Universiteit Leiden
The Netherlands

Opleiding Informatica

Recovering from Controllers Failures

in Software Defined Networks

J. Q. Bouman

Supervisors:
M. M. Bonsangue & Feng Hui

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

27/06/2019

Abstract

The separation of the data and control plane of networks is at the core of the emergence of Software Defined Networks. The separation of these two planes boils down into introducing a separate piece of software in the network, the controller, the brain of the network. Therefore a resilient connection between infrastructure and controller is of great importance. In this thesis we approach the problem of a failing controller in a multi controller setup, by comparing the implementation of Open vSwitch with a self build solution on the infrastructure side of the entire network. Few experiments show that the self made implementation under perform when compared to the existing implementation. Approaches from different perspectives, e.g. from the control plane, build into the controller, would be a logical solution.

Contents

Chapter 1

Introduction

The current form in which computer networks work start to reach it's limits. Therefore new and more andvanced way of networking is required. Since pure hardware accelaration reached the limit of their potential, multiple companies independently started to create ways in which their networking products become more superior above the others. All these new ideas are revolving around the concept of having one hypervisor or control server, which has an understanding of the full networking tree, controls all the switches within the tree. The problem with all these implementations is a lack of interoperability, this problem can be solved with the use of Software Defined Network Controller. Even though one controller can control a large network, when the controller stops working the networking can come to a halt. Therefore redundancy needs to be implemented. In this thesis we discuss the problem of recognizing when a controller halts combined with the reassignment of switches to working controllers.

1.1 Thesis Overview

This chapter contains the introduction; Chapter 2 includes an overview of Software Defined Networks and of all software tools we used in our experiments; Chapter 3 explains the setup of the experiments and evaluates the results; Chapter 4 discusses related work; and we draw some conclusions in Chapter 5.
This work is presented as a bachelor thesis at LIACS under the supervision of M.M. Bonsangue and Feng Hui.

Chapter 2

Software Defined Networks

2.1 Model

SDNs are a new vision on networking, the base of the idea is the separation of the control and the data plane. Nowadays switches need to know a lot about their neighbouring network devices in order to get a more efficient connection. The switches need to know not only which devices are directly connected to their ports, but also the devices connected to their neighbours. Especially in larger networks like datacenters this amount of information will quickly add up. As a consequence, knowledge about the network, switches hardware need to rise with it, which results in more expensive switches. In order to tackle the problem, plane separation seems to be a plausible solution. The implementation of the separation in Software Defined Networks is by defining a controller which has knowledge about the whole network, or at least about the part it controls By removing the control out of the switches the networking hardware can be significantly simplified.
Moving towards Software Defined Networks is a big step for network manufactureers to make, setting aside all their privatly funded research and implementations, their slight edge above the competitors and potentially the lost of locked in customers.

The solution is more Openness. At this moment, due to the different implementations between manufacturers, when starting to build up a large network one is almost obligated to use the same brand network gear, since most of the software does not play well with others. If we can combine the introduction of SDNs together with more openness of the protocols, the network manufacturing market will be opened up for newcommers and hopefully, due to competition, drive down the costs for switches even further. This openness is what drives some of the software used in the expirements.

According to [1] a network is considered a software defined network if it has the following characteristics.

- Plane Separation

- Centralized Control
- Open
- Automated Network
- Simplified Device

Software Defined Networks is first and foremost the separation of the data plane from the control plane. The control plane will, in the new model be situated in a position where it will have more insight about the total network then one piece of the network will ever be able to know.
As partly insuated above the plane separation is acompanied by the centralisation of control, such that the central body has more information about the entirety of the network. The centralisation creates an easier base on which whole network applications can be implemented, since only one device has to be changed for a change in the entire network.
To be open SDN needs to have a non-propriatary, well documented API for both the North- and Southbound API, see the section below for more information about these terms.
In order to not roll back in features compared to the current generation networking hardware, the new SDN needs to be an automated network, which means that every newly added device in the network needs to able to connect without problems and be able to automatically discovered and added to the network topology.
Due to the plane separation and the centralized control, devices inside the network can be simplified. In an SDN setup, evere piece of the infrastructure itself does not need a great computing power to keep track of all the other devices in the network, but instead only needs to follow a specific rules.

