SDN Controller Robustness and Distribution Framework

Ficky Fatturrahman

Master of Science Thesis

Delft University of Technology

Network Architectures and Services

Faculty of Electrical Engineering, Mathematics and Computer Science
Network Architectures and Services Group

SDN Controller Robustness and Distribution Framework

Ficky Fatturrahman
4311809

Committee members:
Supervisor: Dr. Ir. Fernando Kuipers
Member: Dr. Zaid Al-Ars
Member: Ir. Rogier Noldus

August 22, 2017

Delft
University of
Technology

Copyright © 2017 by Ficky Fatturrahman
All rights reserved. No part of the material protected by this copyright may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without the permission from the author and Delft University of Technology.

Acknowledegement

I would like to thank Ministry of Communication and Informatics (MCIT) of Indonesia for giving me a scholarship to pursue my master's degree. I would also like to thank my supervisors Fernando Kuipers and Niels van Adrichem for their guidance. Special thanks to Niels for the ACRoDiF name.

Furthermore, I would like to thank my fellow students at TU Delft, my fellow housemates and my EWI classmates.

Last I want to thank my mother and father, my sister and brothers.

Abstract

SDN improves network flexibility which is constrained by network protocol in a conventional network by decoupling the control plane and the data plane of the network. This is the reason why many companies and universities migrate their network to SDN, there will be more SDN network in the future. Yet SDN network mainly depends on the controller in the control plane. Hence, SDN controller robustness becomes an important issue, because a controller failure will result to a network outage.

OpenFlow is arguably the standard protocol for SDN network. Thus, it is necessary to investigate the robustness of the OpenFlow control plane. Several open source controllers such as OpenMul, Floodlight, Opendaylight, and ONOS have multiple controllers framework to tackle a controller failure. They provide failover mechanism, when there is a controller failure, a backup controller can take over to control the network. In this thesis a benchmark is conducted to measure how long the failover time of those open source controllers. Unfortunately their failover time is in order of seconds, which is way higher than 50ms, the acceptable standard of carrier-grade recovery time.

This thesis presents a solution that can improve SDN robustness: A Controller Robustness and Distribution Framework (ACRoDiF). ACRoDiF is compatible with several open source OpenFlow controllers such as Ryu, OpenMul, Floodlight, Opendaylight, and ONOS. ACRoDiF provides failover mechanism that has lower failover time than in the open source controllers: 76ms. It can also eliminate failover time completely if using two active primary controllers.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1-1 Background

IP is arguably the most important network protocol in this digital age. The Internet, which now has become a critical infrastructure runs on top of IP. IP is also used in data centers around the world for communication between servers, both inside the data centers and to the outside world. A large number of critical applications traverse the Internet, such as financial data transactions between data centers. But IP was not designed that purpose. Present day IP networks have evolved into much more complex networks than how it was originally invented, e.g. current routing tables of the Internet have reached over 600k routes. It continues to increase steadily. This is a burden for the network control plane, especially the Ternary Content Addressable Memory (TCAM) in routers. There is a limit to what a control plane can do with the current design of an IP network device. A current network device has both control plane and data plane incorporated into one device. The control plane performance of such a device is limited to the hardware itself. Most routers and switches run on hardware with limited resources. It is costly to design a router with powerful control plane hardware, because an IP network consists of a lot of routers connected with each other. Too expensive for any organization or ISP to invest on plenty of routers with powerful control plane hardware. Instead of solving the problem with IP network limitation, most networks only hinder the complexity via a workaround solution.

Software-Defined Networking (SDN) is possible solution. SDN decouples the control plane and the data plane into separate devices. The data plane spreads around as switches in the network. The data plane only forwards flows in the network. The control plane is logically centralized and controls the whole network. It organizes how the traffic flows in the network. As a separate device, the control plane can have its own powerful hardware that can perform difficult computation. For example, the control plane can run complex routing algorithms without being restricted by routing protocol or hardware.

As a critical infrastructure and the carrier of critical applications, the robustness of IP networks is an important issue to address. Network outage on the Internet or data centers will result in huge revenue loss. Thirty to twenty years ago, one minute of downtime would not have cost a dime. But now, a minute of downtime means thousands of dollars revenue loss. Research conducted by various organizations shows the cost of network outage. Downtime costs around $700 billion to North American organizations a year according to IHS research [11]. The research mentioned that the main cause is because of network failure. Although the servers and applications are working fine, they cannot communicate with each other. Separate research from Ponemon Institute shows that a minute downtime can cost organizations thousands of dollars [12]. The research indicated an increasing trend in downtime cost over time, see Figure 1-1.

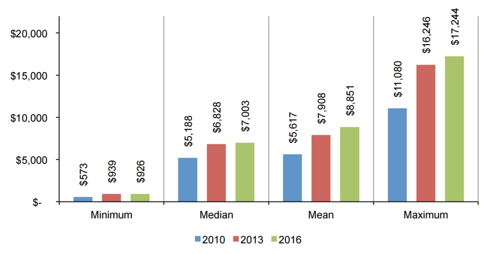


Figure 1-1: Total cost per minute of an ICT outage [12]

1-2 Problem Definition

In the past few years, OpenFlow from Stanford University has received significant attention as the protocol for SDN. Not only in the academic world, OpenFlow has also gained popularity in the industry. As any other SDN protocol, OpenFlow separates the control plane and the data plane of the network into OpenFlow switch as the data plane and OpenFlow controller as the control plane. The separation gives flexibility for the user on how to design the network, not restricted to the limitation of conventional network protocols.

Although OpenFlow offers flexibility to its user on how to program the network, it is not without problem. In the early specification of OpenFlow, the original architecture only had one controller in the control plane. This single controller design introduces availability issues in control plane robustness: Single Point of Failure (SPoF). Every OpenFlow switch in the data plane relies on the controller. Controller failure will bring the whole network down. The previous section mentioned the importance of robustness and how much network outage can cost. It is also relevant to OpenFlow network. In the newer specification, starting from version 1.2, OpenFlow has an additional feature on control plane robustness. However, it only demonstrates how to implement an OpenFlow network with multiple controllers, not how failover works between those controllers in the control plane. So, it is up to the network programmer to develop highly available controllers, since a failover mechanism is not part of the OpenFlow specification. Fast failover mechanism between multiple controllers is one of the key factors to provide robust OpenFlow control plane.

1-3 Research Objectives

The previous section mentioned the importance of SDN robustness, specifically on OpenFlow control plane failover. Therefore the main objective of this thesis is to propose a novel approach to improve failover performance on the OpenFlow control plane. From this objective the following sub-objectives are derived:

- Review related works on SDN robustness and multiple OpenFlow controllers solutions.
- Identify existing conventional network protocols that can be applied to improve SDN control plane robustness.
- Implement conventional network protocols to provide failover on OpenFlow control plane and measure and evaluate their failover performance.
- Identify existing open source OpenFlow controllers with failover solution on the control plane.
- Implement open source OpenFlow controllers to measure and evaluate their failover performance.
- Develop and propose a new solution that can improve failover performance and measure and evaluate the proposed solution.

1-4 Research Questions

Ideally, OpenFlow control plane should provide robust service to the data plane. When controller failure occurs, there should be backup controllers that can detect it quickly. The control plane should also provide a failover mechanism with the minimum handover time from primary controller to the backup controller. Another important issue is that the control plane also decide which backup controllers that will take over and control the data plane during failover, otherwise it will cause race condition, a situation where two or more controllers trying to take over the data plane at the same time. Based on the previous description, the research questions for this master thesis are the following:

- What kind of improvements are needed for the SDN control plane to overcome SPoF?
- How can the control plane detect failure when failure occurs in one of the controllers?
- How to minimize failover period when there is controller failure?
- How to avoid race condition between controllers in during failover?

1-5 Thesis Structure

This Master Thesis is organized as follows. First, in Chapter 1 the SPoF issue in OpenFlow is formulated into research objectives and research questions. Next, Chapter 2 presents an overview of SDN and OpenFlow. Related works in controller robustness, conventional redundancy protocols, and failover mechanism in open source controllers are discussed in Chapter 3. Chapter 4 evaluates various failover mechanisms for OpenFlow using conventional redundancy protocols and using open source controllers. In Chapter 5, failover mechanism for multiple controllers is proposed, experimented, measured, and analyzed. Finally, Chapter 6 concludes this master thesis and recommends future work.

Chapter 2

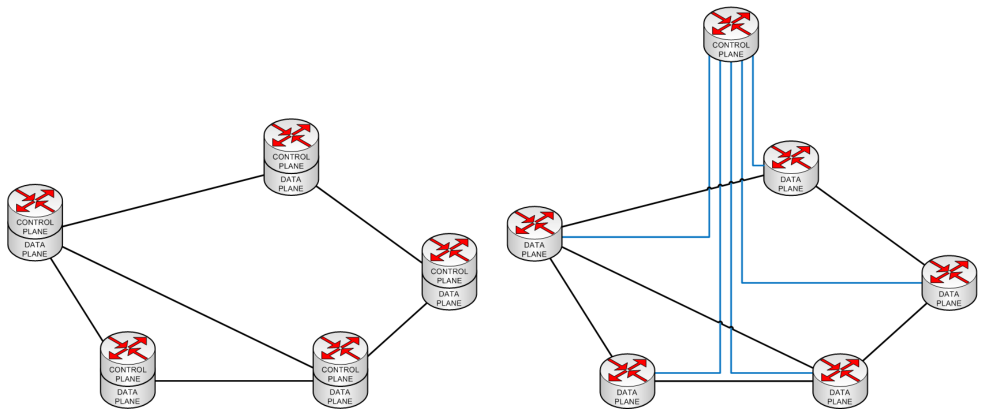
SDN

2-1 SDN Overview

The first chapter briefly discussed the problem that conventional IP networks are becoming too complex. IP was not designed to accommodate current critical applications' requirements. Even though IP has many limitations, it is still widely used, e.g. in the Internet and in the datacenter network. We come up with workaround solutions to overcome its limitation rather than innovate with better solutions. IP networks are getting convoluted with numerous workaround devices. Those devices have different functions, from various vendors, with different operating systems. For example, Network Address Translation (NAT) boxes were invented to counter the IPv4 exhaustion problem, Firewall, Intrusion Detection System (IDS) and Intrusion Prevention System (IPS) were invented to prevent security attacks and load balancers were invented to distribute traffic. This is a major problem for network administrators and engineers: they have to understand how all those devices work and how to integrate all of those devices. Upgrading is also another problem, since adding new features or hardware upgrades to the control plane means downtime for those devices, because the control plane and the data plane reside in the same device.

SDN introduces a new solution to improve networks. SDN decouples the control plane and the data plane into separate devices, see Figure 2-1. The data plane can perform any function, it can function as a router, a switch, firewall, or any application instructed by the control plane. The separation of the control plane and the data plane enables a logically centralized control plane. This simplifies the network, everything can be controlled from a single control plane, no need to integrate multiple devices from various vendors with different operating systems. An SDN control plane or can be installed on a server as an SDN controller, which has more powerful CPU than a CPU on a commodity router or switch. It is capable to implement more complex algorithm and it is easier to add more features, i.e. it enables programmability of network services.

Figure 2-2 shows SDN Architecture that comprises of three layers [7]:



(a) Conventional Network **(b)** SDN

Figure 2-1: Conventional Network versus SDN

- **The Data Plane Layer**
  The Data Plane is the network infrastructure that is controlled by the control plane. It consists of Network Elements (NE)s with forwarding capabilities such as a switch or a router. It is the underlying physical infrastructure that can be orchestrated by the controller through the Southbound interface in the control plane.

- **The Control Plane Layer**
  The control plane is a logically centralized system that translates the abstract form of the network application in the application plane layer into instruction or action for NE in the data plane. It communicates with the application plane using its Northbound interface. An example of Northbound interface API is REST API. It communicates with the data plane using its Southbound interface. There are several protocols exist for SDN Southbound interface, e.g. OpenFlow, NETCONF, OVSDB.

- **The Application Plane Layer**
  The application plane is the network application that can program the NE in the data plane to operate the desired network behaviour. It communicates to the controller through the Northbound interface in the control plane. Some examples of the network applications are network virtualization hypervisor such as FlowVisor [20] and DelftVisor [24], internate exchange point such as Software-Defined Internet Exchange (SDX) [8], and network monitoring system such as OpenNetMon [25].

2-2 OpenFlow

OpenFlow started out as part of the Clean Slate Program at Stanford University. The program objective is what if the Internet is designed from scratch. Current IP networks are too difficult for researchers to innovate new features on a realistic network, such as new routing protocols. OpenFlow solves this problem with network programmability. A programmable network allows a researcher to experiment on a production network without having to worry about

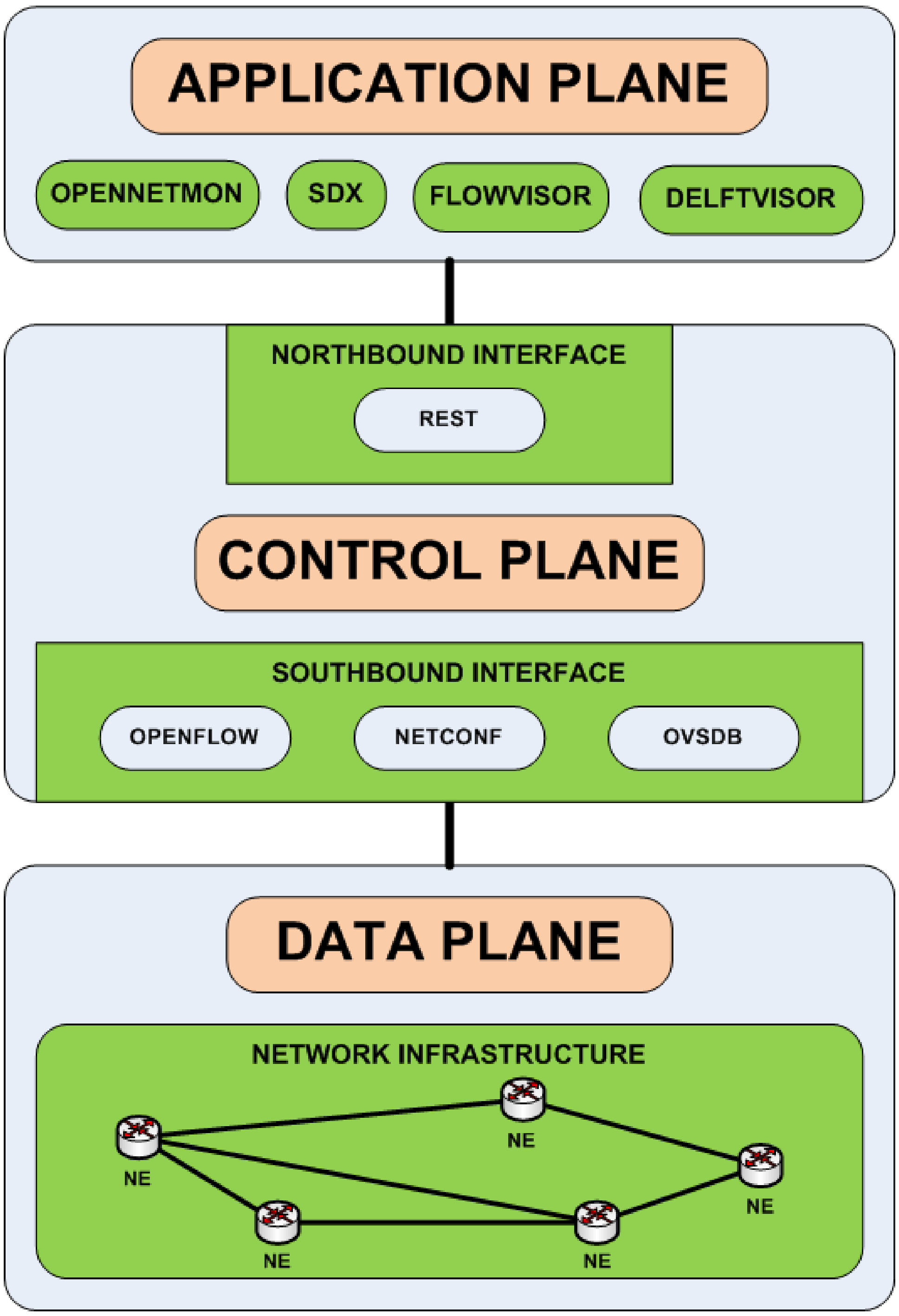


Figure 2-2: SDN Architecture

disrupting real traffic in the network. A researcher can program the network to provide a slice of the network, isolated from the real traffic.

