Path Computation Enhancement in SDN Networks

by

Tim Huang

Bachelor of Computer Science in ChengDu College of University of Electronic Science and Technology of China, ChengDu, 2011

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Applied Science

in the Program of

Computer Networks

Toronto, Ontario, Canada, 2015

Path Computation Enhancement in SDN Networks

Master of Applied Science 2015

Tim Huang

Computer Networks

Ryerson University

Abstract

Path computation is always the core topic in networking. The target of the path computation is to choose an appropriate path for the traffic flow. With the emergence of Software-defined networking (SDN), path computation moves from the distributed network nodes to a centralized controller. In this thesis, we will present a load balancing algorithm in SDN framework for popular data center networks and a fault management approach for hybrid SDN networks. The proposed load balancing algorithm computes and selects appropriate paths based on characteristics of data center networks and congestion status. In addition, a solution that supports proper operations of a hybrid SDN network will also be proposed. The evaluation shows the proposed load balancing algorithm performs better than classic shortest path algorithms. We also demonstrated that the proposed solution for hybrid SDN networks can support proper operations in complicated hybrid SDN networks.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Ngok Wa Ma for his patient support and encouragement through my graduate study.

Special thanks to the Computer Networks Master program and the School of Graduate Studies at Ryerson University for their financial support and the opportunity granted me to study in this great citadel of learning.

Last, but not the least, I would like to thank my wife, jojo jiang, for her continuous support over years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 General

Since the first establishment of point-to-point networks in the last century, networking industry has experienced fast development, especially after the establishment of the Internet. The network traffic is growing exponentially over time. Meanwhile, managing networks has become more complicated and difficult. Firstly, it involves too many types of network devices, from switches and routers to middle boxes such as firewalls, load balancers. Secondly, there are many network protocols to address all sorts of information asymmetry problems among network nodes, like using Spanning Tree Protocol (STP) on switches to avoid Layer-2 loop, Open Shortest Path First (OSPF) on routers to choose the shortest paths, Network address translation (NAT) on firewalls to protect internal Internet Protocol (IP) address[12].

Network devices have control and data plane. The control plane is the brain of network node, which decides what to do with incoming packets. The data plane is the limbs of network node, which takes action on the packet according to control plane decision. On control plane of legacy network devices, network device vendors usually implement the standardization protocols drafted by some organizations, such as Internet Engineering Task Force (IETF) and Institute of Electrical and Electronics Engineers (IEEE). The job of network operators is to configure different vendors' devices to make the protocol work properly among devices. In many cases, different networking hardware vendors implement the same protocol differently. In addition, these vendors may introduce proprietary features to the standard protocol. This results in unnecessary configuration complexity for network operators and unpredicted mismatching between different vendors' devices.

The issues raised in the last paragraph are caused by the fact that the control plane of legacy networks is a distributed system. Every network node is independent and the nodes have different views for the same network. The purpose of most of the network protocols is to help network nodes to exchange network information among themselves. In most situations, a single network node does not have the complete information of the network. If there is something wrong with the network, the network operator has to go through the network device by device to inspect whether the configuration conforms

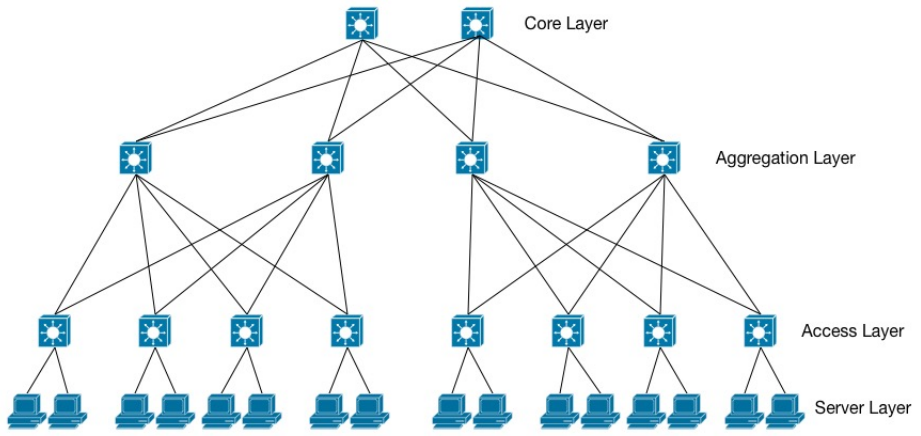


Figure 1.1: A typical data center network topology

with the design of the network and to check the consistency of protocol parameters among network nodes. Nowadays, protocols such as Simple Network Management Protocol (SNMP), Path Computation Element Protocol (PCEP) and The Network Configuration Protocol (NETCONF) can control the whole network to some extent. However, there are too many limitations for this type of protocols.

The SDN paradigm provides a method to design and mange the network more efficiently. The networking functions, such as path selection and policy deployment, are centralized on the controller. Using a centralized controller has several obvious advantages. Firstly, the complete network information resides on the controller. The network operator can utilize all kinds of network information to design the policies for the network. Secondly, when the operators need to do some troubleshooting for the network, they do not have to go to network node one by one. What they only need to do is checking the relevant information on the controller. Because all the network information is in the controller, the operators can be more accurate and faster to manage, provision and trouble shoot. Lastly, the controller provides possibility of programming to the operators. Thus, the operator can optimize the network according to the upper layer application.

1.2 Research Problem

In current data centers, the structure of the network consists of several layers of switches, usually consisting of access layer, aggregation layer and core layer[7]. Several servers connected to an access layer switch, several access layer switches connected to an aggregation layer switch, and aggregation layer switches connected to core layer switches. The Figure 1.1 shows a typical data center network topology.

The main function of a data center is to provide the services to the numerous clients, so using multiple servers to load balance same service is very common in the data center. That is why implementing load balancing in the data center can be significant. A number of papers[25, 16, 20, 17] on the topic of load balancing can be found in the literature. In this thesis, we proposed an algorithm that utilize the characteristics of data center networks to compute best path on the controller. A simple but efficient path selection algorithm is also proposed.

The evolution from legacy network to SDN networks takes time, so the legacy and SDN networks will coexist for a while to form hybrid SDN networks. The current solution for hybrid SDN networks is not sufficient to handle some general topologies. In this thesis, a solution is proposed to address the deficiency of the current approach.

1.2.1 Contributions of the thesis

1. Design an efficient load balancing algorithm based on the popular topologies used in data center networks.

2. Propose a solution to interconnect SDN network islands with legacy network islands.

1.2.2 Thesis Organization

- **Chapter 2** presents the background information for legacy networks and architecture of SDN networks, followed by the related work on the thesis.

- **Chapter 3** describes the detail of proposed algorithms for path computation and path selection.

- **Chapter 4** represents the solution for hybrid SDN networks.

- **Chapter 5** depicts the implementation of chapter 3 and chapter 4.

- **Chapter 6** shows the experimental results with the analysis.

- **Chapter 7** concludes the thesis and introduces some future work.

Chapter 2

Background

2.1 Legacy Networks

Ethernet switch is one of the basic network elements used in Local Area Networks (LANs). It uses hardware addresses to forward the frame at the data link layer of the (Open Systems Interconnection)OSI model. The switch operates at data link layer of the OSI model to create a separate collision domain of every switch interface such that the interfaces can transmit and receive the data simultaneously.

The switch forward data frames based on Media Access Control (MAC) table. When a frame arrives at a switch, the switch will put the source MAC address and correspond incoming interface number in MAC table as the basis for forwarding new frames. Then the destination MAC address will be inspected. If the destination MAC address is multicast address or unknown unicast, it will forward the frame to all the interfaces except the incoming interface. Otherwise, the frame will be forwarded to specific interface according to the MAC table.

When the switch floods a frame to the network, it may create the traffic loop in the network whose topology consists of loops. To solve this, legacy switch usually uses a spanning tree protocol that blocks some interfaces so that the resulting logical LAN topology is a tree. Through the spanning tree protocol, traffic loops can be prevented.

2.2 SDN Network

2.2.1 OpenFlow Protocol

SDN, as stated above, uses a controller to implement the control plane. The architecture makes the network become dynamic, cost-effective, and adaptable.When the control plane is decoupled from the data plane of networking devices, the architecture of networks changes. Generally speaking, the architecture of SDN networks[6] consists of three layers as shown in Figure 2.1. The layers are depicted as follows:

- Infrastructure layer: This layer is the data plane of legacy networks, containing all kinds of network

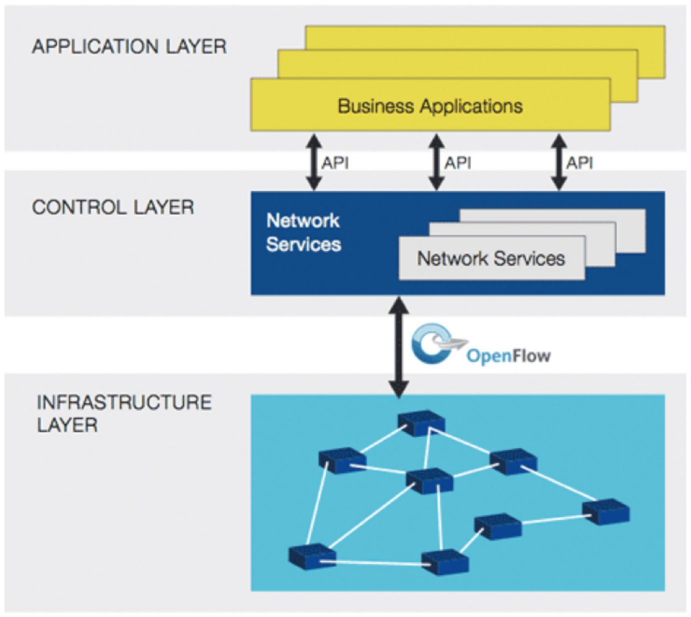


Figure 2.1: The SDN architecture

forwarding elements.

- Control layer: This layer is the summation of control planes for all legacy networks devices. The control layer instructs the infrastructure layer how to forward the traffic and provides the application program interface (API) for the upper layer.

- Application layer: Since the control layer can provide the API for the upper layer, the network can be essentially managed by the upper layer applications.

As the Figure 2.1 shows, the protocol, which is used to support the communications between infrastructure layer and control layer, is the OpenFlow protocol[21]. It should be noted that other protocols can also be used between infrastructure layer and control layer. By using OpenFlow protocol, the control layer can determine the flow paths remotely and exchange the statistics with infrastructure layer.

2.2.2 OpenFlow Switch

The OpenFlow switch is the basic element in infrastructure layer. It relies on the flow tables to forward the traffic rather than the MAC table. When the frame arrives at the switch, it will attempt to match the header information with the flow table entry in priority order. If it matches, the action associated with flow entry will be executed. Otherwise, the packet will be dropped or forwarded to the controller based on flow entry's configuration. The main components of an OpenFlow switch are presented in Figure 2.2.

The flow entry's instructions include the actions or modifying pipeline processing. The actions include

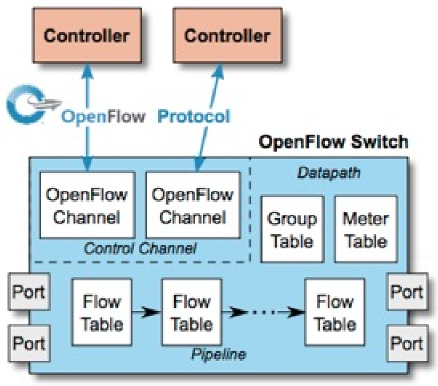


Figure 2.2: Main components of an OpenFlow switch

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

Table 2.1: Main components of a flow entry in a flow table

the packet's forwarding, packet modification. Pipeline processing allows packets to be forwarded other tables for further processing.

Flow entry usually forward the packet to an interface. The interface can be physical, logical or reserved interface. The reserved interface specifies forwarding actions such as sending to controller, flooding or normal. Normal means forwarding the packet as the legacy switch. Table 2.1 shows the main components of a flow entry.

Every flow entry includes:

- match fields: matching with packets. These consist of the incoming interface and packet headers.
- priority: matching priority of the flow entry.
- counters: updated when packet is matched.
- instructions: to modify the action set or pipeline processing.
- timeouts: maximum time or idle time before a flow gets expired.
- cookie: opaque data value chosen by the controller.
- flags: flags alter the way flow entry is managed.

As we describe above, the workflow of an OpenFlow switch processing a packet is presented in Figure 2.3[23].

