A Scalable and Fault Tolerant OpenFlow Controller

A report submitted in fulfilment
of the requirements for the degree

of

Masters in Computer Science

at

The University of Waikato

by

Karthik Neelakanta Sharma

The supervisor for this thesis is Dr Richard Nelson

2015

Abstract

OpenFlow provides a protocol for updating flow tables in switches. Most current OpenFlow deployments rely on a single controller to control all switches. However, as the number and size of production networks deploying OpenFlow increases, relying on a single controller for the entire network might not be feasible for several reasons. First, the amount of control traffic destined towards the centralized controller grows with the number of switches. Second, since the system is bound by the processing power of the controller, low setup times can grow significantly as demand grows with the size of the network. Finally single controller architecture has zero fault tolerance which makes it non-ideal for large enterprise level deployments. In this thesis, the existing work that has been done to build scalable and fault tolerant controllers has been explored. After learning and understanding different systems we have built our own database backed scalable and fault tolerant controller. The database that was used for this purpose is Titan Graph database, with a Cassandra backend. A custom version of a simple switch application was built to demonstrate the scalability and fault tolerance of our architecture. Some performance comparisons between our version of simple switch and the original version were also carried out. Finally in this thesis some future enhancements that we would like to implement are outlined.

Acknowledgements

I would like to take the time to thank the following people for their guidance and support during this project: Richard Nelson, for his supervision of this project, Umesh Krishnaswamy and Jonathan Hart, for their continued patience with my enquiries about ONOS, James Thornton and Stephen Mallette for answering my questions on Bulbs, Stephen Donnelly and Stuart Wilson for supporting me during this thesis and last but not least to my wife Kanchana for her unconditional support during my thesis.

Contents

List of Figures

List of Tables

Chapter 1 Software Defined Networking

1.1 Introduction

The simplicity in the internet’s design has led to a tremendous innovation in the internet, but the network itself remains quite hard to change and surprisingly difficult to manage. The root cause of this problem in a traditional network lies primarily in the complicated control plane running on top of all switches and routers throughout the network. These networking devices are manufactured by different network vendors and used proprietary protocols to control the data plane. In these devices, proprietary firmware on the control plane of the switch determines where packets of data are forwarded by the data plane. Distributed optimization of network control was inherently difficult since control plane was a part of individual network devices.

Software Defined Networking (SDN) is a relatively new approach to computer networking which evolved from some preliminary research and work done at UC Berkeley and Stanford University in 2005. SDN introduces a layer of software between bare metal network components and the network administrators who configure and set them. This software layer gives network administrators an opportunity to make their network device adjustments through a software interface instead of having to manually configure hardware and actually physically access network devices giving them a very good control over their networks. This is achieved by decoupling the system that makes decisions about where traffic is sent (the control plane) from the underlying systems that forwards traffic to the selected destination (the data plane). SDN adheres to open standards and is vendor-neutral, i.e. it can theoretically operate with any vendor's network hardware. This gives organizations the ability to avoid vendor lock-in for a host of network products.

Most current SDN deployments however currently rely on a single SDN controller. However, as the number and size of production networks deploying OpenFlow increases, relying on a single controller for the entire network might not be feasible for several reasons. First, the amount of control traffic destined towards the centralized controller grows with the number of switches. Second, since the system is bound by the processing power of that single controller, low setup times can grow significantly as demand grows with the size of the network. This clearly introduces a serious limitation on the scalability and fault tolerance of the controller. We aspire to design and build an open source, database backed scalable and fault tolerant OpenFlow controller. Our controller is intended to be used for rapid prototyping and research environments.

1.2 Software Defined Networking

Traditional networking devices such as switches and routers can be divided into three different logical planes. They are the data plane, control plane and management plane. Data plane refers to the hardware part where the packet forwarding takes place, and control plane refers to the part that implements the routing protocol. Typically in networking devices, control plane is implemented in proprietary firmware developed by equipment vendors. Management plane is used for network monitoring and controlling purposes.

Software Defined Networking is a new and emerging network architecture which separates data and control plane in a networking device, and makes the control plane independent and programmable. The separation of the control plane from the data plane abstracts the network infrastructure from the applications and treats network as a virtual entity. In this thesis the focus will be mainly on the data and the control plane of networking devices. Figure 1.1 shows the data and control plane of a traditional networking device.

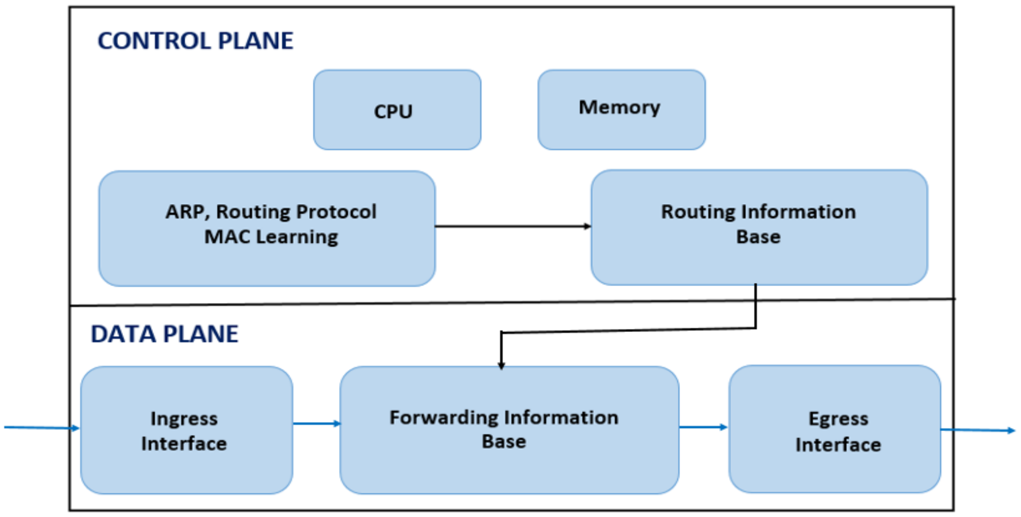


Figure 1.1 Data and Control plane in traditional networking hardware

1.3 Architecture of Software Defined Networks

A Software Defined Network can be logically divided into three different layers. The infrastructure layer refers to the actual forwarding hardware. This layer consists of network devices such as Layer 2 switches in a LAN centric environment. The control layer, also known as the SDN controller is where the real intelligence of a Software Defined Network is situated. This layer implements the basic network services which can be used by various networking applications in the application layer. The switches that are located in the infrastructure layer are not traditional network switches. These switches need to support some mechanism whereby the control layer can talk to and program the switches in the infrastructure layer.

The Figure 1.2 depicts the architecture of a Software Defined Network.

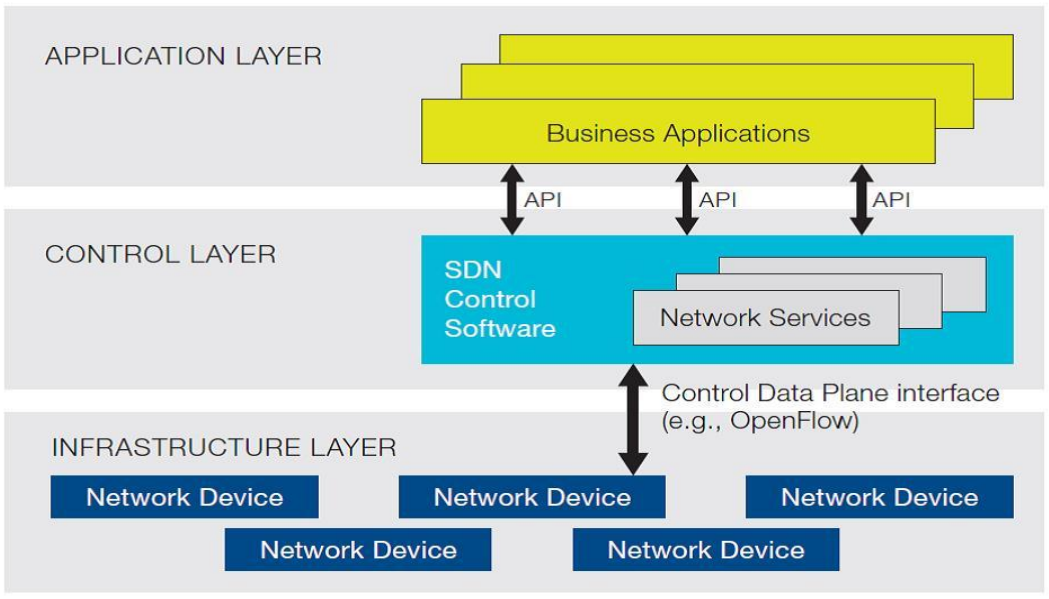


Figure 1.2 Architecture of a Software Defined Network [1]

In a Software Defined Network (SDN) architecture, southbound application program interfaces (APIs) are used to communicate between the SDN Controller and the switches of the network. They can be open or proprietary. The most popular and well known southbound interface is the OpenFlow protocol. The northbound application programming interface (API) on a SDN control layer enables application layer to program the network and request services from it. The Northbound API is evolving rapidly but currently there are no standards for it. Each OpenFlow controller provides their own set of interfaces.

The control plane can have one or more control nodes. The nodes in control layer are called as SDN controllers (commonly OpenFlow controllers) and they send routing and switching information to the data plane nodes that they control. After receiving the information from controller, the networking devices update their forwarding tables according to the information that they receive from the SDN controller.

1.4 The OpenFlow protocol

As mentioned in the above section OpenFlow protocol is the most popular and widely accepted protocol for the southbound Application Programming Interface. OpenFlow protocol intends to provide access to the data plane of the switches. It does this by specifying a language that a switch can recognize and use to update its forwarding tables. OpenFlow is a language for generically defining characteristics of a particular flow of traffic and a set of actions to be executed when the switch encounters packets that matches such characteristics.