An OpenFlow network consists of OpenFlow controller in the control plane and OpenFlow switch in the data plane. According to OpenFlow whitepaper [14], an OpenFlow switch consists of at least three of the following: a flow table, an OpenFlow channel, and OpenFlow Protocol, see Figure 2-3. Flow table is a forwarding table based on rules configured by the controller. An OpenFlow switch has one or more flow table, each flow table has one or more flow entry. A flow entry uses any of the information within the packet such as TCP port, VLAN tag, MAC/IP address, or incoming port to determine what to do with the packet, e.g. forward to the switch's other port(or ports), drop the packet, send to the next flow table, or encapsulate and forward the packet to the controller. A flow entry has three fields: match fields, counters for statistics, and a set of instructions to apply to matching packets. When there is an incoming packet arrive at an OpenFlow switch's port, the switch will look up to its flow table to find matching flow entry for that particular packet. If the switch find a matching flow entry, it will apply the instruction set to that packet, if there is no matching entry, it will send the packet to the controller. The controller has the privilege to send or modify flow entry in the switch via an OpenFlow channel. An OpenFlow channel connects a controller and a switch, it runs on top of TCP or encrypted using TLS. The communication between a controller and a switch is using OpenFlow Protocol.

2-2-1 OpenFlow Protocol

OpenFlow uses an OpenFlow channel as the interface that connects the control plane and the data plane. The OpenFlow channel can be an in-band or out-of-band link between control plane and data plane. Communication between the control plane and the data plane is using OpenFlow Protocol via the OpenFlow channel. The OpenFlow protocol has three types of messages:

- Controller-to-Switch
  Controller-to-Switch are the messages sent from the control plane to the data plane. It may expect a reply from the data plane.
- Asynchronous
  Asynchronous messages are the type of message that can be sent from the data plane to the control plane without request.
- Symmetric
  Symmetric messages are the type of messages that can be sent both from the control plane to the data plane and the other way around.

See Table 2-1 for the list of OpenFlow protocol messages.

2-2-2 OpenFlow Multiple Controllers Mechanism

As mentioned in the previous chapter, the original OpenFlow protocol specification did not support multiple controllers implementation, which is a SPoF issue. To address that problem,

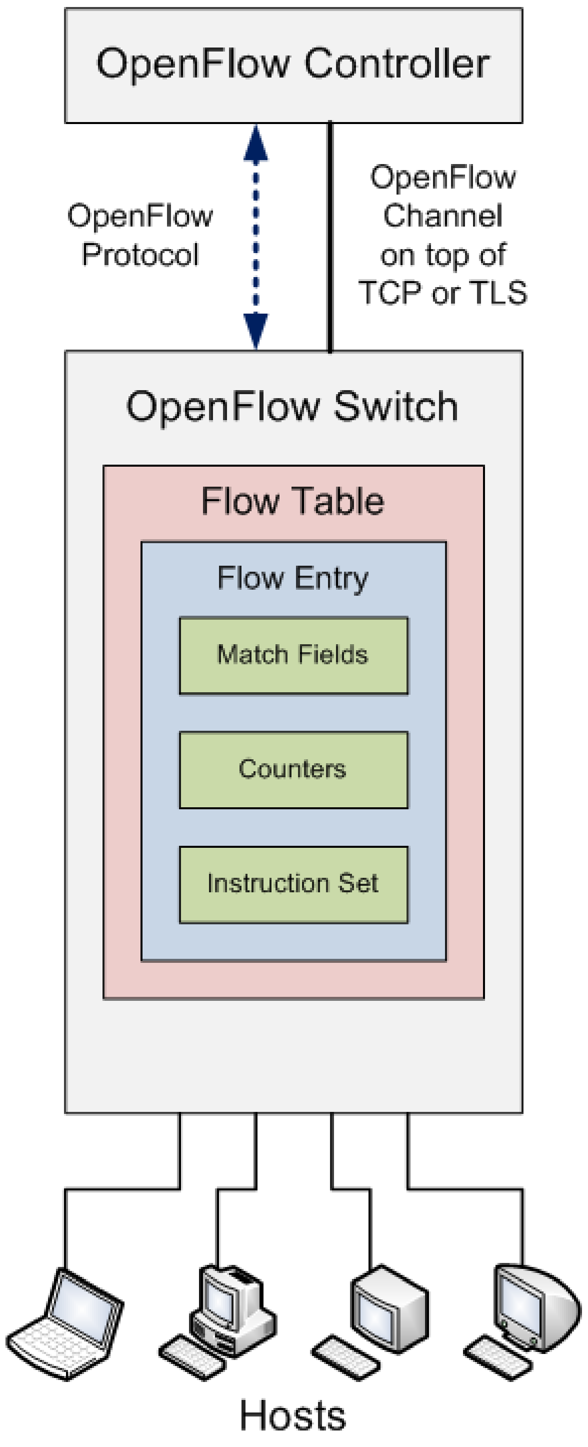


Figure 2-3: OpenFlow Architecture

Table 2-1: OpenFlow Protocol

Protocol	Message	Description
Controller-to-Switch	Features	This message requests for the switch to reply with its capabilities. This message is usually sent during the initialization of OpenFlow channel between the control plane and the data plane.
	Configuration	This message sets and queries configuration parameters in the switch.
	Modify-State	This message can add, delete, or modify flow/group entries in the switch's flow table.
	Read-State	This message requests for the switch to reply with various information such as configuration, statistics and capabilities.
	Packet-out	This message is sent from the control plane to respond to packet-in message from the data plane. It can also be sent from the control plane to send packet to a specific port on the switch.
	Barrier	This message is used to ensure message dependencies have been met or to receive notifications for completed operations.
	Role-Request	This message states the role of the controller to the switch connected to it, see 2-2-2.
	Asynchonous-Configuration	This message can customize the type of asynchronous messages that can be sent from the data plane to the control plane.
Asynchronous	Packet-in	Upon receiving a packet, the data plane first look up its flow table. If there is no match in the table, it will forward the packet to the control plane as packet-in message.
	Flow-removed	Flow entry in the flow table has a timeout. When the flow entry reaches its timeout, it will get removed from the table. Then the data plane sends flow-removed message to notify the control plane.
	Port-status	This message is sent from the data plane to notify the control plane of any changes in its port status, e.g. port is down because of the link disconnected.
	Error	This message is sent to notify the control plane of problems in the data plane.
Symmetric	Hello	This message is exchanged between a controller and a switch during initial setup of the OpenFlow channel.
	Echo	This message is used to monitor the liveness of OpenFlow channel between the control plane and the data plane. If echo request is sent, the receiver must respond with echo reply message.
	Experimenter	This message is used to offer additional functionality within the OpenFlow message type space.

newer OpenFlow specifications are updated with a new feature to support multiple controllers. In the newer specification, OpenFlow recognize three different roles on how controllers can manage an OpenFlow switch:

- The first type role is MASTER. This type of controller role has full access to the OpenFlow switch. It can send flow entry or flow modification to the switch's flow table. This controller role can also receive all type of messages from the switch, e.g. packet-in message or port statistics message. In an OpenFlow network, there can only be one controller with MASTER role active in the network.
- The second type of role is EQUAL. OpenFlow controller with EQUAL role has the same privilege as MASTER. But unlike MASTER, there can be multiple controllers active in the network with EQUAL role. Those controllers have the same access privilege to the OpenFlow switch connected to them. One possible application for EQUAL role is to make load-balancing controllers, with multiple controllers interchangeably controlling the switch.
- The last type of role is SLAVE. This type of role has the least privilege of them all. Like EQUAL, there can be multiple controllers with SLAVE role in the network. Controller with SLAVE role can only received statistical messages, it cannot send the switch any messages that can modify the switch's state, such as packet-out or flow modification message. If a controller with SLAVE role sends a message that can modify a switch's state, the switch will drop the message and reply with an error message.

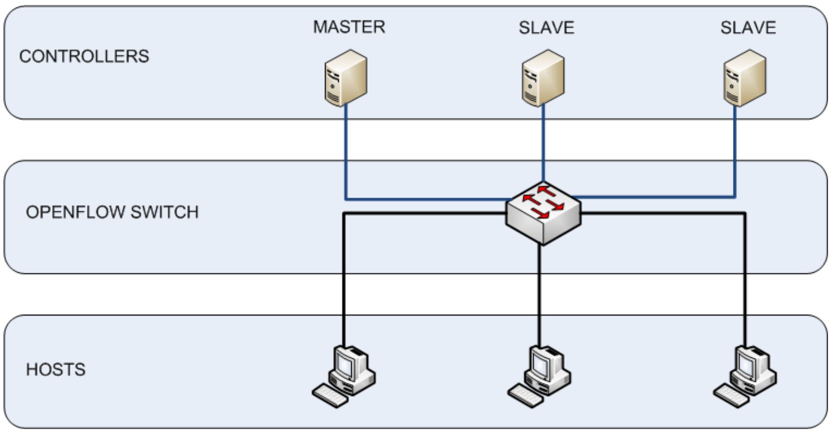
An OpenFlow controller can state its role as MASTER/EQUAL/SLAVE by sending an `OFPT_ROLE_REQUEST` message. See Table 2-2 for controller roles summary.

Multiple controllers implementation in OpenFlow is different from most high availability solutions used in a client-server model. Usually the client is unaware that there exist several servers providing the service. In comparison to the client-server model, in OpenFlow the data plane is similar to a client that sends a request to a server, in this case the control plane. Different from that model, the switch in the data plane, aware that multiple controllers exist in the control plane. When connected to multiple controllers, an OpenFlow switch maintains separate OpenFlow channels for each controller.

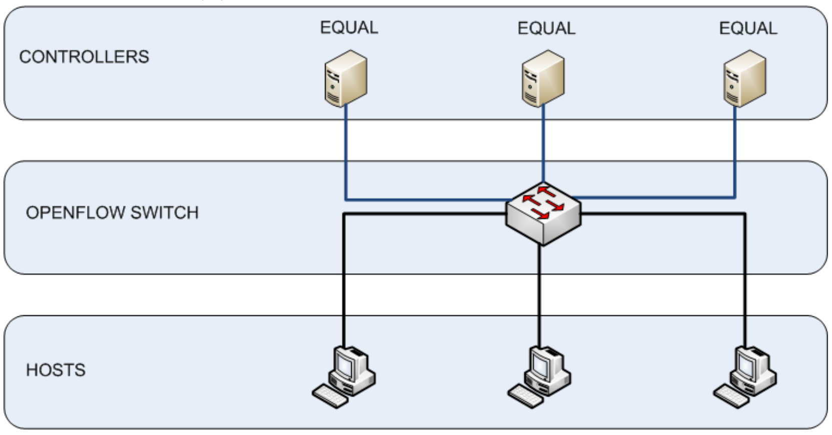
Although the data plane is aware that it is connected to multiple controllers in the data plane, the coordination between controllers is handled by the controllers themselves, e.g. the implementation of load-balanced multiple controllers with EQUAL role or failover handling between controllers. The control plane must decide the role for the controllers as MASTER, EQUAL, or SLAVE in the initial connection to the data plane. When there is a failure in a controller with MASTER role, the control plane should be able to detect it, then resolve the problem by electing a new controller to take over the MASTER role from one of the controllers with SLAVE role. The data plane can detect if the OpenFlow channel is disconnected, but it cannot elect a new controller, it waits for the control plane to send role request message with new MASTER. OpenFlow role request message has a `generation_id` field. The `generation_id` is a number that should be incremented, so the data plane can recognize newer state in the control plane during MASTER/SLAVE failover. See Figure 2-4 for an example of an SDN network with different controller roles.

Table 2-2: OpenFlow Controller Roles

Role	Access	Messages			Number
		Controller-to-Switch	Asynchronous	Symmetric	
MASTER	Full	All	All	All	At most only one
EQUAL	Full	All	All	All	Multiple
SLAVE	Read-only	No messages that modify the state of the switch, no packet-out	None, except port status	All	Multiple



(a) One MASTER with several SLAVEs



(b) Several controllers with EQUAL role

Figure 2-4: SDN Controllers with different roles: MASTER/EQUAL/SLAVE controllers

Chapter 3

Failover Mechanisms and Related Work

3-1 Related Work

Several solutions have been proposed for SDN resilience in the past few years. Yet not all of them tackle failover problem between multiple controllers. SDN resilience research can be categorized into the following:

- **distributed controllers**, this approach improves controller robustness by distributing the controllers in the network, using multiple controller and divide the network into several domain for each controller to handle, the examples are HyperFlow [23] and Kandoo [9]
- **state replication**, this type focuses on state replication aspect between multiple SDN controllers, assuming that there are multiple controllers in the network and there is a failover mechanism between controllers whenever a controller failure occurs, the examples in this category are CPRecovery [5] and SMaRtLight [3]
- **load balancing/load sharing**, in this subtopic the load in a controller is distributed between multiple controllers, reducing the possibility of CPU or other resource overload that can cause failure in a controller, preventing SPoF, the example is Balanceflow [10] and DALB [28]
- **devolving controller**, this approach delegates some of controller functions (control plane functionality) back to the switches, to avoid resource overload in the controller, it also aims to distribute some of the control funtions to the switches to avoid SPoF problem in the centralized controller SDN network, the examples in this category are DAIM [1], DIFANE [27], DevoFlow [4] and [21]
- **data plane resilience**, Not only in the control plane, but also resilience in the data plane. In the data plane, similar work has been done to provide fast recovery during a link failure, the example for this category is in [26]

3-1-1 Distributed Controllers

HyperFlow

HyperFlow [23] is an OpenFlow controller application that run on top of a NOX controller to provide a distributed control plane. HyperFlow was designed to improve shortcoming of the first OpenFlow specification, see Section 1-1. In the early OpenFlow specification, it only supports to implement one controller. But it is not scalable when production network grows in size and number, HyperFlow was designed to solve that problem. Scalability problems that HyperFlow wants to solve are:

- increasing control traffic when the number of switches is increasing, when the number of switches is increasing, control traffic is also increasing, but the traffic can be distributed to distributed controllers
- high flow setup latency because of the distance between controller and switch, specifically in a network with large diameter, if the controllers are distributed, the network can be partitioned into smaller partition with its own controller
- increasing flow setup time when the network grows larger, the resource of one controller can be exhausted if the network grows larger

HyperFlow has a physically distributed but logically centralized control plane. The distributed controllers in HyperFlow partitioned the network into separate domain that connected to each other. Every event in each domain is handled locally by its own controller. HyperFlow uses publish/subscribe messaging paradigm to enable communication between controllers. The publish/subscribe mechanism enables every event on each domain gets propagated to the other domain. This enables the control plane has a network-wide views of the network topology. It also improves resilience and robustness to network partitioning

HyperFlow uses advertisement packet in the control plane as a healthchecking mechanism to monitor the controller availability. It declares controller failure if there is no advertisement from a controller after three advertisement intervals. When there is a controller failure, the controller IP address configuration in the orphan switch has to be reconfigured to connect to an active controller. HyperFlow relies on proprietary configuration interface from the switch's hardware vendor to change the controller IP address in the switch. Unfortunately HyperFlow's paper does not mention how long the advertisement interval and it also does not measure the failover time.

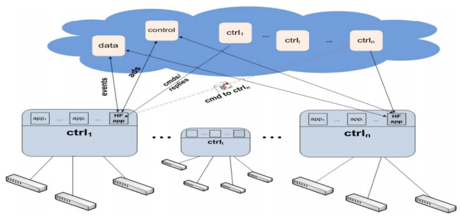


Figure 3-1: High-level Overview of HyperFlow [23]

Kandoo

Kandoo [9] is a distributed OpenFlow controllers architecture that has hierarchical controllers design. Kandoo objective is similar to devolving controller, but instead of reverting some control plane functionality back to the switches, it puts local controllers close to the switches, so it is still compatible with OpenFlow specification. Kandoo approach is using distributed controllers, load sharing, and controllers placement to create hierarchical distributed controllers. Kandoo has two layer of controllers: root and local, see Figure 3-2. Local controllers deal with most of the flows from local events and application that does not require network-wide state view of the topology. Root controllers handle flows that require network-wide state view of the topology such as elephant flows. Root controller is centralized for all the local controllers. Kandoo works in the following steps. First, the local controllers handle all local events. But if there are specific flows/events that pre-specified for the root controller, local controllers forward the packet-in from that flow to the root controller, e.g. elephant flows that is predefined to have more than 1 Mbps throughput. After receiving the packet-in, root controller sends the flow entry through local controllers so they push it to the switches. Although Kandoo focuses on the distributed controllers solution, unfortunately there is no mention on the failover performance on the controllers.

