DELAY-AWARE ELEPHANT FLOW REROUTING IN SOFTWARE-DEFINED NETWORKING (SDN)

HNIN THIRI ZAW

UNIVERSITY OF COMPUTER STUDIES, YANGON

SEPTEMBER, 2019

Delay-Aware Elephant Flow Rerouting in Software-Defined Networking (SDN)

Hnin Thiri Zaw

University of Computer Studies, Yangon

A thesis submitted to the University of Computer Studies, Yangon in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

September, 2019

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research and has not been submitted for a higher degree to any other University or Institution.

…..…… ……..……

Date Hnin Thiri Zaw

ACKNOWLEDGEMENTS

I would like to thank Ministry of Education for giving me the opportunity to study PhD Courses by providing support that allowed me to perform the research in University of Computer Studies, Yangon, one of the best universities in Myanmar.

I would like to express very special thanks to Dr. Mie Mie Thet Thwin, the Rector, the University of Computer Studies, Yangon, for allowing me to develop this thesis and giving me general support during period of my study.

I would like to express my gratitude to my supervisor, Dr. Aung Htein Maw, Professor, University of Information Technology for his helpful guidance and suggestions throughout my PhD research. Working with him has been my most valuable experience, broadening and enriching my research.

I would like to express my appreciation to Dr. Teck Chaw Ling, Associate Professor from University of Malaya, Malaysia for his kind assistance, valuable discussion, practical advice and insightful comments along the path to graduation.

I would like to acknowledge and special thanks to Dr. Thandar Phyu, Director of Technology, ATG Company Ltd. for her constructive comments that help to improve this thesis.

I would like to extend many special thanks also to Dr. Khine Moe Nwe, Course-coordinator of PhD 9th Batch, for her support and encouragement throughout all stages of the research.

I would like to express special thanks to Daw Aye Aye Khine, Associate Professor, Head of English Department for her patience and valuable suggestions which help to improve the thesis writing.

I further would like to thank Dr. Zin May Aye and all my colleagues from the Cisco Lab for being good friends and sharing every time their honest opinion concerning my work.

Last but not least, I would like to express my appreciation to my parents, my younger brother and sister, and my husband for always being there for me and supporting me without boundaries through all these years. None of this would be possible without family.

ABSTRACT

Conventional tree topology networks and single shortest path routing will create congestion along overloaded links. Therefore, load-balanced path selection is necessary to reduce congestion and enhance the performance of the application. In order to solve network congestion problem and to balance the network load, Equal Cost Multipath (ECMP) algorithm is used to handle traffic flows. However, ECMP algorithm lacks the ability to schedule the traffic flow dynamically. Moreover, ECMP does not consider any current network and traffic conditions.

Software-Defined Networking (SDN) is an advanced network infrastructure offering cost-effectiveness and high interoperability for end-user control and network programmability in networks. The primary difference between a conventional networking and SDN networking is decoupling data plane and control plane in SDN based networks. In SDN infrastructure, the whole network is centrally managed by a dedicated controller, which interacts with network switches using OpenFlow protocol. The network congestion detection and dynamic scheduling mechanisms can be implemented by using OpenFlow protocol in SDN infrastructure.

Traffic Engineering (TE) in SDN is a critical application that is attracted tremendous amount of research to find optimized routing decision based on analysis of network and traffic conditions. Following traffic engineering in SDN, most of literature reviews differentiate the traffic flows in networks: small-volume (short-lived) flow which is referred to as mice flows and large-volume (long-lived) flow which is referred to as elephant flow. Among these two types of flows, elephant flows can affect the network performance due to consuming large amount of resources. Therefore, differentiation types of flows and scheduling elephant flows are important issues in TE.

A large number of elephant flow scheduling approaches in traffic engineering have been proposed. At first, various examples of existing traffic measuring and rerouting methods are analyzed, and then their advantages and disadvantages are compared, based on a number of significant criteria such as the improvement of network throughput, the reduction of network delay and packet loss. Recent research on traffic engineering has focused on collecting flow statistics, byte counts and bandwidth utilization; however, a majority of the solutions does not address delay-related issues.

This dissertation presents a study towards an elephant flow scheduling approach for various types of traffic such as TCP, UDP, HTTP and FTP. To this end, a new traffic management method is proposed, that is, Delay-Aware Elephant Flow Rerouting (DAEFR), aiming to minimize the static ECMP's flow collision problem and network congestion, thereby optimizing network throughput and, reducing flow completion time and packet loss. Since the proposed method is flow-based rerouting method, the risk of packet reordering can also be reduced without additional network overhead. The proposed design for dynamic multipath-based traffic management approach in the SDN comprises five main tasks: detecting elephant flow, computing shortest paths, estimating end-to-end delay and bandwidth utilization, calculating least cost path, and rerouting traffic flow from the ongoing path to the uncongested path.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

CHAPTER 1
INTRODUCTION

The network administration has to consider many challenges, such as network policies (Access Control Lists (ACL) or rights management), security (suspected threats) and traffic engineering (Quality of Service). Nowadays, the nature of network management is demanding more dynamics because of Bring-Your-Own-Device (BYOD) policies, Firewalls, Intrusion Detection Systems (IDS) and Cloud Computing. Networks, however, are lag in behind and still choosing old-fashioned technologies with specific interfaces for manufacturer. Managing the networks is therefore complex and tending to cause errors. Designed on the concept of centralized management in traditional network, Software-Defined Network (SDN) solves these issues through separating the data plane and the control plane.