2.2.3 OpenFlow Controller

The controller is the basic element in the control layer. The control layer consists of one or several physical controllers to form a logical controller. In most situations, the application layer can also be

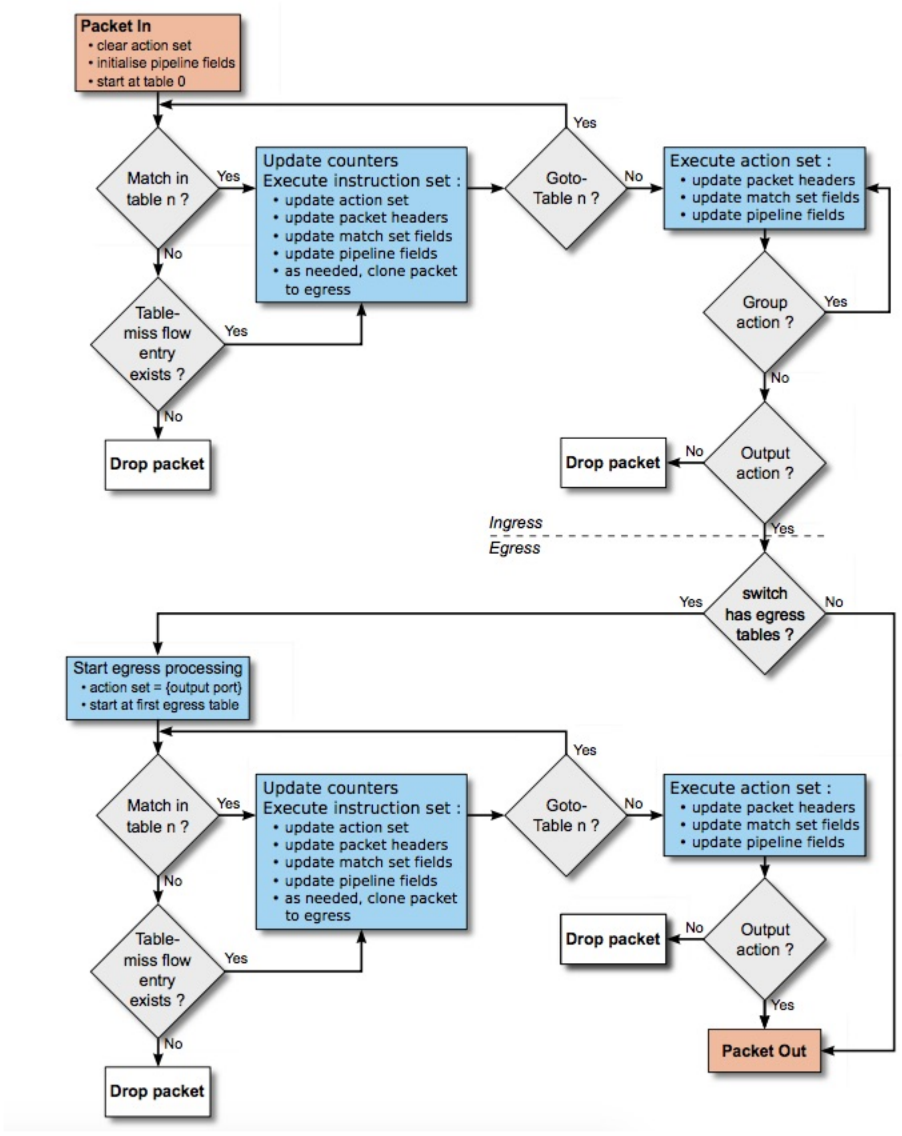


Figure 2.3: The flowchart detailing packet flow through an OpenFlow switch

integrated to the controller. In SDN architecture, the main function of a controller is to communicate with switches. By adding, deleting and modifying the flow entry reactively and proactively, the controller can change the flow path. The controller can also send OpenFlow message to query or change switch's status.

Nowadays most controllers provide Representational State Transfer service (RESTful) API to make the controller be accessible readily by the application layer.

2.3 Network Topology in SDN

In SDN networks, almost all the controllers use Link Layer Discovery Protocol (LLDP) to discover the topology of the network. LLDP is an open standard used by network devices for discovering their neighbors in legacy networks. In SDN networks, the controller structures the LLDP frame and sends it to all the interfaces of all the switches. When the switch receives the LLDP frame from its peers, it will forward the frame to the controller. Through this method, the controller can learn the complete network topology.

After learning the network topology, the controller, like legacy networks, will overlay a spanning tree on the network. The spanning tree is used to prevent traffic loop during data flooding and broadcasting. Unlike legacy networks, the unicast traffic will not follow the spanning tree. Instead, the unicast traffic will follow the shortest path between the source and destination.

2.4 Path Computation in SDN

The path computation is the core work for the controller after it learns the network topology. Dijkstra's and Floyd Warshall algorithms[11, 13] are commonly used for the shortest path computation. Dijkstra's algorithm computes all the shortest paths between a single-source to all possible destinations; while Floyd Warshall algorithm computes the shortest paths for all possible source/destination pairs.

In a legacy network, each network node computes the shortest path from the node to all destinations, so the Dijkstra's algorithm is the best option. In SDN networks, the controller knows the complete network topology and is in charge of setting up paths for all possible source/destination pairs, thus Floyd Warshall algorithm is the perfect choice.

2.5 Hybrid SDN Networks

A hybrid SDN network is the network which the legacy network and the SDN network are used and operate in the same environment. Since the legacy network is not controlled by the SDN controller, the controller does not know the complete network topology. The consequence is that the network is broken down into a number of SDN islands separated by non-SDN island(s) as illustrated by Figure 2.4. Since the controller does not know the topology of the non-SDN island, it cannot compute the spanning tree and the path between hosts residing in different SDN islands.

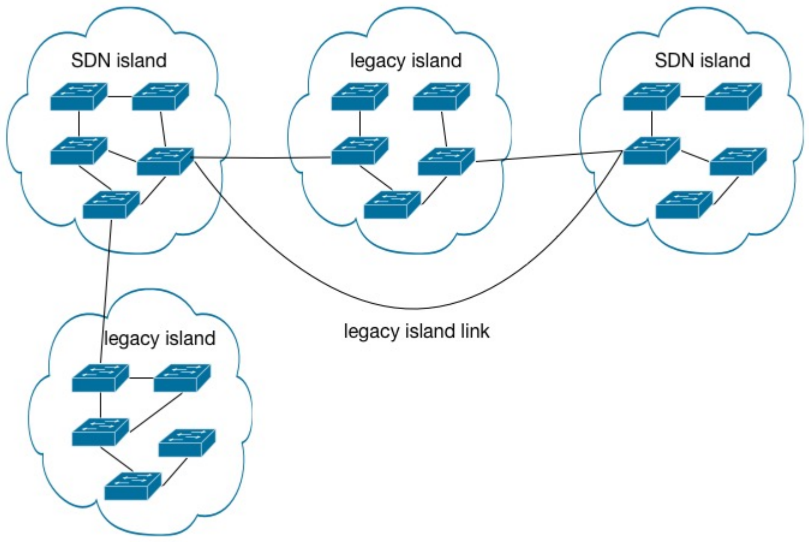


Figure 2.4: A example of hybrid SDN network

Here are the basic terminologies and concepts we will use in explaining our approach to solve the connectivity program in hybrid SDN networks:

- Legacy island: The part of the network which consists of legacy (non-SDN) switches.
- Legacy island link: The logical link from a SDN switch to another SDN switch through legacy island. There can be many physical links and legacy network nodes in between two SDN switches. However, from the view of SDN controller, the traversed legacy network nodes and links are treated as one single logical link.
- SDN island: The part of the network which consists of SDN switches.

The Figure 2.4 shows an example of a Hybrid SDN network.

2.6 Related works in Load Balancing

Load balancing occupies an important position to solve over-load traffic problem in the network. It has been one of the first appealing applications in SDN networks. These are some commonly used load balancing approaches[8]:

- Random: This approach randomly distributes the traffic to the available paths. Generally, hash function is used to map requests to available paths.
- Round Robin: This approach distributes the request to the paths in sequence, starting from the first path to the last one in rotation continuously.

- Weighted Round Robin: This approach assigns weight for each path, then distributes requests sequentially with respect to the assigned weights.
- Least Connections: This approach forward the request to the path that has the least number of current connections.

Wang et al.[25] proposed an algorithm to achieve load balancing across replica servers for the same service. Their algorithms create a binary tree to map the servers. By matching source IP address in the packet with the wildcard, the destination server is selected. The controller then replaces destination IP address for the service by the IP address of the selected server. By spreading the traffic among replica servers, the controller achieves the traffic load balancing. Since only the first packet of the traffic flow is sent to the controller, communication between the switches and the controller is minimal. However, the paper did not mention how to make a path selection when there are multiple equal-cost paths.

Handigol et al.[16] proposed a load balancing algorithm in which the controller periodically inquires the status of the network and servers. The algorithm tracks Central Processing Unit (CPU) utilization of servers and network link utilization to choose the best server dynamically. However, they did not account the overhead of the query traffic. They also did not address path selection when there is a congestion.

Li et al.[20] proposed an algorithm to select the path according to the switch statistics. The algorithm utilizes three-tier network design and queries the switch statistics to select link with maximum available bandwidth for every hop. However, selecting the maximum available bandwidth for every hop may not be the optimal path from overall situation. For example, choosing a maximum available bandwidth interface at the beginning may cause the worse route to be chosen in following hops. The algorithm also does not take advantages of some desirable data center networks characteristics.

Jiang et al.[17] proposed an algorithm extending Dijkstra's algorithm to select shortest path on the controller. The algorithm leverages the fact that the controller has the complete view of the network. Therefore, path calculation can use not only link weight but also node weight. The extension can be helpful for the general network topology, whereas the characteristics of data center networks are not used at all.

In chapter three of this thesis, we will discuss an efficient way to compute the shortest paths based on popular data center network topology. We will also address the issue of load balancing when there is a congestion in the network.

2.7 Related works in Hybrid SDN networks

As stated above, there are many researchers realized that SDN and legacy switches will coexist in a network. Currently, how to deal with hybrid SDN networks becomes another hot topic. A number of approaches[22, 1] are found in the literature to deal with the interconnection of SDN and legacy networks.

Ogrodowczyk et al.[22] proposed designing Hardware Abstraction Layer (HAL) for legacy switch in hybrid SDN networks. Hardware Abstraction Layer resides between controller and legacy switch i.e. the controller can only control the legacy network through HAL. There are two fundamental tasks for HAL.

One is communication with controller, sending and receiving OpenFlow packets, the other is parsing and translating OpenFlow packets such that legacy switch can understand. The idea adds another control layer for the network, however, makes network more complicated. The physical hardware needs to be purchased for upgrading the network is an additional expense for network maintenance.

Floodlight controller[2] proposed a solution[1] to send the special broadcast frames to discover the legacy islands in the hybrid SDN network. When Floodlight sends LLDP frames to discover the network topology, it will set countdown timers for the LLDP packets. If the controller does not receive the LLDP frame before the timer expires, it will send the special broadcast frame to discover possible legacy island between OpenFlow switches. This approach can handle the hybrid SDN network only if there is no loop in the network. Moreover, this solution can not compute paths that span the whole network.

In chapter four of this thesis, we will discuss how to discover and manage hybrid SDN networks without adding new hardware. A solution for loop prevention and path computation in hybrid SDN networks will be proposed.

Chapter 3

Proposed Path Computation and Path Selection Algorithms

This chapter introduces and discusses the path computation and path selection algorithms. First, the overview of the proposed algorithm will be given. Afterwards, the path computation and path selection algorithms will be discussed in great length.

3.1 Overview of Proposed Algorithms

The proposed algorithm consists of path computation algorithm and path selection algorithm. The path computation algorithm uses basic network information to calculate all equal shortest-paths for the network. The output of path computation algorithm and network congestion status will be used as input to the path selection algorithm.

Figure 3.1 presents the flowchart of the proposed algorithm, which consists of following modules:

- Path computation module: Path computation algorithm is implemented in this module. It utilizes the characteristics of data center network to find all shortest paths in the network.
- Path selection module: Path selection algorithm is implemented in this module. It uses the path computation module's result as input to select an appropriate path based on network congestion status.

3.2 Path Computation

3.2.1 Characters in Data Center Networks

Data center networks consists of two-level or three-level trees of switches[7]. A three-tiered network design has a core tier as the root, an aggregation tier in the middle and an edge tier at the leave of the

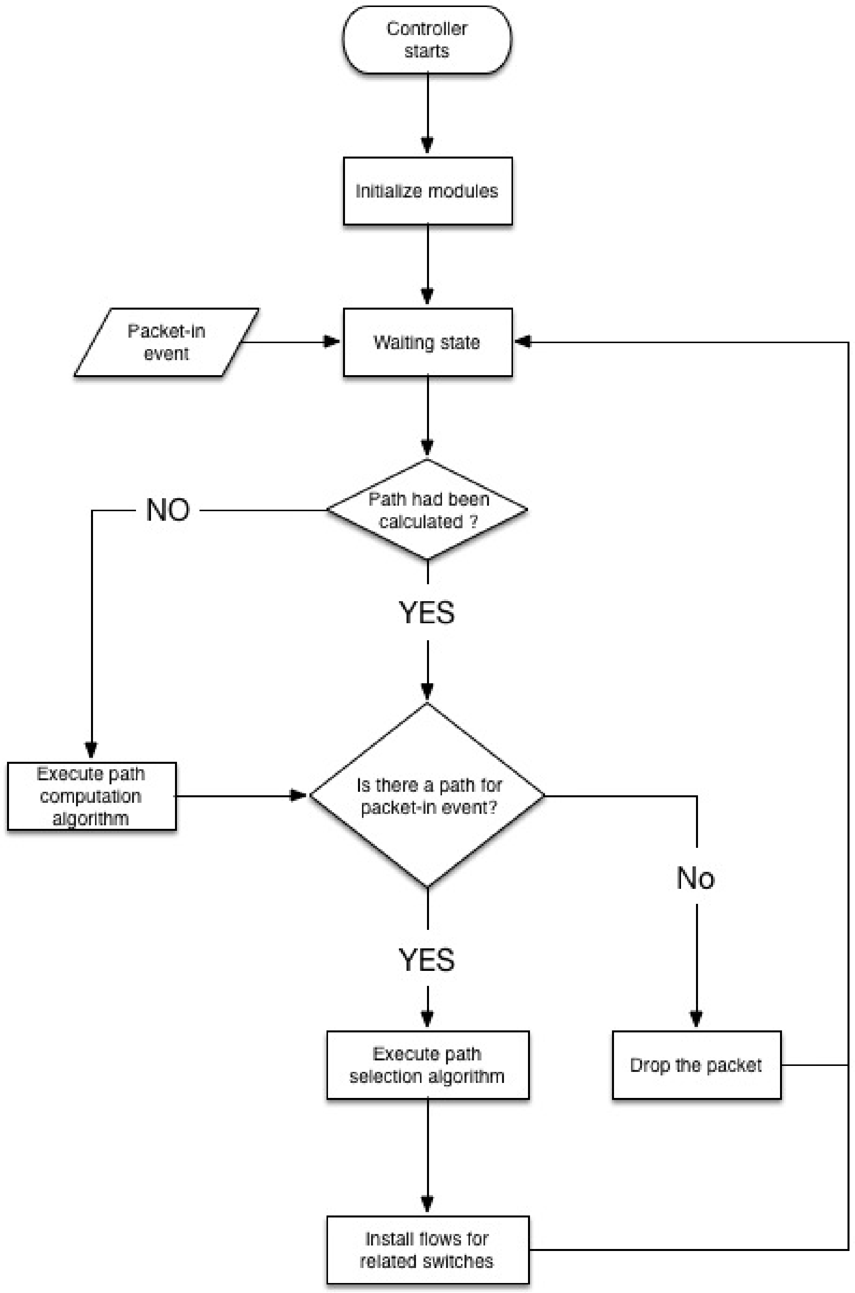

Controller starts
Initialize modules
Packet-in event
Waiting state
Path had been calculated ?
NO
YES
Execute path computation algorithm
Is there a path for packet-in event?
No
YES
Execute path selection algorithm
Drop the packet
Install flows for related switches

