice was mainly influenced by the fact, that POX's libraries are in Python and additionally tools like NetworkX library for working with graphs are also available in Python. For more information about Python programming language please see [17].

NetworkX library was used for path calculations. Floyd-Warshall and Dijkstra's shortest path calculation algorithms are implemented in NetworkX. For more information about the library please see [6].

10.2 POX and its components

Some of components being a part of the POX package were used in order to speed up the load balancing application development. POX's code is distributed under the GNU General Public License and thus the resulting load balancing application is carrying the GNU GPL license as well.

POX currently supports OpenFlow v1.0 (corresponding component `openflow.of_01` is automatically loaded during startup), therefore the resulting application will take advantage only of the OpenFlow v1.0 features.

10.2.1 Openflow.spanning_tree component

POX's `openflow.spanning_tree component` was used. It is ran alongside the resulting load balancing application. The `openflow.spanning_tree` component sets the `NO_FLOOD` bit on switchports that are not in the calculated spanning tree. Hence, when an ARP request is received by to the controller, `OFPP_FLOOD` action is set and the packet sent back to the switch. The packet is then flooded through all the switchports, which do not have their `NO_FLOOD` bit set. This way all the „wanted“ flooding functionality is preserved, while „unwanted“ functionality is prevented.

10.2.2 Openflow.discovery

POX's `openflow.discovery` component was introduced to handle the Link Layer Discovery Protocol (LLDP) packets. It is required by both `openflow.spanning_tree` and

`forwarding.l2_multi` components, where it is used to establish a topology adjacency table (see Section 10.2.3).

10.2.3 Forwarding.l2_multi component

POX's `forwarding.l2_multi` was used as a base for the load balancing tool development. Some of it's basic functionality such as `OFPT_PACKET_IN` events handling or installation of newly calculated flow entries on particular connected switches was preserved. However, the whole path calculation logic was replaced in accordance with the algorithm development process (see Chapter 9).

The component builds an adjacency table representing switches' interconnections. It is essentially a dictionary keyed by `switch_1` and `switch_2` identifiers containing the port number of the interface pointing from `switch_1` to `switch_2`.

The `forwarding.l2_multi` component also implements the original version of Floyd-Warshall algorithm, which was later tweaked during the application development process (as described in Section 9.4). Path installation process is handled by the component as well. Firstly, the newly calculated path is checked to be valid (according to adjacency table) and then corresponding flow entries are deployed on switches along the path using `OFPT_FLOW_MOD` messages.

The component waits for `openflow.spanning_tree` and `openflow.discovery` (see Sections 10.2.1 and 10.2.2) applications to be ready and running in order to start performing correctly. This requirement was preserved in the resulting application as well, otherwise flooding loops would occur in the network.

10.3 Load balancing application

The resulting application variants using Dijkstra's or Floyd-Warshall algorithm were named `load_balancing_d.py` and `load_balancing_fw.py` accordingly (see Appendix A and B for installation and application use guidelines). Each tools is then ran as a POX component (see Appendix B).

Chapter 11

Evaluation of results

The final application was subject to a series of tests. The load calculation equation (see Equation 9.2) was adjusted for the tests as seen in Equation 11.1. Reason for such adjustment was the fact that it was not possible to run near-full load tests as it was not possible to generate e.g. 5 Gb/s flows in the virtual test environment.

$$port\_tx\_load = \frac{tx\_bytes\_speed}{port\_bandwidth} * 1000 \tag{11.1}$$

The final application was tested on both virtual (Section 8.3) and physical (Section 8.4) networking testbeds, as one of the tasks was to test support for OpenFlow on Hewlett-Packard switches (available in OFELIA). Section 11.1 describes the application's runtime test and brings its functionality evaluation. Section 11.2 then describes deviations in behavior when running the application on a slice of OFELIA's physical network, compared to virtual network environment runtime.

11.1 Virtual testbed results

The application was initially tested in the virtual environment (see the topology in Figure 11.1). Table 11.1 depicts virtual host computers and corresponding switches connections.

Host	IP address	Switch ID
H1	10.0.0.1	00-00-00-00-00-01
H2	10.0.0.2	00-00-00-00-00-02
H3	10.0.0.3	00-00-00-00-00-03
H4	10.0.0.4	00-00-00-00-00-09
H5	10.0.0.5	00-00-00-00-00-0a

Table 11.1: Host - switch bindings including host IP addresses.