2.1.1 The Controller

The controller is the base for software defined networks, the embodiement of the seperated control plane. It processes previously undefined packets and sets a pathway for potential likeable packages. Communications from and to the controller is devised into two parts, the Southbound API, which comunicates with the switches and other networking equipment, and the Northbound API which handles communications with the extensions. See Figure 2.1 for an overview of the controller.

Southbound API

The Southbound API is the part of the controller which communicates the devices in the network. Communication with the networking equipment consists of receiving packets and sending new rules or filters to the network switches. An example of one of these communication protocols is Openflow, however often OEMs have their own proprietary protocol.

Northbound API

The Northbound API refers to the protocols used by the the controller to connect to other applications. These applications can be anything to extend the functionality of the controller or in some cases the only way for the controller to inspect the packages. For example they can have extra information about traffic in the network, so the infrastructure can be prepared if a heavy network load is incomming. Another application type which is often suggested is for security purposes, to prevent hackers from reaching hosts connected to the infrastructure.

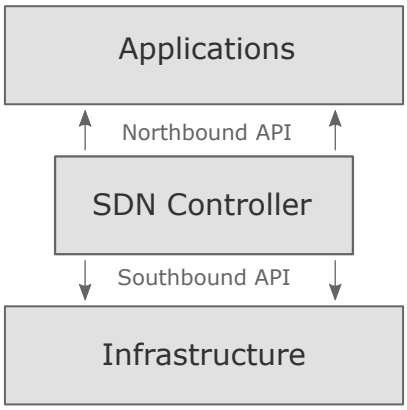


Figure 2.1: SDN Controller Diagram

2.1.2 Switches

The switches which are in use nowadays can be devided in three categories, depending on the amount of control the controller can have over these switches.
The most advantagious type of switch is a switch with full support for the OpenFlow protocol. Due to the characteristics of the OpenFlow protocol, the controller has full control of physical and virtual ports of the switch. See section 2.1.3.
On the middle ground, the manufacturer specific API. These kinds of switches only comunicate with the API build by the seller of the switch. Any form of control over these switches completely relies on the openness of the API and implementation of plane separation with software build by the manufacturer.
The least usefull switch in a software defined network setup is a switch with has no management over the ports.

2.1.3 OpenFlow

All packets received by a switch can be either dropped, forwarded or inspected. The action performed is determend based on the characteristics of the packet. These characteristics could for example be an IP Adres, a MAC Adress or a VLAN ID. Characteristics are stored by the switch in flow tables consisting of rules, every rule specifies the corresponding action. Rules do not only include physical but can also refer to virtual ports

on the switch. Every incoming package to a switch is handled via the OpenFlow protocol in the following way:

If a switch encouters a package which has properties that correspond to any of the stored rules in the switch, the package is forwarded/dropped/modified according to the action of the rule.
However when a switch encounters a package which does not correspond to any of the existing rules stored in the switch, the package or information about the package is send to the controller for inspection.
During the inspection the controller can not only extract the basic information about the package, like source, size, destination, lifetime, etc, but the controller can also, when in possession of the full package, extract information about the data inside the package and potentially peal off any layers of packaging containers, like IP, TCP/UDP, SMTP, etc.
After inspecting the package the controller decides what to do with this specific packet and creates a rule for future packets which will be filtered based on one or more properties of the package. The resulting rule will be send to the switch to update the flow table.
For all the following packages complying to the new rule received by the switch will follow the same action. These actions can consist of sending the package further through the network, changing the package and/or it's content or deleting the package entirely.
All the rules set by the controller are intended to be set by a specific time, however exceptions with indefinately lasting rules do occur.

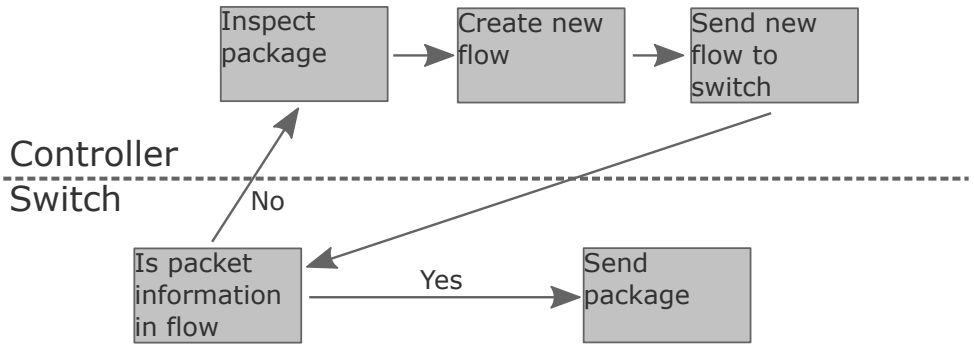


Figure 2.2: OpenFlow Packet Diagram

2.2 Software

In this section we explain all software tools and packages used in this work.