Today's privately owned or public sector data centers offer a variety of applications and cloud-based services. As their traffic characteristics differ significantly, it is a difficult task to handle traffic properly for the underlying network structures in various network situations. Network congestion is a critical problem in networks with large volumes of traffic. The network link is overloaded when it transmits the traffic more than its capacity. At this time, the packets are delayed or lost, thereby reducing the overall network performance. This eventually reduces network application’s or service’s performance and leads to unsatisfactory experiences for end-users. Traffic Engineering (TE) systems that implement load-balancing functionalities try to achieve this by computing optimal routing paths for traffic flows. Aiming to achieve this by computing optimum routing paths for the traffic streams, TE systems implement load-balancing functionalities.

1.1. Problems and Motivations

A commonly used routing strategy is the Equal-Cost Multi-Path (ECMP) [35] routing, which per-hops calculates and randomly allocates multiple best paths to the destination. As the flow-to-path mapping does not take into consideration of the flow sizes, congestion situations are still taking place with high volume flows, which are called elephant flows, since they can be sent over the same network links. Furthermore,

the forwarding paths are static and ECMP cannot therefore react to changing traffic features. In particular, traditional network architectures are not appropriate for developing advanced TE systems because a number of the desired characteristics are missing. No network entity can readily collect and aggregate statistical information from all network devices to form a global network view that allows a TE app to understand the network current situation and simplify routing path calculations. In addition, in traditional networks, the routing behavior is rather static and can not be programmatically modified in the short run. This makes reacting to changing traffic characteristics practically impossible. The Software-Defined Networking architecture concept has all the above features. SDN enables the network application at the control layer to gather information from network devices and alter the network's routing behavior dynamically. Since SDN has been appropriate infrastructure for TE, numerous different TE systems for the prevention of congestion in the SDNs have already been developed. Most of the studies on traffic management has focused on bandwidth utilization (current load of the network); however, they do not consider network latency (end-to-end delay) problem. When the network has high latency, data transmission time will take a long time. Long data transmission time causes bottlenecks in the network nodes. Therefore, the proposed Delay-Aware Elephant Flow Rerouting (DAEFR) mainly focuses not only bandwidth utilization but also network latency to tackle the congestion problem. In this dissertation, the new traffic management method has been proposed which redirects the elephant flows from the ongoing path to the least cost path by measuring end-to-end delay and bandwidth utilization. There are two main tasks in the proposed approach. The first task is to differentiate elephant flow and mice flow. Second, when the new flow becomes elephant flow, it is rerouted to the least cost path. Otherwise the route decision for new flow is made by reactive forwarding method which is a default application in ONOS [66] controller.

1.2. Objectives

The objectives of this dissertation are as follows:

- To design an effective traffic re-scheduling scheme in SDN by estimating end-to-end delays and bandwidth utilization in order to avoid link congestion

- To improve network throughput and to reduce flow completion time (FCT) by rerouting elephant flows, which have an impact on network performance over a period of time, from ongoing path to least cost path

1.3. Contributions

Two main contributions are provided by the proposed work presented in this dissertation. The first is a detailed analysis of existing traffic management methods, which is implemented in SDN environment, which supports useful research information in this field such as traffic measurement, traffic management, problems of performance, analysis, and comparison in previous works of the existing traffic rerouting methods. The second contribution is a new traffic engineering method, Delay-Aware Elephant Flow Management (DAEFR), which can optimize the network throughput and reduce the flow completion time (FCT) effectively by handling elephant flows based on current traffic load and end-to-end delay of available paths.

1.4. Structure of Dissertation

The dissertation structure is organized as follows:

Chapter 2 presents why Software-Defined Networking has been emerged and analyze the current existing SDN TE solutions.

Chapter 3 explains the background of Software-Defined Networking (SDN) and components of multipath routing in SDN.

Chapter 4 deals with the proposed DAEFR architecture which consists of elephant flow detection, end-to-end delay and load utilization measurement, and dynamic elephant flow rerouting components.

Chapter 5 describes the hardware and software requirements for demonstration prototype. Moreover, this chapter covers the technical choices and the detailed implementation of proposed traffic rerouting application using ONOS controller.

Chapter 6 discusses the results with experimental scenarios provided by demonstration prototype.

Chapter 7 concludes with the advantages and disadvantages of the proposed DAEFR and remarks some future work following this dissertation.

CHAPTER 2
LITERATURE REVIEW

This chapter firstly identifies the problems of traditional network and presents why Software-Defined Networking (SDN) has been emerged. Then, traffic engineering in SDN is explained in terms of traffic measurement and management. It also describes and analyzes the existing traffic measurement and management methods with their internal functions.

2.1. Limitations of Conventional Networking

Conventional TCP/IP network nodes are linked to CLIs, configuration files and GUIs specific to the vendor and the device. This forces operators to gain insight into the quirks of each device or management interface if they want to manage a heterogeneous system. Due to the user interface types available, network configuration often needs a separate configuration of each switch and router across multiple UIs. Furthermore, the UI of a single switch only gives an administrator a view of the switch's status instead of the entire network. This leaves network-wide behaviors to be defined by side effects from each device's behaviors. Although cumbersome and highly error-prone, this network configuration approach per device is normally the only method available to an administrator.

The current Internet's capacity is quickly getting inadequate to accommodate the huge bulks of traffic models provided by new techniques and services (e.g. mobile equipment, server virtualization, cloud services, big data) generated by numerous users, sensors and applications. The increasing Big Data in data centers requires high network capacity and network scaling. Network devices become more complex to support these needs. In addition, it would be difficult and time-consuming for administrators to configure individual devices because of small network changes, such as adding or writing anything to modify a device. Many multivendor switches and routers, ACLs, should be reconfigured that can cause inconsistency and errors. Existing multi-tiered networks with tree structured static Ethernet switches are not appropriate for dynamic computing and storage requirements in current and future companies, campuses, carrier environments and data centers. In the meanwhile, new network infrastructures are required to be supplied energy saving, reliability and high performance. It should also

enhance the scalability and robustness with the effective creation and provision of diverse, high quality of network services (QoS). Due to their limited capabilities, existing network equipment cannot fulfill these requirements.