The actual mechanisms used to program flows into switch hardware vary greatly depending on the vendor of the particular hardware. Instead, OpenFlow provides a way to describe desired flow state within an agent running locally on the forwarding device. All switches that are OpenFlow enabled will have the OpenFlow agent that will interpret the OpenFlow commands. The OpenFlow specification also includes ways for the OpenFlow controller, which is remote and located in the control plane to make modifications to this information. The OpenFlow agent, armed with the flow information programmed into it by a controller, acts like the control plane on traditional switches. The only difference is that it does not have to run routing protocols, or make decisions locally. All the decisions are made by the remote OpenFlow controller and the OpenFlow agent stores these OpenFlow entries, and pushes them into the flow tables on the hardware device. The following Figure 1.3 shows an idealized OpenFlow switch where the flow table is controlled by a remote OpenFlow controller. [2]

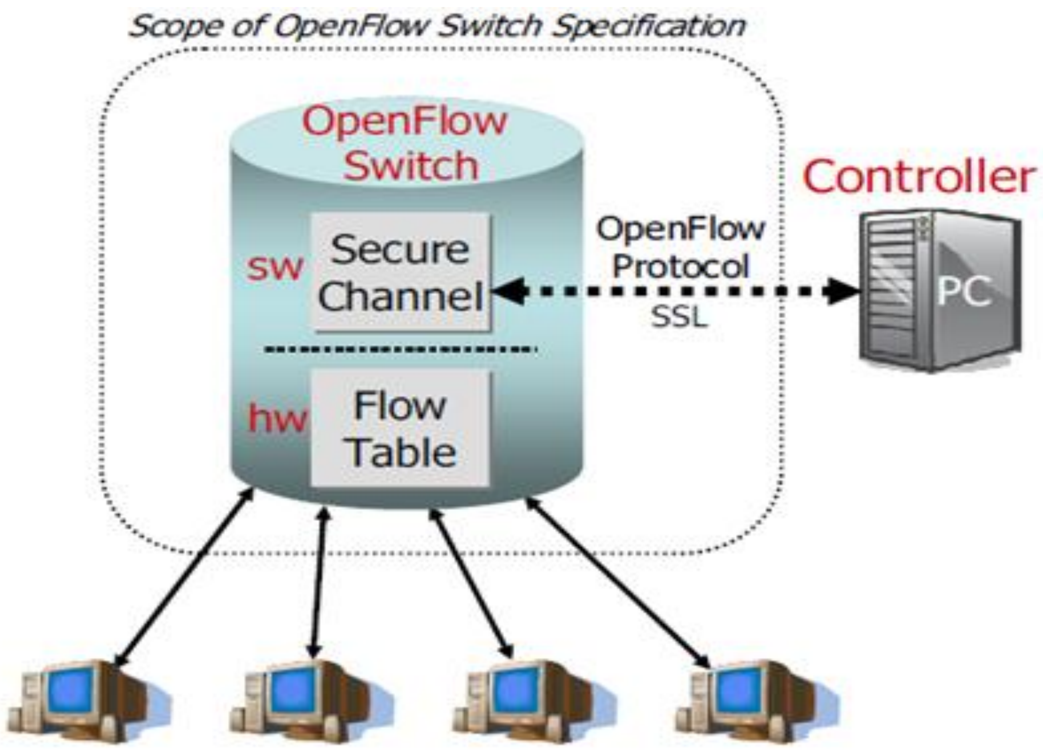


Figure 1.3 Idealized OpenFlow Switch [2]

The OpenFlow controller has a unified view of the whole network. It runs the routing or switching protocols to collect the relevant routing or switching information. There are two different ways in which the OpenFlow controller can program the switches in the network.

The first packet of each new flow can trigger the controller to insert flow entries down to the switches and the switch makes efficient use of flow table where every flow needs small additional flow setup time. The other approach is that the flow tables in switch can be prepopulated by the OpenFlow controller ahead of time for all traffic matches that could come into the switch. By predefining all the flows and actions ahead of time in the switch flow tables, the packets can be forwarded at line rate as this approach does not require any additional flow setup time per individual flow. However this approach often requires aggregated or wildcard rules.

1.4.1 Types of OpenFlow messages

OpenFlow protocol supports three types of messages that are exchanged between the switch and the OpenFlow controller: controller-to-switch, asynchronous and symmetric messages that are discussed briefly here [3].

The controller-to-switch messages are initiated by the controller and may not always require a response from the switch. These messages are used to configure the switch, manage the switch's flow table and acquire information about the flow table state or the capabilities supported by the switch at any given time. Examples of this type of messages are Features, Config, Modify-State, Read-State, Packet-Out, Barrier, Role-Request and Asynchronous-Configuration.

The asynchronous messages are sent without solicitation from the switch to the controller and denote a change in the switch or network state. One of the most important asynchronous messages is the packet-in message. The packet-in message is a way for the switch to send a captured packet to the controller. There are two reasons why this might happen; there could be an explicit action as a result of a match asking for this behaviour, or from a miss in the match tables, or a ttl error. The OpenFlow controller can then examine the packet headers in the packet-in messages and can program the switches to take appropriate action. Some other asynchronous messages include flow removed and port status.

Symmetric messages are messages that are sent without solicitation in either direction between the controller and the switches. These messages are typically used to assist or diagnose problems in the connection between the switch and the controller. Some examples of symmetric messages are Hello, Echo and Experimenter messages.

1.4.2 Connection establishment between switch and the OpenFlow controller

A switch that is configured in OpenFlow mode typically initiates the connection to the OpenFlow controller. It does this by sending TCP sync messages to the OpenFlow controller IP addresses (which is configured into the switch on start-up) to its default port 6633. After TCP handshake is completed between the switch and the controller the connection is two way. Following the TCP handshake process a set of messages are exchanged between the controller and the switch such as Features Request, Features Reply and Set Config.

Even though the normal mode is for the OpenFlow switches to initiate the TCP connection to OpenFlow controller, it is common for some switches to implement what is referred to as passive ports where the switch itself would listen to a connection initiated from another device. This feature is typically used for troubleshooting. There are some command line tools that are intended to be used this way (for e.g. dpctl). From a computer with dpctl installed, a connection to the OpenFlow switch can be initiated. An OpenFlow switch with passive ports enabled will accept such a connection from the computer. This command line utility allows the user to dump the contents of the OpenFlow table on to the switch. It can also be used to interrogate the switch in terms of its capabilities. It can even be used to install entries in the OpenFlow table on the switch.

1.4.3 The OpenFlow table

The OpenFlow table is a data-structure that resides on the high speed data plane of the OpenFlow switch. Its contents decide the forwarding and packet handling behaviour of the OpenFlow switch. OpenFlow table contains one or more flow entries. Each flow entry has a set of components which include Header Fields, Actions, Priorities, Counters and Timers. Header Fields are used to identify which packets to perform the actions on. These consist of the ingress port and packet headers. Actions are the set of actions that are applied on the matched packets. The priority field matches the precedence of the flow entry. The counters are updated when the packets are matched. Timeouts refer to the maximum amount of time or idle time before the flow is expired by the switch. A flow table entry is identified by its match fields and priority: the match fields and priority taken together identify a unique flow entry in the flow table. The flow entry that wildcards all fields (all fields omitted) and has priority equal to 0 is called the table-miss flow entry [3]. There are two flow expiry mechanism supported by OpenFlow. This can be done either at the request of the controller or by the flow expiry mechanism of the OpenFlow switch. The switch flow expiry mechanism is based on the state and configuration of the flow entries. Each flow entry has an idle timeout and a hard timeout associated with it. Idle timeout is the inactivity timeout. If there are no packets matching the flow entry for this timeout period the flow gets deleted. In case hard timeout is set, the flow gets deleted after the timeout period even if there are packets matching the flow entry during that period.

Chapter 2 Related Work

2.1 Introduction

This chapter explores the previous work that this project is based upon. It discusses in detail the various approaches that have been explored before. As explained in the previous chapter, OpenFlow provides a protocol for updating flow tables in switches. Most of the current deployments rely on a single controller to control all the switches in the topology. However, as the number and size of production networks deploying OpenFlow increases, relying on a single controller for the entire network might not be feasible for several reasons. First, the amount of control traffic destined towards the centralized controller grows with the number of switches. Second, since the system is bound by the processing power of the OpenFlow controller, low setup times can grow significantly as demand grows with the size of the network. Finally single controller architecture has zero fault tolerance which makes it unsuitable for large enterprise level deployments. There has been several proposals in the academic literature to achieve redundant or distributed controllers some of which are quite complex. We will have a detailed look into some of these in the following sections. The focus of the following sections will be on the architecture of the system with a focus on how it implements scalability and fault tolerance.

2.2 Distributed OpenFlow controller using co-ordination framework

Volkan Yazıcı et.al in [4] discusses the need for a distributed OpenFlow controller in real world data centres. The real world data centres need to handle around a 150 million flows per second. But todays OpenFlow controllers are known to handle around 6 million flows per second using a high end dedicated server with 4 cores. This clearly indicates that either distributed controller architecture or an appropriate main frame computer with sufficiently many cores is required to achieve the required scalability and reliability. [4]

The authors [4] have decided to approach the problem using distributed cluster architecture for OpenFlow controllers. The flow on impact of the distributed cluster design is that it is inherently scalable and fault tolerant. We can add more OpenFlow controller nodes to the cluster when the requirement arises and the presence of multiple nodes offers more reliability and fault tolerance than using a single main-frame computer.

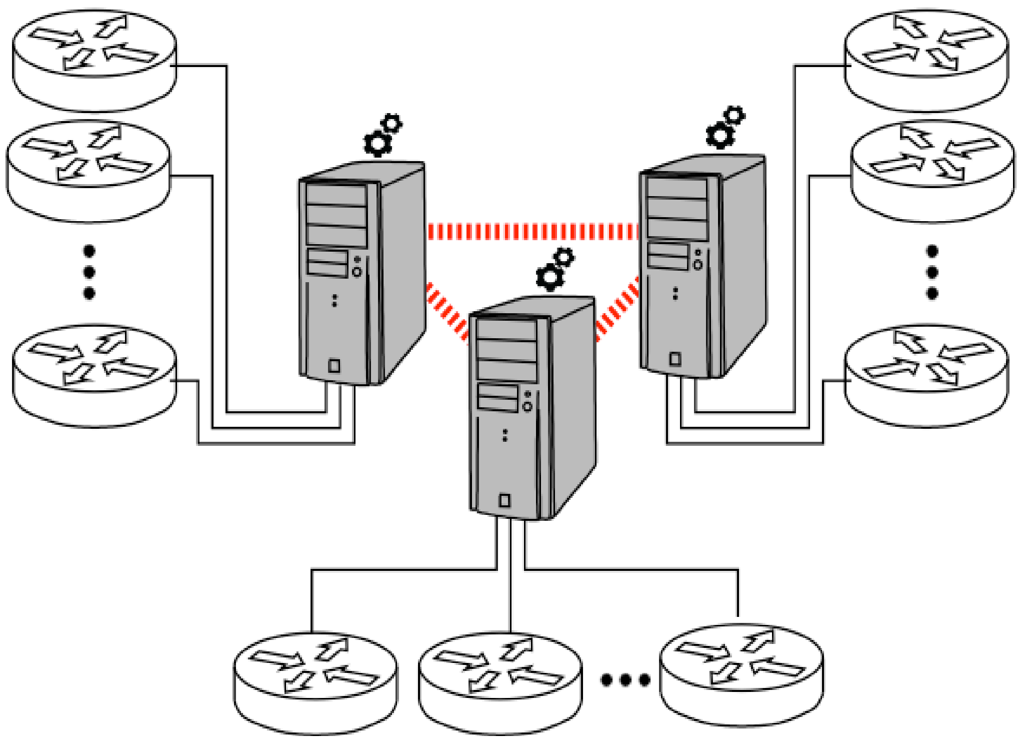


Figure 2.1 Distributed OpenFlow Controller Architecture [4]

From Figure 2.1, it would become clear that some kind of coordination framework is required to enable the coordination between multiple instances of the OpenFlow controllers in the cluster. They have decided to use JGroups [5] membership notifications and infrastructure to enable the clusters in the distributed framework to communicate with each other. JGroups is a mature, robust and flexible group communication library.

The multiple controllers in the distributed cluster elect a master controller among themselves. This master controller maintains the global controller to switch mapping of the network. This master node is periodically monitored by the other node in the cluster. If the master node is found to be inaccessible it is immediately replaced by one of the other nodes in the cluster. This mechanism avoids the exposure of a single point of failure.