2.2.1 Oracle VM VirtualBox

Building machines for every different aspect of the network was not feasible for this project, therefore we used virtual machines instead. For the decision on Oracle's VirtualBox [2] was the recommandation by Mininet [3] the decisive factor, even though the software is slightly slower in tests compared to VMware [4]. We choose

VirtualBox, since VirtualBox is free and works on almost every system (OS X, Windows and Linux). The use of a virtual machine supports the possibility of suddenly stopping the controller software in order to simulate a crash.

2.2.2 Mininet

In this paper we focus on the type of switches with integrated OpenFlow protocol. Because these kind of physical switches were not easily accessible to us during the research period, we simulated the switches in a virtual machine running a software called Mininet [3]. Mininet can create hosts, switches and controllers to simulate a network and is focussed towards research, teaching and development.
Hosts within the network are end users or machines with no switching capabilities, however they can run (simple) applications.
Mininet has a collection of a few switch types, including Indigo Virtual Switch, Open vSwitch, User-space switch and the Linux Bridge. Indigo Virtual switch is a switch based on the Indigo Project by Floodlight [5]. Open vSwitch is a virtual switch with a wide variaty of connection capabilities, see 2.2.6. The Linux Bridge is a standard linux bridge from the bridge-utils library in the Linux kernel [6].
Within mininet there are four different types of controllers: NOX, OVSController, Ryu and the Remote controller. The NOX, OVSController and Ryu controller implementation do all start their respective controller from Mininet. The Remote controller starts a connection with a controller outside the reach of Mininet. The controller is specified with a name, ip address and a port on which the controller is listening.
A part of the Mininet software is a Command Line Interface, CLI for short. In this enviroment the whole network created within Mininet can be controlled. A feature of CLI is the ability to give commands to a host to start pinging another host. We use this feature of the CLI for our experiments in Chapter 3.

2.2.3 OpenDaylight

We use OpenDayLight [7] as controller in our work. Our choice is motivated by the numer of tutorials online using this specific controller, the reason being the ease of use. With the base installation, the controller does not have the logic to create new flows for switches, therfore a few features need to be installed as explained in 3.1.3. Features are extensions for Opendaylight which make use of the northbound API to connect to the controller.
The controller does have software to use the OpenFlow protocol preinstalled.

2.2.4 FlowVisor

Flowvisor calls itself *"... a transparant proxy between OpenFlow switches and multiple OpenFlow controllers."* [8] In this way it can control and redirect the data which is send within, in what they call a slice. The slices are

isolated and therefore cannot control traffic in another slice. A benefit of this slicing is that forwarded packages in the data layer are not delayed, this is inhereted from the model described by OpenFlow. FlowVisor creates an abstraction layer in which controllers and switches do not connect to eachother directly, the goal is to make the redistribution of connections between the control and data layer easier.

2.2.5 Wireshark

Wireshark [9] is a program which can sniff up and analyze network packets. Most of the openly available SDN controllers do not have a graphical user interface in which one can find the traffic to the controller or within the network. Wireshark however can pickup the packets send between the nodes in the Mininet network and between the mininet switches and the OpenDayLight controller, making easier the solving any problems within the network.

2.2.6 Open vSwitch

Open vSwitch is a virtual switch constructed to bridge traffic between different virtualisation enviroments. The switch is an upgrade from the Linux Bridge, since it has the capabilities to connect virtualisation enviroment from different machines together. Open vSwitch has many protocols build-in, one of which is the OpenFlow protocol. This vritual switch is implemented in the Mininet enviroment.

Chapter 3

Recovering Controllers Failures

3.1 Setup

The complete setup of our work is based in multiple virtual machines. Mininet will create a network in which the devices and switches do reside. Via the OpenFlow messaging protocol the switches inside Mininet communicate with the controller, Flowvisor is not used in the experiments as explained in 3.1.2. The controller used is the OpenDayLight controller, it has a simple setup and can be quickly setup to control a basic to medium network without the need of expertise of the complete software.