In addition, today's protocols are often isolated and are designed to address a particular problem with no basic abstractions. Without abstraction, network engineers also need to set up many thousands of network equipment and protocols for network policy implementation and support of new services. This limitation prevents the implementation of consistent QoS, security and other strategies. As many thousands of network elements are added, which need to be managed and configured, networks become much more complex. These network elements have their control and data forwarding layers, closed in boxes as shown in Figure 2.1. Thus, a few outside interfaces only (for example, packet transmission) are standardized; but all internal flexibility is obscured. Absence of standard open interfaces bounds network operators' capability to customize and improve hardware and software in their network environments. A new network equipment structure is thus necessary, which decouples the router's transmission and control planes into forwarding elements and control elements dynamically. In the Section 2.1.1, the detailed limitations of existing protocol such as (such as Spanning Tree Protocol) are highlighted.

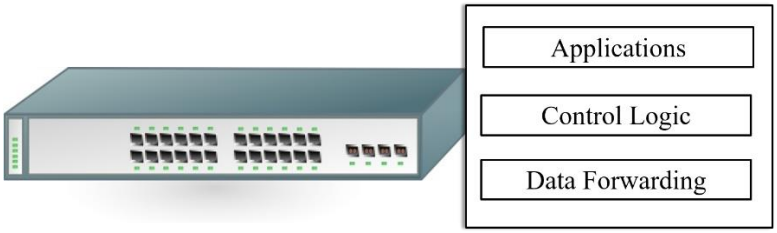

Figure 2.1 Conventional Network Element

2.1.1. Spanning Tree Protocol (STP)

Spanning Tree (SPT) is the common link-layer medium for various applications from communication companies to cloud carriers to corporation to everyday users. The improved Rapid Spanning Tree Protocol (RSTP) [87] and Spanning Tree Protocol (STP) can avoid loop in traditional Ethernet network. To be a single spanning tree connects the whole network without any cycles, the network elements (such as switches) within the same segment have to agree on which links will be use and collaborate in STP and RSTP. To avoid network loops, the basic concept of STP is blocking the links. Therefore, SPT chooses the single path for all traffic in Ethernet

network even though the network has the multiple redundant links. So, the path selection of SPT can cause the network-wide congestion. For this reason, traditional Ethernet network cannot take the advantage of redundant links without any special configuration.

RSTP was modified to the Multiple Spanning Tree Protocol (MSTP) [40] to solve the disadvantages of single spanning tree. MSTP allows to group and map Virtual LAN (VLAN) into different multiple spanning tree instance (MSTI). When the link break down, one of the spanning tree instances can associate each VLAN in the network. Accordingly, MSTP can enhance the network reliability and availability. The incoming network traffic load can be distributed over concurrent links. However, in MSTI, participants are still subject to the time when STP service is not repaired and available. So VLANs may be severely impacted while some VLANs may have little impacts. This may happen if no physical partitioning was caused and the redundant links do not exist within network. Because of using non-tree topologies, without essential planning and complicated manual configuration, some of the redundant links are usually left unused although traditional Ethernet networks are commonly designed with redundant links. This challenge cause the performance degradation in Ethernet network.

2.1.2. Traffic Engineering (TE) in IP-Based Network

Traffic Engineering is a primary mechanism for maximizing data network performance because the data transmission actions can be dynamically evaluated, predicted and controlled. To fulfill traffic engineering requirements in conventional networks, the simple shortest path routing is used by setting up link weights with the current network graph and traffic demand. Most of IP networks install the protocols, for example, Open Shortest Path First (OSPF) protocol [46] that chooses according to static weights, which are assigned to each link. OSPF protocol is used to update link weights and assemble an entire view of the network graph within autonomous system (AS). To forward packets of traffic flow to the subsequent hop along its route, every router has to selects the least hop paths and generates a forwarding table. Shortest path routing is, however, not enough for most network applications to achieve traffic engineering main objectives. Further, the modifications of static link weight can also have an effect on the routing styles of the whole set of traffic flows. The accurate view of network graph and having timely are very important to select appropriate link

weights. Therefore, to collect network information such as network device status and to track the network topology, the Simple Network Management Protocol (SNMP) is used in conventional network. However, the network operator needs to estimate the traffic size between routers.

In Equal-Cost Multipath (ECMP) routing [35], the network may have more than one shortest paths between routers. The OSPF protocol cannot decide the route to forward the traffic flow among multiple shortest paths because shortest path protocols use static link weight. Dynamically increasing traffic criteria, the volumes of traffic diverge realistically over time, the shortest path routing related to the cost of the links is not the efficient solution to improve network performance. Moreover, collecting the accurate network parameters is hard. To balance the traffic load, ECMP is based on hashed values of the packet headers to distribute the incoming traffic among equal paths. The limitation of ECMP is that it uses static hashed mapping of traffic flows to paths and neither network parameters are taken into account (such as bandwidth and delay) or traffic volume. With ECMP, two long flows can collide being routed over the same output port of the switch initiating hotspots in the network. As a result, the network throughput degradation and the latency of path travel across the congested link increases, increasing the flow completion time (FCT).

2.2. Software-Defined Network (SDN)

With the introduction of the Software-Defined Network (SDN) concept, network operators hope to address some of the aforementioned shortcomings. SDN has two fundamental properties in its core: (1) SDN separates the control plane from the data plane, and (2) SDN can consolidate the control plane in a way in which a single software control program can control multiple data plane elements. A third is derived from these two properties: programmability. As providing centrally controlled views of a complete network infrastructure, new features can be added if necessary.