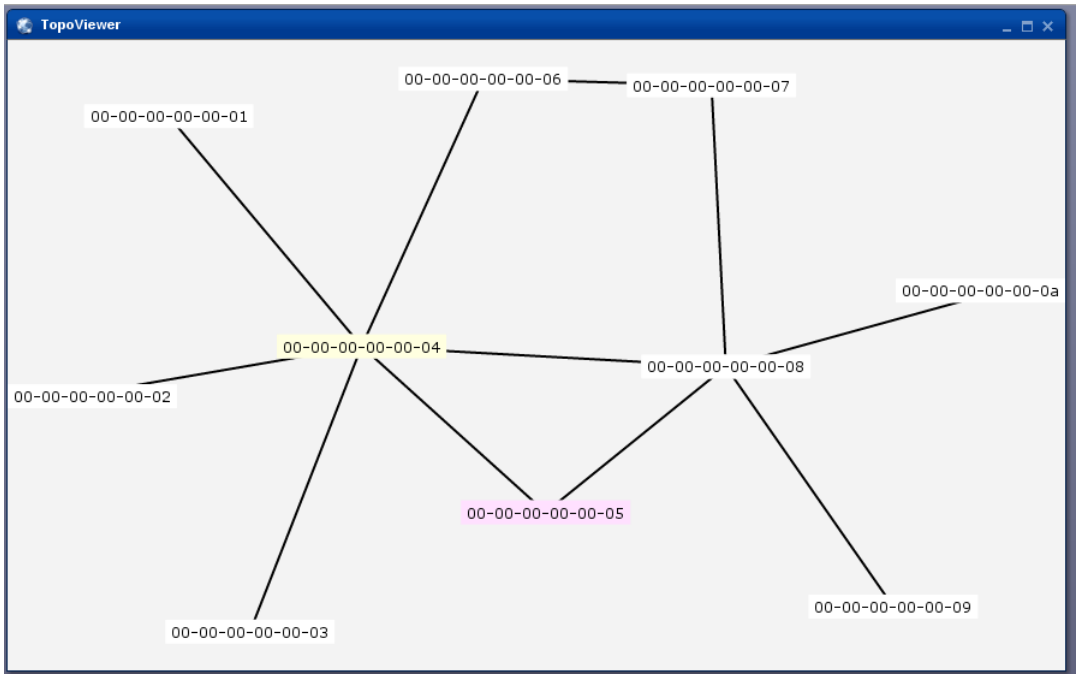


Figure 11.1: Virtual networking testbed (including switch IDs) in the Mininet virtual machine (see also Figure 9.2).

The application start is depicted in Figure 11.2. Initially, flow entry hard and idle timeouts as well as link-load threshold parameter values are shown, followed by output of the `openflow.of_01` component showing established connection with a particular switch, `openflow.discovery` showing detected interconnecting links and `openflow.spanning_tree` showing spanning tree building process.

```
./pox.py samples.pretty_log openflow.discovery openflow.spanning_tree load_balancing_d
POX 0.1.0 (betta) / Copyright 2011-2013 James McCauley, et al.
|Hard timeout: 30  |Idle timeout: 20  |Load threshold: 0.5
[core                   ] POX 0.1.0 (betta) is up.
[openflow.of_01         ] [00-00-00-00-00-02 1] connected
[openflow.of_01         ] [00-00-00-00-00-03 8] connected
[openflow.of_01         ] [00-00-00-00-00-08 10] connected
[openflow.of_01         ] [00-00-00-00-00-09 6] connected
...
[openflow.discovery     ] link detected: 00-00-00-00-00-08.5 -> 00-00-00-00-00-0a.2
[openflow.discovery     ] link detected: 00-00-00-00-00-01.2 -> 00-00-00-00-00-04.1
[openflow.discovery     ] link detected: 00-00-00-00-00-08.3 -> 00-00-00-00-00-07.2
...
[openflow.spanning_tree ] 6 ports changed
[openflow.spanning_tree ] 3 ports changed
...
```

Figure 11.2: The application start showing switches connecting, interconnecting links being discovered and spanning tree operation.

11.1.1 Zero link-load threshold test

First series of tests was performed with the link-load threshold value set to zero. Hence, even very low link-load (e.g. caused by ping command) would trigger corresponding edge's weight update in the network's graph representation. Such change is then taken into account when a new path calculation takes place (either a new flow appears or the current flow expires). This allows the test to be based simply on „pinging“ between connected hosts, because no „larger“ flows are required.

Command `h1 ping h4` was issued in the Mininet's console. Therefore, it was expected to see a path being established between switches 00-00-00-00-00-01 and 00-00-00-00-00-09 (based on Table 11.1. The application's behavior in such setup is shown in Figures 11.3 and 11.4.

```
...
[load_balancing_d       ] Constructing graph G:
[load_balancing_d       ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-03 with a weight of 0 (threshold not reached)
[load_balancing_d       ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-01 with a weight of 0 (threshold not reached)
[load_balancing_d       ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-02 with a weight of 0 (threshold not reached)
[load_balancing_d       ] Adding edge between 00-00-00-00-00-05 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d       ] Adding edge between 00-00-00-00-00-06 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d       ] Adding edge between 00-00-00-00-00-07 and 00-00-00-00-00-06 with a weight of 0 (threshold not reached)
[load_balancing_d       ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-07 with a weight of 0 (threshold not reached)
[load_balancing_d       ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d       ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-05 with a weight of 0 (threshold not reached)
[load_balancing_d       ] Adding edge between 00-00-00-00-00-09 and 00-00-00-00-00-08 with a weight of 0 (threshold not reached)
[load_balancing_d       ] Adding edge between 00-00-00-00-00-0a and 00-00-00-00-00-08 with a weight of 0 (threshold not reached)
[load_balancing_d       ] Installing path: [00-00-00-00-00-01, 00-00-00-00-00-04, 00-00-00-00-00-08, 00-00-00-00-00-09]
[load_balancing_d       ] Installing path: [00-00-00-00-00-09, 00-00-00-00-00-08, 00-00-00-00-00-04, 00-00-00-00-00-01]
...
```

Figure 11.3: Application output during the Zero link-load threshold test - right after start, no link-load statistics on interconnecting links, thus link weights set to 0. The path installed is the shortest path.

As shown in Figure 11.3, initially all link weights are set to 0 as as no link-load statistics are available right after start. They have to be calculated at based on at least two consecutive statistics gatherings. Once the `h1 ping h4` command is issued in Mininet's console, the best path is chosen. In this case it is the shortest path going through switches `[00-00-00-00-00-01, 00-00-00-00-00-04, 00-00-00-00-00-08, 00-00-00-00-00-09]` as there is no known load on the links yet (the reverse path is installed as well in order to provide source-destination reachability in both directions). However, with next measurement of link loads the link weights are updated as shown in Figure 11.4.

```
...
[load_balancing_d       ] Constructing graph G:
[load_balancing_d       ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-03 with a weight of 0.00136666666667
[load_balancing_d       ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-01 with a weight of 0.0216633333333
[load_balancing_d       ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-02 with a weight of 0.00150333333333
[load_balancing_d       ] Adding edge between 00-00-00-00-00-05 and 00-00-00-00-00-04 with a weight of 0.00123
[load_balancing_d       ] Adding edge between 00-00-00-00-00-06 and 00-00-00-00-00-04 with a weight of 0.00136666666667
[load_balancing_d       ] Adding edge between 00-00-00-00-00-07 and 00-00-00-00-00-06 with a weight of 0.00136666666667
[load_balancing_d       ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-07 with a weight of 0.00150333333333
[load_balancing_d       ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-04 with a weight of 0.02139
[load_balancing_d       ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-05 with a weight of 0.00123
[load_balancing_d       ] Adding edge between 00-00-00-00-00-09 and 00-00-00-00-00-08 with a weight of 0.0215266666667
[load_balancing_d       ] Adding edge between 00-00-00-00-00-0a and 00-00-00-00-00-08 with a weight of 0.00136666666667
[load_balancing_d       ] Installing path: [00-00-00-00-00-09, 00-00-00-00-00-08, 00-00-00-00-00-05, 00-00-00-00-00-04, 00-00-00-00-00-01]
[load_balancing_d       ] Installing path: [00-00-00-00-00-01, 00-00-00-00-00-04, 00-00-00-00-00-05, 00-00-00-00-00-08, 00-00-00-00-00-09]
...
```