3.1.1 Mininet

Inside Mininet the hosts and switches are setup in a representation of a server rack setup, where multiple devices in a rack are connected to one switch and the switches are connected with each other as well. Only two server racks are used in this experiment to test the connection. Two server racks are enough since disconnecting one of the switches from their own controller and transfering it over to the other controller gives the complete cycle of changing controllers.
The Mininet virtual machine is a copy of the virtual machine provided by Mininet on their website [3]. The program which runs the infrastructure setup is based on a Python script with the Mininet API loaded in. The infrastructure created here is based on a datacentre setup with hosts in racks and switches on the top slot of rack to connect all the hosts and neigbouring switches. The setup consists of three hosts per switch with two racks, an overview can be found in Figure 3.1. The complete code for the created infrastructure can be found in Appendix A.

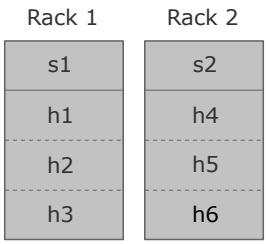


Figure 3.1: Rack Setup in Mininet

3.1.2 Flowvisor

On the middlegrounds between the controller and the switches we created a virtualization layer. This is implemented in such a way that both the controller and the infrastructure have no indication of the existence of the virtualization layer existence. Therefore controllers and parts of the infrastructure can be switched without any problems. However this implementation would only shift the problem from one place to another. For example, what if this virtualization layer is not accessible anymore? Probably the whole network would not be available. Therefore an implementation with virtualisation layer resulted a less practicle solution that we have abandoned.

3.1.3 OpenDaylight

The Opendaylight server is installed in a virtual machine with a base image of Debian.
As explained in 2.2.3 the OpenDaylight controller does out of the box not have the capabilities to create flows based on the information it receives from the switches, therefore the following features are installed:

- odl-l2switch-all
- odl-dlux-core
- odl-dluxapps-yangui
- odl-dluxapps-yangutils
- odl-dluxapps-topology
- odl-dluxapps-nodes

The “odl-l2switch-all” feature makes the controller capable of setting the right flows. This feature does add flows in such a way that the switch acts as a regular layer 2 switch. This means, at the moment a switch makes contact with the controller, OpenDaylight will directly send flows to the switch at the moment it detects a host, this is after the first ping.
The “odl-dlux-core” creates the graphical user interface via a webserver, however this is only the core feature. In order to display anything on the webserver, four dluxapps are installed, namely “odl-dluxapps-yangui”, “odl-dluxapps-yangutils”, “odl-dluxapps-topology” and “odl-dluxapps-nodes”.

"odl-dluxapps-yangui" and "odl-dluxapps-yangutils" create the basis for the webpages. "odl-dluxapps-topology" and "odl-dluxapps-nodes" show the user the topology of the discoverd network by the controller and a list of all the nodes with their respective flows. Webpages are used to check if a switch made a valid connection with the controller. The user interface is not needed for running our experiments.