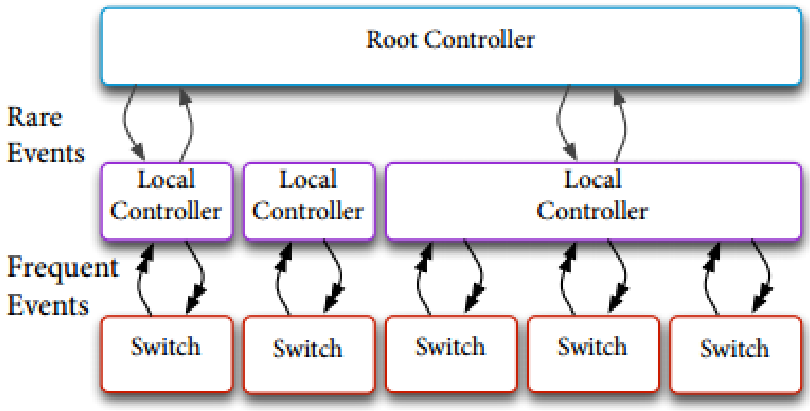


Figure 3-2: Kandoo Architecture [9]

3-2 Conventional Failover Protocols

Not only in SDN, SPoF is also a problem in a conventional network. Basically, an SPoF is a part of the network, which if it fails, causes the entire network to stop operating. Redundancy is one of the solution to increase network robustness against SPoF. Making an SPoF part of the network redundant can decrease the risk of SPoF. Redundancy protocol enables multiple devices to operate in the network. It also provides failover mechanism between redundant devices. There are several redundancy solution in the conventional IP network. In the network layer, there is Virtual Router Redundancy Protocol (VRRP), and in the transport layer there is Linux Virtual Server (LVS).

3-2-1 Network Layer: VRRP

Originally VRRP was designed as gateway redundancy protocol in the network layer. The main objective of this type of network protocol is to provide redundant network gateway that transparent to its hosts. VRRP works using multiple network devices taking the role of one Virtual IP (VIP) address. Figure 3-3 shows an example of a simple LAN network that uses VRRP. The hosts are in 10.0.0.0/24 network with a gateway to VIP 10.0.0.1. There are two routers that take the role as the virtual gateway. Each physical router has its own unique IP address but shares the same VIP address. Only one physical router active as the MASTER, the other one standby as BACKUP. When a host sends an ARP request for VIP hardware address, the MASTER replies it with its virtual MAC address: 00-00-5E-00-01-XX, with XX is the Virtual Router Identifier (VRID). The MASTER periodically sends VRRP advertisement packet to the BACKUP to notify its availability. If the BACKUP does not receive VRRP advertisement packet for longer than three times of advertisement timer, it declares that the MASTER is down and initiates failover process to take over the MASTER role by sending gratuitous ARP packet to the network. Upon receiving the ARP packet, every host updates its own ARP cache for MAC address of the gateway with virtual MAC address of the new MASTER.

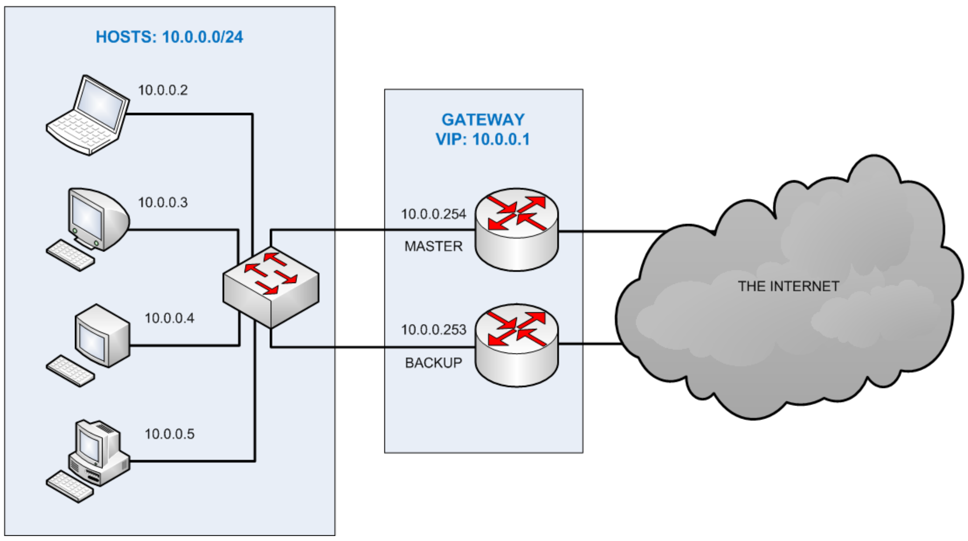


Figure 3-3: VRRP Network

3-2-2 Transport Layer: LVS

Another option that can be applied to provide redundancy and failover for SDN controller is LVS (Linux Virtual Server). LVS is an architecture that provides high availability at the transport layer, commonly used on web server. LVS consists of two components: 1) a single or a pair of load balancers, 2) a cluster of servers. Load balancer role is to distribute packets to the cluster of servers, e.g. HTTP request packets to the cluster of web server. The load balancer also performs Network Address Translation (NAT) service to the cluster. Similar to VRRP, logically there is only one virtual IP address visible to the host. The host is unaware of the use of multiple servers in LVS's cluster. NAT enables multiple servers in the cluster to share one virtual IP address even though each server has its own unique physical IP address. LVS also has healthchecking mechanism to probe the availability of the servers, similar to how VRRP advertisement. But unlike VRRP, the load balancer in LVS provides both network layer and transport layer healthchecking service to the cluster, to make sure those servers are still available. The load balancer periodically sends TCP healthcheck packet to the servers, checking the availability of the TCP port of the servers, e.g. HTTP at port 80. Because the load balancer has the responsibility to forward packets from the host to the servers, when a server fails, the load balancer removes that particular server from the pool of active servers. Every upcoming packet received by the load balancer is forwarded to other active servers. Figure 3-4 shows an example of an LVS network.

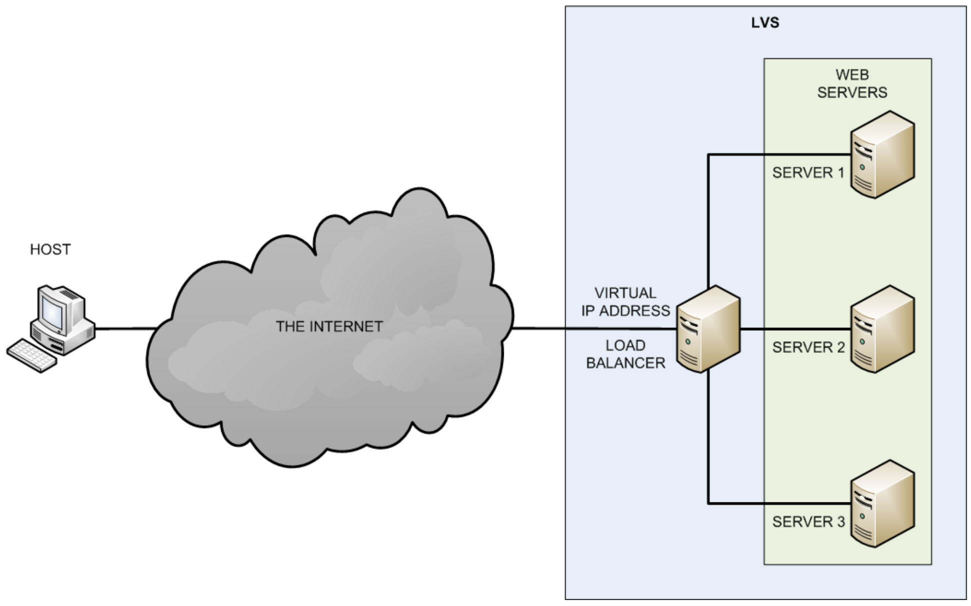


Figure 3-4: LVS Network

3-3 Open Source Controllers

Currently, there are several open source OpenFlow controllers available made from different programming languages, e.g. java-based, python-based, ruby-based, etc. Some of them are NOX/POX, Floodlight, Maestro, Beacon, Ryu, etc. But not all of them have built-in high availability feature. The following are open source controllers that have high availability feature: OpenMUL [18], Floodlight [19], OpenDaylight [15], and ONOS [2].

3-3-1 Ryu

Ryu is a python-based SDN controller, it is licensed under the Apache 2.0. In its Southbound interface, it supports several protocols such as OpenFlow, OF-Config, NETCONF. Ryu's Northbound interface has REST API. Ryu does not have multiple controllers or high availability feature. But as a standalone controller, Ryu is the easiest to deploy. In this master thesis, Ryu is used as the base of the proposed framework.

3-3-2 OpenMUL

OpenMUL is a lightweight SDN controller. It is an open source version of MUL which is developed by KulKloud. OpenMUL is licensed under GNU General Public License version 2 (GPLv2). In its core, OpenMUL has a C based multi-threaded infrastructure.

OpenMUL uses REST API in its Northbound interface. In its Southbound interface, other than OpenFlow, OpenMUL also supports several other protocols such as NETCONF and OVSDB. The main components of OpenMUL are the following:

- MUL director/core, it is the core component of OpenMUL which in charge of :
  - handling all low level switch connections and OpenFlow processing
  - providing API interface
  - providing Southbound protocols, e.g. OpenFlow
  - supports hooks for high speed low-latency learning (reactive) processing infrastructure
  - making sure all flows, groups, meters and other switch specific entities are kept in sync across switch, controller reboots/failures
- MUL infrastructure services, this component provides basic services built on top of MUL director/core, the following are some of the services:
  - **topology discovery service**, this service is responsible to discover the network topology using LLDP packets, it also responsible to detect and prevent loops in the network
  - **path finding service**, this service is responsible to calculate the shortest path between two nodes in the network using Flloyd-Warshall algorithm, it also supports ECMP with the possibility to influence route selection behavior using various external parameters such as link speed and link latency, it is designed to be fast and scalable, in its web, it was mentioned that OpenMUL tested upto 128 nodes, it also provides an extremely fast shared memory based API interface to query routes
  - **path connector service**, this service is responsible to provide a flexible interface for application to install flows across a path, it also responsible to hide network complexities from the Application and App developer, another feature of this service is separating SDN domain into core and edge
- MUL system apps, the apps components are built using API provided by the previous two components. The following apps are compatible with OpenFlow:
  - **L2switch**, an application to implement layer 2 learning switch
  - **CLI app**, an application to configure Mul components in command line
  - **NBAPI webserver**, python-based webserver that provide REST API for OpenMul controller.

OpenMUL supports multiple controllers implementation for high availability. Multiple controllers in OpenMUL can be deployed as hot/warm-standby failover in active-active or active-standby mode. In OpenMUL's blog [22], MuL development team released the commercial version of OpenMUL named Mul 3.2.5 on 12th February 2014. It claims that it has HA failover feature with less than one second failover time. KulKloud release BEEM on its website [13], the current commercial version of OpenMUL that also has hot standby HA feature, which it claim can mitigate outages in less than 1 second.

3-3-3 Floodlight

Floodlight is a java-based OpenFlow controller. It is licensed under Apache License version 2.0 and supported by Big Switch Networks. Floodlight is based on Beacon controller from Stanford and Big Switch Networks. It only supports OpenFlow protocol in its Southbound interface. The follwoing are the hightlighs from Floodlight's features:

- module loading system that make it simple to extend and enhance
- requires minimal dependencies and easy to set up
- supports various virtual and physical OpenFlow switch
- capable of handling mixed OpenFlow and non-OpenFlow networks
- multithreaded
- supports OpenStack cloud orchestration platform.

Floodlight has the following built-in networking applications:

- **OpenStack Quantum Plug-In**, a plug-in to connect Floodlight to OpenStack as network backend
- **Virtual Switch**, an application to implement layer 2 learning switch
- **ACL (stateless FW)**, an application to implement Firewall based on Access Control List (ACL)
- **Circuit Pusher**, an application to build bidirectional circuit using permanent flow entry on all switches between two hosts,

3-3-4 OpenDayLight

OpenDaylight is a java-based SDN controller that is built on Model-Driven Software Engineering (MDSE) principles called Model-Driven Service Adaptation Layer (MD-SAL). MD-SAL uses abstract representations of underlying network devices and network applications as objects or models, whose interactions are processed within the Service Adaptation Layer (SAL). The SAL is the control logic int the control plane layer of OpenDaylight, its Northbound interface translates the network application from the application layer to the data plane layer via its Southbound interface, see Figure 3-5 for OpenDaylight architecture. In addition to OpenFlow, OpenDaylight also supports other protocols in its Southbound interface, e.g. OF-Config, OVSDB or NETCONF.

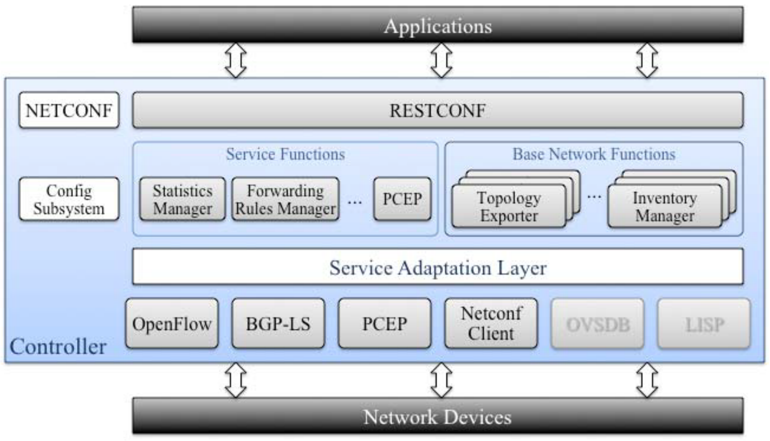


Figure 3-5: OpenDaylight Architecture [15]

3-3-5 ONOS

ONOS is a java-based SDN controller, it is created to be an open source SDN controller targeted for service provider. In its Nortbound interface, ONOS has two Northbound abstraction framework: the Intent Framework and the Global Network View. The Intent Framework enables a network application to apply a service in the network without knowing how the service will be performed in the data plane. The Global Network View provides the application layer a view of the network's resources such as the hosts, switches, links, utilizations. ONOS's Northbound interfaces isolates the application layer from the complex underlaying network infrastructure. The Southbound interface in ONOS provides an abstraction of network resources as objects, it also enables ONOS to manage network element with different protocols, e.g. OpenFlow, OVSDB or NETCONF. Figure 3-6 shows ONOS architecture. ONOS is designed with the following features:

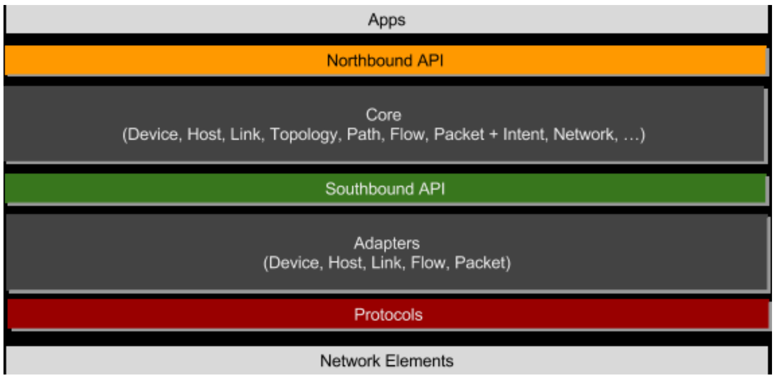

Apps
Northbound API
Core
(Device, Host, Link, Topology, Path, Flow, Packet + Intent, Network, ...)
Southbound API
Adapters
(Device, Host, Link, Flow, Packet)
Protocols
Network Elements

Figure 3-6: ONOS Architecture [17]

- **Distributed Core**, this feature allows ONOS to run as a cluster of controllers that provides scalability and high availability
- **Northbound API**, this feature enables higher-level network application to be translated to the data plane

- **Southbound API**, this feature facilitates ONOS to control the data plane, not only OpenFlow-based switch, but also legacy devices
- **Software Modularity**, this feature makes ONOS easy to develop, maintain, debug, and upgrade.