Figure 3.1: The workflow of proposed algorithms in chapter 3

tree. A two-tiered network design only has the core and the edge tiers. Since the three-tiered network can support much more hosts at the edge tier and the network topology is more complicated, we will show path computation algorithm for the three-tiered network in the following part. It should be noted that the path computation algorithm can be easily modified to apply for the two-tiered data center network.

In a three-tiered network, all the hosts connect to the edge tier. The main purpose of the data center is to provide services to the external users, so the majority traffic of the data center network is between the hosts (data center servers) and the core switches. There is also minor traffic between the hosts from edge to edge within the data center. Usually, hosts are not connected to the aggregation layer. Therefore, we can assume no traffic starts from or ends at aggregation tier. From the network perspective, there are only two kinds of traffic in the data center network: traffic between core and edge and edge to edge.

The purpose of aggregation tier is to aggregate the traffic of several edge tier switches and forward it to the core tier switches, so the uplink interface bandwidth is usually larger than the downlink interface bandwidth. In addition, data center networks usually use symmetric Ethernet link (the bandwidth is the same in both directions), which makes the graph to be undirected graph.

Dijkstra's and Floyd Warshall algorithms do not take advantages of any of the characteristics described above to compute the paths. In this thesis, we will propose a new path computation algorithm to use these characteristics in order to reduce the processing time.

3.2.2 Path Computation Algorithm

We assume that every switch can be configured as either edge, aggregation or core switch. When the switch establishes a connection with controller, the switch identifies the tier it belongs to the controller. Based on this information, the controller can build an adjacency table with tier information for every switch. Let denote E_i, A_j and C_k an edge switch, an aggregation switch and a core switch. Let denote N_1, N_2, N_3 the complete sets of edge and aggregation switches, respectively. In addition, let be $|X|$ the size of set X. Thus,

$$N_1 = \{E_1, E_2, \cdots, E_{|N_1|}\}$$

$$N_2 = \{A_1, A_2, \cdots, A_{|N_2|}\}$$

$$N_3 = \{C_1, C_2, \cdots, C_{|N_3|}\}$$

Let denote T_e^j and T_c^j as the edge and core layer adjacency table of A_j, and T_a^j as the aggregation layer adjacency table of E_i or C_i . Thus,

$$T_e^j = \{E_1, E_2, \cdots, E_{|T_e^j|}\}$$

$$T_c^j = \{C_1, C_2, \cdots, C_{|T_c^j|}\}$$

$$T_a^i = \{A_1, A_2, \cdots, A_{|T_a^i|}\}$$

Algorithm 3.1 Pseudocode of the path computation algorithm

```
for each $A_i$ in $N_2$ do
    for each $E_j$ in $T_e^i$ of $A_i$ do
        for each $E_k$ in $T_e^i$ of $A_i$ $(k > j)$ do
            get the path from $E_j$–$A_i$–$E_k$ and put the path in $P$
            using $E_j$ and $E_k$ as the input for $H_0$ to get the element $(E_j, E_k)$ in $N_0$
        end for
        for each $C_l$ in $T_c^l$ of $A_i$ do
            get the path from $E_j$–$A_i$–$C_l$ and put the path in $P$
            add element for $R_j$ of $E_j$
        end for
    end for
end for
while $N_0$ is not empty do
    pop first element $(E_m, E_n)$ in $N_0$
    for each $C_p$ in $R_m$ of $E_m$ do
        tag $C_p$ in $L_0$
    end for
    for each $C_q$ in $R_k$ of $E_n$ do
        if $C_q$ is tagged in $L_0$ then
            get a path from $E_m$–$C_q$–$E_n$ and put the path in $P$
        end if
    end for
end while
```

It should be noted that the maximum length of T_e^j should be much smaller than the number of edge layer switches in a three-tirered network.

Let denote N_0 as the set that contains all the unique pairs from N_1.

Let denote H_0 as the hash table whose input are the pairs in N_1, and the output are the address of corresponding elements in N_0.

Let denote R_i as the reachability table for E_i. It contains the core switches that can reach through an aggregation switch. Thus,

$$R_i = \{C_1, C_2, \cdots, C_{|R_i|}\}$$

Let denote L_0 as the list which equals to N_3.

Let denote P as the set to store the results of the path computation algorithm.

Using the above definitions, we have the following pseudo code in Algorithm 3.1 to compute the equal cost shortest paths for all the pairs and store them in P.

At the beginning, we iterate all aggregation switches in aggregation layer. For every aggregation switch A_j, we can find a path from an edge switch E_j to another edge switch E_k through A_j by picking its unique pairs in T_e^j. Once the paths are found, E_j and E_k will be used as the input element in H_0 to find and remove the corresponding element in N_0. After the removal, the remaining elements in N_0 will be the pairs which are not directly connected through aggregation switches. For every aggregation switch A_j, we can iterate core switches C_k in T_c^j, a path from an edge switch E_j to a core C_k through A_j can be found. After finding the path, the C_k will be added in reachability table R_j.

Lastly, we pop the remaining elements in N_0 until N_0 becomes empty. For every pop, we can get a pair of edge switches E_j and E_k, which are not directly connected through aggregation switch. By comparing the R_i and R_k to find overlapped core switch C_j, we can find the path from E_i to E_k through C_j. After the operations listed above, all the paths in the network are found and stored in P. After the calculating, the time complexity of proposed algorithm should be O($N_2 \times N_3$).

3.3 Load Balancing

3.3.1 Load Balancing Background

To perform load balancing, we first compute and store shortest paths for all source/destination pairs. When multiple shortest paths are available for a given source/destination pair, we need to compare them and select an appropriate path.

3.3.2 Path selection Algorithm

The path selection algorithm utilizes the congestion status of the links in the networks. The link congestion status has three possible levels. A level of 0 indicates that there is no congestion, a level of 1 a moderate congestion and a level of 2 a high congestion. Two thresholds, upper and lower thresholds, associated adjacent levels are defined. The upper threshold is used to decide if the congestion status should change from lower congestion level to higher congestion level. Conversely, lower threshold is used to decide if the congestion status should change from higher congestion level to lower congestion level. Let th_u^1 and th_l^1 be the upper and lower thresholds associated with level 0 and level 1, respectively. Thus, the congestion status is changed from level 0 to level 1, if the current congestion value (it will be defined in next paragraph) is greater than th_u^1 , and is changed from level 1 back to level 0 if the current congestion value is lower than th_l^1. Similarly, let th_u^2 and th_l^2 be the upper and lower thresholds associated with level 1 and level 2, respectively. Thus, the congestion status is changed from level 1 to level 2 if the current congestion value is greater than th_u^2 and is changed from level 2 back to level 1 if the current congestion value is smaller than th_l^2.

Since OpenFlow protocol does not allow us to carry customized message, we need to extend OpenFlow packet type to support network congestion status reporting from switch to controller. On the switch, we retrieve the tx_bytes in ofp_port_stats filed every second as data rate to calculate the congestion value. Thus, we define the congestion value of a port as:

$$congestion\ value = Data\ rate/Bandwidth \tag{3.1}$$

We also predefine the thresholds on the switch and compare the congestion value with the appropriate threshold to determine the current congestion status. If the congestion level changed, the switch will send the latest congestion status to the controller proactively. The controller will parse the link congestion status and update congestion information of corresponding paths.

The selection mechanism will then create a list Q, which contains equal cost paths with the same minimum congestion level and minimum congestion weight. The steps used to derive Q is as follows.

On the controller we define

$$P_j = \{l_1, l_2, \cdots, l_N\} \tag{3.2}$$

as a set of N links that form path j and let Con_ibe the congestion level of linkl_i, then the congestion status,P_j^m, of path j is the maximum congestion level of all the links of path j:

$$P_j^m = max(Con_{1,}Con_2, \cdots, Con_N) \tag{3.3}$$

Where Con_i is the congestion level of link i. We also define the congestion weight of path j , P_j^w, as the sum of all the links congestion level in path j:

$$P_j^w = sum(Con_{1,}Con_2, \cdots, Con_n) \tag{3.4}$$

For a given source/destination pairs, let there k be equal cost shortest paths whose congestion levels are $P_1^m, P_2^m, \cdots, P_k^m$, respectively. Among these paths, we define a set S_x such that:

$$S_x = \{x | P_x^m = min(P_1^m, P_2^m, \cdots, P_k^m)\} \tag{3.5}$$

Thus S_x contains all the paths that have the same minimum congestion level. We then select the paths in S_x with lowest congestion weight and put them in set S_y:

$$S_y = \{y | P_y^w = min(P_x^w, x \in S_x)\} \tag{3.6}$$

Finally, the paths in S_y will be put into list Q, and one of the paths in Q will be selected using hashing to serve a new flow request.

When a flow request forwarded to the controller, we will extract the MAC and IP addresses and port numbers in User Datagram Protocol /Transmission Control Protocol (UDP/TCP) for both source and destination, then using OR operation with all source and destination MAC and IP addresses, and port numbers to get the index. The index will be used to select a path in Q. As the source and destination MAC and IP addresses and port numbers are random, the final path selection will be also randomly. This random selection to balance the load in the network. The Figure 3.2 presents the flowchart of path selection algorithm.

3.4 Summary

This chapter describes path computation and path selection algorithms. First, we discuss the network characteristics of data center network. Based on these characteristics, we design a path computation algorithm to find all equal cost shortest paths and store the results. Afterwards, we extend current OpenFlow packet type to let switch sends congestion status to controller proactively. On the controller, we used the result of the path computation algorithm and network congestion status from switches to derive a set of path with equal minimum congestion level and congestion weight. Finally, a path from list Q is selected based on hashing the IP addresses, MAC addresses and port numbers.

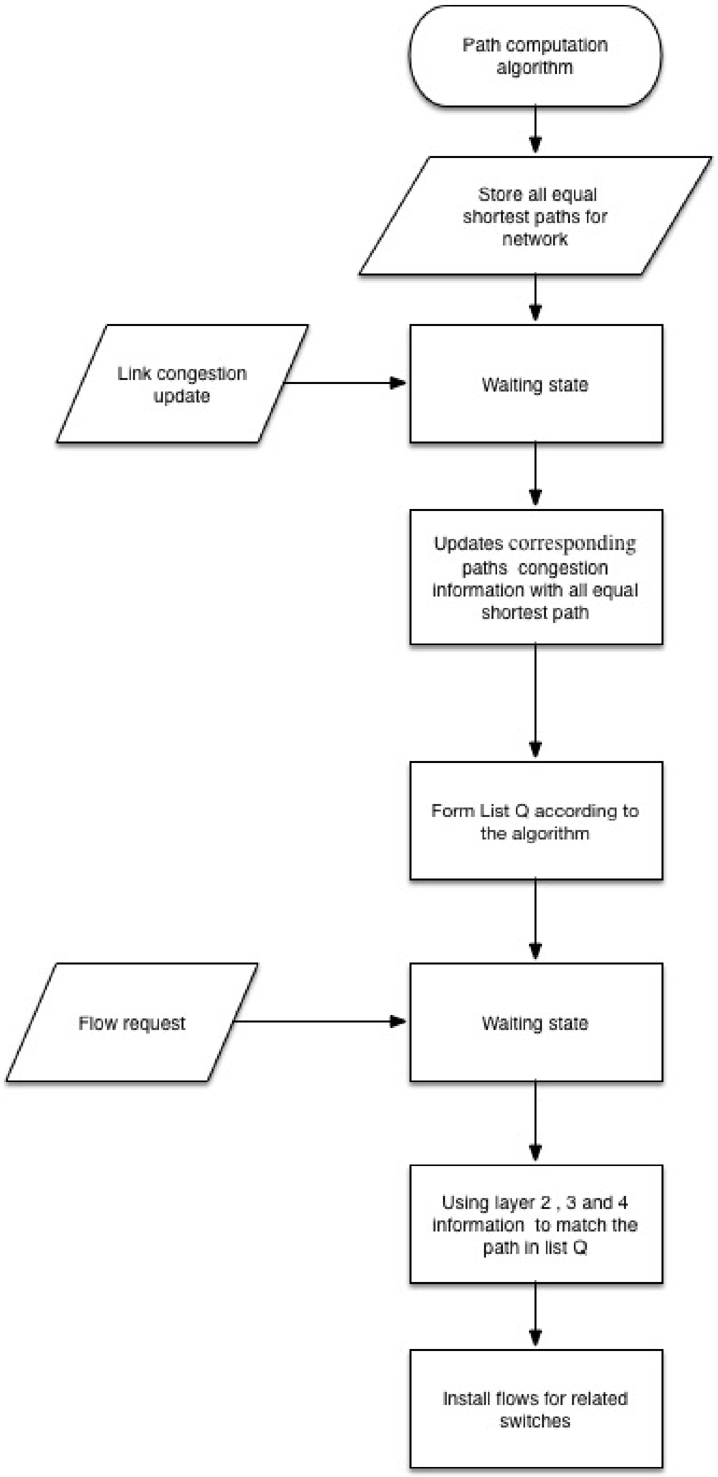

Path computation algorithm
Store all equal shortest paths for network
Link congestion update
Waiting state
Updates corresponding paths congestion information with all equal shortest path
Form List Q according to the algorithm
Flow request
Waiting state
Using layer 2 , 3 and 4 information to match the path in list Q
Install flows for related switches

Figure 3.2: The flowchart of path selection algorithm

Chapter 4

Proposed Solution For Hybrid Networks

This chapter covers the proposed algorithm for SDN hybrid networks. The possible problems are highlighted and the current solution is introduced, since they lay the foundation for the proposed solution.