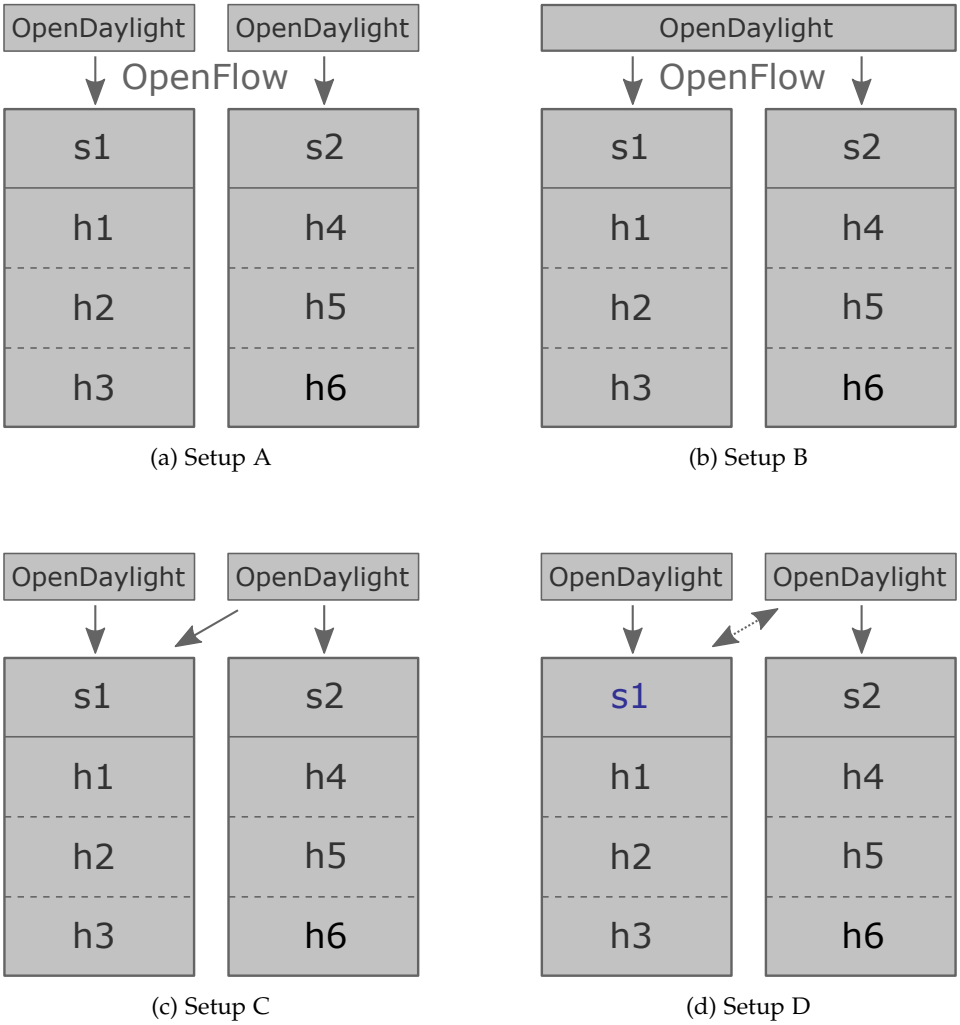
3.1.4 Implementing reconnections after a controller failure

In the experiments, the switch is purposly disconnected from the controller by shutting down the virtual machine in which the controller is running. Therefore the switch needs to be reconnected to another controller. The most elegant solution would be to change the connection by introducing the switch to another controller via a northbound API call. However the preinstalled northbound API and the extra installed features did not include any way to add a new switch to the controller. The only way for a switch to be added to the controller is via a southbound API call, in this case OpenFlow.
The next prossible way is to change the direction of the packets, however this has not the preference because of the following reasons. First of all this is a change which is not specified in the OpenFlow protocol, which would be hard to implement without deeper knowledge of the protocol. Second is it not sure that a new connection will be made at the moment the controller receives a random packet from a network device. Finally, it is not scalable, if the network increases dramatically the modification of packets would probably become a bottleneck for the Flowvisor program.
Changing the controller information in the switches itself seems to be the best possible option for now. Given the new controller information to the switch, the switch does automatically connect to the newly assigned controller. The main obstacle in this setup is the necessity for the switch to know where to find the second best controller.

3.1.5 Tests

The experiment is conducted with the setup as described in 3.1.1 and a double installation of the OpenDaylight controller as in 3.1.3. The combination of these two parts resulted in four different test setups as shown in Figure 3.2.

To create a baseline for the metrics we have collected, two different setups have been created. Setup A 3.2a is a dual controller with both controlling a separate switch. Setup B 3.2b has a single controller controlling both the switches, this setup is to confirm that controlling two switches does not have an impact on performance. The two following setups are two different implementations of reconnecting a switch to another controller. Setup C 3.2c uses the implementation of the Open vSwitch to connect to multiple controllers. Setup D 3.2d is our own implementation indicated by the blue text on the switch.
The Open vSwitch implementation in setup C connects to both controllers at the same time and mirrors all the



(a) Setup A

(b) Setup B

(c) Setup C

(d) Setup D

Figure 3.2: Four different test setups

messages to the controller. In our own implementation a Python thread keeps checking, in this case every one second, if the switch has lost connections. When the connection is lost the controller entry is removed from the switch, all the flows are removed and the new controller is added. The removal of flows is necessary to prevent collition of flows, when new flows are received from the second controller.

In both setup C and setup D, the left controller in the figure is stopped to check the connection. No controller is stopped in setup A and setup B.

In order to test the connections, ping requests are performed between “h1” and “h4”. Ten pings are performed back to back and the result of the pings are added to the result list. The command used in the Mininet CLI is the following *h1 ping -c 10 h4*. After this the Mininet enviroment is shutdown and restarted for the next test.