SDN controller, also simply called the network operating system (NOS), is the crucial element of the Software-Defined Network. SDN observes an abstract global network view and updates it in order to enable the central control logic. Bottom-up detailed description of the SDN abbreviations is as Forwarding Devices, Data Plane, Southbound Interface, Control Plane, Northbound Interface, and Management Plane [47]. In Software-Defined Networking, the network operating system centrally controls the entire network through Southbound Interface. SDN enables network engineers to

develop features and functionality in contrast to the conventional network once the network is deployed. New functionality can be added to software without reconfiguring the switches.

SDN has been extensively researched. Mappings of virtual abstractions and physical architectures are generally provided one-to-many, e.g. a switch abstraction is implemented using a distributed set of physical devices. In [28], it shows that the traditional per packet consistency is demonstrated to be insufficient for correct one-to-many mapping and introducing new directions for the research: (1) investigating in better techniques for mapping or (2) limiting the SDN API to provide only safe abstractions. In [11], it develops a distributed architecture that offers a global network view of such applications with high availability and scale-up on physically distributed servers. In [61], it extends SDN infrastructure to aware application in SDN data plane by inspecting packets beyond layer 2-4 headers.

2.3. Benefits of Software-Defined Networking (SDN)

This section first identifies limitations of conventional TE solutions and then shows how each of them can be improved by SDN.

2.3.1. Unrealistic Ratios of Traffic Splitting

The minimization of congestion is often achieved via several paths, but some limitations exist in current traffic splitting mechanisms. Basically traffic sharing per packet results in a reordering sequence number of packets at the target destination host, which is unwanted in TCP based applications in particular. The per-packet multipath routing as explained in [52] may cause TCP segments to come with no sequentially to the destination host, unnecessarily disable the congestion control mechanism of TCP and result in a reduction of the application throughput and the entire network throughput. Jitter may also occur, which requires large buffers to store packets that are receivable temporarily. On the other hand, a traffic division by flow allows distinguishing every single TCP or UDP flow, thus eliminating the problem of traffic rearrangement. Nevertheless, the network devices and the hash function used for the distribution of traffic are determining the distribution of granularity. This granularity does not necessarily have to be that of the TE solution which lead to inappropriate traffic ratios being allocated to each route. Consequently, the actual performance of the TE congestion management mechanisms and its capacity may not be ideal.

Furthermore, the possibility of shortest routes congestion for traffic load balancing cannot be taken into account by traffic dividing mechanisms such as the ECMP [35]. SDN can help to overcome this limitation with the higher granularity on network elements.

2.3.2. Algorithms for Non-optimal Path Computing

There are also some limitations to the path calculation and the resource management needed in Multiprotocol Label Switching-Traffic Engineering (MPLS-TE). In [69], it found that there were some latencies in some of the links in an over-loaded network. It analyzed the MPLS-TE solution and found that latency inflation is the result both of the used CSPF (Constrained Shortest Path First) algorithm and constant path changes resulting from the bandwidth-allocation algorithm supplied by numerous MPLS vendors to adapt the retained LSPs' (Label Switching Paths) bandwidth automatically according to traffic requirement. With the ability to perform path calculation in a logically central control system, this shortcoming can be overcome by SDN.

2.3.3. TE Databases Are Not In Real Time Reflecting the State of the Network

Even though the PCE-based (Path Computation Element) architecture can enhance such a shortcoming of typical MPLS-TE, it also has its own constraints. The data related to TE stored in the traffic-engineering database (TED) is often used to calculate the path in the PCE-based architecture. Nevertheless, the TED does not really represent the situation in the network in actual time according to RFC 4655[74]. If the TED is not configured accurately to the network state that this may happen at some times, it can increase the possibility for miscalculated routes. The logically centralized control plane, which knows the network state in real-time, can help SDN to overcome this limitation.

2.3.4. Distributed Protocols' Long Convergence Time

Their dependence on RSVP-TE (Resource Reservation Protocol) is another important restriction that MPLS-TE and the PCE-based architecture can find. The PCE answers with the calculated route information when a network element requires a route. The distributed RSVP-TE protocol is then used by the network element to notify additional hosts. As a result, it takes longer than the path that a centralized control plane

installs with programming capabilities. This directly affects the stability of the network and affects already established data flows. The scalability of RFC 4655 also affects this fact, because RSVP-TE is an in-band signaling protocol. Such a shortcoming presented by most solutions based on MPLS-TE, because they all depend on RSVP-TE. The high level of network programming that SDN offers, in particular when network resource control is done out-of-band, may help remove this limitation. In summary, in MPLS networks, TE has developed significantly. The usage of network resources is becoming more and more improved from early implementation of more complex PCE strategies with MPLS-TE.

2.4. Traffic Engineering in Software-Defined Network

Traffic Engineering has been widely exploited in the past and current data networks. However, these past and current networking paradigms and their corresponding TE solutions are unfavorable for the next generation networking paradigms and their network management due to two main reasons. First, today's Internet applications require the underlying network architecture to react in real time and to be scalable for a large amount of traffic. The architecture should be able to classify a variety of traffic types from different applications, and to provide a suitable and specific service for each traffic type in a very short time period. The second is that a compliable network management should be able to improve use of resources in order to increase systems performance in view of the rapid growth in cloud computing and thus the demand of massive data centers. Therefore, more smart and efficient TE tools and network architectures are urgently needed.