4.1 Overview of current Hybrid SDN Network Solution

As we discussed in chapter 2, there are two different approaches to handle SDN hybrid networks. One is adding hardware for legacy switches to support OpenFlow protocol. The drawbacks for this approach are extra cost for hardware and the lack of robustness, which means the network still can not work properly when the controller loses OpenFlow connection with switches.

The other approach, used by the Floodlight controller is discovering and managing legacy network on the controller[1]. Floodlight's solution only solves the simplest scenario where there is not any loop among SDN and legacy islands. In addition, the path computation module of Floodlight cannot compute the path that traverses through the SDN islands. Figure 4.1 shows the example hybrid SDN network that Floodlight can handle:

However, when there are loops among islands in the hybrid SDN network, the Floodlight cannot

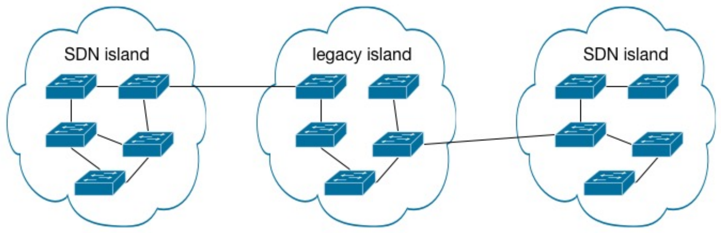


Figure 4.1: A example of hybrid SDN network that Floodlight can handle

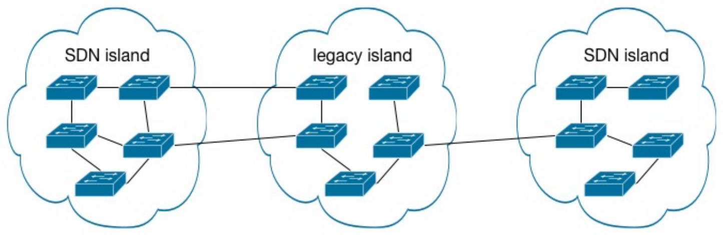


Figure 4.2: Example 1 of hybrid SDN network that Floodlight can not handle

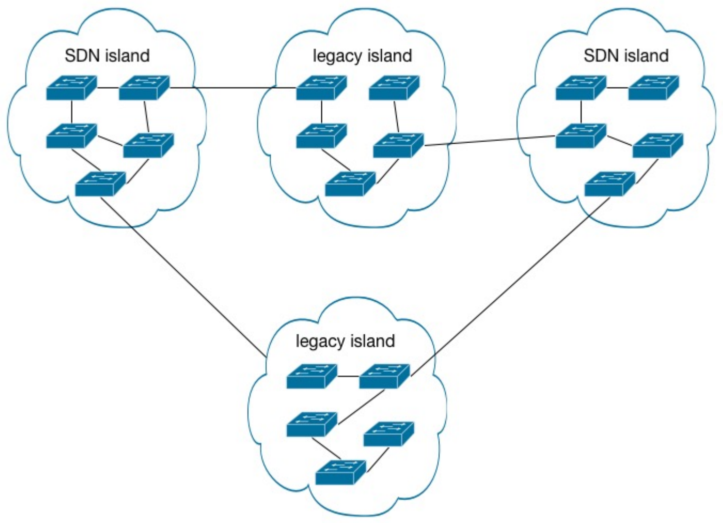


Figure 4.3: Example 2 of hybrid SDN network that Floodlight can not handle

prevent the traffic loop problems. Figure 4.2 and 4.3 show the topology that Floodlight can not handle. The Floodlight does not have any built-in mechanism to avoid the loop between or among islands, and the path computation module only calculates path within SDN island.

The proposed solution in the next section uses the similar approach as Floodlight with enhancements. Essentially, the proposed solution can handle more complicated hybrid SDN network, like Figure 4.2 and 4.3.

4.2 Overview of proposed Solution for hybrid network

The workflow of the proposed solution works as follows: The controller sends special broadcast to discover possible legacy islands in the SDN hybrid network. After the discovery phase, the controller analyzes and configures the border switches of SDN islands to avoid possible loop. Finally, path computation module is used to calculate the path for the whole network. Figure 4.4 shows the flowchart of the proposed solution.

Three modules are involved in proposed solution:

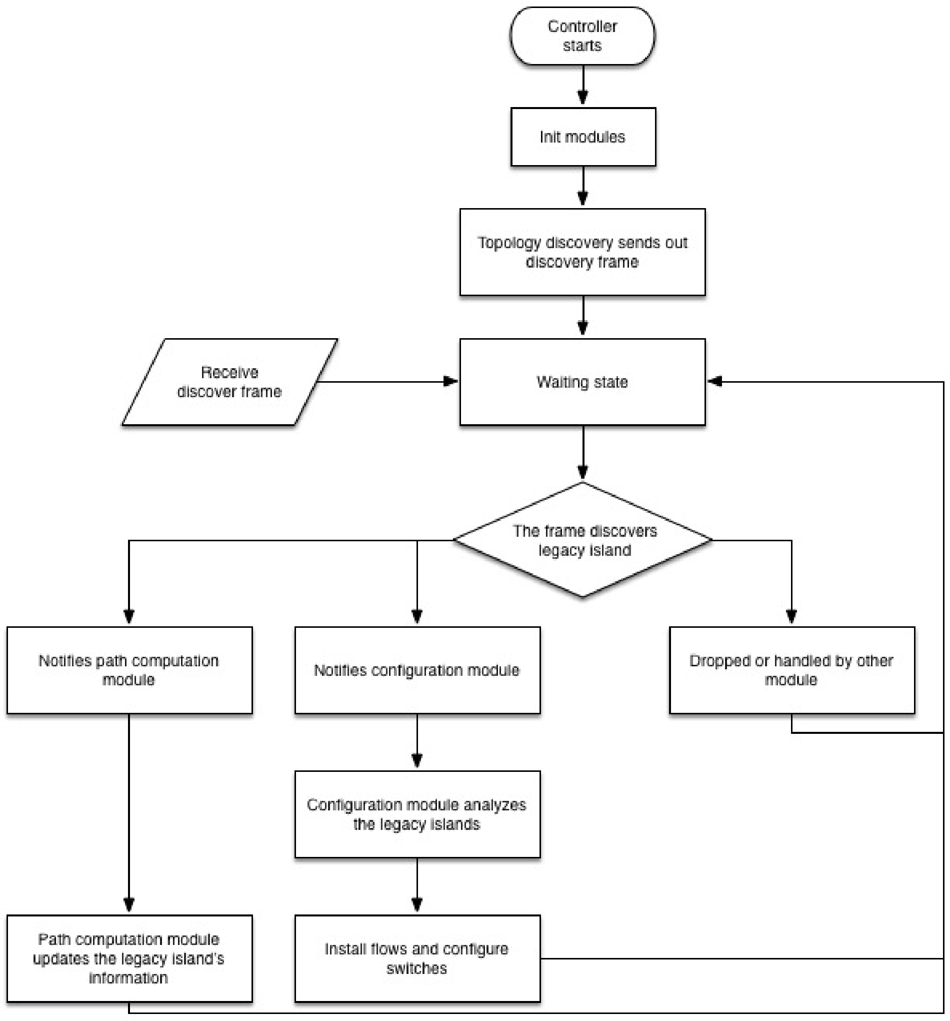

Controller starts
Init modules
Topology discovery sends out discovery frame
Receive discover frame
Waiting state
The frame discovers legacy island
Notifies path computation module
Notifies configuration module
Dropped or handled by other module
Configuration module analyzes the legacy islands
Path computation module updates the legacy island's information
Install flows and configure switches

Figure 4.4: The workflow of proposed solutin for hybrid network

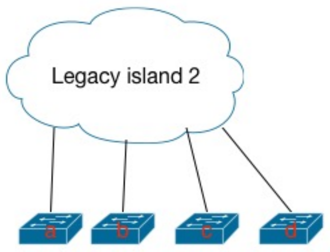


Figure 4.5: The switches a,b,c and d are connected to legacy island 2

- Discovery module: Sending, receiving and parsing special broadcast to discover legacy island.
- Configuration module: Getting notification from discovery module, analyzing the legacy island and configuring the border switches of the SDN islands.
- Path computation module: compute the path both for SDN and legacy switches.

4.3 Proposed Solution for Hybird Network

In the discovery module, whenever a notification of a new interface is received, the controller sends a standard LLDP frame and a special broadcast frame that contains same content as the LLDP frame to the new interface using packet_out event. Since the LLDP frame will be parsed and dropped by any legacy switch, only the broadcast frame can traverse the legacy islands in the network. When the controller discovers that a switch “A” receives a special broadcast frame but not a standard LLDP frame from a sending switch “B”, it can conclude that there must be a legacy island link between the receiving switch and the sending switch. Once a legacy island link is discovered, the discovery module will notify configuration module about the discovery.

Eventually the configuration module will collect all legacy island links. It then analyzes these links to derive all legacy islands and SDN border switches which connected to legacy islands. In our solution, maximum clique algorithm[9, 10, 24] is used to solve this problem. From the output of maximum clique algorithm, we can derive all legacy islands. Let take Figure 4.7 as the example to illustrate the results after the clique algorithm. In the first step, suppose the controller discovers all the legacy island links a-b, (a-b means the sending switch is "a" and the receiving switch is "b", and the other links follow the same name convention) a-c, a-d, b-a, b-c, b-d, c-a, c-b, c-d, d-a, d-b, d-c, e-f, e-g, f-e, f-g, g-f and g-e. Using all the legacy island links as input of the maximum clique algorithm, it can work out that the switches a, b, c and d are connected to the same legacy island, as shown in Figure 4.5, and switches e, f, g are connected to another legacy island.

For every legacy island, we need further analysis for SDN border switches to check which SDN island(es) they belong to. In the example, we need to know whether switches a, b, c and d are in the same SDN island or not. This can be done by just checking if these switches can reach the others using SDN links only. The switches that can be reached from each other through the SDN links belong to

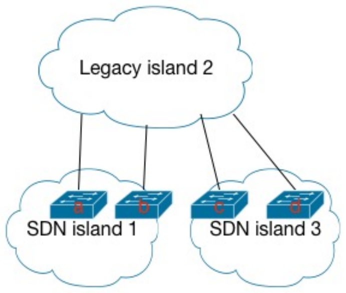


Figure 4.6: a,b are in a group

the same SDN island. Figure 4.6 illustrates that switches a and b and switches c and d belong to their respective SDN islands.

We need to avoid possible loop in the case where a SDN island and a legacy island have multiple direct links. Therefore, for every SDN island, the SDN border switch with the highest dpid is elected as the designated switch to flood/receive traffic to/from the legacy island, the non-designated switch(es), which will not flood traffic to legacy island and drop the traffic, except LLDP frames and the special broadcast frames, received from the legacy island. The interfaces facing legacy islands on the non-designated switches are called non-designated interfaces. Non-designated switches are installed the flow which only accept LLDP and the special broadcast from the non-designated interfaces. In addition, a non-designated interfaces are configured as NO_FLOOD to avoid loop shown in Figure 4.7. In the example, if we assume switch a is elected as the designated switch, then switch b will be configured with no flood traffic to legacy island 2 and only accepting LLDP and the special broadcast frame from legacy island 2.

Same approach is applied to c, d, e, f and g, we assume that c, e and f are the designated switch, then d and g will be configured and installed flows. By this approach, the loops in Figure 4.7 are avoided.