3-3-6 Summary

Table 3-1 shows the comparison between open source SDN controllers.

Table 3-1: Open Source Controllers Comparison

Controller	Southbound Protocol	Program	Multiple Controllers Feature
OpenMul	OpenFlow, OF-Config, OVSDB, NETCONF	C	Yes
Floodlight	OpenFlow	Java	Yes
ONOS	OpenFlow, OF-Config, OVSDB, NETCONF	Java	Yes
OpenDaylight	OpenFlow, OF-Config, OVSDB, NETCONF	Java	Yes
Ryu	OpenFlow, OF-Config, NETCONF	Python	No

Chapter 4

Failover Evaluation

This chapter evaluates failover time between controllers. First, different types of controller failures are discussed in section 4-1. The second section explains the testbed environment. Next, the failover times using conventional redundancy protocol are measured in section 4-3, and failover times of several open source controllers are measured in section 4-4. The last section compares those failover times.

4-1 Type of Simulated Failure

There are several events that can cause controller failure. The following are three main types of failures that can trigger failover:

- Network failure

  Network or connectivity failure is a failure of the link between two devices. It can be categorized into two types: between the controllers and between the controller to the switch. Network failure can be simulated by using iptables tool in the controller. The result of applying iptables with the rule to drop any packet is blocking the connectivity in a link. When applied to all interfaces in a controller, it will block the OpenFlow channel from the controller to the switch and also block the link between controllers.

- Power failure

  Similar to network failure, power can also be categorized into two types: graceful shutdown and non-graceful shutdown. Simulating graceful shutdown can be done by executing the shutdown command in the controller. It is possible to configure a controller to send other controllers a message when the graceful shutdown command is executed. So, the other controllers can get notified and take over the role if necessary, e.g. failover between controllers to take over MASTER role. In a non-graceful shutdown, the controller does not have the opportunity to notify other controllers. So the failover process depends on liveness monitoring mechanism between controllers, failover is executed when other controllers notice that they do not receive any reply after a period of time.

- Software failure

  Software failure occurs when the process/PID that handles OpenFlow requests is killed in the controller's operating system. To simulate software failure, a command that kills the PID process is executed in a controller. It will trigger the controller to send FIN packet to the switch and other controllers. Thus closing the OpenFlow channel, initiating the failover process.

4-2 Testbed Environment

This section describes the testbed environment. Ubuntu Linux 14.04 servers used in the experiment. The aforementioned linux distribution is the most compatible version with every controllers deployed in the testbed.

The following subsections explains the network topology and software tools used for measurement.

4-2-1 Topology

To simulate and measure failover time, the simplest topology is sufficient, see Figure 4-1. At least three controllers are needed to simulate failover from one controller to a second controller. Although two controllers should be enough to simulate failover, some open source controllers such as OpenDaylight and ONOS require minimum three controllers to enable failover feature. Because they implement quorum mechanism in their failover solution, see section 4-4 for further explanation. Other nodes in the topology are one OpenFlow switch and two hosts to generate traffic that needs to be sent to the controllers as packet-in. The first host generates traffic and then the second host receives the traffic.

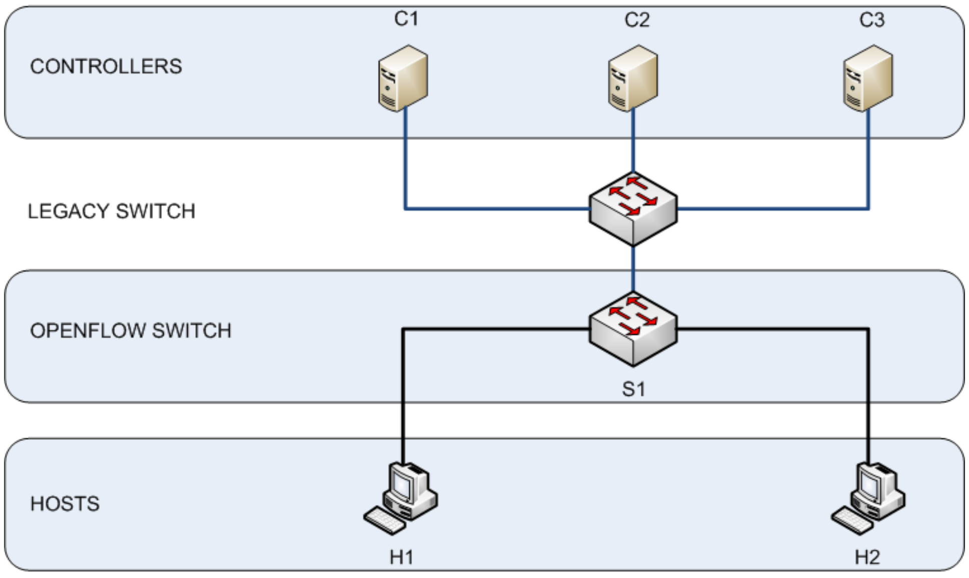


Figure 4-1: Topology

4-2-2 Software Tools

This experiment uses software-based OpenFlow switch, the openvswitch (OVS). To generate traffic, pktgen is used. Pktgen offers higher accuracy compared to other similar tools. Because pktgen runs in the kernel space, unlike other packet generator tools that run in the user space. Tcpdump is used to capture the packets. Network failure is simulated using iptables. To automate the process, all the software tools are run automatically using crontab.

4-2-3 Measurement Scenario

Figure 4-2 shows the traffic generated from H1 to H2. Failover time can be measured by calculating the packet loss during a fixed period of time. See the numbered arrows in Figure 4-2. The arrows show the path of generated packets from H1 to H2. In an ideal condition, the traffic goes from H1 to H2 in the following steps:

1. H1 generates random packets to its interface that is connected to S1. These random packets have random source/destination IP/MAC addresses.
2. Since S1 does not have a flow entry for those packets received from H1, S1 encapsulates them as packet-in messages and sends them to the C1, controller with MASTER role.
3. The controller also does not have the MAC addresses of every packets it received because they are randomly generated. It does not know where to send those packets. Because of that, it sends them back as packet-out messages to S1 with instruction to broadcast them to all interfaces except the source where S1 receive them at the first time.
4. Upon receiving packet-out messages, S1 reads the instruction to broadcast them to all interfaces except the source. Since it has only 2 interfaces, those messages are forwarded to the interface connected to H2. H2 receives randomly generated packets from H1 and captures the packets it received using tcpdump.

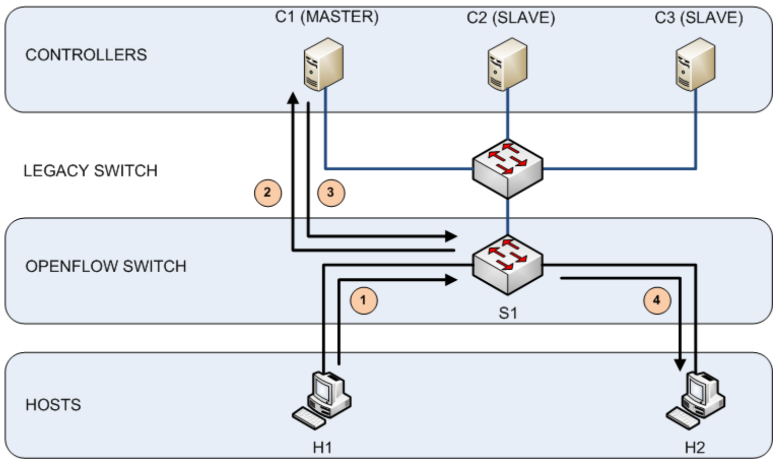
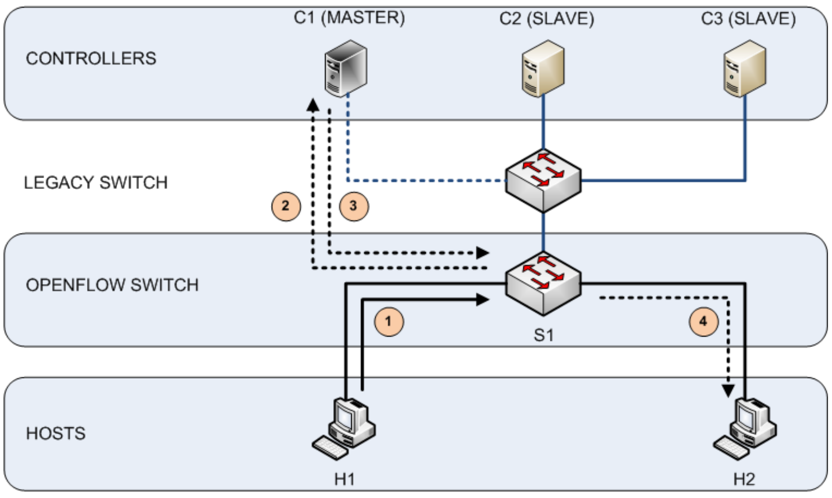
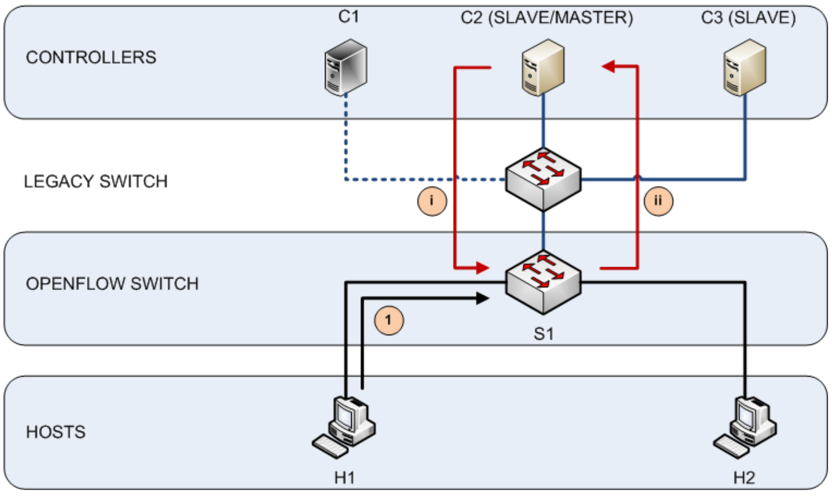


Figure 4-2: Normal Traffic

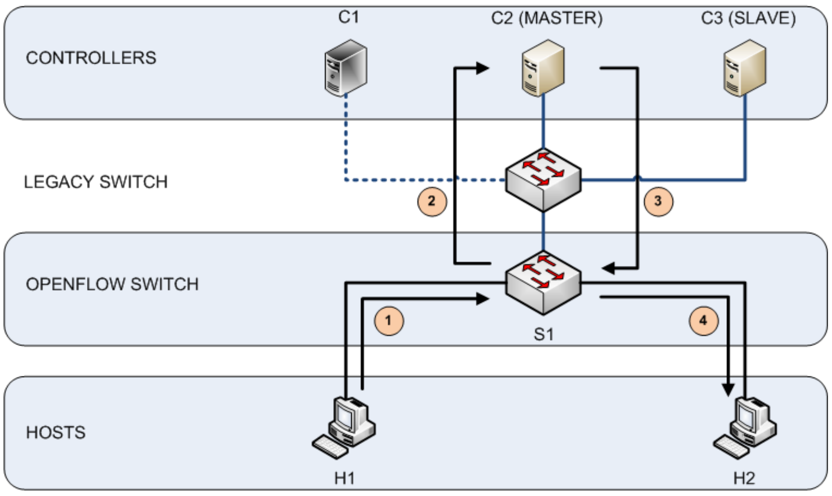
The scenario during the failover process is explained in the following picture at Figure 4-3. In Figure 4-3 there are three phases of the failover process:



(a) Link from MASTER controller is disconnected



(b) SLAVE controller take over MASTER role



(c) The network continue to operate

Figure 4-3: Failover

1. In the first phase, link from the MASTER controller disconnected from the network, see Figure 4-3a. The link from MASTER controller is disconnected by configuring iptables command to DROP every packet it received. H1 still continuously generates packets to its interface that is connected to S1. After S1 detects there is no connectivity to the MASTER controller, all incoming packets get dropped. S1 also does not try to forward those packets by itself because it is configured in `fail_mode: secure`. So there are no packets forwarded to H2 until another controller takes over the MASTER role.

2. The next phase is the take over phase. In Figure 4-3b, S1 still drops every packet from H1 when there is no controller with MASTER role available. Upon detecting the unavailability of C1, C2 notices that there is no controller with MASTER role. C2 changes its role from SLAVE to MASTER by sending `OFPT_ROLE_REQUEST` message to S1. S1 changes C2 role from SLAVE to MASTER if the role value in the message is `OFPCR_ROLE_MASTER`. S1 replies back a `OFPT_ROLE_REPLY` message to acknowledge it. Now S1 has C2 as the new controller with MASTER role.

3. Figure 4-3c shows the final phase, in this last phase the network resumes to operate as usual, the packets traverse from H1 to H2, but through the new controller with MASTER role: C2.

In one session pktgen in H1 generates 60,000 packets with an interval of one millisecond between packets, one minute from the first packet to the last packet. In total there are one hundred sessions executed in a period of every four minutes between sessions. At the same time, H2 starts tcpdump to capture those packets and write those out as separate pcap files for every sessions.

If there is no failure in the controllers, tcpdump reports that it captured 60,000 packets. It means there is no failover between controllers. When there is a failover, tcpdump captures less packets, those lost packets are the number of failover time in ms.

During failover simulation in Figure 4-3, packet loss before the new controller takes over the role of MASTER controller. Since there is one ms interval between packets, the packet loss can be used to calculate how long the failover times in millisecond.

4-3 Conventional Redudancy Protocols

In this section the performance of conventional redundancy protocols is evaluated. VRRP and LVS are implemented in the control plane. A controller failure is simulated. Then the failover time is measured.

4-3-1 Network Layer: VRRP

In this thesis, VRRP is adapted into SDN network. Multiple controllers share one VIP address as one controller, see Figure 4-4. The gateways are the controllers in the control plane, and the hosts are the OpenFlow switches in the data plane. One controller as MASTER, which actively providing services to the switch, and the other as BACKUP, standing by waiting to take role as MASTER controller when the original MASTER fail.

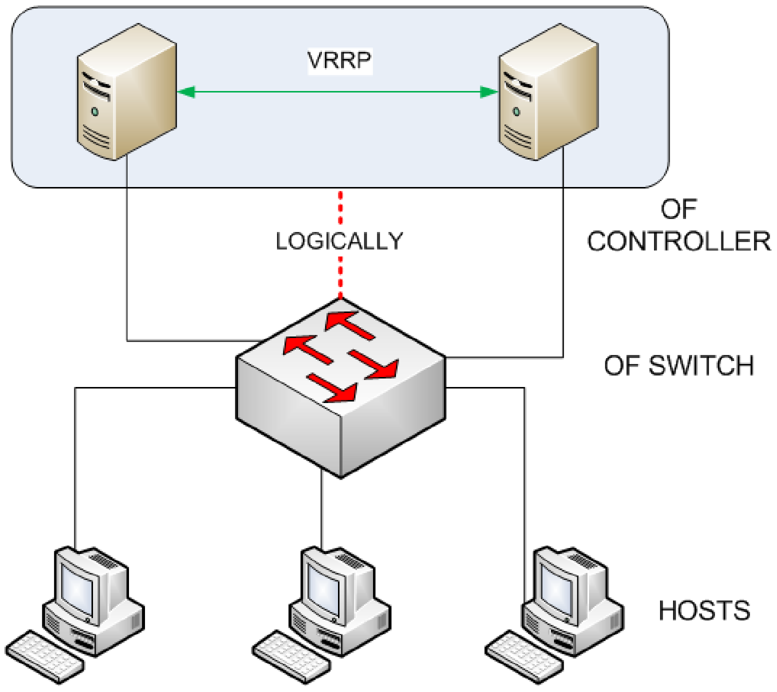


Figure 4-4: SDN Controllers with VRRP

Experiment Evaluation

To demonstrate VRRP on SDN network, Ryu OpenFlow controller and keepalived were used in the experiment. Keepalived is a software tool in Linux that implements VRRP protocol. In popular linux distribution such as Ubuntu, keepalived is part of the distribution package. The configuration for VRRP in keepalived is located at the configuration file: keepalived.conf. In the file, a user can configure the initial state of the controller as MASTER or BACKUP, the VIP and the IP addresses of other controllers. VRRP can work with only two controllers and also supports more than two controllers. If more than two controllers are used, the priority field in the keepalived.conf can be configured to determine the sequence of next controller taking over during failover. Since the sequence of next controller is already set, there is no race condition.