Figure 11.4: Application output during the Zero link-load threshold test - reviewed load statistics on interconnecting links, hence link weights adjusted according to current link-load. A longer then initial path was chosen based on the load statistics.

Figure 11.4 shows an update of edge weights in the graph, which is forced by link-load statistics being updated. The weights show, that links on the previously chosen path `[00-00-00-00-00-01, 00-00-00-00-00-04, 00-00-00-00-00-08, 00-00-00-00-00-09]` are indeed being utilized more than others. Once the initial flow entry hard timeout expired, new path was calculated since the ping command was still running. Updated edge weights were taken into account and new path going through `[00-00-00-00-00-01, 00-00-00-00-00-04, 00-00-00-00-00-05, 00-00-00-00-00-08, 00-00-00-00-00-09]` was chosen to be installed. This path clearly avoids the direct link between `00-00-00-00-00-04` and `00-00-00-00-00-08`, which is currently running at roughly 15x higher load then the links on subpath `[00-00-00-00-00-04, 00-00-00-00-00-05, 00-00-00-00-00-08]`.

The ping command itself never failed. As expected, differences in packet latency were observed, as the first packet, has to travel from the switch to controller and back in order to be inspected, so that appropriate flow entries may be installed on all switches involved. All following packets are then simply forwarded, which significantly reduces their latency. Mininet's console output is shown Figure 11.5.

```
...
mininet> h1 ping h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_req=1 ttl=64 time=157 ms
64 bytes from 10.0.0.4: icmp_req=2 ttl=64 time=2.23 ms
64 bytes from 10.0.0.4: icmp_req=3 ttl=64 time=0.617 ms
64 bytes from 10.0.0.4: icmp_req=4 ttl=64 time=0.676 ms
64 bytes from 10.0.0.4: icmp_req=5 ttl=64 time=1.09 ms
64 bytes from 10.0.0.4: icmp_req=6 ttl=64 time=0.468 ms
64 bytes from 10.0.0.4: icmp_req=7 ttl=64 time=0.482 ms
64 bytes from 10.0.0.4: icmp_req=8 ttl=64 time=0.656 ms
64 bytes from 10.0.0.4: icmp_req=9 ttl=64 time=0.510 ms
....
64 bytes from 10.0.0.4: icmp_req=104 ttl=64 time=0.546 ms
^C
--- 10.0.0.4 ping statistics ---
104 packets transmitted, 103 received, 0% packet loss, time 103140ms
```

Figure 11.5: Mininet's console output during the Zero link-load threshold test. Latency of the first packet is significantly higher than the latency of all following packets, as only the first one has to be examined by controller.

11.1.2 Non-zero link-load threshold test

The non-zero threshold test was performed by appropriate use of `--th` command line parameter of the application. The application was started by executing `./pox.py samples.pretty_log openflow.discovery openflow.spanning_tree load_balancing_d --th=0.043`, thus setting the threshold value to 0.043. The value was chosen based on the Zero link-load threshold test resulting link-loads (see Section 11.1.1) in order to require more then one ping session to break the threshold value, and thus induce higher weight of the corresponding edge in graph. The link will then appear to suffer higher load then others, which will in the end force some longer path with better capacity to be chosen.

An xTerm instance was started from the Mininet's console for hosts H1, H2 and H3 in order to run parallel pings towards hosts H4 and H5.

Firstly, only ping between H1 and H4 was ran, which did not generate enough link load to break the threshold, thus the same path has been chosen every time recalculation occurred.

Secondly, ping between H2 and H5 was started. At this point both paths between H1 and H4 as well as H2 and H5 were using the shortest physical path (over link `00-00-00-00-00-04` and `00-00-00-00-00-08`), however even this additional amount of traffic did not cause the threshold to be broken on link between `00-00-00-00-00-04` and `00-00-00-00-00-08`.

In the end ping between H3 and H5 was started. Initially the shortest physical path was chosen as well, but once new statistics were gathered, edge weights update was triggered, which shown broken threshold on link `00-00-00-00-00-04 - 00-00-00-00-00-08`. This then resulted into longer, thus better paths being calculated, when a new flow occurred.

The application output was as expected. Initially, all link weights in the graph were set to 0. The same physically shortest paths were being calculated until the threshold on one of the links was broken. Accordingly, the edge weight changed from 0 to a higher value on the link, that has reached the link-load threshold. All the following path calculations were then taking this update into account, and hence paths were calculated so that they would avoid the heavily loaded link.