The cluster of controller presents the applications running on top of them, a view as if it were a single centralized controller. In other words the switches and the applications are unaware of the switch to controller mapping in the network. This makes it easier for the switch to controller assignments and reassignments to happen seamlessly when new controller are added or for proper load balancing. This feature makes the above architecture scalable.

2.3 HyperFlow – A distributed control plane for OpenFlow

Amin Tootoonchian and Yashar Ganjali [6] discuss HyperFlow a distributed event based control plane for OpenFlow which allows the network operators to deploy any number of OpenFlow controllers in their networks.

HyperFlow keeps network control logically centralized and provides scalability to the network. [6]. All controllers in the distributed control plane share the same consistent network wide view and serve the switches connected to it locally without needing to contact other controllers. This helps to minimize flow setup times.

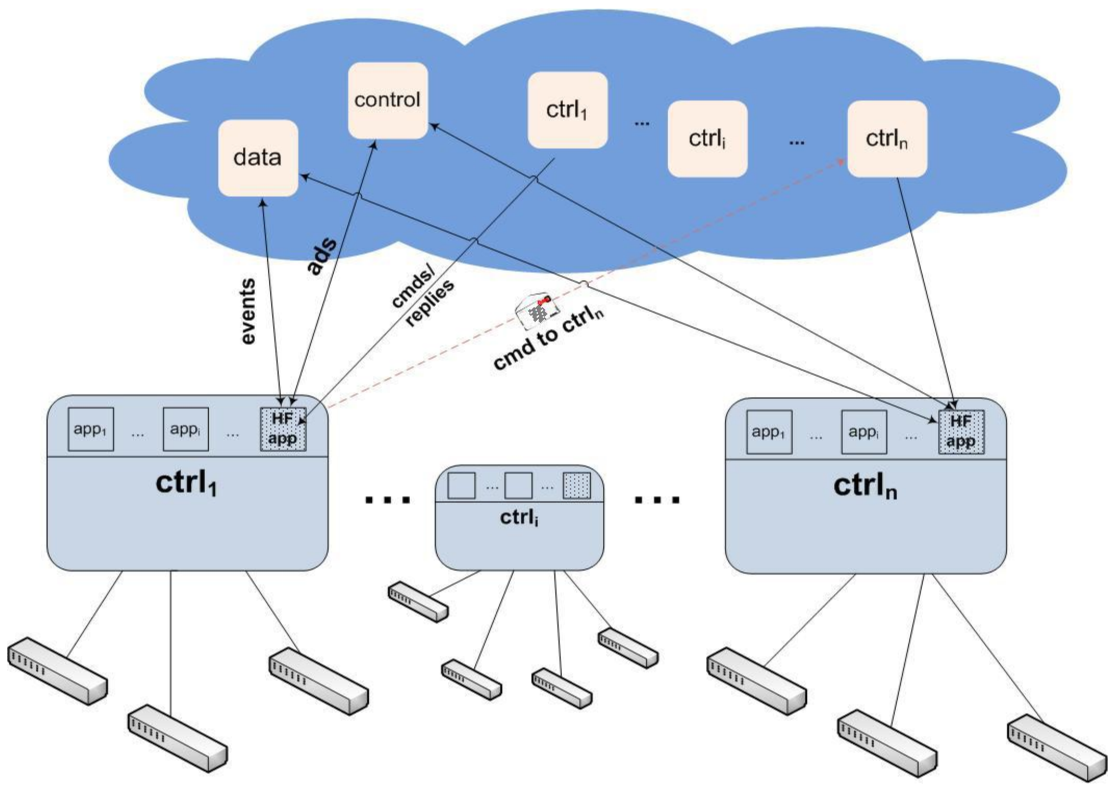


Figure 2.2 High-level Overview of HyperFlow

A HyperFlow based network is composed of OpenFlow switches, NOX controllers each running an instance of the HyperFlow controller application and an event propagation system for cross-controller communication. All the controllers have a consistent network-wide view and run as if they are controlling the whole network. All instances of the NOX controller run the exact same controller software and set of applications. Each switch is connected to the closest controller in its proximity. If that controller fails, then the switches that were connected to that controller must be reconfigured to connect to a nearby controller. Each controller directly manages the switches connected to it. Each controller can also indirectly query as well as program the switches connected to other controllers by communicating with other controllers. Figure 2.2 illustrates the high-level view of the system.

In Figure 2.2, each controller runs NOX with the HyperFlow application atop, subscribes to the control, data, and its own channel in the publish/subscribe system (depicted with a cloud). Events are published to the data channel and periodic controller advertisements are sent to the control channel. Controllers directly publish the commands targeted to a controller to its channel. Replies to the commands are published in the source controller. [6]

HyperFlow achieves a network-wide view for all the controller instances in the cluster. This is achieved because the HyperFlow controller application running on each instance of the OpenFlow controller selectively publishes events that change the state of the system through a publish subscribe mechanism. The other controllers in the system replay the published events to reconstruct the state of the network. Since these state synchronisations is done by the HyperFlow controller application, individual applications such as routing and load balancing does not need to do state synchronisation.

2.4 Onix: A Distributed Control Platform for Large-scale Production Networks

Teemu Koponen et.al [7] discusses Onix, a distributed control platform designed for large scale production networks. Onix provides useful and general

API for network control that allows scalable applications development. Onix's API contains a data model that represents the network infrastructure. In the Onix platform the controller consists of Network Information Base, Switch import/export and Distribution import/export. Network Information Base (NIB) is a data structure that tracks the network state. NIB stores the graph of all network entities within the topology. NIB stores a collection of network entities as key value pairs and is identified by a flat, 128-bit, global identifier. The NIB is at the heart of the Onix control model and basis for its distribution model. NIB is decentralized and distributed over several Onix nodes. The controller programmer manages the network by reading and writing to the NIB. If a change is made to a local NIB instance on one of the Onix nodes, then these modifications are propagated to other NIB on other Onix nodes. The switch import/export component interprets the instructions from the Onix node and configures the switches. The distribution import/export component makes the different NIB's consistent with each other.

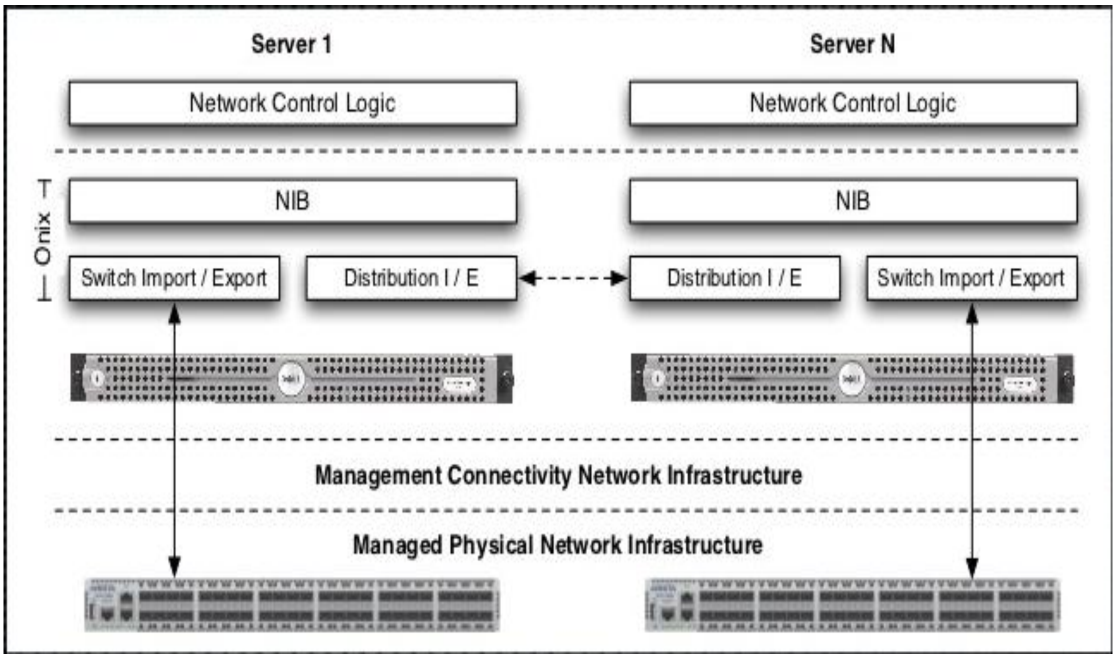


Figure 2.3 The components in an Onix Controlled Network [7]

Figure 2.3 shows four components in an Onix controlled network. They are managed physical infrastructure, connectivity infrastructure, Onix, and the control logic implemented by the management application. This figure depicts two Onix instances coordinating and sharing (via the dashed-arrow) their views of the underlying network state, and offering the control logic a read/write interface to that state. [7]

Onix developers have proposed some strategies to make the architecture more scalable. The easiest approach seems to be partitioning the workload of the controller. Since NIB is decentralized and distributed over multiple Onix nodes, each Onix instances might take different tasks. Another approach is for the application to aggregate a topology as a single logical node and use that as the unit of event dissemination between instances. For example, the topology can be divided into logical areas and each area can be managed by a distinct Onix instance. The last approach is consistency and durability. To distribute the NIB with consistency, the authors suggest two methods which developers can choose. Onix offers a replicated transactional database and, for volatile state that is more tolerant of inconsistencies, a memory-based one-hop DHT.

To handle reliability issues and to make the architecture more fault tolerant, Onix suggests some methods. To deal with link failure issues, Onix provides API’s. If one of the Onix instances fail the controller program can make the running Onix instances to take over the responsibilities of the failed instance. Or a more reliable option would be to have each network element be managed by more than one Onix instance.

2.5 ONOS – A database backed distributed OpenFlow controller

Pankaj Berde et.al [8] discuss ONOS which uses a database backed approach to build a system that run across multiple servers to support scalability and fault tolerance. ONOS maintains a global network view to manage and share network state across ONOS servers in a cluster. Each of the network elements such as switch port and host is modelled using a graph database (Titan Graph database) which is backed by Cassandra key-value store to make it distributed.

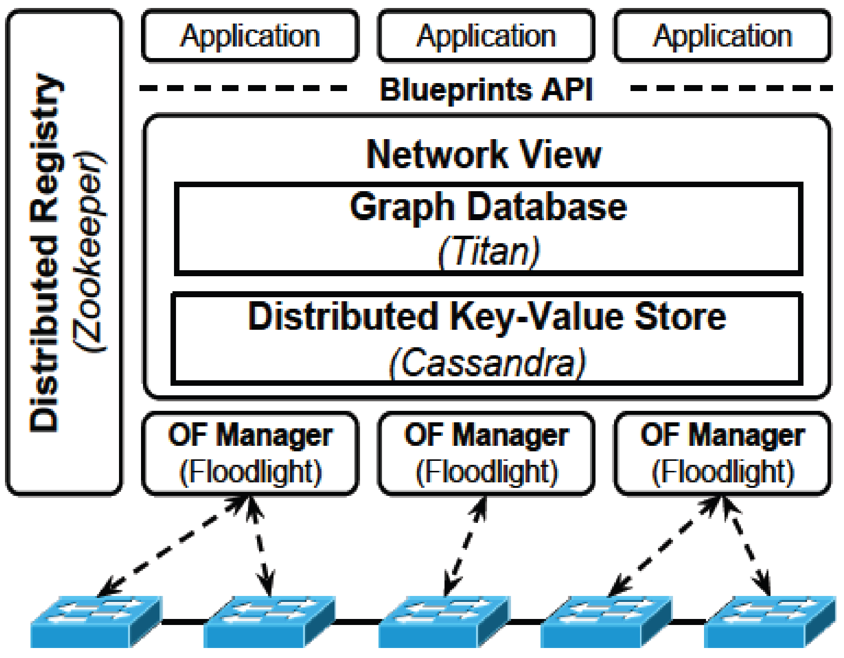


Figure 2.4 ONOS Architecture Diagram [8]

As mentioned above the global network view is implemented on the Titan Graph database. As ONOS runs on multiple servers, each instance of ONOS is the master controller for a subset of switches in the topology. As the data plane capacity grows or demand on the control plane increases, additional instances can be added to the ONOS cluster to distribute the control plane workload.