Result and Analysis

The same measurement scenario applied for VRRP as in the open source controller experiment. The average failover time measured from using VRRP is 13,8 second. One possibility why VRRP failover time is in order of second is because VRRP advertisement interval can only be configured in seconds. In the newer specification: VRRPv3, the advertisement interval is in order of centiseconds. Although there is a patch for keepalived to implement VRRPv3, but no manual on how to use it yet, only VRRPv2 implementation was measured in this experiment.

Unfortunately there are some issues using VRRP as robust SDN controller solution. First, VRRP only works if all the controllers are in the same subnet. Because VRRP failover solution relies on ARP, which is a protocol that translate network address into hardware address. Second, VRRP does not provide load-balancing mechanism. It is possible to distribute controller function in VRRP, but it is not practical. Although VRRP supports multiple BACKUP controllers, but only one MASTER controller can serve all switches in the network at a time. So, to distribute controller function we need to configure multiple IP addresses on every controller and configure different controller's IP addresss on every switches, basically a lot of configuration on every devices. Third issue is transport layer failure, since VRRP is a network layer protocol, it can only detect network layer failure. VRRP works well on network layer devices such as router. If there is an application failure on the MASTER controller, VRRP cannot detect it. The BACKUP controller assumes that the MASTER controller is still functional, since it is still reachable at network layer. Another problem is reconnection overhead. Because the switch is only aware of one controller, it only has one OpenFlow channel to the MASTER controller. If the MASTER failed, the switch initiates a new OpenFlow channel to the new controller.

4-3-2 Transport Layer: LVS

Since network layer redundancy network protocol such as VRRP cannot solve transport layer failure, the next option is to use redundancy solution in the transport layer: LVS. LVS can provide redundancy in SDN control plane. Instead of multiple servers, there are multiple controllers. The load balance monitors the availability of the controllers, different from VRRP which the function resides in the BACKUP controller. Figure 4-5 shows LVS adapted into an SDN network.

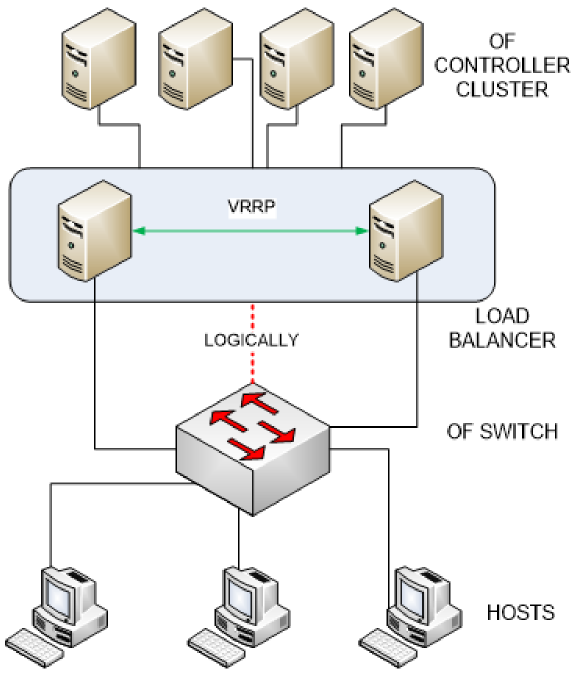


Figure 4-5: SDN Controllers with LVS

Experiment Evaluation

The same tools were used to implement LVS and VRRP: keepalived and Ryu controller. We install keepalived on the load balancers and Ryu on the controllers. The IP addresses of the controllers are written down in the keepalived.conf configuration file. In the file, we can also write down the virtual IP address and the OpenFlow port so the load balancer can check controllers' availability in both the network layer and the transport layer. We can also specify what kind of algorithm the load balancer uses to send packet to the controllers, e.g. using round robin or weighted round robin. There is no minimum or maximum number of controllers can be used. There is also no synchronization between controllers, but the load balancer periodically sends TCP check packet on the OpenFlow port of every controllers. We can also specify the interval of TCP check packets, but unfortunately the interval is in order of seconds, not ms. The number of attempts before declaring that the controller is down can also be specified in the file, e.g. after 3 retries with no reply. If the load balancer does not get any reply after 3 attempts, it will removes that particular controller from its list of available controllers. Any new packet will get forwarded to the other controllers.

Result and Analysis

The same measurement scenario also applied in this experiment. LVS improves several VRRP shortcomings. First, LVS supports load balance feature. Unfortunately this feature does not work for OpenFlow. OpenFlow reuires separate OpenFlow channel for each connection between an OpenFlow switch and an OpenFlow controller. Because LVS is using virtual IP, the switch only acknowledge the existence of single controller. Every time the load balancer distribute OpenFlow message from the switch to a different controller, the switch initiates an OpenFlow channel. Second, LVS provides transport layer TCP check mechanism. Although LVS solves VRRP problem with transport layer failure, it still does not solve reconnection overhead issue. Both VRRP and LVS use single OpenFlow channel from the switch to the controller. Because the switch still only has one OpenFlow channel to the virtual IP address of the controllers. So failover process requires the switch and the new controller to setup a new OpenFlow channel. This unnecessary channel reconnection overhead adds more time to failover process. The average failover time using LVS is 10.7 seconds. LVS's failover time is also in order of seconds because the TCP check packets are sent in interval of seconds too, in a way similar to VRRPv2 advertisement interval which is also in order of seconds.

4-4 Open Source Controllers

The previous section shows that conventional redundancy solution such as VRRP and LVS enable multiple controllers implementation on OpenFlow, but the failover performance is still quite slow for carrier grade network. This section discusses multiple controllers implementation and failover mechanism of the open source controllers. First the setup is evaluated. Then the result is presented.

4-4-1 OpenMUL

Experiment Evaluation

Installing OpenMul was quite easy to do, its documentation is comprehensive. First, download OpenMul source code from git. Next, compile build.sh to build the necessary binaries. In this experiment, we start OpenMul by running mul.sh script and we only enable two features in the script: `l2switch` and `start-ha`. Both options enable OpenMul to implement layer 2 learning switch and high availability feature, respectively. Running OpenMul with two controllers is straightforward, just execute mul script with start-ha option and the name of the peer controller: `./mul.sh start-ha l2switch <peer controller's IP address>`. The command `./mul.sh` starts OpenMul, `start-ha` enables HA feature, `l2switch` runs layer 2 learning switch application. It is suggested in the documentation that the both controllers need to be connected first in HA mode before connecting to the switch. So, the controllers can establish a negotiation session between themselves to elect a controller with MASTER role.

Result and Analysis

During the experiment we trace the OpenFlow packets between the controllers to the switch and between the controllers themselves. Compared to other open source controllers in this experiment, OpenMul was the easiest controller to deploy and to implement the multiple controllers feature. Although easy to deploy, unfortunately OpenMul lacks in terms of features. Specifically the multiple controllers feature, since in this experiment we evaluate that particular feature. The shortcoming that we encounter in OpenMul HA feature is that it supports no more than two controllers. It is unknown whether the commercial version of OpenMul can support more than two controllers or not. The OpenMul controllers use TCP port 7745 to communicate with each other, e.g. to check the other controller availability. The other shortcoming is that a user cannot configure the initial role of the controllers, and this cannot specifically which particular controller gets assigned the first MASTER role.

In the initial setup, each controller sends `role_request` messages with its role as MASTER or SLAVE respectively, so no race-condition occurs. Since there are only two controllers, the SLAVE immediately takes over when the MASTER is disconnected from the network, it sends `role_request` message to the switch with incremented `generation_id`. We were able to automate the process of calculating failure recovery time by using cron. The process repeated for 200 times. On average, OpenMul's failover time is 11.5 second.

4-4-2 Floodlight

Experiment Evaluation

Installation manual for Floodlight is available in its wiki page. The instruction is very clear, to run Floodlight, all of the prerequisite packages need to be installed, e.g. java, git, and ant. Its multiple controller feature can be enabled by modifying floodlightdefault.properties configuration file. In the file, all of the controllers' IP addressess, their initial role as MASTER or SLAVE, and the port they use to communicate with each other need to be written down. In order for them to communicate with each other, first java keystore is generated and copied to each of the controllers. There is no minimum number of controllers that Floodlight need to enable multiple controllers, it also supports more than two controllers.

Result and Analysis

The same process as in OpenMul experiment is implemented to calculate Floodlight's failover time. During tracing the packets, it is discovered that Floodlight does not have a mechanism to determine which controller is next in line to take over the MASTER role. If only two controllers are used, the failover process works just fine. But if more than two controllers are used, race conditions occur after the MASTER controller gets disconnected. Every SLAVE controller attempts to take over the MASTER role at the same time by sending `role_request` message. There was a period of time that the switch changed the MASTER controller more than once. The switch chooses the controller that sends the last `role_request` packet it received. The race condition occurs because Floodlight does not increment the `generation_id` field in its `role_request` message, the switch will change the controller with MASTER role every time it received `role_request` message with MASTER role. On average, Floodlight's failover time is 29.9 second.

4-4-3 OpenDayLight

Experiment Evaluation

There are several stable versions available, we install version 4.1 Berrylium. OpenDaylight documentation is very comprehensive, every release has its own separate manuals for installation, user, and for developer. Although OpenDaylight is much more complex than OpenMul or Floodligth, it was actually easy to install and to implement multiple controllers. After downloading and extracting OpenDaylight on each controller, to enable multiple controllers feature we only need to edit three configuration files: akka.conf, modules.conf, and module-shards.conf. The akka.conf defines all the controllers IP and port for synchronization, modules.conf defines what kind of data shards from ODL database can be stored, module-shard defines the data-shards are replicated to which controllers. Those files can be edited either manually or automatically. Enabling multiple controllers automatically can be done by running configure_cluster.sh script: `configure_cluster.sh <index> <seed_nodes_list>`. OpenDaylight supports more than two controllers, but the failover part of the feature only works when there are at least three controllers. If implemented with two controllers, OpenDaylight can still work with the switch, one controller as MASTER, the other as SLAVE, but there will be no failover when the MASTER is down.

Result and Analysis

After starting the controllers, they negotiate with each other to determine which one is the MASTER. Each of them sends a join command message to the other controllers. Those controllers comunicate using port 2550, but it can be modified into other port in the akka.conf configuration file. They also periodically send messages to each other for healthchecking purpose. Since they already negotiated which controller is the MASTER, when a switch connected, those controllers send `role_request` message with their respective role, either MASTER or SLAVE. They also already negotiated which controller is next in line to take over if the MASTER down. When the SLAVE controllers detect that the MASTER is down, the successor sends `role_request` with incremented `generation_id` to take over MASTER role. Since the next MASTER controller is already set, there is no race conditions occurs. The average ODL's failover time is 12.7 second.

4-4-4 ONOS

Experiment Evaluation

ONOS has several stable versions available. We use version 1.5.1 Falcon. Its documentation is available in its wiki page. Similar to OpenDaylight, ONOS is much more complex than OpenMul or Floodlight, but it was easy to install. Just download ONOS source code from git and then compile it to build the necessary binaries. ONOS provides script to execute remote install: `onos-install <controller IP address>`. ONOS multiple controllers feature is the easiest to install compared to other controllers. It has its own script to automate remote install for multiple controllers and it also automates the clustering configuration between the controllers: `onos-form-cluster <controllers IP addresses>`.

Result and Analysis

ONOS provides very useful CLI. It can select or change the role for each controller. It makes easier during measurement. We can configured the MASTER on C1 manually. ONOS does not has maximum number for controllers but it has similar quorum mechanism as OpenDaylight. In order for failover feature to work, there must be three ONOS controllers. ONOS can still works with only two controllers but it cannot execute failover when the MASTER controller fails. ONOS does not increment generation ID field in its role request message during failover, but it has its own mechanism to decide which SLAVE controllers will take over the MASTER role during failover, preventing race condition.

4-5 Comparison

The table 4-1 is the result of measured failover time from different open source controllers. Failover time is the total amount of time from network disconnection of C1 or the time H2 received the last packet-out from C1 to the time H2 received the first packet-out from C2. Failure detection time is the time measured from network disconnection of C1 to the time S1 received role request message from C2. Take over time is the time measured from the time the S1 received role request message from C2 to the time H2 received first packet-out from C2.

Table 4-1: Benchmark

Controller	Failover time (s)	Min.	Max.	Race condition	Incremented generation_id
OpenMul	11.5	2	2	None	Yes
Floodlight	29.9	2	None	Yes	None
ONOS	2.9	3	None	None	None
OpenDaylight	12.7	3	None	None	Yes
VRRP	13.8	2	None	None	-
LVS	10.7	2	None	None	-

Chapter 5

ACRoDiF: A Controller Robustness and Distribution Framework

In the previous chapter, the failover time of several open source controllers has discussed. Unfortunately the failover time is still quite high. The fastest failover time is in ONOS with average of 2.9 second. This is still much higher than 50ms, which is the acceptable carrier-grade recovery time [16].

This chapter proposes ACRoDiF (A Controller Robustness and Distribution Framework) to improve SDN control plane robustness using multiple controllers. ACRoDiF is inspired by the Content Delivery Network (CDN) concept. CDN uses DNS to direct client traffic to the closest server based on the source IP address of the client.

5-1 CDN Overview

CDN is a distributed system that content provider uses to distribute web pages and content to be cached in the geographically closest web server to the client. Conventional client-server model that a web server uses is a centralized web server. Nowadays, the problem with a centralized web server is that many content provider have their content accessed from clients around the world. The latency is quite high when a client accesses content from a web server located in a different continent. Using CDN, content provider duplicates its own web server, distributes them around the world, and provides cached content such as text, image, or video on itw distributed web servers. CDN enables client to access a web page faster than centralized model, see Figure 5-1.