Application output of this test is depicted in Figure 11.6. Commented lines show, when and which of the ping commands mentioned earlier was started.

```
$ ./pox.py samples.pretty_log openflow.discovery openflow.spanning_tree load_balancing_d --th=0.043
POX 0.1.0 (betta) / Copyright 2011-2013 James McCauley, et al.
|Hard timeout: 30  |Idle timeout: 20  |Load threshold: 0.043
[core                  ] POX 0.1.0 (betta) is up.
...
[load_balancing_d      ] Constructing graph G:
[load_balancing_d      ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-02 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-01 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-03 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-05 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-06 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-07 and 00-00-00-00-00-06 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-05 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-07 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-09 and 00-00-00-00-00-08 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-0a and 00-00-00-00-00-08 with a weight of 0 (threshold not reached)
######## H1 ping H4 started
[load_balancing_d      ] Installing path: [00-00-00-00-00-01, 00-00-00-00-00-04, 00-00-00-00-00-08, 00-00-00-00-00-09]
[load_balancing_d      ] Installing path: [00-00-00-00-00-09, 00-00-00-00-00-08, 00-00-00-00-00-04, 00-00-00-00-00-01]
...
[load_balancing_d      ] Constructing graph G:
[load_balancing_d      ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-02 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-01 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-03 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-05 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-06 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-07 and 00-00-00-00-00-06 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-05 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-07 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-09 and 00-00-00-00-00-08 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-0a and 00-00-00-00-00-08 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Installing path: [00-00-00-00-00-01, 00-00-00-00-00-04, 00-00-00-00-00-08, 00-00-00-00-00-09]
[load_balancing_d      ] Installing path: [00-00-00-00-00-09, 00-00-00-00-00-08, 00-00-00-00-00-04, 00-00-00-00-00-01]
######## H2 ping H5 started
[load_balancing_d      ] Installing path: [00-00-00-00-00-02, 00-00-00-00-00-04, 00-00-00-00-00-08, 00-00-00-00-00-0a]
[load_balancing_d      ] Installing path: [00-00-00-00-00-0a, 00-00-00-00-00-08, 00-00-00-00-00-04, 00-00-00-00-00-02]
...
[load_balancing_d      ] Constructing graph G:
[load_balancing_d      ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-02 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-01 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-03 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-05 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-06 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-07 and 00-00-00-00-00-06 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-05 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-07 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-09 and 00-00-00-00-00-08 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-0a and 00-00-00-00-00-08 with a weight of 0 (threshold not reached)
######## H3 ping H5 started
[load_balancing_d      ] Installing path: [00-00-00-00-00-01, 00-00-00-00-00-04, 00-00-00-00-00-08, 00-00-00-00-00-09]
[load_balancing_d      ] Installing path: [00-00-00-00-00-02, 00-00-00-00-00-04, 00-00-00-00-00-08, 00-00-00-00-00-0a]
[load_balancing_d      ] Installing path: [00-00-00-00-00-0a, 00-00-00-00-00-08, 00-00-00-00-00-04, 00-00-00-00-00-03]
[load_balancing_d      ] Installing path: [00-00-00-00-00-03, 00-00-00-00-00-04, 00-00-00-00-00-08, 00-00-00-00-00-0a]
[load_balancing_d      ] Installing path: [00-00-00-00-00-0a, 00-00-00-00-00-08, 00-00-00-00-00-04, 00-00-00-00-00-02]
...
[load_balancing_d      ] Constructing graph G:
[load_balancing_d      ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-02 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-01 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-04 and 00-00-00-00-00-03 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-05 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-06 and 00-00-00-00-00-04 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-07 and 00-00-00-00-00-06 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-05 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-07 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-08 and 00-00-00-00-00-04 with a weight of 0.0533033333333
[load_balancing_d      ] Adding edge between 00-00-00-00-00-09 and 00-00-00-00-00-08 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Adding edge between 00-00-00-00-00-0a and 00-00-00-00-00-08 with a weight of 0 (threshold not reached)
[load_balancing_d      ] Installing path: [00-00-00-00-00-01, 00-00-00-00-00-04, 00-00-00-00-00-05, 00-00-00-00-00-08, 00-00-00-00-00-09]
[load_balancing_d      ] Installing path: [00-00-00-00-00-03, 00-00-00-00-00-04, 00-00-00-00-00-05, 00-00-00-00-00-08, 00-00-00-00-00-0a]
[load_balancing_d      ] Installing path: [00-00-00-00-00-09, 00-00-00-00-00-08, 00-00-00-00-00-05, 00-00-00-00-00-04, 00-00-00-00-00-01]
[load_balancing_d      ] Installing path: [00-00-00-00-00-0a, 00-00-00-00-00-08, 00-00-00-00-00-05, 00-00-00-00-00-04, 00-00-00-00-00-03]
######## longer paths with lower load were chosen - subpath [00-00-00-00-00-08, 00-00-00-00-00-05, 00-00-00-00-00-04]
######## instead of subpath [00-00-00-00-00-08, 00-00-00-00-00-04]
...
```

Figure 11.6: Application output showing the course of the Non-zero link-load threshold test is shown including ping commands chronologically as they were issued during the tests.

11.1.3 Dijkstra's vs Floyd-Warshall algorithm

Both application variants (using Dijkstra's or Floyd-Warshall algorithm) were tested. The considerations for using one or the other version were already stated in Sections 9.7 and

9.7.1.

Unfortunately, the test environment did not offer an opportunity to create large and complex enough environment to truly test the full load performance of the application. This is caused by the fact, that the physical computer's resources have to be distributed across all the virtual hosts, switches and controller. Some links were flapping even in the relatively simple test topology used. Hence, in this test setup no differences in the application's performance were observed as the network virtualization itself was taking too much computational power.

However, the principle remains that Floyd-Warshall variant would be better off in dense networks (with large number of interconnecting links) and with a large number of flows emerging within the 30 seconds statistics collection period. In such case, the path calculation for all source-destination pairs would appear only once per 30 seconds. On the other hand, in less complex networks with a new flows emerging relatively sporadically, the Dijkstra's variant would perform better, as path calculations would appear only on demand. In such case, all the computations together would take less time than running the Floyd-Warshall algorithm.