The distributed architecture of ONOS allows the system to continue operating when an ONOS instance fails by redistributing work to other remaining instances. ONOS also allows us to have a single running controller with a redundant controller waiting to take over in case the primary controller fails. A switch has the option to connect to multiple ONOS instances, but only one instance is chosen as the master controller for each switch. The master instance alone is responsible for controlling and programming the switch. When an ONOS instance fails, the remaining instances elect a new master for each of the switches that were previously controlled by the failed instance. They use a consensus based leader election to make sure that at most one ONOS instance is in charge of each switch. They have used Zookeeper to manage switch-to-controller mastership, including detecting and reacting to instance failure.

Chapter 3 Our Proposal

3.1 Introduction

We propose to investigate the feasibility of using a database backed controller architecture to build redundant and distributed OpenFlow controllers. In this architecture the controllers would be essentially stateless with the state information being stored in a database server. Even though this makes the database server the single point of failure, it is easier to replicate databases across multiple servers and then to distribute the load using load balancers keeping the controller stateless. This approach localises the decision making process to a single point, i.e. database server, allowing multiple controllers to control the whole network or parts thereof. The different OpenFlow controllers controlling the network will be connected to the same database server. The multiple OpenFlow controllers controlling different parts of the network can communicate among each other by means of a publish subscribe framework.

3.2 Choice of database

ONOS has used a graph database for representing the network elements in the topology. It was decided to adopt this idea, since graph databases, which primarily represents information in terms of a set of vertices and indicates the relationship among them by the edges connecting the vertices, lends itself very beautifully to represent different network elements in a network topology. The graph database that was chosen to be used was Titan Graph DB. Titan is a distributed, real-time, scalable transactional graph database developed by Aurelius. Titan leverages various storage backends for persistence including

Cassandra, hbase and Hazelcastcache. In this project Cassandra was chosen as the database backend. If Titan was completely removed, then the data would have to be written directly to Cassandra's key-value store rather than to Titan's graph data model. A schema would have to be defined for the applications data in Cassandra's data model. But Titan Graph DB provides a data model that fits well for the network-oriented data that the application uses. Hence the application running on the OpenFlow controller will not store data in Cassandra directly; all database access goes through Titan.

Titan exposes a graph data model, where everything is either a vertex or an edge. A graph schema for its data needs to be defined, and the data has to be written to Titan. Titan stores this graph model in Cassandra's key-value store underneath. The details of how Titan stores data in the key-value store are abstracted completely from application running on the OpenFlow controller.

Since Titan Graph DB backed by Cassandra is implemented in Java and run as a Java Virtual Machine, some mechanism is required to allow the application, written in python, to store and retrieve data from Titan. Blueprints API is an open-source community developed Java interface for graph databases that expose a property graph data model.

A property graph is a graph with the following elements. It has a set of vertices and a set of edges. Each vertex in the set has a unique identifier, a set of outgoing and incoming edges. Each vertex also has a collection of properties defined by a map from key to value. Each edge in the edge set has a unique id, an outgoing tail vertex, an incoming head vertex, and a label that denotes the type of relationship between the two vertices. Each edge also has a collection of properties defined by a map from key to value. Titan natively implements Blueprints API which means blueprints is the core interface for Titan.

Bulbs is an open-source Python persistence framework for graph databases. It is like an ORM for graphs, but instead of SQL, it uses the graph-traversal language Gremlin to query the database. Groovy is a programming language and Gremlin Groovy is a graph traversal language built on top of it using groovy's meta-programming facilities. Bulbs can connect to several graph-database servers, including Neo4j Server and Rexster server. Rexster server that runs inside Titan Graph DB hosts Blueprints implementations and exposes elements of that API (and Gremlin) over REST using JSON. The graph model programmed into the Titan Graph DB will be written to Cassandra backend.

Figure 3.1 shows how the python application can communicate with the Titan Graph database with Cassandra backend.

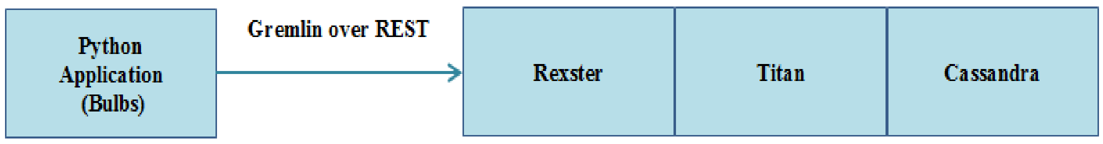
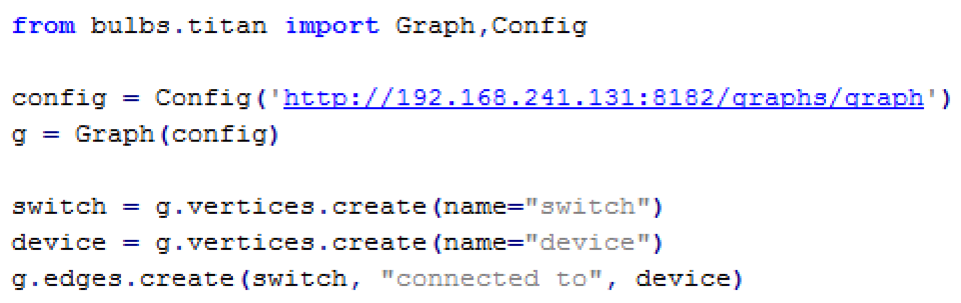


Figure 3.1 Python application communicating with Titan Graph database

In the Figure 3.1, Cassandra is used as the underlying data storage. Titan provides graph database functionality on top of Cassandra. Rexster exposes the Titan graph to remote applications via the network. All three of these systems run within the same JVM so calls between them are performant.



```
from bulbs.titan import Graph,Config

config = Config('http://192.168.241.131:8182/graphs/graph')
g = Graph(config)

switch = g.vertices.create(name="switch")
device = g.vertices.create(name="device")
g.edges.create(switch, "connected to", device)
```

Figure 3.2 Creating vertices and edges on Titan DB from Bulbs

3.3 Choice of OpenFlow controller

The OpenFlow controller chosen for this project is the Ryu OpenFlow controller that is written in Python. When different OpenFlow controllers were considered, the decision to go with an OpenFlow controller implemented in Python was made, since it would be easier to prototype the application. The two such OpenFlow controllers considered were Pox and Ryu. The reason for choosing Ryu over Pox was that, at that point Ryu already had implemented OpenFlow 1.3. Even though this thesis did not use any features of OpenFlow 1.3 per se, it would be useful in case it was decided to extend this project to add more features. Also the documentation available for Ryu was very good and so was the support via mailing lists.

3.4 Choice of publish subscribe framework

As mentioned in the above sections the communication between multiple instances of Ryu OpenFlow controllers is facilitated by means of a publish subscribe framework. The following three publish subscribe libraries ActiveMQ, RabbitMQ and ZeroMQ were considered. RabbitMQ is one of the leading implementation of the AMQP protocol [10]. Therefore, it implements broker architecture, meaning that messages are queued on a central node before being sent to clients. This approach makes RabbitMQ very easy to use and deploy. However, it also makes it less scalable and slower because the central node adds latency and message envelopes are quite big. ActiveMQ can be deployed with both broker and P2P topologies. Like RabbitMQ, it is easier to implement advanced scenarios but usually at the cost of raw performance.

ZeroMQ is a lightweight message orientated socket implementation. It is also suitable for inter-process asynchronous programming. It is faster than TCP. It carries messages across inproc, IPC, TCP and multicast. It can connect N-to-N via fanout, publish subscribe, pipeline and request reply. We use publish subscribe mechanism in ZeroMQ for communicating between multiple instances of OpenFlow controllers. ZeroMQ is a very lightweight brokerless messaging system specially designed for high throughput/low latency scenarios. Advanced features have to be implemented by the user by combining different features such as sockets and devices. It certainly looks like ZeroMQ would be ideal candidate for our needs.

Around October last year, while pondering over the best mechanism to enable communication between multiple Ryu instances, we came across a study conducted by Adina Mihailescu where he compares and benchmarks different message brokers.

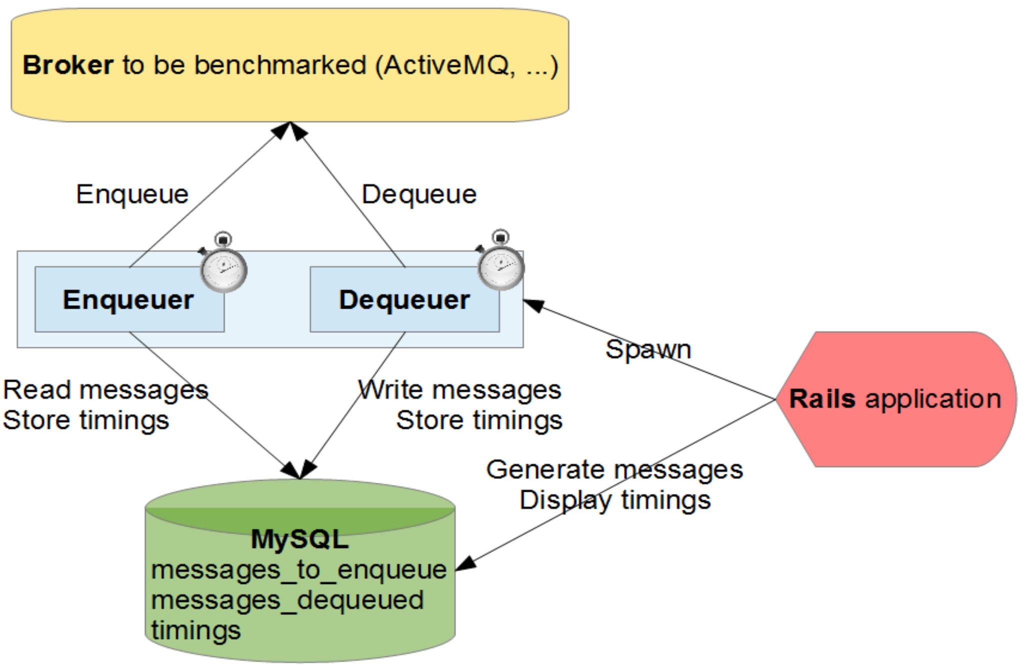


Figure 3.3 Setup to benchmark different message brokers [10]

The setup used for test is described in the above diagram. Since the different brokers were using different protocols, they have built a little Rails application piloting a binary that was able to enqueue/dequeue items taken from a MySQL database. The test done on different brokers with multiple message sizes and the results are published in [10]. The message brokers that got tested and benchmarked were ActiveMQ, RabbitMQ, HornetQ, Appollo1, QPID and ZeroMQ. Since ZeroMQ did not have a message broker, an in-memory broker without persistence was implemented for ZeroMQ. The details of the study are available at [10]. The conclusion of the study was that ZeroMQ outperforms all other messaging systems. Unless there is a need for complex broker features, ZeroMQ is a perfect message dispatcher among processes. This study confirmed that ZeroMQ has low latency and high throughput advantage over other messaging systems. Because of this ZeroMQ has been used for communication between multiple Ryu instances.