At first glance, it seems that we can utilize non-designated link to setup flow as FabricPath does. However, using the non-designated link to setup flow may cause the MAC address flapping in the legacy island. That causes returning traffic dropped by the non-designated switch from the legacy island back to SDN island. Since the legacy island may flood the frame to the SDN island, the non-designated switch will also receive the flooded flow. On non-designated switch, we have to differentiate whether it is flooded flow or not. The mechanism that we can use is that if the coming flow matches the flow entry on the non-designated switch, it will be the returning traffic from the legacy island. Otherwise, it is the flooded flow from the legacy island, then we need to drop the flow on non-designated switch to prevent the duplicate flow.

Let us take Figure 4.8 as example to demonstrate how the mechanism may fail. We assume h1 setups a flow to h2 by using designated link, then h1 setups another flow to h3 by using non-designated link. This will cause MAC address of h1 flaps from s4 to s6 in the legacy island. When h2 sends the frame back to h1, the frame will get to the s3 according to the MAC tables in the legacy island. However, s3 will drop the packet according to its flow table. In this case, the non-designated switch dropped the

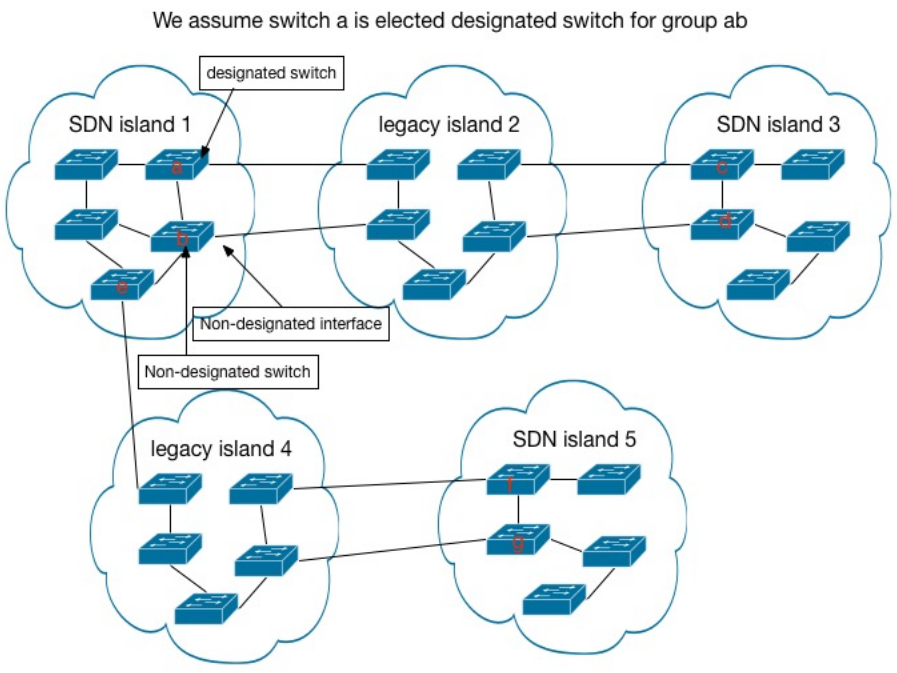


Figure 4.7: Scenario 1 that the proposed solution handles

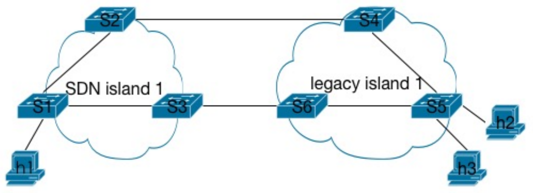


Figure 4.8: The example that we can not use non-desiganated switch

returning flow.

Network loop may also form among the islands as shown in the Figure 4.9 below. To avoid this kind of loop, the SDN and legacy islands are abstracted as nodes in the graph. In this graph, minimum spanning tree algorithm[18] is used to break the loop.

We will use network topology in Figure 4.9 to demonstrate how the spanning tree is applied. Since we already solve the loop problem in the case where a SDN island has many links to a legacy island, we can assume that every SDN island has one link to legacy island. In this situation, we will abstract all SDN islands (islands 1 and3 in the example) and legacy islands (islands 2 and 4) as nodes. By calculating the spanning tree for this graph, one of the links that form the loop will be disabled (a-b link in this example), thus, avoid traffic loop. For similar reason we mentioned above, we will not use the blocked link for non-flood traffic.

In the path computation module, legacy islands need to be abstracted as special switches. Adja-

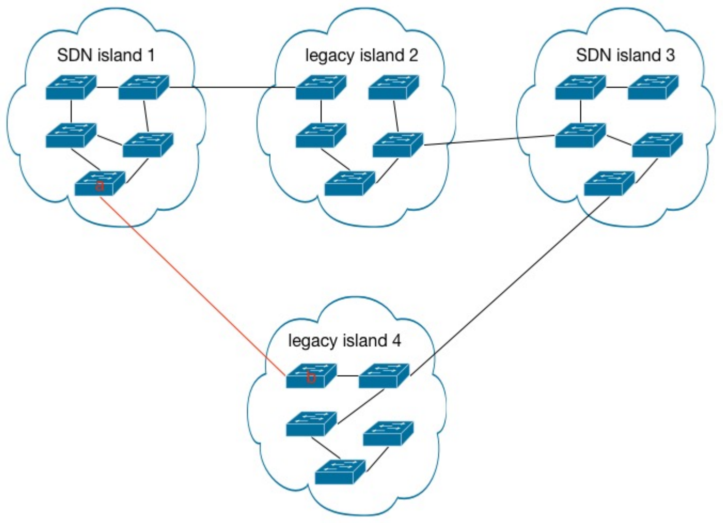


Figure 4.9: Scenario 2 that the proposed solution handles

cency and MAC table for the special switches also need to be maintained. It should be noted that only designated switches will be used to receive and forward the traffic from and to legacy island, so designated switches and legacy islands will consider each other as the adjacency. Then we can use the path computation algorithm for SDN hybrid networks.

4.4 Summary

This chapter has introduced the solution to solve the problems in hybrid SDN network. First, the drawbacks of other solutions dealing with hybrid SDN network are overviewed, and then the proposed solution is explained in detail: Through the special broadcast to detect the legacy island, the configuration module could analyze and configure the switches remotely to avoid the loop in layer 2 network. The modification to path computation module is made to handle legacy island for path computation of the SDN hybrid network.

Chapter 5

Implementation

A prototype of the test network is simulated using Mininet[19], Open vswitch[4] and the POX controller[5]. The POX controller is used to run the proposed algorithms.

5.1 System Environment

We are using the Parallel Desktop as the hypervisor on a 1.7GHz Intel Core i5 CPU machine, with 4GRAM, Mac OS 10.10 Operating System. On the hypervisor, Ubuntu12.04 is used as the operating system, where we run Mininet 2.1 and Open vswitch 1.9.3 as the hosts and switches. The POX 0.4 running on the host machine uses network to communicate with Open vswitch. The OpenFlow version between controller and switch is 1.0.

Mininet Mininet is the network simulator, which can create a virtual network with hosts on a single machine. It runs a collection of hosts, switches and links on Linux Kernel system and uses lightweight virtualization approach to make a single Linux system as a complete network. Figure 5.1 illustrates an OpenFlow network with 2 hosts, 1 switch and 1 controller in the Mininet simulator. The hosts are in separate namespaces and connected to the network using virtual Ethernet. The datapath, which uses Open vswitch in this thesis, provides connectivity for hosts and switches.

Open vsiwtch Open vswitch(OVS) is an open source production-quality implementation of a switch. We can easily modify its source code to achieve our own object. In addition, it can be used as OpenFlow or legacy switch.

POX POX is a SDN controller, which is mainly used for the research. It is a great platform for researchers to develop their own code by Python. It supports the high-level SDN API and virtualization. It can be used on Windows, Linux and Mac OS. In this thesis, we used a third party Python libary networkx[15] when implementing the algorithm.

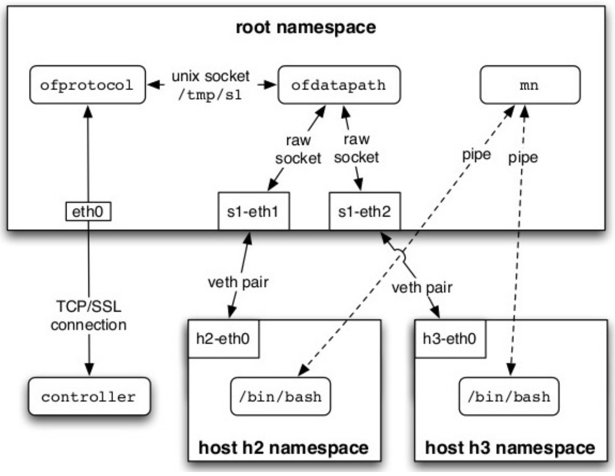


Figure 5.1: The architecture of Mininet

5.2 System Design

The key benefits of this system are to reasonably use the computation and link resources in the network and to increase availability for hybrid SDN network. The whole system is implemented based on the existing POX and Open vswitch architecture.

The built-in modules in POX are used to discover network topology, fundamental network setup and path computation. To achieve high availability for hybrid SDN networks, discovery module is modified to support legacy islands discovery, configuration module is created to analyze and configure the network, and path computation module is modified to support legacy islands.

To achieve efficient utilization of network resource, packet type of OpenFlow protocol is expanded that makes Open vswitch proactively send network congestion status. On the controller, we need to parse the network congestion status and to notify path computation module. In path computation module, the proposed algorithm and network congestion status are used to select the appropriate path.

The whole system is based on original POX framework, using following built-in modules:

- Topology Discovery module: This module is used to discover the link between OpenFlow switches. Packet_out and packet_in events are used to send and parse LLDP packet for discovering and maintaining the connectivity between OpenFlow switches. Then the network topology can be drawn.

- Spanning Tree module: This module is used to calculate the spanning tree for preventing broadcast storm. By latest network topology drawn by topology discovery module, spanning tree can be calculated dynamically and the ports which are not in the spanning are set to be NO_FLOOD to avoid the broadcast storm.

- Path computation module: This module is used to calculate the shortest path for the whole network topology. By latest network topology drawn by topology discovery module, path computation module uses Floyd Warshall algorithm to calculate all pair shortest path for whole network. When packet is forwarded to controller, path computation module will install the flows to corresponding switches for flow forwarding.

The following modifications are made to achieve our goal:

- Topology Discovery: discovering and maintaining legacy islands in the hybrid SDN network. By sending and parsing the special broadcast to discover and maintain possible legacy islands. Through the special broadcast, we can discover how legacy islands connect to SDN islands. Finally, the whole hybrid SDN network topology can be drawn.
- Configuration module: analyzing the legacy islands and configuring the SDN border switches. The detail workflow had been described in Chapter 4 above.
- parsing module: parsing new OpenFlow packets which controller can get network congestion status from switches.
- path computation module: computing shortest paths for the whole network and selecting the appropriate path based on network congestion status. After the configuration by the configuration module, path computation module will use the algorithm which mentioned in chapter three to compute and store the all pair shortest paths. When the path computation module receives new flow installation request, it will select a path from the paths stored in list Q.

5.3 Summary

This chapter has ended with the implementation details and the system architecture for chapter three and four. The experimental results evaluation will be presented in next chapter.

Chapter 6

Experimental results and Analysis

This chapter investigates the performance of the proposed path computation and path selection algorithms and tests the validity of the proposed hybrid SDN network solution. The proposed path computation algorithm will be compared with classic path computation algorithms. In addition, the proposed path selection algorithm will be compared with the hash approach and periodic query approach (hash and periodic query will be talked specifically in 6.4). Finally, the proposed solution of SDN hybrid network will be verified.

The following outlines what we will study in this chapter:

- Comparison of the network path computation times among the proposed path computation algorithm, Dijkstra's and Floyd Warshall algorithms for data center network with the fat-tree topology.
- Comparison of the controller setup a new flow for the hosts when using proposed path selection algorithm versus hash and the periodic query.
- The network throughput among proposed path selection algorithm, hash and periodic query.
- How controller discovers and manages SDN hybrid network.

6.1 Experiment Parameters and Environment

The software used for simulations and experiments are: POX controller, Open vswitch and Mininet in Ubuntu 12.0.4 The client-server environment is setup by Iperf[3].

6.2 Performance Metrics

To analyze the performance of the algorithms, the following metrics are needed to be considered: algorithm convergence time, host end-to-end delay and throughput utilization.

Algorithm 6.1 The Floyd–Warshall algorithm

```
Let dist be a $|V| \times |V|$ array of minimum distances initialized to $\infty$
for each vertex $v$ do
    dist[$v$][$v$] $\leftarrow 0$
end for
for each edge $(u, v)$ do
    dist[$u$][$v$] $\leftarrow$ w$(u, v)$
end for
for $k$ from 1 to $|V|$ do
    for $i$ from 1 to $|V|$ do
        for $j$ from 1 to $|V|$ do
            if dist[$i$][$j$] $>$ dist[$i$][$k$]+dist[$k$][$j$] then
                dist[$i$][$j$] $\leftarrow$ dist[$i$][$k$]+dist[$k$][$j$]
            end if
        end for
    end for
end for
```

- Algorithm convergence time: This metric measures the time that takes to compute all equal cost shortest paths for the whole network.
- New flow setup time: This metric measures the delay for the first round trip between hosts.
- Throughput utilization: This metric measures the network throughput by sending UDP traffic from client to server and measuring the average transfer rate from server side.