11.2 Physical testbed results anomalies

Transition from virtual network environment to a slice of network with physical devices was not seamless and different OpenFlow protocol errors, namely `OFPET_BAD_ACTION - OFPBAC_BAD_OUT_PORT` and `OFPET_BAD_REQUEST - OFPBRC_BAD_TYPE` occurred (see Figures 11.7 and 11.8 correspondingly). Additionally, the POX's `spanning_tree` component was not working properly either, even though the POX controller version was the same, etc. The spanning tree malfunction caused endless flooding of flooding across the entire test topology (see Figure 9.1). Even though the flooding occurred, paths for emerging flows were calculated correctly. However, in the end they failed to install on the physical switches.

```
...
[openflow.of_01        ] [02-08-00-00-00-02|520 2] OpenFlow Error:
[02-08-00-00-00-02|520 2] Error: header:
[02-08-00-00-00-02|520 2] Error:   version: 1
[02-08-00-00-00-02|520 2] Error:   type:    1 (OFPT_ERROR)
[02-08-00-00-00-02|520 2] Error:   length:  76
[02-08-00-00-00-02|520 2] Error:   xid:     79
[02-08-00-00-00-02|520 2] Error: type: OFPET_BAD_ACTION (2)
[02-08-00-00-00-02|520 2] Error: code: OFPBAC_BAD_OUT_PORT (4)
[02-08-00-00-00-02|520 2] Error: datalen: 64
[02-08-00-00-00-02|520 2] Error: 0000: 01 0d 00 10 00 00 03 88  70 b2 cc d1 00 1a 00 00   ........p.......
[02-08-00-00-00-02|520 2] Error: 0010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
[02-08-00-00-00-02|520 2] Error: 0020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
[02-08-00-00-00-02|520 2] Error: 0030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
...
```

Figure 11.7: Application output showing the `OFPET_BAD_ACTION - OFPBAC_BAD_OUT_PORT` error.

```
...
[openflow.of_01        ] [00-24-a8-da-80-00|4000 1] OpenFlow Error:
[00-24-a8-da-80-00|4000 1] Error: header:
[00-24-a8-da-80-00|4000 1] Error:   version: 1
[00-24-a8-da-80-00|4000 1] Error:   type:    1 (OFPT_ERROR)
[00-24-a8-da-80-00|4000 1] Error:   length:  20
[00-24-a8-da-80-00|4000 1] Error:   xid:     300
[00-24-a8-da-80-00|4000 1] Error: type: OFPET_BAD_REQUEST (1)
[00-24-a8-da-80-00|4000 1] Error: code: OFPBRC_BAD_TYPE (1)
[00-24-a8-da-80-00|4000 1] Error: datalen: 8
[00-24-a8-da-80-00|4000 1] Error: 0000: 01 12 00 08 00 00 09 97                          ........
...
```

Figure 11.8: Application output showing the `OFPET_BAD_REQUEST - OFPBRC_BAD_TYPE` error.

The issues were investigated. The spanning tree malfunction was discussed earlier on the openflow-discuss mailing list. As the POX's developer Murphy McCauley suggests in his post, the FlowVisor's discovery mechanism interferes with POX's discovery mechanism (the `openflow.discovery` component) [22]. The mechanism is required when running both the resulting application as well as the `spanning_tree` component (needed to cope with topology loops).

In the same e-mail Mr McCauley also suggests, that for the very same reason also the `forwarding.l2_multi` component is not functioning properly together with FlowVisor software. It could not install an end-to-end path in FlowVisor environment. This is most likely the explanation of why the resulting application was not functioning properly in the OFELIA testbed either, as it is based on the `forwarding.l2_multi` component. Thus, the OpenFlow `OFPET_BAD_ACTION - OFPBAC_BAD_OUT_PORT` (bad output port in flow entry action) error occurs as shown in Figure 11.7. Another hint, why this error is probably bound with FlowVisor is that flow entries failed to install on both Hewlett-Packard and NEC switches, thus the error is probably not vendor specific.

Unfortunately, these issues were not known at the time when the POX controller and OFELIA testbed were being chosen for development.

As far as the `OFPET_BAD_REQUEST - OFPBRC_BAD_TYPE` error is concerned, it occurred only on the Hewlett-Packard ProCurve 3500 (ID: 00-24-a-da-80-00 as seen in Figures 11.8 and 11.9) switch, right after connection between the switch and controller is established. The fact, that this error never occurred on NEC switches nor in the virtual testbed, suggests some deviation in Hewlett-Packard's implementation of the OpenFlow v1.0 standard.

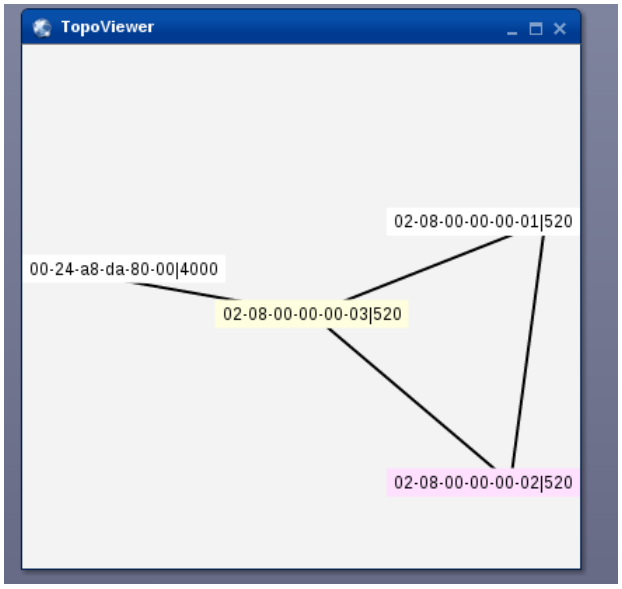


Figure 11.9: Physical networking testbed (including switch IDs) in OFELIA (see also Figure 9.1).

11.3 Comparison with CBP in load balancing

Evaluation of the application calls also for comparison with current best practices in load balancing, particularity in legacy networks (the techniques were discussed in Section 7).