3.5 Choice of network simulator

To test the application, some mechanism is required to simulate the OpenFlow switches and hosts. Mininet is a network emulator. It runs a collection of end-hosts, switches, routers, and links on a single Linux kernel. It uses lightweight virtualization to make a single system look like a complete network, running the same kernel, system, and user code. A Mininet host behaves just like a real machine and allows users to ssh into it. The user can start up sshd and bridge the network to the host and run arbitrary programs, including anything that is installed on the underlying Linux system. The programs that the users run can send packets through what seems like a real ethernet interface, with a given link speed and delay. Packets get processed by what looks like a real ethernet switch, router, or middle box, with a given amount of queueing. It uses openvswitch to simulate the switches that support OpenFlow. In fact Mininet is the de-facto standard when it comes to OpenFlow network simulators. Mininet was used to simulate our network.

3.6 Proposed Architecture

The following Figure 3.4 diagrammatically represents our proposed architecture. It shows three instances of Ryu OpenFlow controller each controlling three network switches. The network wide topology as well as other associated information will be stored in the database server running Titan Graph database that is backed by Cassandra key-value store.

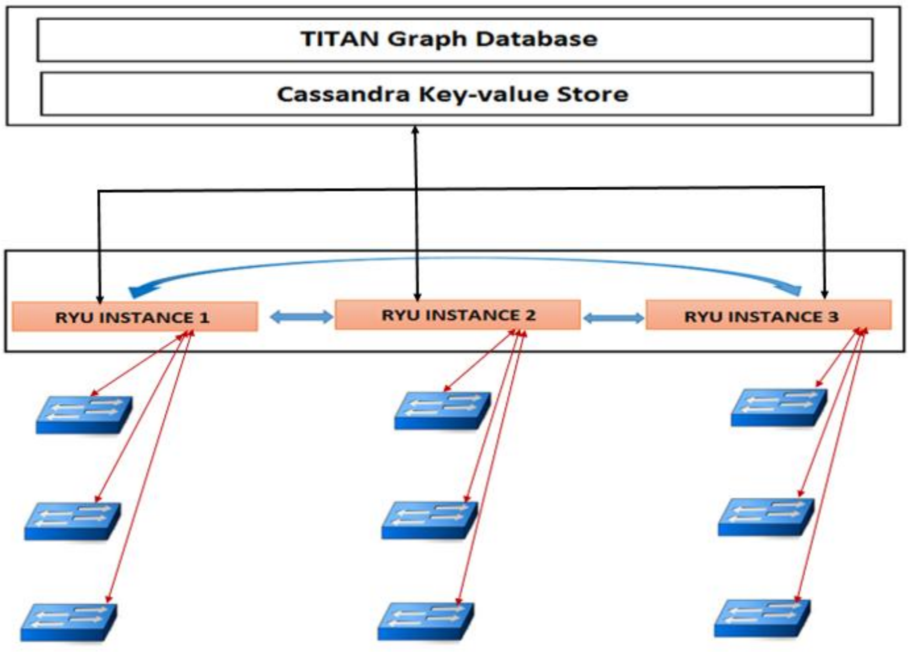


Figure 3.4 Proposed Architecture

The OpenFlow controller instances themselves can talk to each other using the ZeroMQ message queues. The messages queues are diagrammatically represented using blue double directional arrows.

Chapter 4 Demonstration of our Architecture

4.1 Introduction

To demonstrate our scalable fault tolerant architecture we are using an L2 switch application running on Ryu OpenFlow controller. As mentioned in the previous chapter Ryu OpenFlow controller application will store network topology data in a graph data model. We use a very basic model where each network element such as switch, port, device etc. are represented using a vertex. Edges are placed between the elements where they are related, e.g. ports have an edge to the switch that they are on and ports have edges between them if they are connected by a link in the network.

One thing to note is that the network is eventually consistent anyway. Even if Ryu application tried to implement a strongly consistent data store, it would still be behind events that are actually happening in the network. So Ryu could think a switch is present even though in the network it has just disappeared. In this case the application has to handle what happens when the switch goes away. Using an eventually consistent data store does not really fundamentally change the nature of the data. What it means is that the application has to be aware that two different instances may have slightly different views of the data at each point in time.

4.2 Data Modelling

To efficiently model network topology and the flow information, five different types of vertices are used in our data modelling. This idea was borrowed from ONOS. The five vertices used are

1. switch
2. port
3. device
4. flow_entry
5. flow

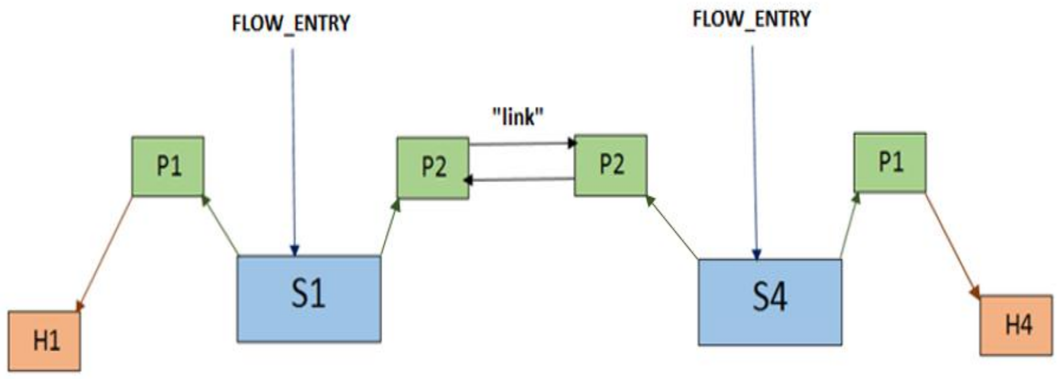


Figure 4.1 Representation of a graphical model of a part of network topology

In the above diagram all the boxes represents vertices. The edges are represented by "incoming / outgoing" arrows. The diagram shows H1 (Host 1) is connected to Port1 of S1 (Switch 1) and H4 (Host 4) is connected to Port 1 of S4 (Switch 4). Port 2 of S1 and S4 are physically connected.

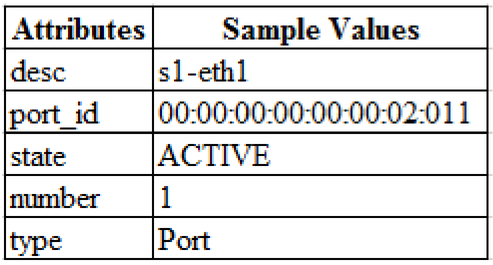
Some general rules that have been followed for data modelling are explained below. Switches will have an "outgoing" edge to all its ports. Consequently ports will have an "incoming" edge from the switch to which it belongs. Hosts/Devices will have an "incoming" edge from the switch port to which it is connected. Flow_entry will have an "outgoing" edge to the switch on which it is supposed to be installed. Consequently switches will have an “incoming” edge from the flow_entry.

In Figure 4.1, the links between the ports of different switching devices falls outside the purview of the OpenFlow protocol. That information can be obtained using Link Layer Discovery Protocol (LLDP). In the above diagram P2 on S1 will have an outgoing edge to P2 on S4. Similarly P2 on S4 will have an outgoing edge to P2 on S1.These two edges will be labelled as “link” to denote physical link between the two switching devices.

The attributes of the vertices listed above are explained in the following section. This will give an idea about how the above vertices are stored in the graph database.

4.2.1 Port

The port vertex is created for each port/interface on the switches. The key that is used to uniquely identify a port is a combination of the data path identifier of the switch with the port number appended to it. This will help to keep the port identifier unique across the network. In addition to the port id, the port number and state of the port are also stored .This information is obtained from OpenFlow Port Status message. We also store a human readable descriptive field (e.g. s1-eth1) that helps to uniquely identify a port in the topology. This information can be used for a human readable output if the need arises. We also use a descriptive field called **‘type’** which for port is **‘Port’.** This will help to differentiate a port vertex from other types of vertices. The following table shows an example port vertex with sample values.

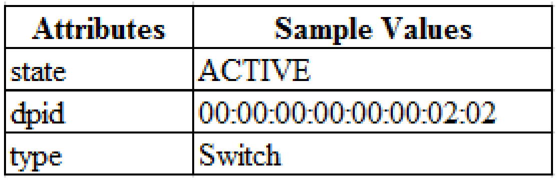


Attributes	Sample Values
desc	s1-eth1
port_id	00:00:00:00:00:00:02:011
state	ACTIVE
number	1
type	Port

Table 4.1 Port vertex

4.2.2 Switch

The switch vertex as the name implies is created for each switch in the topology. The key that is used to identify a switch is the data path identifier (dpid) of the switch. We also store the state of the switch and the ‘type’ field (in this case ‘Switch’) to identify switch vertices from other types of vertices. Below table shows an example switch vertex with sample values.



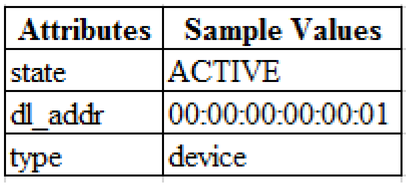
Attributes	Sample Values
state	ACTIVE
dpid	00:00:00:00:00:00:02:02
type	Switch

Table 4.2 Switch vertex

4.2.3 Device

The device vertex as the name implies is created for each device. Device is any kind of endpoint sending packets on the network. In mininet these will be the hosts in the network. Devices are tracked based on packets observed in packet-in messages by the controller. When the controller sees a packet-in, it records the in port, source mac address and source IP address if it is an ARP packet. All this information constitutes a device.

The above three types of vertices are what can be described as fundamental vertices. These vertices mirror the network topology of the underlying network. We use two mere vertex types to store some additional information that will enable us to forward packet from one host to another host via a series of switches across the network. These vertices are created after computing the path between the host (device) vertices on the network. Below table shows an example device vertex with sample values.



Attributes	Sample Values
state	ACTIVE
dl_addr	00:00:00:00:00:01
type	device

Table 4.3 Device vertex

4.2.4 Flow

This vertex type is created after computing the path between different pairs of source and destination hosts. We store the source and destination mac addresses of the communicating hosts. We also store the source switch (i.e. the switch which is connected to the source host) and destination switch (i.e. the switch which is connected to the destination host) and the ports on those switches, source port that is connected to the source device on the source switch and destination port on the destination switch that is connected to the destination device. We also store the path summary of the path that connects the source and destination devices. As in case of other types of vertices, the type field on this type of vertices have the value ‘flow’ and can be used to distinguish this from other types of vertices. The flow vertex gives a snapshot of the communication path between source and destination hosts that communicate with each other. The following table shows an example flow vertex with sample values.