CDN exploits the DNS server function to distribute network traffic to the closest server. In CDN, there is a hierarchy how DNS server works. Figure 5-2 shows two clients from different continents access the same URL: example.com. Because example.com uses CDN, the domain resolves to several different IP addresses, e.g. each IP address for a web server in each continent. The process how a client access a web page from example.com is the following:

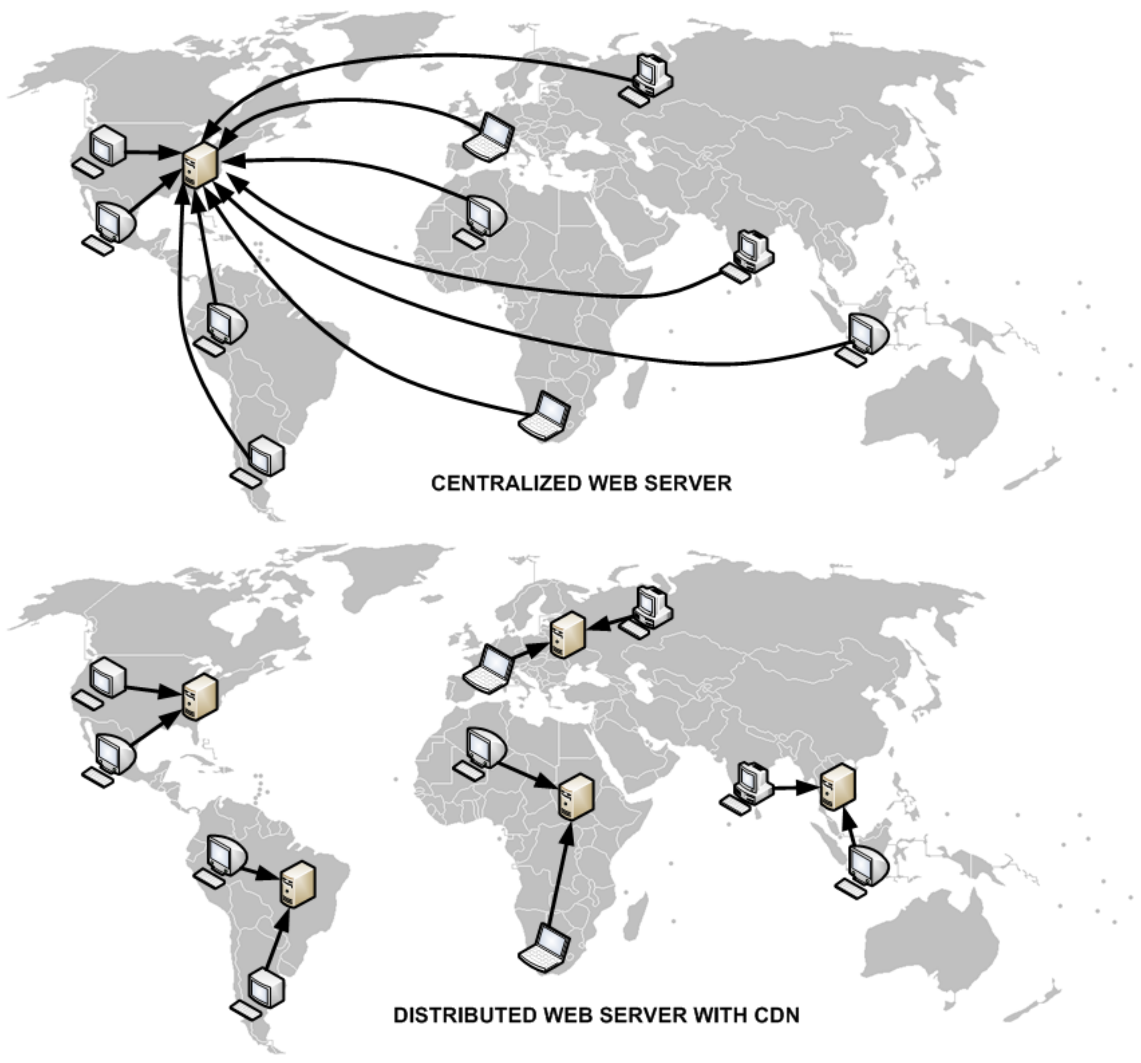


Figure 5-1: CDN

1. When a client access a web server, first the client query the IP address of the web server's domain from a recursive DNS server.

2. The recursive DNS server queries the requested domain to the authoritative DNS server of that domain.

3. The authoritative server looks up the client IP address to identify the client's geographic location. Then the authoritative server replies the query with the IP address of the closest web server to the client.

4. The recursive server replies to the client with the information from the authoritative server.

5. Each client accesses the web server of example.com from its own continent.

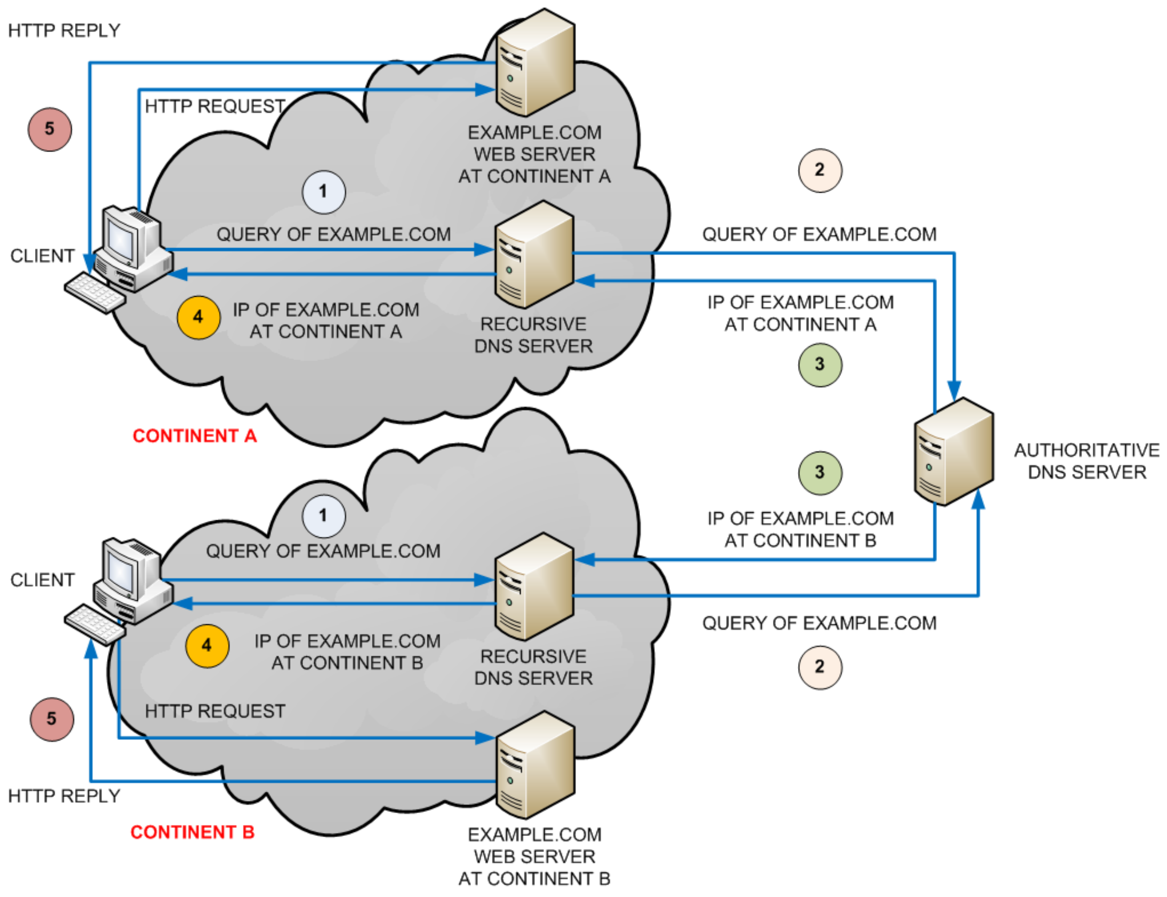


Figure 5-2: DNS in CDN

5-2 Proposal

5-2-1 ACRoDiF Requirements

The previous chapters has discussed the problems with multiple controllers solutions for OpenFlow. There are several problems that can be improved, specifically in terms of controller failover. The following are the requirements for ACRoDiF to enhance controller failover feature in an OpenFlow controller:

- **More than two controllers support**: Not all multiple controllers feature support more than two controllers in the control plane, e.g. OpenMul. ACRoDiF should be able to support more than two controllers implementation. There is no limit on how many controllers can be implemented in the network. In ACRoDiF, there are two types of controller: primary controller that controls the data plane and backup controller that takes over the primary role when the primary controller fail.
- **Liveness monitoring**: ACRoDiF must have liveness monitoring mechanism to identify a controller failure. Since OpenFlow has echo request/reply message, ACRoDiF can send echo request/reply message to monitor both the control plane and the data plane. ACRoDiF can use liveness monitoring to detect controller failure.

- **Failover mechanism**: ACRoDiF must be able to execute failover upon detecting a controller failure. When ACRoDiF detects failure in the primary controller, it promotes the backup controller to primary controller.

- **Eliminate race condition**: Although Floodlight supports more than two controllers in the control plane, unfortunately it has race condition problem. ACRoDiF failover mechanism must have a solution to avoid race condition. In this experiment, we manually configure the priority list of the controller to be promoted as primary during failover.

5-2-2 ACRoDiF Overview

ACRoDiF expands distribution concept borrowed from CDN to be applied on multiple OpenFlow controllers. The switch and the controller in OpenFlow connection resembles the connection between a client and server. The client, in this case an OpenFlow switch sends a request message to the server or the controller, and the switch gets a reply message from the controller. ACRoDiF introduces a proxy layer between the control plane layer and the data plane layer, see Figure 5-3. This proxy layer is responsible for distribution function in ACRoDiF, similar to the DNS server role in CDN. The controller and the switch are oblivious to the proxy layer. The proxy layer decides which controller is active as the primary controller to receive packet-in from the data plane. It also monitors the status of every controller in the control plane. Controller only detects the proxy as the switch. The switch detects the proxy as the controller. This middlebox approach enables ACRoDiF to use any OpenFlow controller, since it does not require any modification on the control plane or in the data plane, see Figure 5-4.

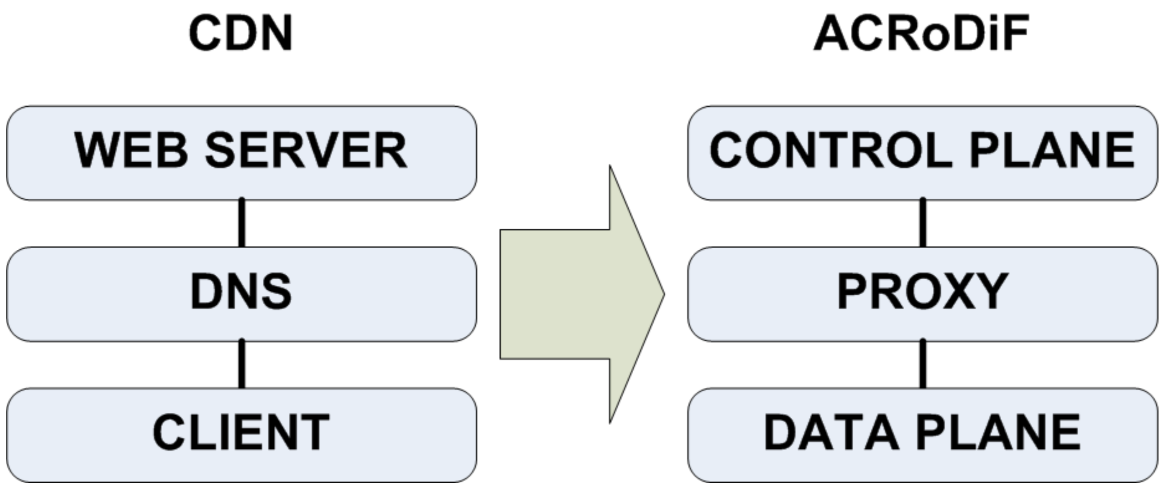


Figure 5-3: CDN and ACRoDiF

The proxy in ACRoDiF is based on a modified version of Ryu controller. As an OpenFlow controller, Ryu already has basic controller function. It listens to OpenFlow TCP port. If a switch initiates a TCP connection to establish an OpenFlow channel, it can create an OpenFlow channel to the data plane. We modify Ryu controller so when an OpenFlow

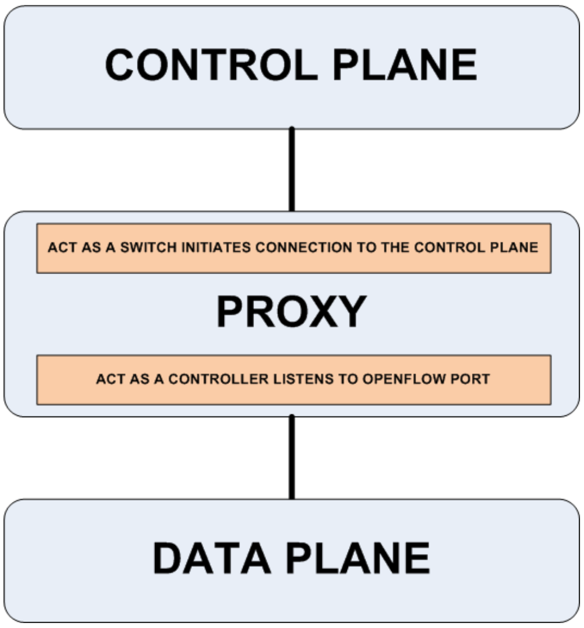

CONTROL PLANE
ACT AS A SWITCH INITIATES CONNECTION TO THE CONTROL PLANE
PROXY
ACT AS A CONTROLLER LISTENS TO OPENFLOW PORT
DATA PLANE

Figure 5-4: ACRoDiF

channel is created from the data plane to the proxy, it acts as a switch to the control plane and initiates TCP connection to create an OpenFlow channel to the controller.

In this experiment we use two controllers, two proxies, one switch, and two hosts, see Figure 5-5. Any network topology is possible, but we decided to implement the simplest topology. The proxy connects to the two controllers if it detects a connection attempt from a switch. Each connection from a switch results to two OpenFlow channel from the proxy to the control plane, one for each controller.

5-2-3 The Proxy Layer

In OpenFlow, the OpenFlow channel between the control plane and the data plane is initiated by the switch in the data plane. In ACRoDiF, the switch is also initiates the channel. First, S1 initiates TCP connection to both P1 and P2. After it connected to the proxies, S1 exchanges hello message to both P1 and P2 which creates an OpenFlow channel between the data plane to the proxy layer. After the OpenFlow channel between proxy layer and the data plane created, P1 and P2 initiates TCP connection to the control plane: C1 and C2. Next P1 & P2 exchanges hello message to C1 & C2 creating an OpenFlow channel between the proxy layer to the control plane. Then P1 sends role request message as MASTER to S1, then S1 responds with sending role reply to both P1 and P2 with different roles: MASTER and SLAVE respectively. In this simple setup we preconfigured P1 role as MASTER and P2 as SLAVE, but it is also possible to implement any algorithm to elect which one is the MASTER, e.g. an algorithm that decide proxy with the lowest latency to the switch as the MASTER, presumably it is the closest proxy. The proxy relays every message between the control plane and the data plane, see Figure 5-6. The following explains how the proxy layer works:

1. H1 generates packets to S1 with destination address of H2.

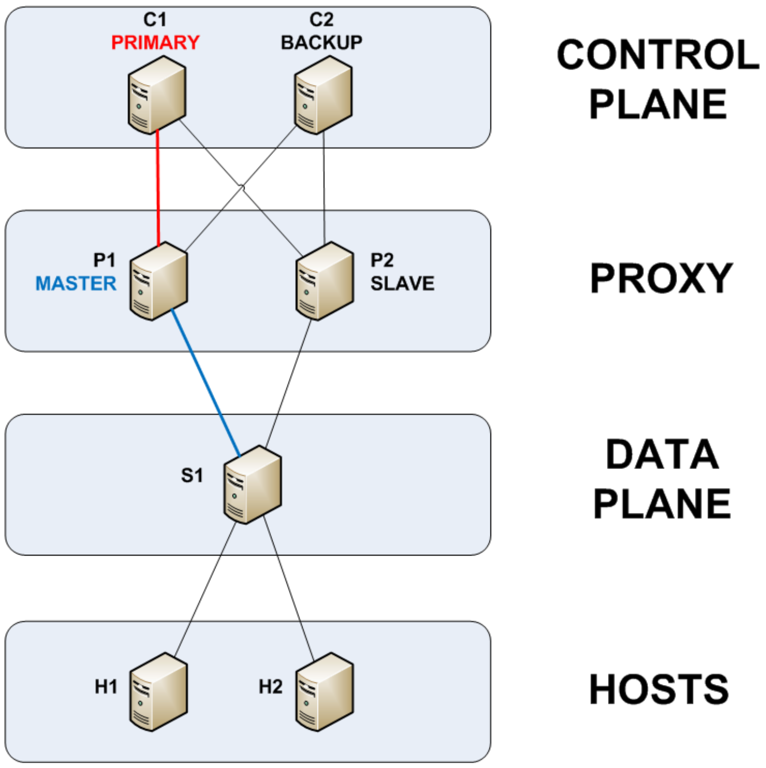


Figure 5-5: ACRoDiF Measurement Topology

2. S1 encapsulates every packet from H1 as a packet-in and forwards it to P1.
3. Then P1 forwards it to the primary controllers: C1.
4. C1 replies the packet-in from P1 with a packet-out.
5. P1 receives packet-out from C1 and then forwards it to S1
6. S1 unencapsulates packet-out from P1 and forwards it to H2

5-2-4 Experiment Evaluation

The measurement scenario is the same as in the previous chapter. When the primary controllers fail: C1 disconnected, Figure 5-7 shows the failover mechanism as the following:

1. C1 disconnected, P1 still receives packet-in from S1.
2. P1 periodically checks the availability of both the controllers and the switch by sending echo request message with an interval of 15ms and receives echo reply message. After not receiving echo reply message from C1 for three times of the echo request interval, P1 declares that C1 is not available, P1 removes C1 from the list of primary controllers. Then P1 promotes C2 role from backup controller to primary controller. The failover time is calculated by counting the packet loss during the failover.