3.2 Results

For every setup described above, twenty tests are performed. The mean of the results is shown in Table 3.1. A full list of all the results can be found in the Appendix B. During the experiments one of the results introduced a significant outlier, while setup D was reconnecting to the controller. To show the impact of this result

Setups	Minimum	Average	Maximum	Packet Loss
Setup A	0.053	0.281	1.160	0.00
Setup B	0.052	0.320	1.078	0.00
Setup C	0.051	0.335	1.438	0.00
Setup D	0.200	12.878	50.906	0.23
Setup D*	0.215	0.403	1.040	0.21

Table 3.1: Mean Ping Time and Packet Loss

Setup D* is added to the result Table 3.1, this result is without the one outlier.
The packets send by the ping command have a time window of a full second to arrive at the receiver. The packet which created the outlier arrived at the end of the window after the new controller took over the control of the switch in which the packet was stuck. Therefore the travel time of this packet was a lot higher compared to the other packets.
Important to note is the packet loss which only occurs during setup D. Furthermore are the minimum and average mostly the same in setup A, B and C. While the average of setup D is slower.

3.3 Evaluating

Adding another switch does impact the controller a little bit, however setup B and setup C do not differ enough to make a hard conclusion out of the results. Our own implementation on setup D however does not give a good result. Improvements can be made by changing the time between checking the availability of the controller. Performing these checks more frequently would increase time spent checking the connectivity and possibly decrease the throughput of the switch. On the other hand, decreasing the frequency could increase package loss. So either the implementation itself needs to be significantly changed in order to minimize the gap compared to setup C or another implementation needs to be chosen. That is why an implementation where the controllers divide the switches between eachother seems to be a more promising option as seen in Chapter 4.

Chapter 4

Related Work

4.1 Towards an Elastic Distributed SDN Controller

In the paper called Towards an Elastic Distributed SDN Controller [10], the same problem as described in this paper, i.e. the abilty for multiple controllers to exchange nodes in the data plane, is approached. Their solution is the creation of a new controller which can have switches in both master and slave configuration. In their paper concerns about mininets abilty to test performance of a controller is brought up, however in this paper we use a base reference in order to check possible differences in response time and not the maximum performance of a controller based on the number of connections. Their implementation is an extension of the already existing OpenFlow protocol without creating more packet types while still adhering to the design rules of the origional protocol.

4.2 DISCO: Distributed Multi-domain SDN Controllers

The second paper called DISCO: Distributed Multi-domain SDN Controllers [11]. Presents an implementation on the connectivity between multiple SDN controllers. Even though the goal of the paper is not the same as discussed here, in their paper every controller needs to know more about the connections outbound to other SDN domains. The faced problems are the same, the robustness of the network in case a controller fails.

4.3 Controller Failover for SDN Enterprise Networks

The last paper called Controller Failover for SDN Enterprise Networks [12] gives an overview of a few different types of implementations to back up a controller failover, taken into consideration are 4 levels of standby,

none, cold, warm and hot, each level with more redundancy. The chosen implementation in their paper is comparable to the DISCO implementation where the controllers share information to avoid the single point of failure. However, in this paper, the owner of the controllers is assumed to be the same, which means there is less concerns about sharing information about the network between controllers.

Chapter 5

Conclusions

Software Defined Networks are a relatively new vision on current networking infrastructure. It has the potential to increase efficiency over existing connections due to the separation of the data and control layer of the network. At the core of the design is a controller which needs to have the power to make this all possible. In order to make the network more robust we inspected a way of reconnecting the network when a controller might not be able to manage the network anymore. The implementation we chose is based on the switch reconnecting to another controller in case the connection to the current controller was lost. Unfortunatly our implementation performed worse than an already existing implementation by the Open vSwitch. However research with another approaches could be more succesfull. Therefore would research in the capabilities of reconnecting switches to another controller based on logic within the controllers or based on an application which communicates to the controller via the northbound API be logical.