The OSPF routing protocol allows load balancing only across (up to four) equal cost paths, hence results comparable with the resulting application (longer path with higher cost being preferred compared to shorter path, which is overloaded) can not be achieved in network running just OSPF. As for EIGRP protocol, it is possible to add not only equal but also higher cost paths to the load balancing scheme. However, it can not operate in reactive manner (e.g. according to changing link loads), as the topology is recalculated only if for example new network is advertised, link goes down, etc. Forcing the EIGRP to recalculate periodically by e.g. flapping one of the links is considered to bring more harm than benefit.

In MPLS networks, load balancing may be performed by running the MPLS-TE (Traffic Engineering) extension. It then uses RSVP(-TE) protocol for resource allocation. Load balancing may be performed across LSPs with the same metric. One or other LSP is chosen randomly or the traffic may be spread across all of them by enabling per-packet load balancing. Additionally, it is possible to assign a portion of the traffic to each LSP (e.g. 50% of the flow will be sent over LSP1, 40% over LSP2 and 10% over LSP3). However, it is only a resource reservation, it does not take dynamically changing network conditions into account.

Compared to the CBP in load balancing for legacy networks, the resulting application for OpenFlow networks takes full advantage of network statistics gathering and topology calculations performed centrally by controller. Hence, most suitable paths are calculated according to current conditions. It does not reserve resources for future use as it is not possible to fully predict, how the network will evolve. Instead, it continuously monitors the network and makes informed decisions based on most recent statistics.

In general, the software-defined networking concept allows, thanks to its layered architecture and standardized southbound API (e.g. OpenFlow), efficient data collection about traffic flows, link events, etc. from the whole network. Therefore, network management applications running above the controller may flexibly react to current network conditions, rather then making uninformed resource reservations for future flows based on expectations about what their characteristics might be (required bandwidth, latency, etc.), which is how the currently used techniques usually work.

Chapter 12

Future work

12.1 Ford-Fulkerson algorithm

Dijkstra's shortest path algorithm may be as well replaced by the Ford-Fulkerson algorithm which is taking into account the remaining capacities of links, when searching for the maximum flow path. Although, similarly to the original solution deploying Dijkstra's algorithm, the new application would require an enhancement to the Ford-Fulkerson algorithm as well. The standard implementation may lead to choosing longer paths (with unreasonably large number of hops) in order to ensure the maximum available capacity of path.

An upgrade to the algorithm would then be looking for shorter paths with not the highest but yet sufficient enough capacity to carry the new flow data.

12.2 Wildcard masks in flow entries

Exact source and destination address match flow entries are being deployed on the switches in the current application version. This was considered to be sufficient for the testing purposes as well as for the small sized networks. However, such model may not scale well in large-scale networks with large amount of hosts. Deployment of exact match rules on all the switches may lead to flow tables' overload and further to decreased performance of devices. Therefore deployment of enhanced (not only protocol) wildcard flow entries especially on the network's core/backbone switches would be desirable in order to prevent the mentioned behavior.

12.3 Flow entry timeouts

Intelligent flow entry timeouts scheme would likely improve performance of the load balancing application especially in terms of lowering the average flow latency. In the current setup, flow expires every 30 seconds. Once the flow entry expires and if the flow is still active, the `OFPT_PACKET_IN` message is received by controller and path calculation occurs. This additional computing is the source of increased latency of the packet.

Adjustments of the timeouts could be implemented for example in the following way: timers would be increased by a value of e.g. 15 seconds every time the `OFPT_PACKET_IN` message from an existing still active flow is received by the controller.

12.4 OpenFlow group tables support

Flow entries may be bundled into groups since the OpenFlow v1.1.0, which allows for more complex forwarding schemes like multipath or link aggregation [10]. Such functionality is highly desirable when considering load balancing deployment. The abstraction enables developers to better utilize multiple switch interconnections, where the current application version can not take advantage of such setup.

Nowadays, it is common that there are multiple parallel physical connections between network's core switches. Those ports are then usually aggregated into port channels. Thus it is desirable to have such an abstraction also in the OpenFlow environment. Sadly, only OpenFlow v1.0 is currently the highest version supported both in hardware and in majority of available OpenFlow controllers.

Group tables functionality would also support the development of server load balancing plug-in to the application. Whereas in current setup the application provides network load balancing only.

12.5 ARP proxy

Implementing an ARP proxy functionality to the application is considered to bring a couple of benefits. Firstly, flooding of ARP requests would be minimized and thus network load would be decreased. Secondly, the address resolution process would occur faster since the ARP proxy controller component would send the ARP replies to the request's source right away.

12.6 Additional statistics sources

Next generations of the load balancing application could take advantage of other common monitoring techniques in addition to flow entry counters and port statistics obtainable through OpenFlow. Simple Network Management Protocol (SNMP) could be used for switch monitoring (port utilization, etc.) or data collected by NetFlow could serve as an additional source of information about the network traffic.

Use of such technologies would be valuable also during transition phase from legacy to software-defined networking. It would enable the load balancing application to make smarter decisions as it would have had knowledge of both legacy as well as the OpenFlow enabled part of the network.

Chapter 13

Conclusion

The principals of software-defined networking (e.g. its layered architecture) were introduced and compared to the traditional „non-software-defined“ networking technology characteristics. The advantages and disadvantages of both approaches were stated. The OpenFlow protocol was introduced including a comparison of its most recent versions. OpenFlow's key architectural components such as flow tables were described as well.

Current best practices of load balancing in computer networks such as MPLS Traffic Engineering were discussed and compared to the resulting OpenFlow load balancing application.