6.3 Path computation performance comparison

The algorithms to be compared with the proposed algorithm are Dijkstra's and Floyd Warshall algorithms. In next subsection, the overview of Dijkstra's and Floyd Warshall algorithms will be presented, and then the test scenario and the numerical results will be discussed.

6.3.1 The Floyd-Warshall algorithm

The Floyd-Warshall algorithm compares all possible paths through the graph for each pair of nodes and stores the shortest one. The idea of this algorithm is to find the best intermediate between source and destination. That is why, in the main loop of the algorithm, all possible intermediate nodes are tried with source and destination. The algorithm mainly consists of a triple loop where every loop need to go through all nodes in the graph. Apparently, The time complexity of the Floyd-Warshall algorithm is O($|V|^3$)(where V is number of the nodes). Algorithm 6.1 shows the pseudo code of Floyd–Warshall algorithm.

Algorithm 6.2 The Dijkstra's algorithm

```
Require: $source, Graph$
dist[$source$] $\leftarrow 0$
create vertex set $Q$
for each vertex $v$ in $Graph$ do
    if $v \neq$ source then
        dist[$v$] $\leftarrow \infty$
        prev[$v$] $\leftarrow$ UNDEFINED
    end if
    $Q$.add_with_priority($v$,dist[$v$])
end for
while $Q$ is not empty do
    $u \leftarrow Q$.extract_min()
    for each neighbor $v$ of $u$ do
        alt = dist[$u$] + length($u$,$v$)
        if alt < dist[$v$] then
            dist[$v$] $\leftarrow$ alt
            prev[$v$] $\leftarrow u$
            Q.decrease_priority($v$,alt)
        end if
    end for
end while
return dist[],prev[]
```

6.3.2 The Dijkstra's algorithm

Dijkstra's algorithm is a classic graph theory algorithm for searching the shortest paths from a single source to all the destinations. It is widely used in network routing protocol, notably Intermediate System to Intermediate System(IS-IS) and OSPF. The original algorithm does not use min-queue to optimize the performance, then the implementation based on a min-priority queue implemented by Fibonacci heap[14] was published in 1984. Through Fibonacci heap, the min_queue operations, which are add_with_priority(), decrease_priority() and extract_min() in this algorithm, can be performed in lower time complexity. By this optimization the time complexity of Dijkstra's algorithm is $O(|E| + |V| \log |V|)$ (where E and V are the number of edges and nodes). The Algorithm 6.2 shows the pseudo code of Dijkstra's algorithm with Fibonacci heap optimization.

6.3.3 Test Case

In this section, we will compare convergence time of the proposed algorithm, Dijkstra's algorithm and Floyd Warshall algorithm for same network shortest path calculation. As Dijkstra's algorithm can only compute the single source to all destinations shortest paths, we will use Dijkstra's algorithm to compute the shortest paths for nodes on edge and core layers.

In this test case, the three-tier fat-tree data center network will be used to compare these three algorithms. In the network, switches on the core and aggregation layers will all connect to each other

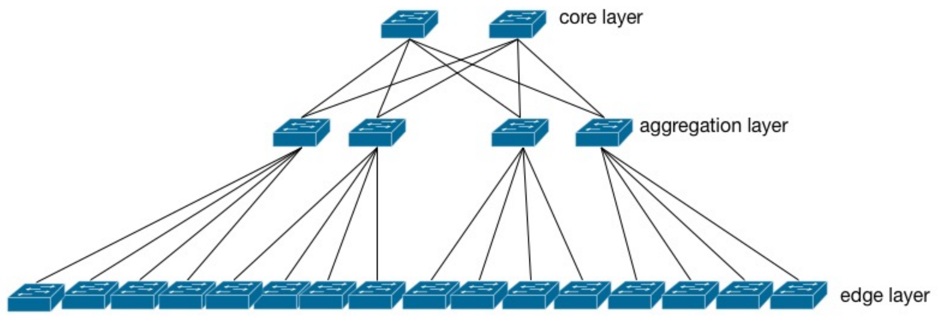


Figure 6.1: The network topology of initial stage

	Core switches number	Aggregation switches number	Edge switches number	Switches number in total
Initial stage	2	4	16	22
1st change	4	8	32	44
2nd change	8	16	64	88
3rd change	16	32	128	176

Table 6.1: Network changes during the test case

and every aggregation node will connect a fixed numbers edge nodes. Figure 6.1 shows the network topology of initial stage and the 6.1 shows how the network scale change in this test case.The Figure 6.2 shows the result of three algorithms.

From the Figure 6.2, we can see the proposed algorithm outperforms the other two algorithms since the time complexity of proposed algorithm is smaller than the other two. At the beginning, when the network is relatively small, the advantage of the proposed algorithm is limited. As the network scales up, the proposed algorithm always requires much less time to compute the paths for the whole network though we used Fibonacci heap as min queue to optimize the Dijkstra's algorithm. Therefore, our study demonstrates that the proposed algorithm can save computation resource on the controller for popular data center network topology.

6.4 Path selection performance comparison

As stated above, the path selection algorithm will follow the shortest path computation. In this part, we will compare proposed path selection algorithm with other two approaches (hash and periodic query) to evaluate the flow setup time and the network throughput. In next subsections, the other two algorithms will be introduced. Then the simulation results will be presented.

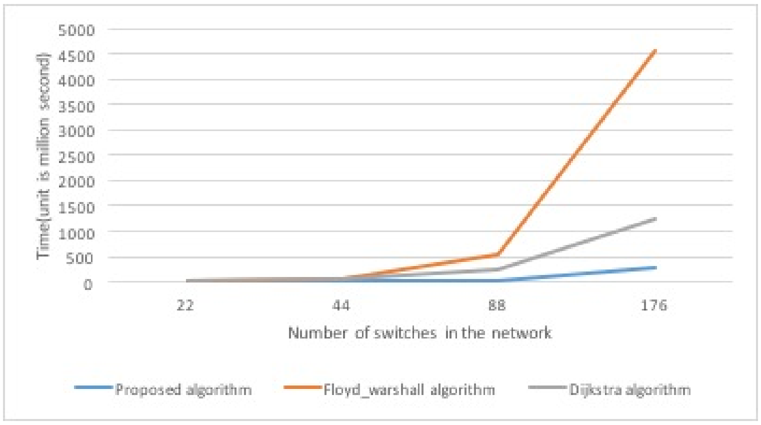


Figure 6.2: Performance comparison among three algorithms

6.4.1 Hash

Hash works pretty straightforwardly. When a flow request is forwarded to the controller, the controller will select a path from all the equal cost shortest paths. The selection will use a hash function with the source and destination MAC addresses, IP addresses and port numbers in UDP/TCP as input to derive a hash value. The hash value will be mapped to one of the shortest path using the modulo operation.

6.4.2 Periodic Query

This algorithm uses OpenFlow protocol to obtain the current traffic information by querying the switch statistics, every second. The differences of tx_bytes for every second are used to evaluated the congestion level. The algorithm will then select the path with the least congestion. As the controller can get precise statistic from the switch and forward the flow to the least congestion path, this algorithm can perform a better load balancing than the proposed algorithm where the congestion measurement is not as precise (only three congestion levels). However, the proposed algorithm requires much less overhead for it does not need to poll all the switches periodically.

6.4.3 Test Case

In this test case, we create the network topology in Figure 6.3 to simulate multiple equal cost shortest paths between clients and server. We will use ping command and Iperf to test the performance of the three algorithms.

New flow setup time comparison In this case, we use ping command to trigger the controller to setup the flow and measure the round trip delay. The ping command is issued from a client to the server, and the idle_timeout and hard_timeout for flows are set to 1 second to make sure that the flow can be timeout from the switches when the next ping packet arrives. We ran the simulation 10 times

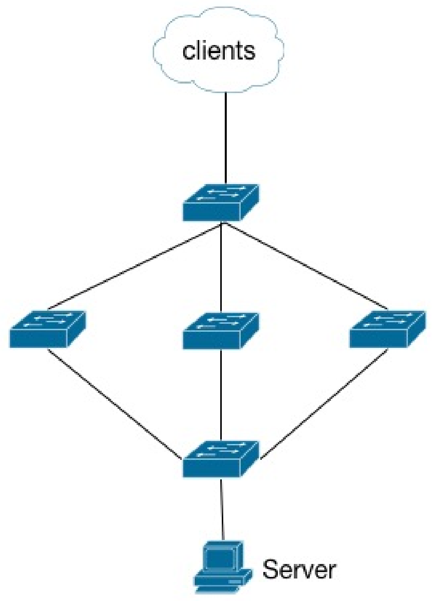


Figure 6.3: Network topology of path selection for test case

ping size	3000 bytes
Ping command repeat times	100 times for every simulation

Table 6.2: The parameters used for ping command

and measured the average delay of the ping command in each run. The parameters used in the ping simulation is given in Table 6.2:

From the Figure 6.4 we can see the average delay for the proposed algorithm and hash outperforms the periodic query approach around 20 percent. The periodic query approach suffers from the fact that the controller needs to query the traffic statistics from every switch and update the network congestion status every second. The combined operation does impact the controller processing the new flow requests.

Throughput comparison In this section, the bandwidth of the link between server and switch will be set to 100Mbps, and bandwidth of the other links to be 10Mbps. We will use 10 clients to send UDP traffic to the server. On every client, we use Iperf to generate UDP traffic with 50 flows for 500 seconds, and the flows will be issued from clients randomly in first 30 seconds. Since there are three equal cost shortest paths and the bandwidth of the links is 10Mbps, the total network bandwidth is 30Mbps. Table 6.3 shows the initial stage and how the traffic changes in every simulation.

On the switch, we set the upper thresholds for moderate and high congestion to be 35% and 70%, respectively. Figure 6.5 shows the overall throughput utilization, which is measured by the overall average transfer rate from the server side.

From the Figure 6.5, we can see that when the traffic load is relatively light, all three approaches

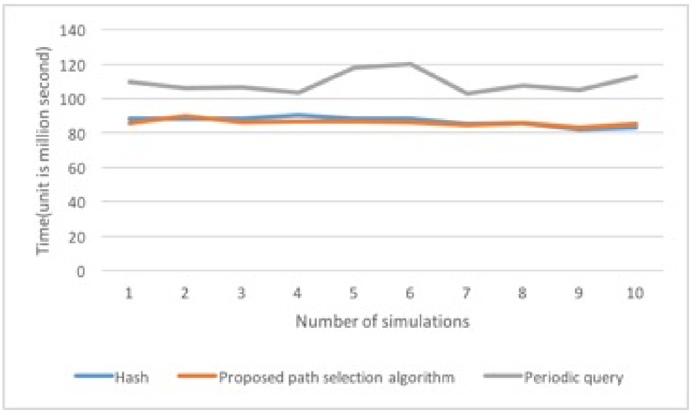


Figure 6.4: Comparison of average delay

	Bandwidth per flow	Bandwidth for all flows	Traffic load/Bandwidth
Initial stage	0.02Mbps	10Mbps	30%
1st change	0.03Mbps	15Mbps	50%
2nd change	0.04Mbps	20Mbps	67%
3rd change	0.05Mbps	25Mbps	83%
4th change	0.06Mbps	30Mbps	100%

Table 6.3: The flows changes during the test case

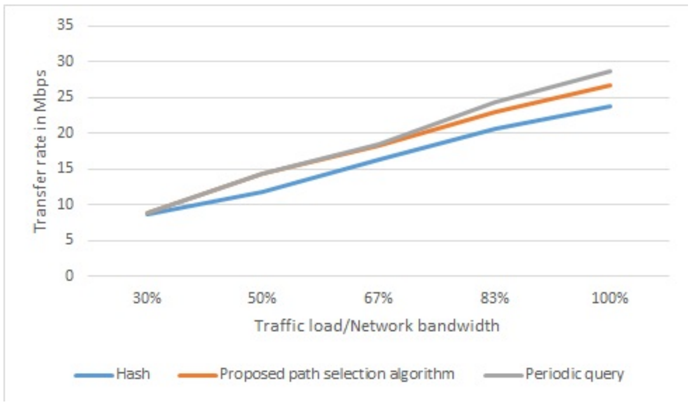


Figure 6.5: Comparison of throughput

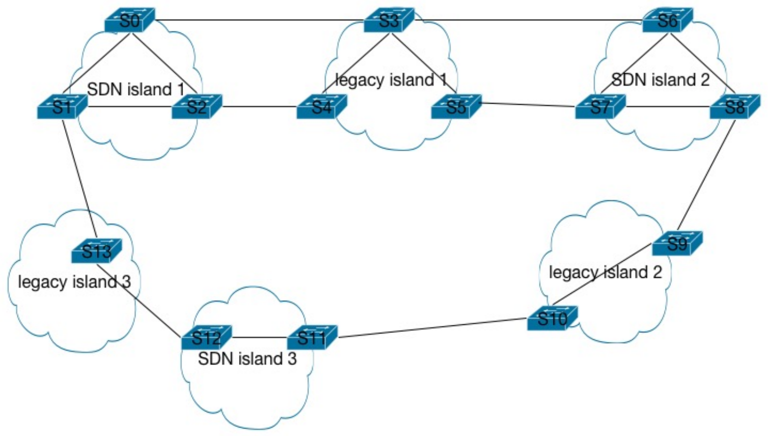


Figure 6.6: The test scenario for proposed solution

perform similarly. As the traffic increases, since the hash approach is not aware of the congestion status in the network, the transfer rate becomes lower than the other two approaches. When the traffic load reaches the high congestion region beyond the upper threshold, all three paths are in the same congestion status, and the proposed algorithm will start to choose the path randomly as the hash does. It should be noted that the proposed algorithm is still a better load balancer than hash when the congestion status does not exceed the high congestion upper threshold. Since in the periodic query algorithm, the controller can still get the precise congestion status by querying the switch statistics to choose the least congestion path, it performs better than the proposed algorithm in high congestion status. However, the observable difference in throughput performance between the proposed algorithm and periodic query only happens in high congestion cases. Moreover, the difference in performance is still relatively small, and if the network scale is big enough, the controller may suffer from the numerous statistics query exchanges with switches, which can decrease the network throughput.