Bibliography

[1] P. Goransson and C. Black, "Software defined networks: a comprehensive approach," 2014. [Online]. Available: http://cds.cern.ch/record/1735404

[2] Oracle. Virtualbox. Accessed: 2019-01-20. [Online]. Available: https://www.virtualbox.org/

[3] M. Team. Download/get started with mininet. Accessed: 2019-02-12. [Online]. Available: http://mininet.org/download/

[4] I. VMware. Vmware. Accessed: 2019-01-22. [Online]. Available: https://www.vmware.com

[5] P. Floodlight. Indigo. Accessed: 2019-01-20. [Online]. Available: http://www.projectfloodlight.org/indigo/

[6] L. Foundation. bridge. Accessed: 2019-05-07. [Online]. Available: https://wiki.linuxfoundation.org/networking/bridge

[7] L. OpenDaylight Project a Series of LF Projects. Opendaylight. Accessed: 2019-01-25. [Online]. Available: https://www.opendaylight.org/

[8] A. Al-Shabibi. Flowvisor. Accessed: 2019-02-12. [Online]. Available: https://github.com/opennetworkinglab/flowvisor/wiki

[9] W. Foundation. Wireshark. go deep. Accessed: 2019-01-20. [Online]. Available: https://www.wireshark.org/

[10] A. A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. R. Kompella, "Towards an elastic distributed sdn controller," *Computer Communication Review*, vol. 43, no. 4, pp. 7–12, 2013.

[11] K. Phemius, M. Bouet, and J. Leguay, "Disco: Distributed multi-domain sdn controllers," *CoRR*, vol. abs/1308.6138, 2013. [Online]. Available: http://arxiv.org/abs/1308.6138

[12] V. Pashkov, A. Shalimov, and R. Smeliansky, "Controller failover for sdn enterprise networks," *MoNeTeC*, 2014. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6995594

Appendix A

Mininet Topography

The topography for Mininet.

```
#!/usr/bin/python
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import OVSSwitch, Controller, RemoteController, CPULimitedHost
from mininet.link import TCLink
from mininet.util import dumpNodeConnections
from mininet.log import setLogLevel
from mininet.cli import CLI
from threading import Thread
from time import sleep

operational = True

def checkConnections( switch, backupController):
  sleep(10)
  global operational
  while (operational):
    if (switch.connected() == False):
      print "Lost connection!"
      switch.vsctl("del-controller 's1'")
      switch.dpctl("del-flows 's1'")
      switch.vsctl("set-controller " + backupController)
      operational = False
    sleep(1)

```

```
26
27
28 def serverRacks():
29
30   link = dict( bw=10 )
31
32   net = Mininet( controller=RemoteController, switch=OVSSwitch, host=CPULimitedHost, link=
33
34   "Define Opendaylight Controllers IP adresses"
35   odlController1_IP = '192.168.56.101'
36   odlController2_IP = '192.168.56.102'
37
38   "Create Server Rack 1 (1 switch, 3 hosts)"
39   h1 = net.addHost( 'h1' )
40   h2 = net.addHost( 'h2' )
41   h3 = net.addHost( 'h3' )
42   s1 = net.addSwitch( 's1' )
43
44   "Create Server Rack 2 (1 switch, 3 hosts)"
45   h4 = net.addHost( 'h4' )
46   h5 = net.addHost( 'h5' )
47   h6 = net.addHost( 'h6' )
48   s2 = net.addSwitch( 's2' )
49
50   "Create Links within Server Rack 1"
51   net.addLink( h1, s1 )
52   net.addLink( h2, s1 )
53   net.addLink( h3, s1 )
54
55   "Create Links within Server Rack 2"
56   net.addLink( h4, s2 )
57   net.addLink( h5, s2 )
58   net.addLink( h6, s2 )
59
60   "Create Link between the two Server Racks"
61   net.addLink( s1, s2, **link ) # **link
62
63   "Create Controllers with the IP adresses specified above"
64   c0 = net.addController( 'co', controller=RemoteController, ip=odlController1_IP, port=66
```

```
65    c1 = net.addController( 'c1', controller=RemoteController, ip=odlController2_IP, port=66
66
67    net.build()
68    #c0.start()
69    #c1.start()
70    s1.start( [c0] )
71    s2.start( [c1] )
72
73    #net.start()
74    #net.staticArp()
75    return net
76
77
78  def stopNetwork( net ):
79    CLI( net )
80    net.stop()
81    global operational
82    operational = False
83
84
85
86  def perfTest( net ):
87    print "START Network Connectivity"
88    net.pingAll()
89    print "END Network Connectivity"
90    print "START Bandwith Test"
91    h1, h4 = net.get( 'h1', 'h4' )
92    net.iperf( (h1, h4) )
93    print "END Bandwitch Test"
94    print "START CLI"
95    CLI( net )
96    net.stop()
97
98  def keepSwitchesOnline( switches ):
99    if switches.connected() == False:
100     switches.cmd( "ovs-vsctl get-controller" )
101
102
103 if __name__ == '__main__':
```

```
104     setLogLevel( 'info' )
105     net = serverRacks()
106     """'c2' tcp:192.168.56.105:6633"""
107     threads1 = Thread(target = checkConnections, args = (net.get('s1'), "'s1' tcp:192.168.56
108     threads1.start()
109     #"perfTest( net )"
110     stopNetwork( net )
111     threads1.join()
```

Appendix B

Results

All results are formated in the following way: min/avg/max/mdev. Except for Setup D which has packet loss added at the end.