The load balancing application was developed in an iterative manner. The application maintains a graph representation of the managed network, with network devices (switches) represented as vertices and interconnecting links represented by edges. The final application adjusts edge weights in the graph based on current load of corresponding links in the network. Dijkstra's algorithm is then used for the shortest path calculation in the graph every time a new flow emerges (specified by a source and destination IP address). The application' s functionality may be adjusted by user through use of command line arguments. Additionally, wildcard match fields are used in flow entries to allow match for any protocol (e.g. ARP, IP) in case source and destination IP addresses remain the same.

Alternatively, variant using Floyd-Warshall instead of Dijkstra's algorithm was developed as well. In this case shortest path calculation occurs only once per the time period between new collection of link-loads and edge weights update takes place. This may lead to overall more effective computations, especially in dense network topologies.

POX OpenFlow controller was used as a base for implementation of the application. A virtual lab test environment was build using Mininet and further used during development process. Additionally, slice of a physical OpenFlow network provided by OFELIA project was used in order to test the resulting OpenFlow application on hardware devices (by Hewlett-Packard and NEC). The application uses OpenFlow protocol v1.0 as at the time of development it is the highest version supported by both POX controller as well as the networking hardware available in OFELIA.

The resulting application was subject to tests in both mentioned environments. No differences in computational performance between the Floyd-Warshall and Dijkstra's version of the application were observed, most probably due to the fact, that it was not possible to create a complex enough test network (with large number of switches and hosts), which would lead to large enough number of path computation requests.

In the virtual test network environment, the application is operating as expected. New flow entries are installed every time a new flow (pair of source and destination IP addresses)

occurs. The path calculation takes current link-loads into consideration. Therefore, links experiencing higher loads are avoided in favor of less utilized links.

As far as tests in the physical network environment are concerned, it was not possible to properly run the application within the OFELIA workspace. OFELIA uses FlowVisor software to divide their physical resources into experimental slices. Unfortunately, POX controller and FlowVisor software interfere with each other, thus preventing OpenFlow applications running above POX to function properly. Yet, basic differences between OpenFlow v1.0 implementation by different vendors were observed.

Further improvements of the application including new features like ARP proxy or enhancements to be possibly brought by higher versions of OpenFlow protocol were discussed as well.

Bibliography

[1] Inc. Arista Networks. OpenFlow - Arista [online]. http://www.aristanetworks.com/en/products/eos/openflow, [cit. 2013-05-05].

[2] Brandon Heller Bob Lantz, Nikhil Handigol and Vimal Jeyakumar. Introduction to Mininet [online]. https://github.com/mininet/mininet/wiki/Introduction-to-Mininet, [cit. 2013-05-06].

[3] Brocade. Brocade MLX Series [online]. http://www.brocade.com/products/all/routers/product-details/netiron-mlx-series/index.page, [cit. 2013-05-05].

[4] Inc. Cisco Systems. How Does Unequal Cost Path Load Balancing (Variance) Work in IGRP and EIGRP? [pdf]. http://www.cisco.com/image/gif/paws/13677/19.pdf, 2009-05-03 [cit. 2012-01-06].

[5] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[6] NetworkX developer team. NetworkX [online]. http://networkx.github.io/, [cit. 2013-05-08].

[7] David Erickson. What is Beacon? [online]. https://openflow.stanford.edu/display/Beacon/Home, 2012-06-22 [cit. 2012-01-05].

[8] Floodlight. Floodlight is an Open SDN Controller [online]. http://www.openflow.org/wp/snac/, 2011 [cit. 2012-01-05].

[9] FlowVisor. FlowVisor [online]. https://github.com/OPENNETWORKINGLAB/flowvisor/wiki, [cit. 2012-01-05].

[10] Open Networking Foundation. OpenFlow Switch Specification Version 1.1.0 Implemented [pdf]. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf, 2011-02-28 [cit. 2013-05-10].

[11] Open Networking Foundation. Software-Defined Networking: The New Norm for Networks [white paper]. https://www.opennetworking.org/images/stories/downloads/white-papers/wp-sdn-newnorm.pdf, 2012-04-13 [cit. 2012-12-09].

[12] Open Networking Foundation. OpenFlow Switch Specification Version 1.3.0 [pdf]. https://www.opennetworking.org/images/stories/downloads/specification/openflow-spec-v1.3.0.pdf, 2012-06-25 [cit. 2012-12-10].

[13] Open Networking Foundation. Members [online]. https://www.opennetworking.org/membership/members, [cit. 2012-01-03].

[14] Open Networking Foundation. HP OpenFlow capable firmware is now GA [online]. http://www.openflow.org/wp/2011/12/hp-openflow-capable-firmware-is-now-ga/, [cit. 2012-12-09].

[15] Open Networking Foundation. HP ProCurve [online]. http://www.openflow.org/wp/switch-hp/, [cit. 2012-12-09].

[16] Open Networking Foundation. OpenFlow Videos [online]. https://www.opennetworking.org/videos, [cit. 2012-12-09].

[17] Python Software Foundation. Python Programming Language [online]. http://www.python.org/, [cit. 2013-05-08].

[18] L.P. Hewlet-Packard Development Company. HP Simplifies Networking with Broadest Choice of OpenFlow-enabled Switches [online]. http://www.hp.com/hpinfo/newsroom/press/2012/120202a.html, 2012-02-02 [cit. 2012-12-09].

[19] L.P. Hewlett-Packard Development Company. Production-ready SDN with OpenFlow 1.3 [online]. http://www.openflow.org/wp/switch-hp/, [cit. 2013-05-05].

[20] Umesh Lakshman and Lancy Lobo. MPLS Traffic Engineering [online]. http://www.ciscopress.com/articles/article.asp?p=426640&seqNum=2, 2006-01-13 [cit. 2012-01-06].

[21] Maestro-platform. Maestro [online]. http://code.google.com/p/maestro-platform/, [cit. 2012-01-05].

[22] Murphy McCauley. [openflow-discuss] The Flowvisor doesn't support spanning_tree [mailing list]. https://mailman.stanford.edu/pipermail/openflow-discuss/2012-November/003795.html, [cit. 2013-05-19].