The newly developed Software-Defined Network paradigm separates a network control plane from the data transmission plane placed as an external entity (called controller) and provides a centralized view of distributed network states for user applications. If the network is a large-scale or broad regional network, there can be more than one SDN controller. The network states are regulated by the control layer in a centralized or distributed way globally through network policy. Because of unlimited access to global network components and resources, the network policies can be updated in time to respond to current flows. In addition, the application layer of the SDN architecture contains SDN applications. There is a support for the communication of a set of application programming interfaces between the application layer and the control layer so that shared network services can be operated, such as routing, traffic

engineering, multicasting, security, access control, bandwidth management, service quality, energy use, etc. In short, interfaces in the network management facilitate different goals.

On the other hand, OpenFlow (OF) [66] can be programmable through OpenFlow controller for data transmission layer and Southbound APIs allow network devices to communicate with the controller. The OF protocol provides access to the network devices, and enables the operation of packet methods and forwarding through the network switches or routing software programs running on OF switches. The programmable switches follow the SDN/OF controller policy and forward packets to establish which path the packets are to be taken over the network or switches or routers. In summary, the SDN paradigm enables a unified and complicated global view of networks through the interactions between these layers and therefore provides for a strong network management control platform over traffic flows. While current TE mechanisms are extensively studied in current networks, it is still unclear how these techniques perform under various traffic patterns, and how enormous traffic and resource information can be obtained efficiently in the entire network. SDN, on the other hand, promises that network management will be dramatically simplified, the operating costs reduced, and innovations and developments promoted in current and future networks.

SDN features are helpful in solving the existing network traffic engineering problems, which can be summed up in the following [4] and shown in Figure 2.5.

- **Traffic measurement**: Scalable global measuring tasks can be deployed in the SDN, which can gather information about the status of the network in real time and central tracking and evaluate traffic by the controller.
- **Traffic scheduling and management**: The requirements of traffic applications can be taken into account to enable flexible granular traffic scheduling in global view.
- **OpenFlow switch**: It has several data forwarding table pipelines that allow for flexible and efficient flow management.

For further SDN applications, consequently, SDN-based TE innovations are very crucial. TE's framework generally covers two roles: parameter based measurement and management by flow scheduling as shown in Figure 2.5. Traffic measurement primarily surveys how the SDN environment monitors, measures and collects status of current information of the network and traffic. The information about the status of

network and traffic involves the status of the topology of the connection, status of port (up or down), port statistics, different types of flow statistics, drops of packets, bandwidth utilization rate, end-to-end delay, other end-to-end network metrics, etc. Based on analysis about network information, the current condition of network can be validated by analyzing packet counter statistics, prediction of the future traffic trend and the avoidance of network congestions and improvement of network efficiency. In next subsection 2.5.1, SDN-TE traffic measurement research works are highlighted in parameters based measurement techniques. Traffic management studies in particular how network traffic is controlled and scheduled according to traffic measurement technologies, in order to meet network applications end-users requirements, such as QoS. SDN-TE traffic management research focus on flow rerouting (or) scheduling.

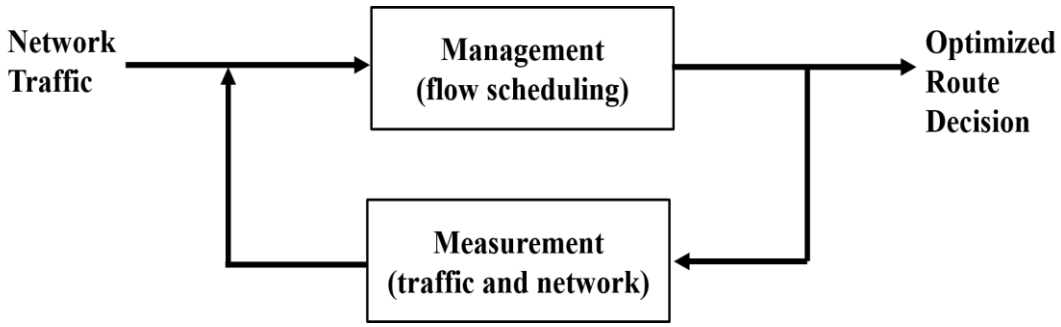

Figure 2.5. Traffic Engineering Framework

2.5. Traffic Measurement

Traffic measurement in the computer network community has been an area of interest for many years. It is usually the measurement of traffic volume and type in a specific network. Some surveillance system today is also able to monitor latency.

Common network parameters based measurement metrics are as follows:

- **Delay** (**latency)** refers to the time it takes to transmit a bit from source to target.
- **Jitter** refers that the delay over time varies. Latency can be seen as long as packet remains on network. This system may be a single device, such as a router or a full communication system with routers and connections.
- **Link utilization** is the average traffic over a certain connection. It is simply a percentage of the overall connection capacity.
- **Packet loss** is the number of packets does not reach their destination.

- **Flow types** is to differentiate the incoming traffic flow is whether long-lived large flow (elephant) or short flow (mice).

2.5.1. Network Parameters based Measurement Techniques

This section describes the measuring systems, which are currently used with SDN capabilities. The phenomenon of elephant and mice flow was treated as a network performance problem. Network resources are utilized in accordance with specific application constraints and requirements. Elephant flows tend to fill out end-to-end network buffers and cause a considerable delay in the small latency-sensitive flows that actually share the same buffers with large flows. This leads to network performance degradation. Moreover, hash-based multipath routing techniques that are currently being used in data networks may have a variety of large flows in one connection while leaving other connections free and reducing network quality [79]. It will therefore be much more appropriate than mice to deal with elephant flows. To achieve this, elephant flows are required to be detected, marked and signaled. In order to manage elephant flows properly, a traffic engineering module at control level can then be informed using SDN. Generally, the detection of elephant flows, such as Hedera [5], can be achieved through regular polling. The technique of Hedera uses the polling period of five seconds. This granularity leads to a likely congestion of the network between the polls. For effective traffic management, long-lived large flows (elephant) that affect network performance need to be focused by the controller. Many small flows (mice) occur and go too quickly for installation of flow entries according to the policy of the controller. The identification of elephant flows is therefore important in the development of a congestion control policy for different types of flows.