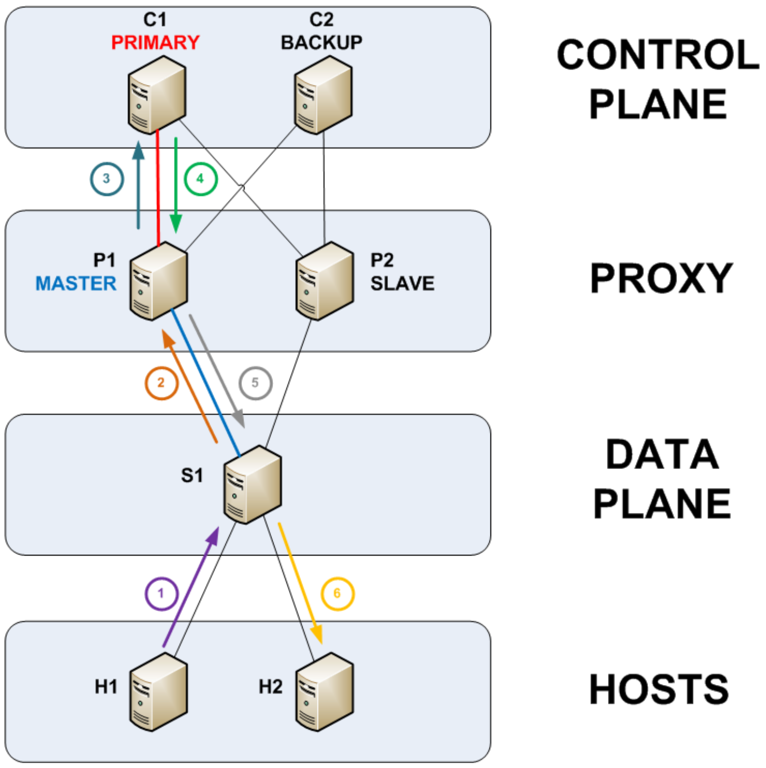


Figure 5-6: ACRoDiF Relays Message

3. The network has new primary controller: C2. P1 forwards packet-in from S1 to C2, and forwards packet-out from C2 to S1.

To calculate the failover time of ACRoDiF, we use the previous open source controllers. Because ACRoDiF features is to overcome the shortcomings of multiple controllers feature on the existing open source controller.

5-2-5 Result and Analysis

Ryu

Since Ryu does not have built-in failover mechanism, each Ryu controller is a standalone controller. Ryu also does not have any feature for failover.

OpenMul

In order ACRoDiF can work with OpenMul, the controller must operate as a stand alone controller. No need to enable multiple controlllers feature. We only run `mul.sh` script to run OpenMul and enable one feature the `l2switch` to implement layer 2 learning switch in the data plane.

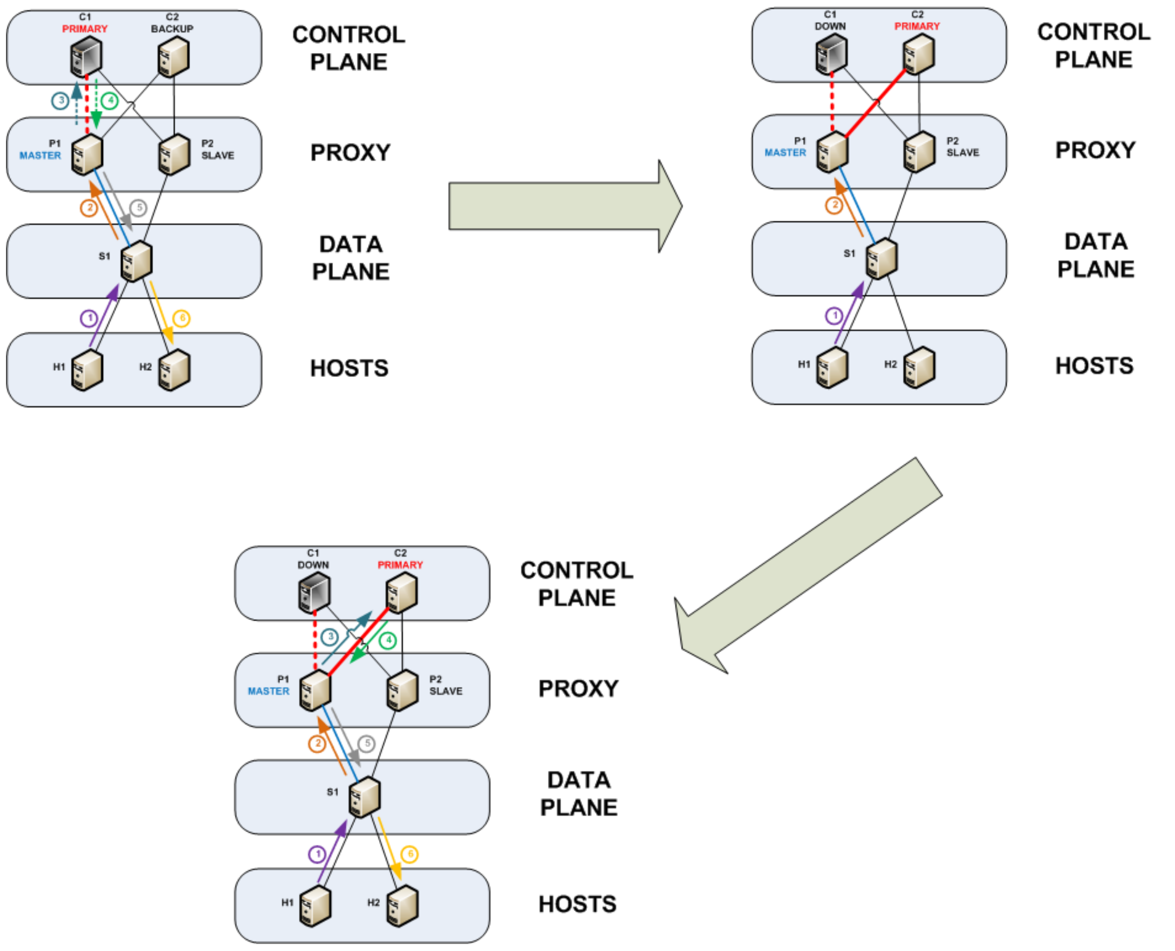


Figure 5-7: ACRoDiF Failover

ACRoDiF improves OpenMUL's shortcoming of not able using more than two controllers. With ACRoDiF, it is possible to deploy more than two OpenMUL controllers in the control plane. The proxy plane is in charge of managing those controllers.

Floodlight

Floodlight's own multiple controllers feature does not have a mechanism to elect next controller that takes over the primary controller role. This causes race condition during failover if more than two controllers deployed in the control plane. Every backup controller tries to take over the primary role. Because Floodlight does not increment generation id field in the role request message, the switch accepts and changes the controller role every time it receives role request message. Floodlight has no function that coordinates roles between multiple controllers. One example of this problem is when the failed primary controller get back online, it sends role request with MASTER role after the OpenFlow channel established. ACRoDiF assumes that function, it decides the next in line controller that takes over the primary role during failover. If the failed primary controller get back online, ACRoDiF puts it in the pool of available backup controllers. ACRoDiF avoid race conditions problem in Floodlight.

OpenDayLight

OpenDaylight does not have race condition problem, but its shortcomings is the minimum number of controllers to enable failover feature and its failover time. Its quorum mechanism makes failover only works when there are at least three controllers in the control plane. ACRoDiF eliminates the quorum requirement of OpenDaylight. The proxy layer can perform failover in the control plane with only two controllers.

ONOS

In terms of failover feature, ONOS is similar to OpenDaylight. It does not have race condition problem. But it has the same quorum mechanism, ACRoDiF eliminates the problem with minimum three controllers for failover.

Failover Time

ONOS has the shortest failover time compared to other open source controllers: 2.9s. It is still higher than standard carrier-grade recovery time: 50ms [16]. ACRoDiF sends echo request to the primary controller with an interval of 15ms. It declares controller failure after not receiving echo reply for 45ms. We specifically decides after 45ms as the period to proceed failover because we want to achieve failover time that is under 50ms. Unfortunately the average failover time measured using ACRoDiF with open source controllers is 76ms, still higher than 50ms, see Table 5-1 for more detail failover time. If the echo request interval is configured lower than 15ms, the proxy becomes too sensitive. It generates false alarm and proceed to failover, although the primary controller is working just fine.

Table 5-1: Failover Time with ACRoDiF

Controller	Failover time (s)	Failover time with ACRoDiF (ms)
OpenMul	11.5	74
Floodlight	29.9	75
ONOS	2.9	74
OpenDaylight	12.7	82
Ryu	-	73

5-3 ACRoDiF with Two Active Primary Controllers

Although ACRoDiF already significantly decreases failover time, but there is still room for improvement. Failover can be eliminated completely if there are more than one active primary controller. For example, if there are two EQUAL controllers in the control plane, there will be no failover when one controller fail, since the other one still active. But the downside of using multiple EQUAL controllers is there are duplicate for every message to the data plane. Each controller replies every message from the data plane. The data plane received duplicates of OpenFlow messages from each EQUAL controller. Duplicate messages generate additional overhead in the OpenFlow channel.

ACRoDiF can solve the duplicate messages problem. Since ACRoDiF is located in between the control plane and the data plane, ACRoDiF can function as a middlebox that filter duplicate messages. It can apply FIFO (First In First Out) algorithm. Every packet-out message has a unique `buffer_id` number that correlates to its packet-in origin. When ACRoDiF received a packet-out message from one of the two active primary controllers, it will check whether it already received other packet-out message with the same `buffer_id` or not. If no matching `buffer_id`, the packet-out will get forwarded to the data plane, if there is a matching `buffer_id`, the packet-out will be dropped since the other controller already sent the same message. There will be no duplicates arrive at the data plane because any duplicates dropped by ACRoDif.

The measurement topology is the similar with the previous topology. The differences are now exists two primary controllers: C1 & C2 and there is a third controller as backup controller in the network: C3, see Figure 5-8. If one of the two primary controllers fail, the backup controller takes over the role to become one of the two primary controllers.

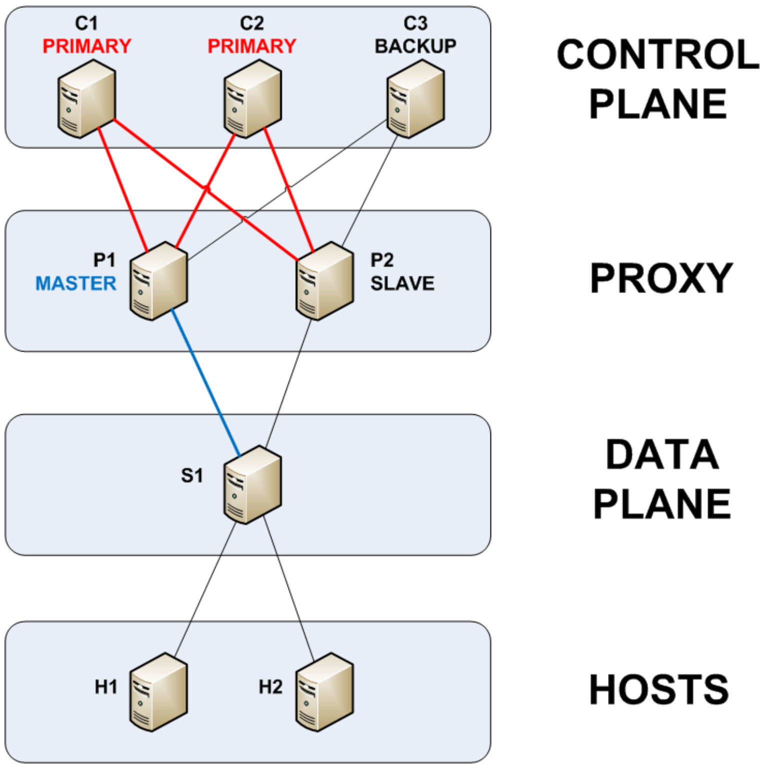


Figure 5-8: Measurement Topology

During normal traffic, packets traverse the network in the following steps in Figure 5-9:

1. H1 generates packets to S1 with destination address of H2.
2. S1 encapsulates every packet from H1 as a packet-in and forwards it to P1.
3. Then P1 forwards it to the two primary controllers: C1 and C2.
4. C1 and C2 reply packet-in from P1 with packet-out.

5. P1 receives duplicates of packet-out from C1 and C2. P1 forwards the first packet-out arrived either from C1 or C2 to S1 and discards the duplicate. Since the proxy decides to forward only the first packet-out arrived to S1, there are no duplicates of packet-out at the switch.
6. S1 unencapsulates packet-out from P1 and forwards it to H2

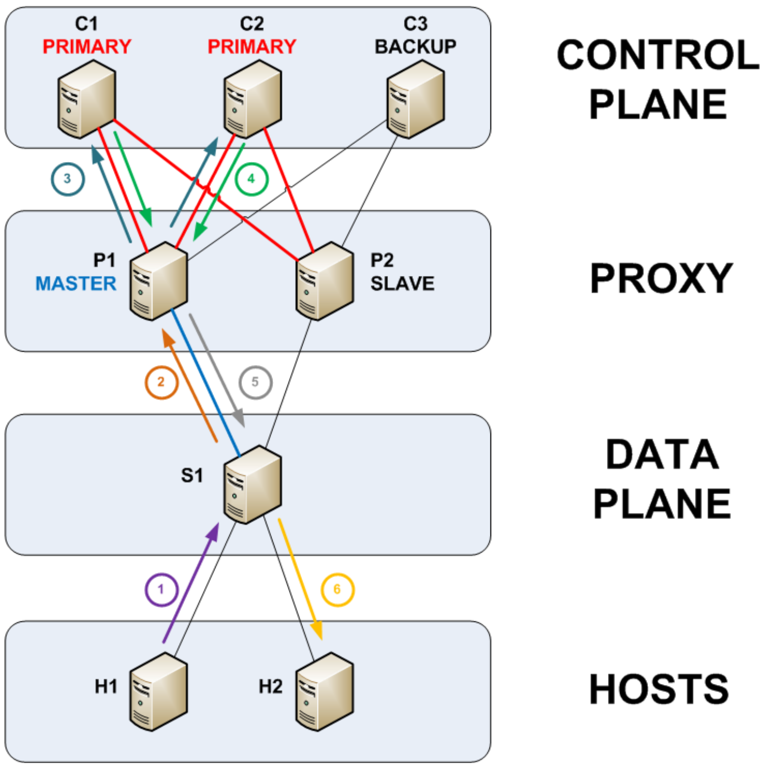
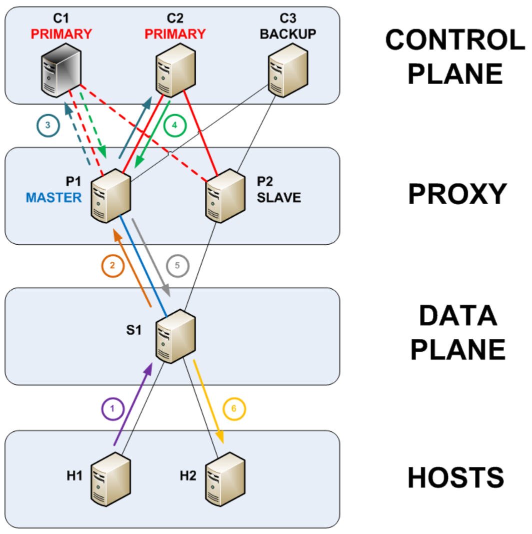


Figure 5-9: Normal Traffic

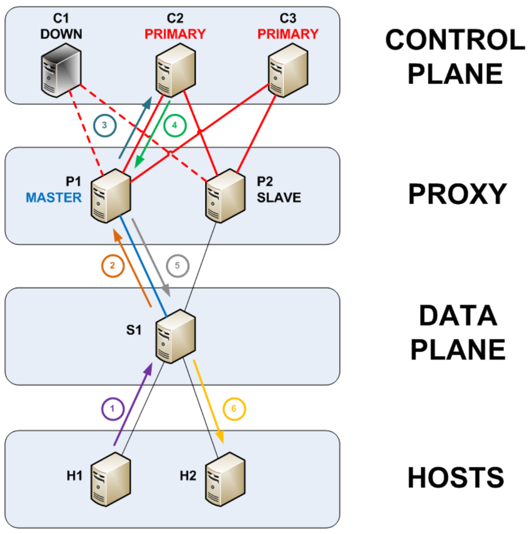
When one of the primary controllers fail, e.g. C1 disconnected, Figure 5-10 shows the failover mechanism as the following:

1. After C1 disconnected, P1 can only sends & receives packet-in & packet-out to & from C2, the only primary controller available.
2. P1 periodically checks the availability of both the controllers and the switch by sending echo request messages with an interval of 15ms and receives echo reply messages. After not receiving echo reply messages from C1 for three times of the echo request interval, P1 declares that C1 is not available, P1 removes C1 from the list of primary controllers. Then P1 promotes C3 role from backup controller to primary controller.
3. The network has two primary controllers, it operates just like how it normally operates with two primary controllers.