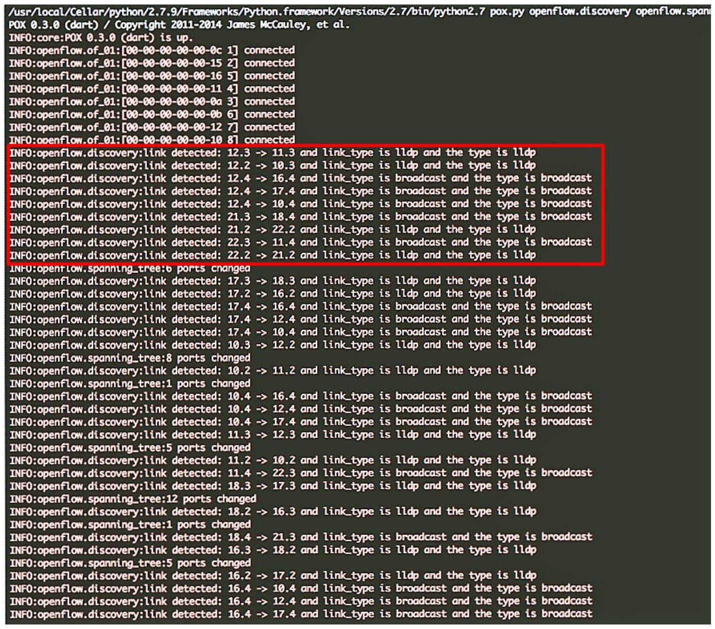
In conclusion, the proposed algorithm can achieve better load balancing than hash approach in terms of throughput. We have to admit that when the network is in high congestion status, the periodic query approach can outperform the proposed algorithm a little. However, the proposed algorithm has relatively much smaller overhead.

6.5 Verification For Proposed Solution of hybird SDN networks

6.5.1 Test case

The Mininet, Open vswitch and POX are used to setup network topology in Figure 6.6. After launching the network, s3, s4, s5, s9, s10 and s13 will fall back to legacy switches. At initial state, there are three legacy islands and three SDN islands. Legacy switches usually run STP to avoid layer 2 loop, so we assume legacy islands are loop free networks.

In the above network topology, legacy island 1 forms local loops with SDN islands 1 and 2. In



```
/usr/local/Cellar/python/2.7.9/Frameworks/Python.framework/Versions/2.7/bin/python2.7 pox.py openflow.discovery openflow.span
POX 0.3.0 (dart) / Copyright 2011-2014 James McCauley, et al.
INFO:core:POX 0.3.0 (dart) is up.
INFO:openflow.of_01:[00-00-00-00-00-0c 1] connected
INFO:openflow.of_01:[00-00-00-00-00-15 2] connected
INFO:openflow.of_01:[00-00-00-00-00-16 5] connected
INFO:openflow.of_01:[00-00-00-00-00-11 4] connected
INFO:openflow.of_01:[00-00-00-00-00-0a 3] connected
INFO:openflow.of_01:[00-00-00-00-00-0b 6] connected
INFO:openflow.of_01:[00-00-00-00-00-12 7] connected
INFO:openflow.of_01:[00-00-00-00-00-10 8] connected
INFO:openflow.discovery:link detected: 12.3 -> 11.3 and link_type is lldp and the type is lldp
INFO:openflow.discovery:link detected: 12.2 -> 10.3 and link_type is lldp and the type is lldp
INFO:openflow.discovery:link detected: 12.4 -> 16.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 12.4 -> 17.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 12.4 -> 10.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 21.3 -> 18.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 21.2 -> 22.2 and link_type is lldp and the type is lldp
INFO:openflow.discovery:link detected: 22.3 -> 11.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 22.2 -> 21.2 and link_type is lldp and the type is lldp
INFO:openflow.spanning_tree:6 ports changed
INFO:openflow.discovery:link detected: 17.3 -> 18.3 and link_type is lldp and the type is lldp
INFO:openflow.discovery:link detected: 17.2 -> 16.2 and link_type is lldp and the type is lldp
INFO:openflow.discovery:link detected: 17.4 -> 16.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 17.4 -> 12.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 17.4 -> 10.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 10.3 -> 12.2 and link_type is lldp and the type is lldp
INFO:openflow.spanning_tree:8 ports changed
INFO:openflow.discovery:link detected: 10.2 -> 11.2 and link_type is lldp and the type is lldp
INFO:openflow.spanning_tree:1 ports changed
INFO:openflow.discovery:link detected: 10.4 -> 16.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 10.4 -> 12.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 10.4 -> 17.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 11.3 -> 12.3 and link_type is lldp and the type is lldp
INFO:openflow.spanning_tree:5 ports changed
INFO:openflow.discovery:link detected: 11.2 -> 10.2 and link_type is lldp and the type is lldp
INFO:openflow.discovery:link detected: 11.4 -> 22.3 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 18.3 -> 17.3 and link_type is lldp and the type is lldp
INFO:openflow.spanning_tree:12 ports changed
INFO:openflow.discovery:link detected: 18.2 -> 16.3 and link_type is lldp and the type is lldp
INFO:openflow.spanning_tree:1 ports changed
INFO:openflow.discovery:link detected: 18.4 -> 21.3 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 16.3 -> 18.2 and link_type is lldp and the type is lldp
INFO:openflow.spanning_tree:5 ports changed
INFO:openflow.discovery:link detected: 16.2 -> 17.2 and link_type is lldp and the type is lldp
INFO:openflow.discovery:link detected: 16.4 -> 10.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 16.4 -> 12.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 16.4 -> 17.4 and link_type is broadcast and the type is broadcast
```

Figure 6.7: The controller discovers the legacy island link and LLDP link

addition, the topology also has a network loop that spans the whole network. For test purpose, we connect a host with every switch.

The dpids of s0, s1, . . . , s13 are 10, 11, . . . , 23. The host's name convention will follow the switch name. For example, the host connected to s0 is h0. The host MAC addresses 00:00:00:00:00:01, 00:00:00:00:00:02, . . . , 00:00:00:00:00:0d will be assigned to h0, h1, . . . , h13.

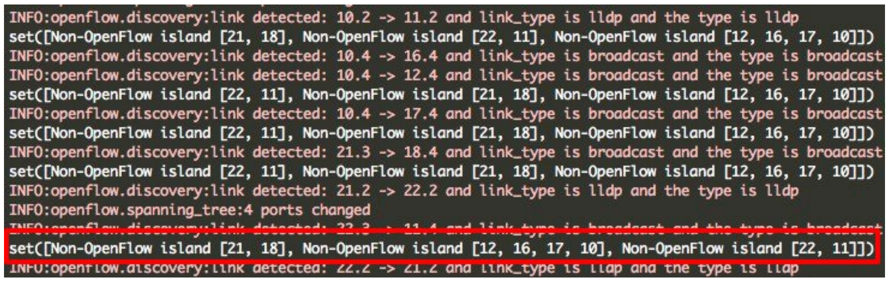
6.5.2 Result and analysis for Hybird network solution

At the beginning, the topology discovery starts sending and parsing LLDP and the special broadcast. The switches will receive both LLDP and the special broadcast within the same SDN island; on the other hand, the SDN border switches will only receive the special broadcast, because the LLDP frame are dropped by switches belong to the legacy islands.

Subsequently, the topology discovery module will notify that a legacy island is detected between SDN border switches. From the highlights in Figure 6.7, we can see that the controller reported the SDN links (referred as LLDP link in the figure) are found within SDN islands and the legacy island links (referred as broadcast link in the figure) are found at the SDN islands borders.

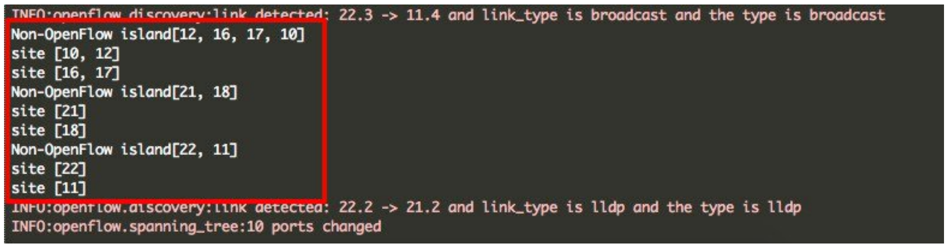
In the configuration module, the notifications of legacy islands will be received. The configuration module can classify legacy islands according to the notifications. The highlights in Figure 6.8 shows the controller finally had analyzed all legacy islands (referred as Non-Openflow island in the figure) and the number behind the legacy islands are the dpids of the SDN borders switches.

For every legacy island, we need to analyze all the SDN border switches to determine if they connect to the same SDN island. For the switches connect to the same legacy island, they will be put in the



```
INFO:openflow.discovery:link detected: 10.2 -> 11.2 and link_type is lldp and the type is lldp
set([Non-OpenFlow island [21, 18], Non-OpenFlow island [22, 11], Non-OpenFlow island [12, 16, 17, 10]])
INFO:openflow.discovery:link detected: 10.4 -> 16.4 and link_type is broadcast and the type is broadcast
INFO:openflow.discovery:link detected: 10.4 -> 12.4 and link_type is broadcast and the type is broadcast
set([Non-OpenFlow island [22, 11], Non-OpenFlow island [21, 18], Non-OpenFlow island [12, 16, 17, 10]])
INFO:openflow.discovery:link detected: 10.4 -> 17.4 and link_type is broadcast and the type is broadcast
set([Non-OpenFlow island [22, 11], Non-OpenFlow island [21, 18], Non-OpenFlow island [12, 16, 17, 10]])
INFO:openflow.discovery:link detected: 21.3 -> 18.4 and link_type is broadcast and the type is broadcast
set([Non-OpenFlow island [22, 11], Non-OpenFlow island [21, 18], Non-OpenFlow island [12, 16, 17, 10]])
INFO:openflow.discovery:link detected: 21.2 -> 22.2 and link_type is lldp and the type is lldp
INFO:openflow.spanning_tree:4 ports changed
INFO:openflow.discovery:link detected: 22.3 -> 11.4 and link_type is broadcast and the type is broadcast
set([Non-OpenFlow island [21, 18], Non-OpenFlow island [12, 16, 17, 10], Non-OpenFlow island [22, 11]])
INFO:openflow.discovery:link detected: 22.2 -> 21.2 and link_type is lldp and the type is lldp
```

Figure 6.8: The controller analyzed all legacy islands



```
INFO:openflow.discovery:link detected: 22.3 -> 11.4 and link_type is broadcast and the type is broadcast
Non-OpenFlow island[12, 16, 17, 10]
site [10, 12]
site [16, 17]
Non-OpenFlow island[21, 18]
site [21]
site [18]
Non-OpenFlow island[22, 11]
site [22]
site [11]
INFO:openflow.discovery:link detected: 22.2 -> 21.2 and link_type is lldp and the type is lldp
INFO:openflow.spanning_tree:10 ports changed
```

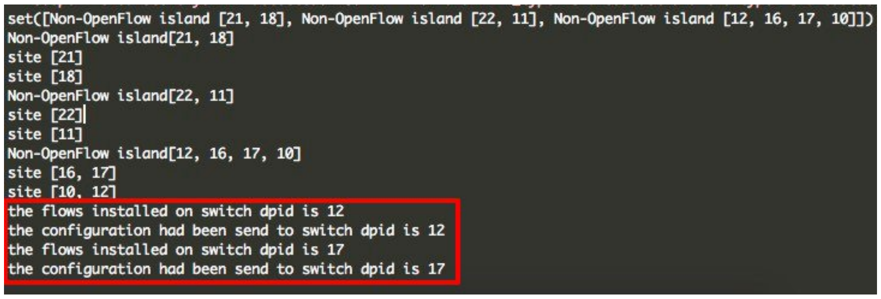
Figure 6.9: The controller discover the groups of legacy islands

same group. As highlights in Figure 6.9 presents, the groups (referred as sites in the figure) of the legacy islands are determined.

For every group of SDN island, the SDN border switch with the highest dpid is elected as the designated switch to flood unknown unicast and multicast, then the configuration module will install flow to non-designated switches and configure its interfaces accordingly. The highlights in Figure 6.10 shows the operations to designated and non-designated switches had been finished.

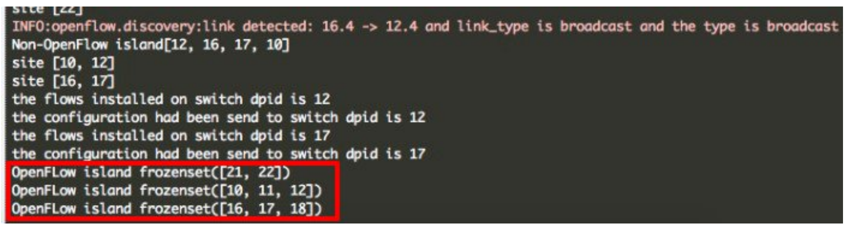
By analyzing LLDP frame, the SDN islands are listed. As Figure 6.11 shows, all SDN islands 1,2 and 3(referred as OpenFlow islands in the figure) are identified. The number behind the SDN islands are the dpids of SDN border switches in the SDN island.