Attributes	Sample Values
matchSrcMac	00:00:00:00:00:01
matchDstMac	00:00:00:00:00:04
src_port	1
dst_port	1
type	flow
src_switch	00:00:00:00:00:00:02:01
dst_switch	00:00:00:00:00:00:02:04
data_path_summary	1/00:00:00:00:00:00:02:01/2; 2/00:00:00:00:00:00:02:04/1

Table 4.4 Flow vertex

4.2.5 Flow_entry

This type of vertex is created for every flow entry that is to be programmed into the switches. This vertex has 'flow_entry' in the type field. It stores the data path identifier (dpid) of the switch into which the flow is to be programmed and also contains actions that is a part of the OpenFlow flow modification message. In addition it stores the 'input port' and the 'action output port'. We also store the 'flow_entry_id' for each flow entry. Below table shows an example flow_entry vertex with sample values.

Attributes	Sample Values
switch_dpid	00:00:00:00:00:00:02:05
actionOuputPort	1
switch_state	FE_SWITCH_UPDATED
matchInPort	2
flow_entry_id	0x4ee30a9400000012
type	flow_entry
actions	[[type=ACTION_OUTPUTaction=[port=1 maxLen=0]];]

Table 4.5 Flow_entry vertex

4.3 Demonstrating fault tolerance and scalability

4.3.1 System setup

The basic setup used for the demonstration of scalability and fault tolerance is described here. We have two servers (virtual machines) “ryu-primary” and “ryu-secondary” running two Ryu OpenFlow controllers. Both of these OpenFlow controllers are backed by the same Titan Graph database running on another virtual machine. The machines ryu-primary and ryu-secondary have IP addresses 192.169.10.1 and 192.168.10.2 respectively. As mentioned above mininet is used to simulate the network topology. The mininet runs on the virtual machine “ryu-primary”. The topology used for demonstration contains six switches and six host devices. The switches are numbered sequentially as s1, s2, s3, s4, s5 and s6. The hosts connected to those switches are similarly numbered as h1, h2, h3, h4, h5 and h6. The hosts have IP addresses assigned from 10.0.0.0/24 subnet with the last octet representing their host number. For e.g. h1 will have an IP address of 10.0.0.1, h2 will have an IP address of 10.0.0.2 and so on. Network topology used is represented below.

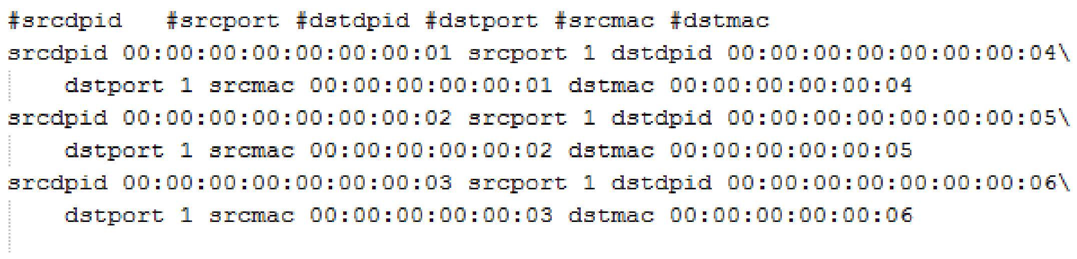
```
h1 --- (eth1) s1 (eth2) --- (eth2) s4 (eth1) --- h4
h2 --- (eth1) s2 (eth2) --- (eth2) s5 (eth1) --- h5
h3 --- (eth1) s3 (eth2) --- (eth2) s6 (eth1) --- h6
```

Figure 4.2 Representation of the network topology

On start-up all the switches in the topology connects to both Ryu OpenFlow controllers. Even though all the switches connect to both the controllers, each switch in the topology can elect one of the controllers as the master controller. This is done by per switch master election using ovs-vsctl set-controller command.

4.3.2 Ryu simple switch application

To demonstrate scalability and fault tolerance, the simple switch application that is bundled with Ryu controller has been taken and modified. On switch start up the application creates port vertices and switch vertices on the Titan Graph database. The information about switches and ports can be obtained via the OpenFlow protocol. The connection between the switches however falls outside the purview of the OpenFlow protocol. To understand the connection between the switches, Link Layer Discovery Protocol (LLDP) has been used. This was not originally a part of simple switch application. When the switches receive a packet-in message it first creates a device vertex for the hosts that are trying to communicate. It also implements some basic switching algorithm and creates the flow entry and flow vertices for all the flows that it computes. The flow entry vertices for each switch will correspond to the entries of the flow table on the switches. When a flow entry is removed, either by the controller or by the flow expiry mechanism in the switches, the switch will send a flow removed message to the controller. On receiving this message the controller application will remove the corresponding flow entry vertex and update the flow vertex in the database. This operation is critical since the database should reflect that updated state of the network. Another modification made to the simple switch application is the addition of two different modes for its operation, proactive mode and reactive mode. In reactive mode, the simple switch application responds to a packet-in message from the switch by pushing down appropriate flow-mod messages down to the switch. In proactive mode, the user specifies a file with '-f' option. This file contains the source mac addresses and the destination mac addresses of the host devices in the network topology that the user wants to connect. It also contains source dpid and the destination dpid of the switches to which the hosts are connected and the respective port numbers. For e.g. if the user wants to establish communication between h1 and h4, h2 and h5, h3 and h6 the user will use the following file.



```
#srcdpid   #srcport #dstdpid #dstport #srcmac #dstmac
srcdpid 00:00:00:00:00:00:00:01 srcport 1 dstdpid 00:00:00:00:00:00:00:04\
    dstport 1 srcmac 00:00:00:00:00:01 dstmac 00:00:00:00:00:04
srcdpid 00:00:00:00:00:00:00:02 srcport 1 dstdpid 00:00:00:00:00:00:00:05\
    dstport 1 srcmac 00:00:00:00:00:02 dstmac 00:00:00:00:00:05
srcdpid 00:00:00:00:00:00:00:03 srcport 1 dstdpid 00:00:00:00:00:00:00:06\
    dstport 1 srcmac 00:00:00:00:00:03 dstmac 00:00:00:00:00:06
```

Figure 4.3 Path configuration file for proactive mode

If the simple switch application is started in proactive mode, it will pre-program the flows corresponding to the source and destination mac addresses mentioned in the files. However it will also respond appropriately to any packet-in messages that may be sent to the controller. The reason for modifying the application to support proactive mode is that it is much easier to demonstrate our scalable application in that mode. It should be mentioned that the same copy of the application is running on all instances of Ryu controller.

4.3.3 Communication between Ryu instances

For demonstrating scalability, communication between the Ryu controller instances needs to be established. The switches in the underlying topology can choose either of the two controllers as its master. The application running on either of these controllers should be agnostic of this choice. In other words each application behaves as if it controls the entire topology. However, only the master controller for a switch can program that switch. We provide ZeroMQ mechanism to allow the Ryu instances to communicate among each other. This is a low latency high throughput messaging mechanism for communication. If the Ryu application has messages that it wants to push down to the switches for which it is not the master controller the message is pushed via ZeroMQ to the other controller and that controller pushes down the messages to the corresponding switches.

4.3.4 Demonstration of fault tolerance

For demonstrating fault tolerance the modified simple switch application described above has been used. We start the application on both Ryu OpenFlow controllers as shown below

```
$ ryu-manager simple_switch.py
```

Figure 4.4 shows that six switches in the network topology is connected to both the Ryu OpenFlow controllers.

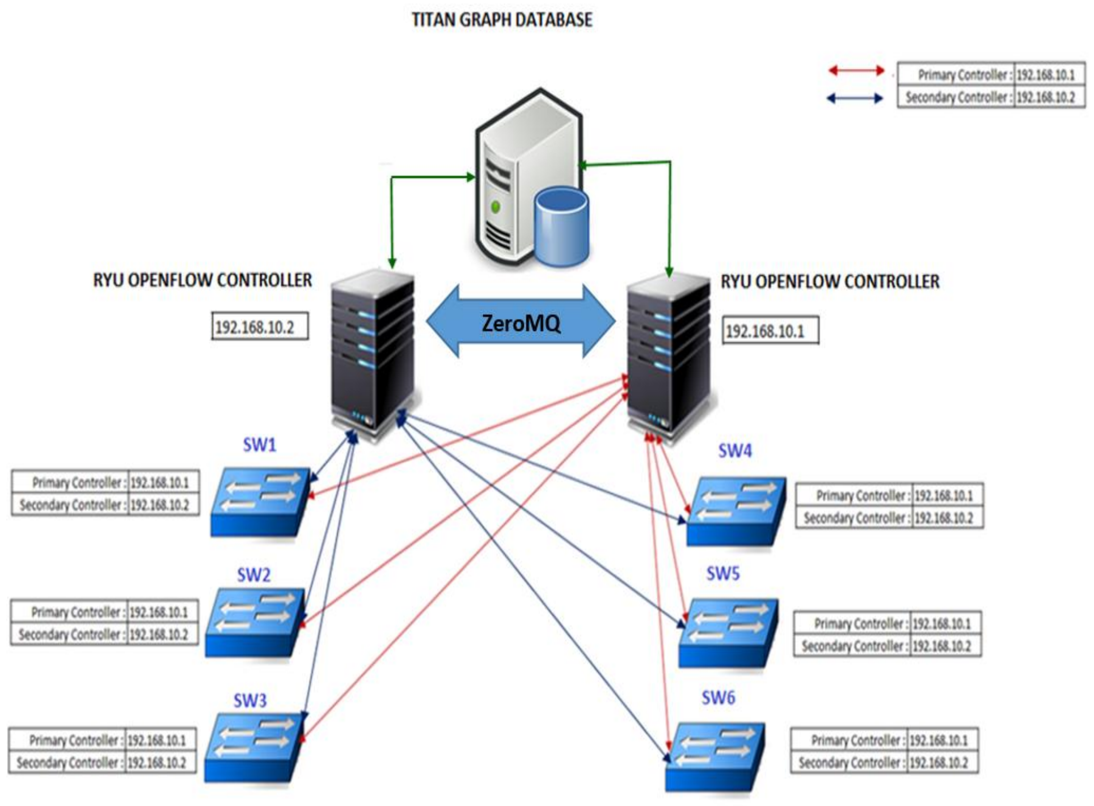


Figure 4.4 Network topology is controlled by two RYU controllers

The Ryu controller running on the Virtual Machine “ryu-primary” (IP address - 192.168.10.1) is designated as the master controller for all the switches. This has been shown by red lines connecting the switches to the controller. In the initial state when the switch flow tables are empty, if a ping test is tried between the connected pairs of hosts as shown in the network topology representation in Figure 4.2, the switches in the data path will send a packet-in message to the controller and the simple switch application running on the master controller will respond by pushing down the appropriate flow modification message down to the switches. Once the flow tables in the switches are populated by the correct flows the connected pairs of hosts will be able to communicate with each other.

Now let us assume a scenario where the server “ryu-primary” goes down for some reason. This scenario is shown in Figure 4.5 below.