B.1 Setup A

```
0.057/0.393/1.802/0.500
0.038/0.127/0.582/0.165
0.051/0.129/0.546/0.152
0.048/0.231/0.582/0.152
0.053/0.321/1.489/0.401
0.050/0.246/0.481/0.137
0.052/0.264/1.027/0.268
0.048/0.138/0.643/0.179
0.044/0.243/1.679/0.486
0.048/0.270/1.020/0.270
0.051/0.466/3.739/1.095
0.064/0.214/0.552/0.138
0.054/0.405/1.551/0.410
0.057/0.278/0.722/0.207
0.060/0.466/1.957/0.532
0.057/0.241/0.691/0.175
0.063/0.384/1.976/0.539
0.061/0.224/0.588/0.147
0.052/0.263/0.495/0.141
0.051/0.314/1.087/0.283
mean:
0.053/0.281/1.160
```

B.2 Setup B

```
0.052/0.133/0.527/0.150
0.047/0.181/0.997/0.280
0.053/0.248/0.482/0.132
0.052/0.295/1.039/0.276
0.053/0.361/1.566/0.417
0.055/0.257/0.467/0.136
0.051/0.327/1.149/0.299
0.048/0.448/2.260/0.618
0.046/0.405/0.785/0.255
0.042/0.412/2.018/0.549
0.050/0.299/0.787/0.212
0.054/0.259/0.504/0.142
0.048/0.354/0.883/0.244
0.051/0.486/2.151/0.587
0.061/0.360/1.561/0.420
0.071/0.406/1.207/0.344
0.053/0.350/1.265/0.332
0.058/0.280/0.557/0.159
0.047/0.312/0.769/0.217
0.053/0.235/0.595/0.156
mean:
0.052/0.320/1.078
```

B.3 Setup C

```
0.064/0.392/1.172/0.336
0.050/0.374/2.143/0.596
0.054/0.275/0.532/0.155
0.053/0.307/0.612/0.183
0.040/0.309/0.824/0.255
0.048/0.171/0.728/0.209
0.052/0.319/1.579/0.429
0.051/0.486/3.260/0.928
0.048/0.225/0.578/0.141
0.060/0.513/1.679/0.457
0.049/0.149/0.571/0.163
0.050/0.504/2.712/0.751
0.053/0.377/1.850/0.505
0.048/0.140/0.633/0.183
0.046/0.159/0.598/0.209
0.047/0.307/1.131/0.309
0.055/0.661/3.531/0.997
0.050/0.274/0.518/0.156
0.048/0.372/1.791/0.495
0.053/0.392/2.316/0.646
mean:
0.051/0.335/1.438
```

B.4 Setup D

```
0.201/0.275/0.528/0.102/0.2
0.292/0.343/0.543/0.083/0.3
0.227/0.733/3.261/0.969/0.2
0.049/0.406/1.261/0.345/0.2
0.238/0.315/0.535/0.088/0.2
0.277/0.339/0.540/0.084/0.3
0.256/0.412/0.656/0.140/0.2
0.179/0.298/0.605/0.125/0.2
0.198/0.309/0.512/0.086/0.2
0.160/249.903/998.358/432.120/0.6
0.225/0.482/1.639/0.445/0.2
0.210/0.318/0.540/0.095/0.2
0.234/0.468/0.649/0.140/0.2
0.086/0.456/1.871/0.539/0.2
0.242/0.481/0.886/0.191/0.2
0.259/0.507/1.808/0.494/0.2
0.236/0.594/2.303/0.658/0.2
0.202/0.276/0.519/0.098/0.2
0.257/0.320/0.557/0.091/0.2
0.220/0.323/0.548/0.132/0.2
mean:
0.200/12.878/50.906/0.23
0.215/0.403/1.040/0.21
```