[23] MurphyMc. POXDesk wiki [online]. https://github.com/MurphyMc/poxdesk/wiki, [cit. 2013-05-12].

[24] MurphyMc. Getting Started [online]. https://github.com/MurphyMc/poxdesk/wiki/Getting-Started, [cit. 2013-05-20].

[25] NOX. About NOX [online]. http://www.noxrepo.org/nox/about-nox/, [cit. 2012-01-05].

[26] NEC Corporation of America. NEC ProgrammableFlow Networking [online]. http://www.necam.com/SDN/, [cit. 2013-05-05].

[27] Ofelia. About OFELIA [online]. http://www.fp7-ofelia.eu/about-ofelia/, [cit. 2013-05-06].

[28] OFELIA. OFELIA tutorial [online].
http://www.fp7-ofelia.eu/assets/Uploads/OFELIA-Tutorial.pdf, [cit. 2013-05-20].

[29] A Linux Foundation Collaborative Project. OpenDaylight. Technical Overview — OpenDaylight [online].
http://www.opendaylight.org/project/technical-overview, [cit. 2013-05-05].

[30] Ivan Pepelnjak. Should I Be Interested in MPLS Traffic Engineering? [online].
http://www.ciscopress.com/articles/article.asp?p=30436&seqNum=2, 2003-01-03 [cit. 2012-01-06].

[31] Enterprise Networking Planet. Cisco and Juniper Reveal SDN Strategies [online].
http://www.enterprisenetworkingplanet.com/datacenter/cisco-and-juniper-reveal-sdn-strategies.html, [cit. 2013-05-05].

[32] SNAC. Simple Network Access Control (SNAC) [online].
http://www.openflow.org/wp/snac/, [cit. 2012-01-05].

[33] Open Networking Summit. Open Networking Summit [online].
http://www.opennetsummit.org/, [cit. 2013-05-05].

[34] Mininet Team. Download/Get Started With Mininet [online].
http://mininet.org/download/, [cit. 2013-05-20].

Appendix A

Installation notes

POX controller, POXDesk web GUI and Mininet require Python programming language libraries installed on the host machine by running `sudo apt-get install python` (in case they have not been installed yet). Additionally, NetworkX package must be present on the machine by executing `sudo easy_install networkx`.

A.1 POX controller

POX controller may be installed simply by downloading the code from POX's repository using the command `git clone http://github.com/noxrepo/pox`. The resulting application component `load_balancing_[d|fw]` should then be copied into `pox/ext` directory in order to be recognized by the controller and hence be ready to load. POX may then be started as `./pox.py component1 component2...componentX`.

A.1.1 POXDesk

For POXDesk web GUI component can be downloaded through `git clone https://github.com/MurphyMc/poxdesk` into `pox/ext` directory [24]. When loading POXDesk (`./pox.py poxdesk`) it is necessary to invoke also POX's components `web messenger messenger.log_service messenger.ajax_transport openflow.of_service poxdesk.tinytopo`.

A.2 Mininet

The Mininet virtual networking testbed can be installed natively on user's Linux system (which requires additional software packages like Open vSwitch to be installed). However, it is recommended to take advantage of ready-to-use Mininet virtual machine provided by Mininet Team, which has already all the needed software packages installed. Steps for the VM's installation are described at [34].

Virtual networking testbed topology (available on the attached CD), may then be installed by copying the source code into `mininet/custom/` folder. To start the topology execute command `sudo mn --topo stat10 --custom staticlb10.py --controller remote`. The portion `--controller remote` tells Mininet, that an external OpenFlow controller will be used (for POX installation see Section A.1).

To ping between hosts issue `hX ping hY`, for additional help type `help`.

A.3 OFELIA

As far as OFELIA physical networking testbed is concerned, it was firstly necessary to register at the project's website. Then a user is allowed to setup VPN connection with the chosen testbed's island. Once the user's connection is established, he/she may specify their project and request a slice of OFELIA's network. An OFELIA tutorial may be found at [28].

The POX controller installation follows the same steps as described in Section A.1.

Appendix B

Application manual

B.1 Requirements

The application requires to be developed and tested in a Linux environment. Python and POX controller have to be both present on the system. Mininet installation or virtual machine is recommended for testing purposes, however it is not necessary to run the application. In order to meet the application's requirement, follow the steps in Appendix A.

The application is distributed under GNU GPL license.

B.2 User specified options

Once the application environment is up and running (see Appendix A), user my run the application by executing `./pox.py openflow.discovery openflow.spanning_tree load_balancing_[d|fw]` in the Linux command line, where `d` and `fw` stand for Dijkstra's and Floyd-Warshall variant. This essential introduces the load balancing application as component of POX and automatically sets the user options to their default values (see Table B.1).

In order to run the controller application on a specific port execute `./pox.py openflow.of_01 --port=PORT` (default OpenFlow port is 6633). For improved readability of the application output it is recommended to invoke the `samples.pretty_log`. Logging level may be set by `./pox.py --verbose|debug|info|warning` (`info` being default).

Sample command line would then be e.g. `./pox.py openflow.of_01 --port=6633 samples.pretty_log openflow.discovery openflow.spanning_tree load_balancing_[d|fw] --ht=30 --it=20 --th=0.5`.

Option	Description	Mandatory	Default value
`--help`	Prints user help message	No	False
`--ht=NUMBER`	Sets flow entry hard timeout to NUMBER seconds	No	30 s
`--it=NUMBER`	Sets flow entry idle timeout to NUMBER seconds	No	20 s
`--th=NUMBER`	Sets link-load threshold	No	0.5

Table B.1: User specified command line options of the application (same for Dijkstra's and Floyd-Warshall variant.

Appendix C

CD contents

Executable source codes of both variants of the application as well as this technical report (including LaTeX source codes) are present on a CD, which comes with the thesis.

Sample Mininet topology (used for application testing) is also attached on the CD.