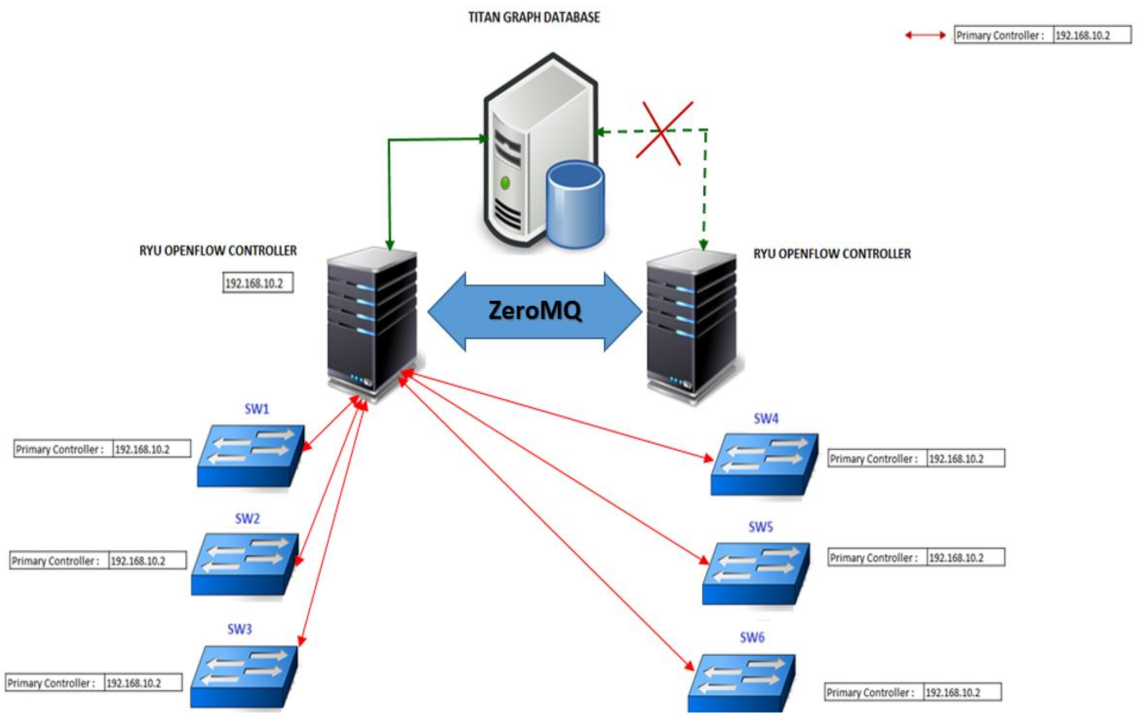

TITAN GRAPH DATABASE
Primary Controller : 192.168.10.2
RYU OPENFLOW CONTROLLER
192.168.10.2
ZeroMQ
RYU OPENFLOW CONTROLLER
SW1
Primary Controller : 192.168.10.2
SW2
Primary Controller : 192.168.10.2
SW3
Primary Controller : 192.168.10.2
SW4
Primary Controller : 192.168.10.2
SW5
Primary Controller : 192.168.10.2
SW6
Primary Controller : 192.168.10.2

Figure 4.5 One instance of RYU controller takes over all the switches

Since all the six switches in the topology had designated the Ryu controller running on “ryu-primary” as their master, the Ryu controller running on “ryu-secondary” cannot program those switches. For the Ryu controller on “ryu-secondary” to take over the switches in the topology the switches have to elect it as their master. This can be done by each switch in the topology by a per switch master election as shown in Figure 4.6. The switches are named sequentially as **s1, s2, s3, s4, s5** and **s6.** We can use ovs-vsctl tool’s set-controller command to designate the Ryu controller running on “ryu-secondary” as their master controller as shown below.

```
$ ovs-vsctl set-controller s1 "tcp: 192.168.10.2:6633"
$ ovs-vsctl set-controller s2 "tcp: 192.168.10.2:6633"
$ ovs-vsctl set-controller s3 "tcp: 192.168.10.2:6633"
$ ovs-vsctl set-controller s4 "tcp: 192.168.10.2:6633"
$ ovs-vsctl set-controller s5 "tcp: 192.168.10.2:6633"
$ ovs-vsctl set-controller s6 "tcp: 192.168.10.2:6633"
```

Figure 4.6 Script to change the master controller of the switches

Once all the switches in the topology has chosen Ryu controller running on “ryu-secondary” as their master then the simple switch application running on “ryu-secondary” can control all the switches in the topology.

We can easily verify this by clearing all the contents of the flow tables in all the switches as shown in Figure 4.7 and then trying to ping between connected pairs of hosts.

```
for i in `sudo ovs-vsctl list-br` ;
    do sudo ovs-ofctl del-flows $i  ;
done
```

Figure 4.7 Script to clear the flow table in the switches

If a ping test is tried between the connected pairs of hosts, then as described above the switches in the data path will send packet-in messages to the

Ryu controller that is the master, which is now the Ryu controller running on "ryu-secondary" virtual machine.

The simple switch application running on Ryu master controller in turn will respond by pushing down the appropriate flow modification messages down to the switches and the switches will be able to communicate between each other.

4.3.5 Demonstration of scalability

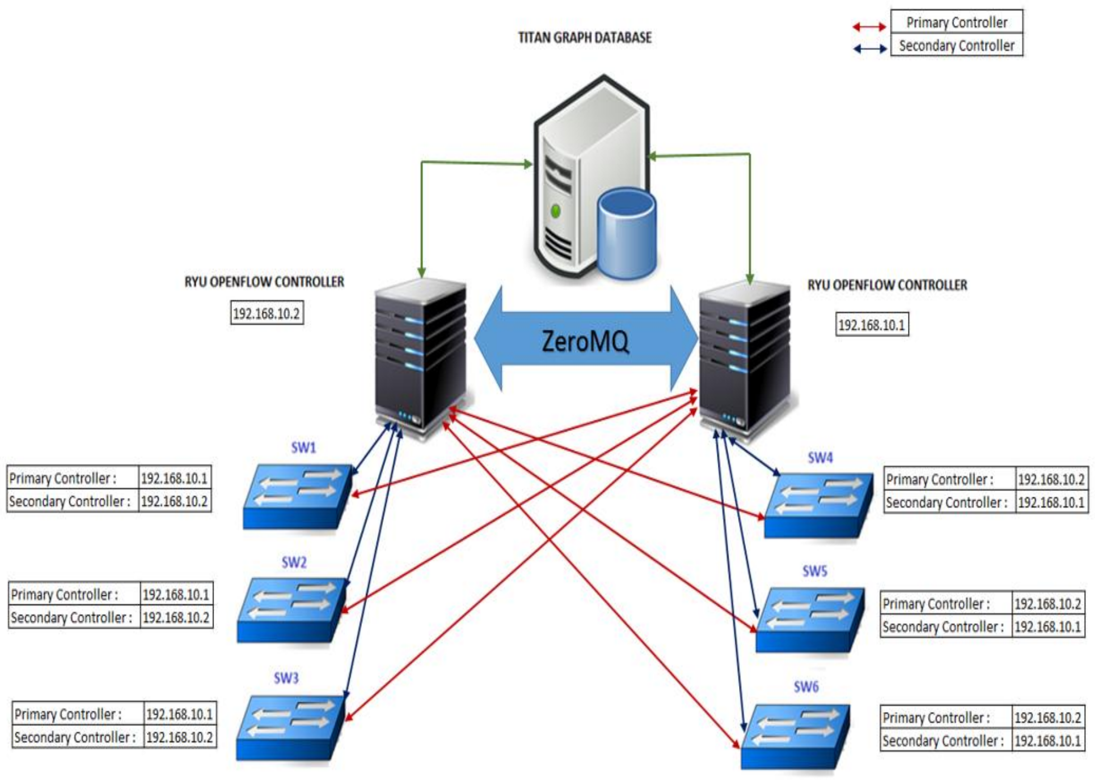
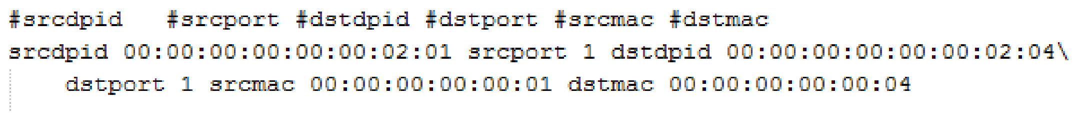

Primary Controller
Secondary Controller
TITAN GRAPH DATABASE
RYU OPENFLOW CONTROLLER
192.168.10.2
ZeroMQ
RYU OPENFLOW CONTROLLER
192.168.10.1
SW1
Primary Controller : 192.168.10.1
Secondary Controller : 192.168.10.2
SW2
Primary Controller : 192.168.10.1
Secondary Controller : 192.168.10.2
SW3
Primary Controller : 192.168.10.1
Secondary Controller : 192.168.10.2
SW4
Primary Controller : 192.168.10.2
Secondary Controller : 192.168.10.1
SW5
Primary Controller : 192.168.10.2
Secondary Controller : 192.168.10.1
SW6
Primary Controller : 192.168.10.2
Secondary Controller : 192.168.10.1

Figure 4.8 Configuration for demonstration of scalability

The setup that was used for the demonstration of scalability is shown in Figure 4.8. On switch start up all the switches establish connection with both the controllers. The switches s1, s2 and s3 chooses the Ryu controller running on "ryu-primary" as their master controller and switches s4, s5 and s6 chooses the Ryu controller running on "ryu-secondary" as their master controller. For this experiment, the modified version of Ryu simple switch application is started in proactive mode on one of the machines say “ryu-primary”. On the other machine the user can choose to run the custom simple switch application in normal mode.

$ ryu-manager simple_switch.py –f path_file.txt

This path file contains the source and destination MAC addresses of the hosts that needs to communicate. It also contains the data path identifiers of the switches to which the hosts are connected as well as the ports on the switches on which the hosts are connected. An entry in the path file will look as below.



```
#srcdpid   #srcport #dstdpid #dstport #srcmac #dstmac
srcdpid 00:00:00:00:00:00:02:01 srcport 1 dstdpid 00:00:00:00:00:00:02:04\
    dstport 1 srcmac 00:00:00:00:00:01 dstmac 00:00:00:00:00:04
```

Figure 4.9 Entry in the path configuration file

Based on this information our custom simple switch application will create switch, port, device, flow and flow_entry vertices in the Titan Graph database. It will also try to push down the flow modification messages to the switches based on the flow_entry vertices it has already computed and stored in the graph database. Now as explained above, the Ryu OpenFlow controller running on “ryu-primary” is the master controller for only three switches in the topology. The other three switches are controlled by the controller running on “ryu-secondary”. This means that this controller can only push down flows to the switches for which it is the master controller. The other flow mod messages will be sent to the controller running on “ryu-secondary” using the ZeroMQ publish subscribe framework. On receiving the flow modification messages the custom simple switch application on “ryu-secondary” pushes down the messages to the corresponding switches.

Chapter 5 Performance Measurements

5.1 Introduction

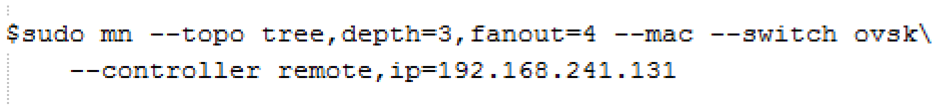
In this chapter, the performance of the system that we built is measured. The application used for demonstration of scalability and fault tolerance is a modified version of simple switch application that is packaged with Ryu source code. We have taken that application and developed it into a database backed scalable and fault tolerant application. It would be interesting to take the original version of simple switch as the benchmark against which the modified application will be compared. It has to be kept in mind that the modified version of the application does offer a lot of features such as scalability and fault tolerance that is not available in the original version.