The equipment supported by NetFlow [73] regularly sends sample packets to a NetFlow Collector, which is a server to access packet's information. In order for information to be collected by tapping a link, NetFlow sampling may be deployed to the network. NetFlow concentrates primarily on network connections of layer 3; for switches of layer 2, this does not apply directly. The NetFlow architecture complicates the configuration and adds complication to NetFlow agents and switches. As a result, it is limiting the scalability. Creating NetFlow sample packets can raise original problems such as where they should be placed and how many to cover the network.

OpenFlow contains three types of statistical counters for each flow (packets, bytes; duration) [95]. By delivering OpenFlow message which is to read network state

periodically to the data plane, the central controller maintains the network state and counter. The OpenFlow controller can be defined as a flow statistics monitor if the polling is frequently enough. Due to its relatively high cost and low performance, it is not suitable for data centers. The OpenTM [84] is estimated by following statistics directly from a network switch for each flow and polling. After 10 queries, the application decides which query to execute and converges to 3 percent error. Several survey algorithms are compared for an interval of 5 seconds in their presenting [84]. In [94], they are also using the fact that each new request for flow is passed through the controller. This enables the traffic to be routed to one of several traffic monitoring systems or traffic recorded or analyzed with an IDS or just a firewall. OpenFlow's "Flow_Removed" and "Packet_In" messages, which are sent by the OpenFlow switch, provide information on the start and end of flow. In addition, the "Flow_Removed" packet accounts for each flow's life and number of packets/bytes. This feature is used in FlowSense [94]. Three perspectives assess measurements: accuracy (compared to polling), granularity (estimated refreshing) and staleness (how quickly usage is estimated).

In addition, a new SDN protocol is being suggested, which is focused on the collection of statistics. Some efforts in this direction are shown in [95], where a new software defines traffic measurement tasks in addition to an OpenSketch enabled network. Tasks include detection of heavy hitters (small flows most of traffic), super-dispersion (source that contacts multiple destinations), changes in traffic, distribution of flow dimensions, and the counting of traffic. Such as OpenSAFE [6] proposes the replication of traffic to monitor the network with a high overhead while FlowSense [94] describes a push mechanism for passive analysis of link usage, it is an effective way of measurement mechanism but not to estimate latency.

The flow entry table for all flows for polling statistics using OpenFlow needs to be established, and each request returns 88bytes of controller bandwidth from the control plane to the data plane. In [94], it describes that a data center with a 100 edge switches generates an average of ten million flows per second. If a polling method is used for statistics collection and the identification of elephant flows, 6.5 GB of control messages will be returned. Therefore, the control plane is able to cover with such a heavy volume of traffic.

The solution [22] sets up a detector agent or applications on the end device, rather than tracking flows by the control plane. It can immediately and accurately detect

elephant flows before transmission. The detector defines that flow as an elephants flow if flow rate measures whether it exceeds a predefined threshold value (e.g. a socket buffer and flow size). It is more precise to use the theory of prediction of traffic patterns to design the adaptive detection system that has attracted researchers' interest. The overhead can be minimized without saving flow statistics. However, the operating system of each end device must be modified in the data center. Virtualization technology in the data center has currently been widely applied. A single host can run several virtual machines in this scenario, which must be changed, and the detection application installed.

A range of device vendors has supported the packet sampling method, sFlow [85]. The sFlow system can be scaled to the volume of traffic, particularly under frequently changing data center traffic. The agent collects and encapsulates sample packets 1 out of k packets of flow in comparison with capturing all flow statistics and sends only the sampled header, which is encapsulated in UDP packets to the central collector. A single sFlow collector [85]can monitor thousands of devices. If the sFlow detection system assumes each end device has 10k flows per second, the 10 million flows per second can be handled, compared to 30% of NOX controller per second. The disadvantage of this approach is that it uses the static predefined threshold for elephant flows to be selected from existing solution [22] and [85].

According to the above literature review, most of the current SDN measurement methods are focused on link utilization and flow types identification. There are very few other network parameters measurement such as latency, jitter and packet loss. Latency is an essential metric in a day-to-day network operation, especially when used to transmit delay or jitter sensitive data from applications. A good VoIP connection requires a latency of less than 50ms. When latency is slightly higher, some dropped frames can be acceptable when the connection is maintained. During the broadcasting of general video, some losses (less than 5%) for most codecs are admissible. An average ~150ms latency is sufficient. If the frames are buffered, it could be acceptable for up to 5 seconds. Interactive videos are more vulnerable to loss of frames and requirements. While certain losses (less than 1 %) are still acceptable, the impact of jitter is far greater. Various techniques are now being employed to measure link usage, end-to-end delay and loss of packets.

2.6. Traffic Management

Network management aims to ensure the availability of the network and to enhance network resource utilization. Reasonable redirection of network traffic is an essential method to enhance network quality. In general, there are several paths in the SDN from the source host to destination host that allow traffic rerouting. As the OpenFlow controller abstracts the graph of network topology globally, several of the above-mentioned network measurement techniques and algorithms of traffic scheduling can be developed that will enable users to reroute dynamically transmitted paths.

2.6.1. Description of Existing Flow Scheduling Techniques in SDN

The current flow scheduling techniques are mainly discussed in this section for traffic in data layers. The main advantage of SDN traffic scheduling, compared with a conventional network, is that transmission centralized and undistributed decision computations, allowing for a more comprehensive planning of a load balance strategy, taking account of several optional rate of link usage and characteristics of traffic flows.