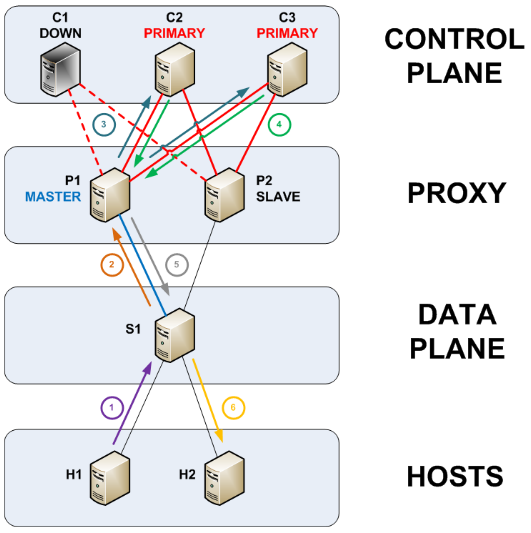
Since there is always duplicates of packet-out from the control plane, so there is no packet loss in the data plane during failover. There is no failover time is calculated when using two active primary controllers in ACRoDiF. The challenge when developing this feature was identifying the packet-outs of the same session from C1 and C2. P1 must able to identify the same packet-outs from C1 and C2 so it does not discard the wrong packet-out message. Fortunately P1 can exploit `buffer_id` field in the packet-out header so it can discard any packet-out with the same `buffer_id` number, see figure 5-11.



(a) C1 is disconnected



(b) C3 promoted to primary role



(c) The network continue to operate

Figure 5-10: Failover

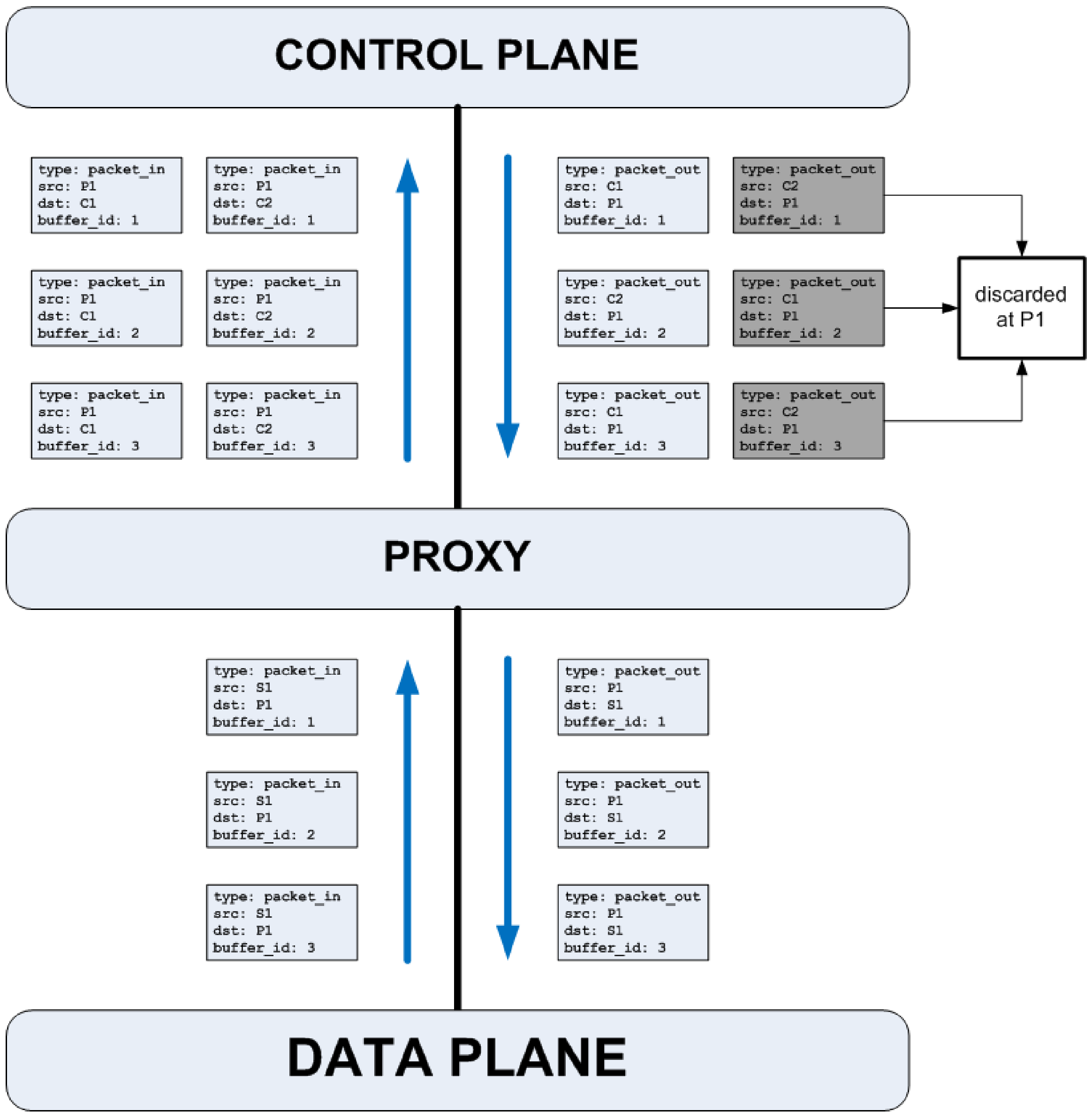

CONTROL PLANE
type: packet_in src: P1 dst: C1 buffer_id: 1
type: packet_in src: P1 dst: C2 buffer_id: 1
type: packet_in src: P1 dst: C1 buffer_id: 2
type: packet_in src: P1 dst: C2 buffer_id: 2
type: packet_in src: P1 dst: C1 buffer_id: 3
type: packet_in src: P1 dst: C2 buffer_id: 3
type: packet_out src: C1 dst: P1 buffer_id: 1
type: packet_out src: C2 dst: P1 buffer_id: 1
type: packet_out src: C2 dst: P1 buffer_id: 2
type: packet_out src: C1 dst: P1 buffer_id: 2
type: packet_out src: C1 dst: P1 buffer_id: 3
type: packet_out src: C2 dst: P1 buffer_id: 3
discarded at P1
PROXY
type: packet_in src: S1 dst: P1 buffer_id: 1
type: packet_in src: S1 dst: P1 buffer_id: 2
type: packet_in src: S1 dst: P1 buffer_id: 3
type: packet_out src: P1 dst: S1 buffer_id: 1
type: packet_out src: P1 dst: S1 buffer_id: 2
type: packet_out src: P1 dst: S1 buffer_id: 3
DATA PLANE

Figure 5-11: Duplicates of packet-out

Chapter 6

Conclusion

6-1 Research Questions Revisited

In this chapter, we revisit our research questions and view how they can be answered:

- What kind of improvements are needed for the SDN control plane to overcome SPoF?
  Chapter 3 shows that distributed controller approach is the solution for SPoF. Chapter 5 presents a new framework for distributed controller: ACRoDiF, it enhances the distributed controller approach with another layer: the proxy layer. The proxy layer in ACRoDiF improves the shortcomings of current SPoF solutions presented in Chapter 4 benchmark.

- How can the control plane detect failure when failure occurs in one of the controllers?
  In Chapter 5, ACRoDiF demonstrates liveness monitoring mechanism using OpenFlow's echo request/reply message. It is possible to delegate the liveness monitoring from the control plane to the proxy. In ACRoDiF, the proxy is responsible for controllers liveness monitoring.

- How to minimize failover period when there is controller failure?
  The experiment in Chapter 5 shows a promising result. The proxy layer in ACRoDiF provides failover mechanism that has lower failover time than the open source controllers. Furthermore, using two active primary controllers in ACRoDiF proves that failover time can be eliminated completely.

- How to avoid race condition between controllers in during failover?
  Because the distribution function resides in the proxy layer. The proxy can decide which controller is next in line to take over the role of the primary controller, no more race condition.

6-2 Future Work

We believe that this thesis is far from complete. There are many aspects that are need to be discussed. These are several recommendations for further research:

- Investigate on why using echo request interval that is lower than 15ms generates false alarm
  Current solution using the interval of 15ms only gives us the average failover time of 76ms, which is still higher than 50ms, the acceptable carrier-grade recovery time standard. Theoretically, lowering the interval gives us lower failover time instead of false alarm.
- Conduct another experiment with different measurement scenario
  The measurement scenario in this thesis only consumes a small amount of network bandwidth because of the low throughput. Further experiment on different scenario is necessary to observe ACRoDiF performance. For example with higher throughput in the network.
- Enhance ACRoDiF feature
  The following ideas present several feature that can be added to ACRoDiF in the future to improve its performance:

  - **Scalability feature**, adding a new controller in the cluster after a switch connects to the proxy is not yet possible. Currently, we need to preconfigured the controllers before a switch connect to the proxy layer
  - **Preempt feature**, in VRRP there is a preempt feature, which means if a failed network element gets back online it will take over the role, in ACRoDiF after a controller is declared not available by the proxy, it remains unavailable eventhough it gets back online

Bibliography

[1] Ameen Banjar, Pakawat Pupatwibuli, and Robin Braun. “DAIM: a mechanism to distribute control functions within OpenFlow switches”. In: *Journal of Networks* 9.1 (2014), pp. 1–9.

[2] Pankaj Berde et al. “ONOS: towards an open, distributed SDN OS”. In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, pp. 1–6.

[3] Fábio Botelho et al. “On the design of practical fault-tolerant SDN controllers”. In: *Software Defined Networks (EWSDN), 2014 Third European Workshop on*. IEEE. 2014, pp. 73–78.

[4] Andrew R Curtis et al. “DevoFlow: Scaling flow management for high-performance networks”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. 4. ACM. 2011, pp. 254–265.

[5] Paulo Fonseca et al. “A replication component for resilient OpenFlow-based networking”. In: *Network Operations and Management Symposium (NOMS), 2012 IEEE*. IEEE. 2012, pp. 933–939.

[6] Open Networking Foundation. *OpenFlow Switch Specification V 1.3. 0 (Wire Protocol 0x04), June 25, 2012.*

[7] Open Networking Foundation. *SDN Architecture 1.0 Overview, November, 2014.*

[8] Arpit Gupta et al. “Sdx: A software defined internet exchange”. In: *ACM SIGCOMM Computer Communication Review* 44.4 (2015), pp. 551–562.

[9] Soheil Hassas Yeganeh and Yashar Ganjali. “Kandoo: a framework for efficient and scalable offloading of control applications”. In: *Proceedings of the first workshop on Hot topics in software defined networks*. ACM. 2012, pp. 19–24.

[10] Yannan Hu et al. “Balanceflow: controller load balancing for openflow networks”. In: *Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on*. Vol. 2. IEEE. 2012, pp. 780–785.

[11] IHS. *Businesses Losing $700 Billion a Year to IT Downtime, Says IHS*. URL: http://news.ihsmarkit.com/press-release/technology/businesses-losing-700-billion-year-it-downtime-says-ihs (visited on 06/14/2017).

[12] Ponemon Institute. *Cost of Data Center Outages, January 2016*. URL: https://www.vertivco.com/globalassets/documents/reports/2016-cost-of-data-center-outages-11-11_51190_1.pdf (visited on 06/14/2017).

[13] KulKloud. *BEEM*. URL: http://www.kulcloud.com/beem/ (visited on 07/20/2016).

[14] Nick McKeown et al. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[15] Jan Medved et al. "Opendaylight: Towards a model-driven sdn controller architecture". In: *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a*. IEEE. 2014, pp. 1–6.

[16] B Niven-Jenkins et al. *Requirements of an MPLS transport profile*. Tech. rep. 2009.

[17] ON.LAB. *Introducing ONOS - a SDN network operating system for Service Providers, November 2014*.

[18] *OpenMul*. URL: http://http://www.openmul.org/ (visited on 07/20/2016).

[19] *Project Floodlight*. URL: http://www.projectfloodlight.org/ (visited on 05/04/2016).

[20] Rob Sherwood et al. "Flowvisor: A network virtualization layer". In: *OpenFlow Switch Consortium, Tech. Rep* 1 (2009), p. 132.

[21] Adrian S-W Tam, Kang Xi, and H Jonathan Chao. "Use of devolved controllers in data center networks". In: *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*. IEEE. 2011, pp. 596–601.

[22] MuL development team. *MuL 3.2.5 Commercially available now!* 2014. URL: https://openmul.wordpress.com/2014/02/12/mul-3-2-5-commercially-available-now/ (visited on 07/20/2016).

[23] Amin Tootoonchian and Yashar Ganjali. "HyperFlow: A distributed control plane for OpenFlow". In: *Proceedings of the 2010 internet network management conference on Research on enterprise networking*. USENIX Association. 2010, pp. 3–3.

[24] Harmjan Treep. "Delftvisor: A network hypervisor for Openflow 1.3, Network virtualization for Openflow 1.3". MA thesis. the Netherlands: Delft University of Technology, 2017.

[25] Niels LM Van Adrichem, Christian Doerr, and Fernando A Kuipers. "Opennetmon: Network monitoring in openflow software-defined networks". In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE. 2014, pp. 1–8.

[26] Niels LM Van Adrichem, Benjamin J Van Asten, and Fernando A Kuipers. "Fast recovery in software-defined networks". In: *2014 Third European Workshop on Software Defined Networks*. IEEE. 2014, pp. 61–66.

[27] Minlan Yu et al. "Scalable flow-based networking with DIFANE". In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 351–362.

[28] Yuanhao Zhou et al. "A Load Balancing Strategy of SDN Controller Based on Distributed Decision". In: *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*. IEEE. 2014, pp. 851–856.

Appendix A

Workaround for ACRoDiF and OpenMUL Testbed

There is one flaw on OpenMUL that require workaround solution, OpenMUL does not reply to echo request (`OFPT_ECHO_REQUEST`) message. Based on the OpenFlow 1.3 specification [6], both echo request and echo reply (`OFPT_ECHO_REPLY`) messages are categorized as symmetric messages, which means echo request message can be sent from either data plane or control plane and must return an echo reply message. Unfortunately OpenMUL only capable of sending echo request message, but not capable replying to echo request message. This is a problem for the ACRoDiF to detect whether the controller is still available or not. ACRoDiF relies on echo request/reply message to verify the liveness of OpenFlow channel to the control plane. It will send echo request message and wait for echo reply from the control plane. When ACRoDiF does not receive echo reply from the control plane after three echo request interval, it will declare echo request timeout, it means controller failure.

Since OpenMUL does not reply to echo request message, iptables cannot be used to simulate link failure to the control plane. Instead of blocking the link by using iptables, the controller's process in the operating system is killed. When the process killed, it will send `FIN` packet to close the socket connection. Upon receiving `FIN` packet from the controller, ACRoDiF will close the OpenFlow channel to that controller, thus activating failover mechanism then switching to the backup controller, bakcup controller become primary controller. After failover, OpenFlow messages from the switch are forwarded to the new primary controller. This workaround still works just like how ACRODiF normally works, the only difference is how ACRoDiF detects controller failure.