5.2 Methodology

The methodology that was adopted for testing the application is as follows. Firstly two different mininet topologies will be defined. These topologies will be connected to Ryu OpenFlow controller. There are three different cases that have to be considered. In the first case the Ryu OpenFlow controller will be running the original simple switch application. In the second case Ryu OpenFlow controller will be running our version of the simple switch application. In this case there will be only one instance of the Ryu OpenFlow controller running. In the third and final case the network topology will be connected to two different instances of Ryu OpenFlow controller running our version of simple switch application. In this case half the switches in the topology will be mastered by the first instance of Ryu and the other half will be mastered by the other instance of Ryu controller. In all the six cases, i.e. two different network topologies connected against three cases of controller application, the method used was to measure the time taken for a simple pingall test on each of topology using time command. This does a simple ping between all the hosts. If there are ‘n’ hosts in the network, each host will ping the other n-1 hosts. Once the pingall test is completed the network is fully programmed and all the hosts in the topology will be able to talk to each other. This test gives an accurate measure of the time taken to program the switches in the topology. The time command has been used to measure the time taken to complete the pingall test.

5.3 Network topologies

The first topology that will be considered is a tree topology. Mininet has the option to create a tree topology of a specified depth, where each leaf switch has ‘n’ hosts connected to it. ’n’ is called the fanout factor. For example, to launch mininet topology with a binary tree of depth 3, and fanout factor of 4, the following command is used.

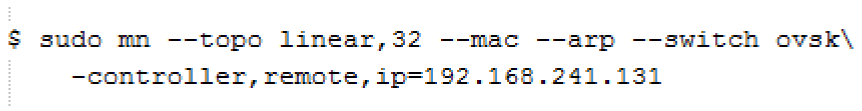


```
$sudo mn --topo tree,depth=3,fanout=4 --mac --switch ovsk\
    --controller remote,ip=192.168.241.131
```

Figure 5.1 Creating tree topology with depth 3 and fanout 4

This topology will have a total of 64 (4 ^3 = 64) hosts in the topology. This topology will be arranged in a tree with a depth of 3. The first level will have a single switch. The second level will have 4 switches and the third level will have 16 switches bringing the total number of switches to 21.

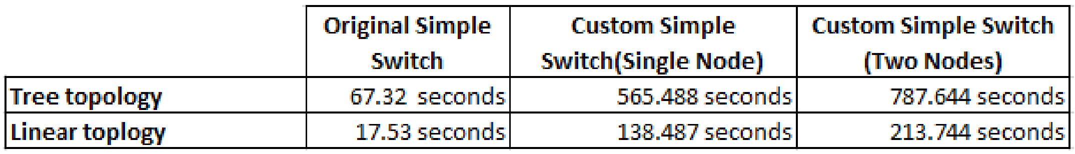
The second topology that will be considered is a linear topology. The topology contains 32 switches and 32 hosts, one hosts connected to each switch. The switches are connected in a linear fashion. To launch this mininet topology, the following command is used.



```
$ sudo mn --topo linear,32 --mac --arp --switch ovsk\
    -controller,remote,ip=192.168.241.131
```

Figure 5.2 Creating linear topology with 32 switches

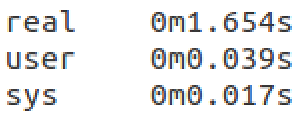
5.4 Results



	Original Simple Switch	Custom Simple Switch(Single Node)	Custom Simple Switch (Two Nodes)
Tree topology	67.32 seconds	565.488 seconds	787.644 seconds
Linear toplogy	17.53 seconds	138.487 seconds	213.744 seconds

Table 5.1 Performance measurements

As it can be seen from Table 5.1, the custom version of simple switch does incur some cost during the flow programming phase. This is because of additional overhead due to creating device, flow and flow_entry vertices in Titan Graph database. In the demonstration setup, the Titan Graph database is run on the same virtual machine as the primary Ryu controller. The creation of vertices and edges on the Titan Graph database is a blocking call. Let us consider creating two vertices and an edge connecting those vertices as the basic unit of operation in a graph database. Time taken for that operation was measured using the time command. The values obtained are shown in Figure 5.3



```
real    0m1.654s
user    0m0.039s
sys     0m0.017s
```

Figure 5.3 Time measured for creating two vertices and one edge

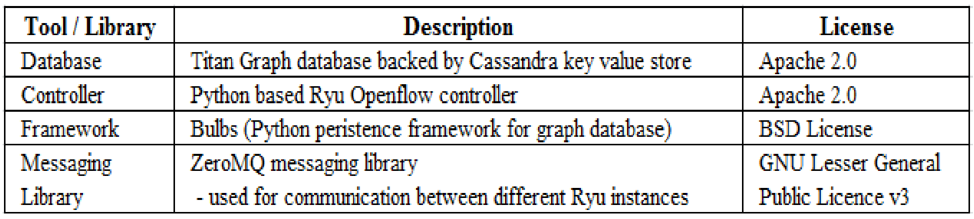
The total amount of CPU time comes to about 0.056 seconds. The rest of the time (1.598 seconds) is spent by the process in a blocked state. This can be speeded up significantly if the database is moved to a dedicated database server. Also the bulbs interface is used to create vertices and edges in the application. This method is described in Figure 3.2. It can be seen that the method has three function calls which in turn are translated to three REST API calls to the Rexster server. The three function call can be reduced to a single call by using a parametrized Gremlin script. This approach has been known to improve the performance time due to the following reasons. There would be a single REST API call instead of three. A batched transaction of three inserts with a single commit instead of three commits in the above case. The parameterized script would be compiled and thus cached. This is important as compiling is quite expensive. The difference in the programming time between our simple switch running on one Ryu instance and the one running on two Ryu instances can be explained by the latency incurred in pushing out flow modification messages across different Ryu instances via ZeroMQ. Considering that our architecture brings the double advantage of scalability as well as fault tolerance to the system, the overhead incurred by our version of simple switch can be justified.

Chapter 6 Future Improvements

6.1 Introduction

While investigating different open source OpenFlow controllers, it was realized that there was not a single open source OpenFlow controller available at that time, which was database backed having features like scalability and fault tolerance. This led to the idea of designing and building a database backed open source controller that is scalable and fault tolerant. It has to be noted that, ONOS, which is a database backed scalable and fault tolerant open source OpenFlow controller was not officially released until December 2014. By this time, the majority of work related to this project was complete. This project also uses an existing controller that is popular for prototyping OpenFlow applications, rather than create a new one as ONOS has. The study was done on several related works on scalable and fault tolerant controllers. The works that were investigated include Onix, ONOS and HyperFlow. After understanding the existing work that has been done, the decision to design and build our own version of a database backed scalable and fault tolerant OpenFlow controller was taken.

This project was implemented entirely using open source tools and libraries. The details of the tools and libraries used are furnished in the following table.



Tool / Library	Description	License
Database	Titan Graph database backed by Cassandra key value store	Apache 2.0
Controller	Python based Ryu Openflow controller	Apache 2.0
Framework	Bulbs (Python peristence framework for graph database)	BSD License
Messaging Library	ZeroMQ messaging library - used for communication between different Ryu instances	GNU Lesser General Public Licence v3

Table 6.1 Open source tools and libraries used

Using the software mentioned in Table 6.1, a database backed scalable and fault tolerant OpenFlow controller was built. To demonstrate that the design is both scalable and fault tolerant, custom developed version of L2 switch application was used. After demonstrating the features of the design, the decision was made to do some performance measurements of the designed controller (both single node and multi node versions) connected to different network topologies. The benchmark used for the comparison was the performance of original L2 switch application that is shipped with Ryu OpenFlow controller.

In the following sections, some suggestion for future improvements and a roadmap that this project should take are presented.

6.2 Separation of Topology Discovery

In the current implementation, topology discovery and creating the topology graph of the network elements is clubbed with the Layer 2 switching application. Ideally this functionality alone should have been a separate application. If that is the case, then the network graph building application can be run along with Layer 2 switching application or in conjunction with any other application.

6.3 Web Interface for Network Topology

Since the Titan Graph DB builds the network topology graph it would be nice if the user can see a visual representation of the same. It would be nice to use the d3 Java Script library for the graphical representation and then serve the web pages using Django framework. Another alternative would be to use KeyLines, which is a fully-featured Software Development Kit for building graph visualization software.

6.4 Rest API for switch to controller mapping

Since the mapping of switches to controllers are dynamic and can change, it would be nice to expose a REST API that the user can use to know the current switch to controller mapping. It would display the list of controllers and the switches that are controlled by the controller.

6.5 Coordination for distributed systems

Zookeeper is a coordination system for distributed systems. It is commonly used to implement coordination mechanisms such as locks and leader elections. We can use it to manage the assignment of switches to controllers. Our fault tolerance model as described in chapter 4, allows a switch to connect to multiple RYU instances, and the instances have to work out amongst themselves which one is the master controller for that switch. This coordination is implemented using a leader election in Zookeeper. When a switch connects to a set of controllers, each of the controllers will contend in the leader election to try and take control of the switch. One instance will become the leader and will become master for the switch. If the master instance dies, Zookeeper can detect this and leadership will be assigned to one of the other instances.

6.6 Replication of Cassandra nodes

The current architecture has only a single instance of Cassandra. This exposes a single point of failure. Since one of the main drivers of the architecture is to make the system fault tolerant, the Cassandra can be configured to be a multi node cluster. During the initial design phase focus was not given for this particular issue since this issue is already known and has a well-documented solution. Now that the basic architecture, that provides scalability and fault tolerance has been implemented it would make sense to implement this feature for the sake of completeness.

References

1. Software-Defined Networking: The New Norm for Networks. Available at: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf
2. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J. (2008). OpenFlow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev., 38 (2), 69–74.
3. OpenFlow Switch Specification v1.3.1. Available at:https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf
4. Volkan Yazıcı1, M. Oğuz Sunay1, Ali Ö. Ercan1. Controlling a Software-Defined Network via Distributed Controllers
5. B. Ban, “Design and Implementation of a Reliable Group Communication Toolkit for Kava,”
6. Amin Tootoonchian, Yashar Ganjali HyperFlow: A Distributed Control Plane for OpenFlow
7. Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievskiy,Min Zhuy, Rajiv Ramanathan, Yuichiro Iwataz, Hiroaki Inouez, Takayuki Hamaz, Scott Shenker. Onix: A Distributed Control Platform for Large-   scale Production Networks
8. Pankaj Berde†, Matteo Gerola‡, Jonathan Hart†, Yuta Higuchi§,Masayoshi Kobayashi§, Toshio Koide§, Bob Lantz†, Brian O’Connor†, Pavlin Radoslavov†, William Snow†, Guru Parulkar†. ONOS: Towards an Open, Distributed SDN OS
9. http://www.amqp.org/sites/amqp.org/files/amqp.pdf

10. http://blog.x-aeon.com/2013/04/10/a-quick-message-queue-benchmark-activemq-rabbitmq-hornetq-qpid-apollo/