The Equal-Cost-Multipath routing technique (ECMP) [35], a traditional effective load balance solution, is based on a hash-mapping algorithm. There are multiple transmission paths in a target network, depending on the ECMP routing. When the first packet of new flow hits a router or switch, it samples the packet header fields for hash calculation from the switch or router and then selects a forwarding path by a hash value that will lead to a redirection of IPs along the same path with the same header.

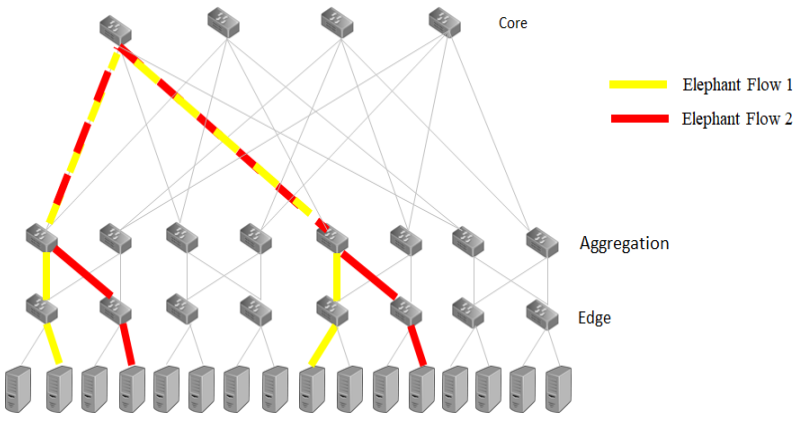

Figure 2.6. Flow Collision in ECMP

The ECMP's obvious defect is that many large traffic flows, defined as elephant flows, may be transmitted to the same path, leading to incoming traffic load and available bandwidth imbalances [35] as illustrated in Figure 2.6.

A simple solution is to divide the original traffic flow into packets, but not flow level. This way of balancing the load significantly improves, but it makes the TCP sending window reordering problem more serious, causing unnecessary TCP sending window reverse shrinkage. A number of improved ECMP schemes have been proposed [63] and [19] to solve this problem. Identifying first the elephant flow and choosing the right path by the controller is their main idea.

In [98], the authors proposed a dynamically scheduled flow to optimize the ECMP algorithm according to the utilization of core bandwidth connections. Every 10 seconds it measures the byte number of the specified switch port. The scheduler module forward some flows to the other link with a smaller bandwidth usage if the use of core link bandwidth exceeds the specified threshold.

DMSP [39] uses the conscious flow scheduling method for bandwidth. DMSP implements a control unit that tracks all links congested. It can redirect or assign the emerging flows to least congested links. After calculating all the shortest paths from the source to a destination, the route can be selected using the Round-Robin policies. Active connections are mapped. When a path is assigned to a particular host, all packets from that host are transferred on a particular port. Statistics of the controller module are used to identify connections congested in the data center. The flow table is updated to provide less congested connections before highly congested connections. To measure the load of a connection, the controller examine each link for the bytes transmitted. The restriction is that the link load parameter only considers that mice flows can be faced to minor delays due to flow table updates.

The dynamic allocation resource model of SDN cloud computing is proposed by Chenhui et al [90]. It classifies user service traffic to QoS flows and best effort flows depending on user needs or application functionality, which originally followed the shortest path. The traffic flow type is distinguished in the IPv4 header and source IP address, depending on the service type (ToS) field when it reaches the controller. The flow and port statistics are collected on a regular basis via OpenFlow protocol to calculate the load for the links. When the load of the link exceeds the load threshold, the optimal route based on packet loss and jitter is chosen. If no optimal path exists, the queue technique is combined to ensure high- priority flow transmission.

The load balance strategy is proposed by OFLoad in [86]. Two steps are taken by OFLoad. In the first stage, elephant flows and mice flows are differentiated. At the end device, the elephant detection and marking mechanism integrates in kernel level shim layer and it monitors the TCP socket buffer. Due to the preset rate threshold, the shim layer identifies and marks the flow as an elephant flow when a buffer bytes exceeds that threshold in a given period, for example by using the IPv4 field of differentiated services (DS). When a flow is tagged as an elephant flow, the elephant flows are redirected to the paths with a minimum hop counts and traffic load, which minimizes traffic congestion in network. With custom OpenFlow polling messages the controller collects information and statistics from switches. In stage two, OFLoad considers for the short (mice) flows and computes the paths by means of the weighted algorithm of multipath routing. The link weights are determined by applying the function of multiplicative dynamically.

Al-Fares et al. detail Hedera [5], which has been proposed the protocol for SDN flow rerouting dynamically. The authors use fat-tree topology in today's production as the most popular network topology. It involves three main steps. In order to identify elephant flows, firstly, it collects global information about flows from edge switches. The second is to compute non-conflicting paths with a placement algorithm, which do not exceed the combined bandwidth requirement of all flows. Finally, the switches are equipped with forwarding tables. The basic idea is that Hedera applies ECMP routing based on flow header for mice flow. After a flow exceeds the predefined threshold (10% of the host NIC capacity), this existing flow is identified as an elephant flow and scheduling dynamically is overtaken by Hedera. In the first place, a path is greedily assigned to the flow; the first path that is suitable for the flow is selected for all links. On the other hand, for each destination end host, Simulated Annealing (SA) algorithm assigns single core device instead of one core device for every flow. Therefore, all traffic flows that pass through that particular core switch destined for a target. It is demonstrated that SA is more effective than Global First Fit (GFF) algorithm. Even with many hosts, Hedera offers improvements to the flow rerouting. Furthermore, at end hosts there are no changes required. These benefits are at the expense of modest overhead communication. Although Hedera resolves the problem of congestion, there are still long queue delays for mice and statistics collected per flow to measure congested route, which is poorly scaled in Hedera.