After SDN and legacy islands are determined, the minimal spanning tree will be calculated. As Figure 6.12 shows, the interface 3 of switch 22 will be blocked to prevent the loop.



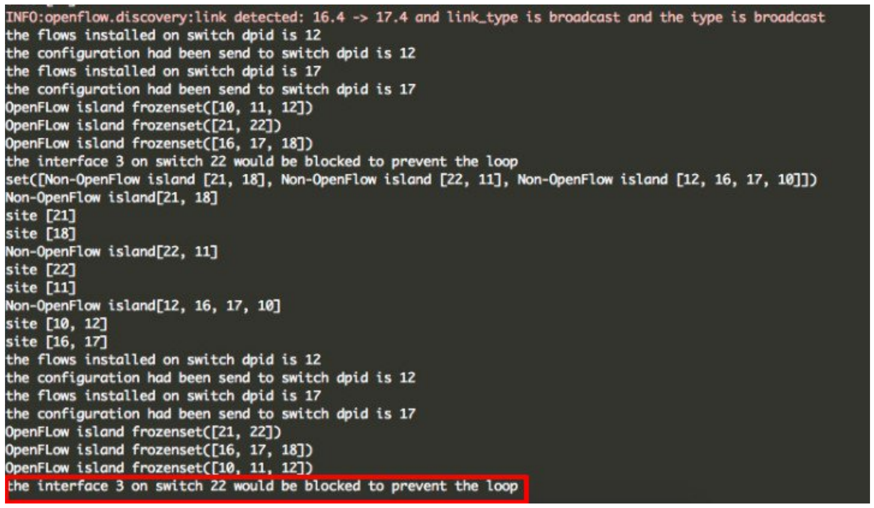
```
set([Non-OpenFlow island [21, 18], Non-OpenFlow island [22, 11], Non-OpenFlow island [12, 16, 17, 10]])
Non-OpenFlow island[21, 18]
site [21]
site [18]
Non-OpenFlow island[22, 11]
site [22]
site [11]
Non-OpenFlow island[12, 16, 17, 10]
site [16, 17]
site [10, 12]
the flows installed on switch dpid is 12
the configuration had been send to switch dpid is 12
the flows installed on switch dpid is 17
the configuration had been send to switch dpid is 17
```

Figure 6.10: The configuration and flows are installed to the switches



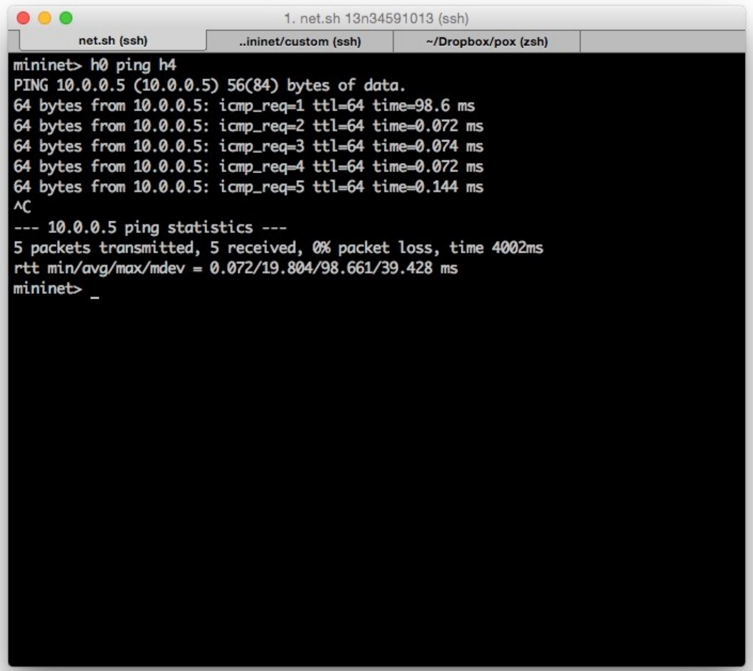
```
site [22]
INFO:openflow.discovery:link detected: 16.4 -> 12.4 and link_type is broadcast and the type is broadcast
Non-OpenFlow island[12, 16, 17, 10]
site [10, 12]
site [16, 17]
the flows installed on switch dpid is 12
the configuration had been send to switch dpid is 12
the flows installed on switch dpid is 17
the configuration had been send to switch dpid is 17
OpenFLow island frozenset([21, 22])
OpenFLow island frozenset([10, 11, 12])
OpenFLow island frozenset([16, 17, 18])
```

Figure 6.11: The controller achieves SDN islands



```
INFO:openflow.discovery:link detected: 16.4 -> 17.4 and link_type is broadcast and the type is broadcast
the flows installed on switch dpid is 12
the configuration had been send to switch dpid is 12
the flows installed on switch dpid is 17
the configuration had been send to switch dpid is 17
OpenFLow island frozenset([10, 11, 12])
OpenFLow island frozenset([21, 22])
OpenFLow island frozenset([16, 17, 18])
the interface 3 on switch 22 would be blocked to prevent the loop
set([Non-OpenFlow island [21, 18], Non-OpenFlow island [22, 11], Non-OpenFlow island [12, 16, 17, 10]])
Non-OpenFlow island[21, 18]
site [21]
site [18]
Non-OpenFlow island[22, 11]
site [22]
site [11]
Non-OpenFlow island[12, 16, 17, 10]
site [10, 12]
site [16, 17]
the flows installed on switch dpid is 12
the configuration had been send to switch dpid is 12
the flows installed on switch dpid is 17
the configuration had been send to switch dpid is 17
OpenFLow island frozenset([21, 22])
OpenFLow island frozenset([16, 17, 18])
OpenFLow island frozenset([10, 11, 12])
the interface 3 on switch 22 would be blocked to prevent the loop
```

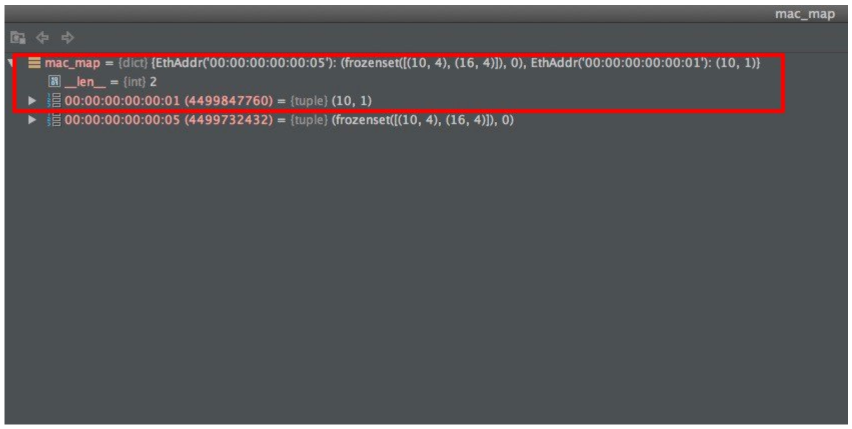
Figure 6.12: The spanning tree is calculated and the interface is blocked to prevent the loop



```
mininet> h0 ping h4
PING 10.0.0.5 (10.0.0.5) 56(84) bytes of data.
64 bytes from 10.0.0.5: icmp_req=1 ttl=64 time=98.6 ms
64 bytes from 10.0.0.5: icmp_req=2 ttl=64 time=0.072 ms
64 bytes from 10.0.0.5: icmp_req=3 ttl=64 time=0.074 ms
64 bytes from 10.0.0.5: icmp_req=4 ttl=64 time=0.072 ms
64 bytes from 10.0.0.5: icmp_req=5 ttl=64 time=0.144 ms
^C
--- 10.0.0.5 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4002ms
rtt min/avg/max/mdev = 0.072/19.804/98.661/39.428 ms
mininet> _
```

Figure 6.13: The between h0 and h4 ping works

In the path computation module, the legacy islands will be abstracted as a single switch, and then the path computation algorithm can calculate the complete network shortest path. Finally, we issue the ping command from h0 to h4. From the Figure 6.13 and 6.14 , we see the ping can work properly and the MAC table of legacy islands can learn MAC address like other switches.



```
mac_map = {dict} {EthAddr('00:00:00:00:00:05'): (frozenset([(10, 4), (16, 4)]), 0), EthAddr('00:00:00:00:00:01'): (10, 1)}
    __len__ = {int} 2
  ▶ 00:00:00:00:00:01 (4499847760) = {tuple} (10, 1)
  ▶ 00:00:00:00:00:05 (4499732432) = {tuple} (frozenset([(10, 4), (16, 4)]), 0)
```

Figure 6.14: The MAC table for legacy island

Chapter 7

Conclusion and Future Work

SDN has been a new approach for network operators to control the network via a central controller. Currently, OpenFlow protocol becomes the most promising southbound interface between controller and switch to exchange network information. In recent years, SDN is moving from pure research area to the industry. Many vendors have published their own SDN solutions to the public. It indicates SDN enters upon the implementation stage.

With the emergence of SDN, application layer services of legacy networks, like load balancing and firewall, can be replaced from the middle boxes to the applications resided on the controller. This thesis focuses on the load balancing in popular data center networks. For load balancing purpose, we proposed a path computation algorithm utilized the characteristics of data center networks to compute all equal shortest paths to save computation resource, and then we proposed a path selection algorithm to select an appropriate path according to the network congestion status. From the simulation in chapter 6, we can see that the path computation algorithm beats the classic Dijkstra's and Floyd algorithm in terms of processing time. The path selection algorithm can achieve better load balancing than hash approach without adding numerous overhead as the periodic query does. As future works, the measurement of the congestion status can be implemented on the switch in a more accurate way. We used tx_byte per second on switch to evaluate the congestion status of an interface. However, the queue length of an interface is a better parameter for us to measure the congestion status. It requires much more modifications in kernel space for Openvswitch source codes.

Problems in hybrid SDN networks will be noticed when people start to upgrade the legacy networks with SDN networks. In this thesis, we evaluated strengths and weaknesses of two different approaches for hybrid SDN networks, and proposed our solution to handle the possible loops in hybrid SDN networks. Through proposed solution, the controller can avoid the possible loops and compute the path among SDN and legacy islands. The demonstration in chapter 6 also validates the proposed solution.

Using blocked interfaces on SDN border switches in path computation module will be the future work for proposed solution in hybrid SDN networks. Blocking the non-designated interfaces to solve the possible loops can be optimized by blocking flooding traffic on non-designated interfaces and using non-designated interfaces to send and receive non-flooding traffic. We did not implement this in the thesis, because some scenarios may cause MAC flapping problem in the hybrid SDN network. If we can find a way to address the problem, then we can utilize the blocked interfaces to compute a better path for certain requests and increase the network throughput.

Bibliography

[1] Floodlight approach to handle hybrid SDN network. `https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Supported+Topologies`.

[2] Floodlight OpenFlow Controller. `http://www.projectfloodlight.org/floodlight/`.

[3] iPerf - The TCP, UDP and SCTP network bandwidth measurement tool. `https://iperf.fr/`.

[4] Open vSwitch. `http://openvswitch.org/`.

[5] POX Wiki. `https://openflow.stanford.edu/display/ONL/POX+Wiki`.

[6] Software-Defined Networking Architecture. `https://www.opennetworking.org/sdn-resources/sdn-definition`.

[7] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63, 2008.

[8] Murat Alanyali and Bruce Hajek. Analysis of simple algorithms for dynamic load balancing. *Mathematics of Operations Research*, 22(4):840–871, 1997.

[9] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[10] Frédéric Cazals and Chinmay Karande. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, 407(1):564–568, 2008.

[11] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[12] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN: An Intellectual History of Programmable Networks. *ACM Sigcomm Computer Communication*, 44(2):87–98, 2014.

[13] Robert W Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[14] M.L. Fredman and R.E. Tarjan. Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms. *25th Annual Symposium on Foundations of Computer Science, 1984.*, 34(3):596–615, 1984.

[15] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, volume 836, pages 11—-15, 2008.

[16] Nikhil Handigol and Srinivasan Seetharaman. Plug-n-Serve: Load-balancing web traffic using OpenFlow. *Acm Sigcomm*, 2009.

[17] Jehn-ruey Jiang, Hsin-wen Huang, Ji-hau Liao, and Szu-yuan Chen. Extending Dijkstra's Shortest Path Algorithm for Software Defined Networking. In *Proc. of the 16th Asia-Pacific Network Operations and Management Symposium*, 2014.

[18] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[19] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[20] Yu Li and Deng Pan. OpenFlow based Load Balancing for Fat-Tree Networks with Multipath Support. In *Proc. IEEE International Conference on Communications*, 2013.

[21] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[22] Lukasz Ogrodowczyk, Bartosz Belter, Artur Binczewski, Krzysztof Dombek, Artur Juszczyk, Damian Parniewicz, Eduardo Jacob, and Jon Matias. Hardware Abstraction Layer for non-OpenFlow capable devices. In *TERENA Networking Conference*, 2014.

[23] Open Networking Foundation. OpenFlow Switch Specification Version 1.5.0 [White paper]. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf`.

[24] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28–42, 2006.

[25] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-Based Server Load Balancing Gone Wild Into the Wild : Core Ideas. *Hot-ICE'11 Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and servicesworks and services*, page 12, 2011.