Elephant flow sensing is carried out at the end host, which send the TCP handshake Elephant flow packet directly to the controller before more than 10 Mbytes are transferred. In this process the available paths are divided in a dynamic way into low latency for mice flows and high bandwidth paths for elephant flows. The low latency paths maintain a low queuing time required by mice, and higher performance tracks generate longer queues to accommodate more packets to meet a hungry demand for performance flows. For measuring network-wide link usage, Freeway [88] used the global view of the SDN based controller. OpenFlow 1.3 adds per-flow meter support, which enables flow and port statistics to be obtained. It uses Greedy algorithm for choosing a low latency path from a set of low latency paths. In Freeway, the controller schedules the elephant flows and mice flows are scheduled locally within network device.

In [21], the authors described a schema for the rerouting of the flow using the sFlow analyzer [85]. When the utilization rate of the link is higher than the predefined threshold, it dynamically reprograms the elephant flow. This scheme is also designed with the Particle Swarm Optimization method to find the global best solution to multiple paths based on a commodity flow problem.

Mahout [22] is another system which allows the management and optimization of SDN traffic, which reduces traffic administration costs via further backend server. As the elephant flow detection in approach of Hedera builds on OpenFlow controller inside network, with long latency and a greater overhead of resources, Mahout uses a backend server instead of transferring elephant flows. This approach results low cost and management efficiency. Mahout therefore uses devices for back-end server detection. Mahout resumes the backend server's special function layer, namely the SHIM layer, which monitors local traffic via a socket buffer. Mahout believes that the respective flow is an elephant flow when the buffer exceeds a specified threshold, it marks the subsequent packets of flow. For flow table regulations, Mahout sets two priorities such as high and low priority. ECMP is used to forward packets that conform to low priority rules as default. If the subsequent of packets of an elephant flow match the rules for high priority, they are delivered to Mahout controller in order to compute the best path, and the relevant forwarding flow rules are modified.

The MicroTE [10] is similar method to that of Mahout [22] approach. MicroTE passively monitors network condition by means of flow statistics. When the statistics of flow change clearly, the aggregation behavior of the flow is activated. MicroTE

evaluates whether or not the incoming flow is an elephant flow depending on the difference between instantaneous flow rate and mean flow rate. MicroTE starts or uses a heuristic ECMP to calculate the routing optimization when the flow speed is exceeded.

With two standard solutions: DevoFlow [63] or DIFANE [96] the wildcard programming technology represents another scheduling mechanism for the flow. The DevoFlow [63] changes the OpenFlow model to create a traffic schema that saves resources and can be scaled. The DevoFlow specifically proposes to redirect elephant flows and not to treat all the elephant flows. This approach can reduce end-to-end delays effectively and checks on costs only after they have been found. A wildcard multipath rule is developed by DevoFlow. A switch uses the flows rule as default transmission. In addition, DevoFlow introduces a traffic statistics collecting method to recognize the elephant flow, in which the elephant flows must be rescheduled using the controller's decisions.

The new efficient traffic wildcard system is DIFANE [96]. DIFANE 's core idea has two aspects. For a given subset of switches, the controller first distributes certain rules. Second, switches are used to find a route for all traffic flows in the data (or) forwarding layer. If the incoming flows do not hit with proactive rules in switches, they are delivered to a subset of suitable authorized switch.

Yan and others [93] offered the guarantee solution of HiQoS SDN QoS. The HiQoS identifies multiple routes by queuing mechanism for different traffic types from source to destination nodes. The results of experiment are verified that the system HiQoS reduce the end-to-end latency and improve performance. The HiQoS can clearly be redirected quickly from the link failure.

In [34], the author used the source routing to convey path and policy information in the packet. This simplifies the core of the network and makes the edge switch to take all routing and policies. The edge switch for packets of a flow embeds the packet with one of the round robin paths selected for it. This approach essentially translates the elephant flows into mice flows and transmits them per packet. Dividing flow over available routing paths divides elephant traffic into mice and spreads it over several paths, thereby ensuring that elephant traffic does no harm latency sensitive mice traffic. The limitation of this approach is that overhead bytes are required in a packet header to implement policy, which increases linearly with the length of the path.

TinyFlow [91] proposed the multipath transmission system with the detection of elephant flow and distributes elephants in many mice flows and at random with the ECMP. OpenFlow switches regularly conduct sampling to detect elephant flows. The dynamic random flow scheduling algorithm alters the egress port of the switch when two samples of the same flow are detected. The principal disadvantage of this method is that TCP out-of-order problem. Moreover, in the case of up-link aggregate switches, the hash collision is still present, leading to poor bandwidth usage. Then, the performance of the whole system depends on the precise detection of an elephant.

TinyFlow [91] and Random-Packet-Spray [63] were motivated in the DifffFlow [14] solution. RPS uses random packet spraying to forward packets via several shortest paths. In DiffFlow, the default routing protocol in switches forward flows, i.e. ECMP, without the participation of the SDN controller. However, the top- of- rack (ToR) switches sample the packets every predefined time, to detect the entry of an elephant flow into the network. The flow duration can be marked as short or long based on the length of the sample period. DiffFlow breaks the elephant into mice flows using a Random Packet Spraying method when the elephant flow is detected. In contrast to ECMP, the RPS transmits each packet to the output ports on the interface ("spraying"). Since this feature is not enabled by default, the OpenFlow switches require further modifications.

Suchandra et al. [15] proposed the multipath scheme, which is dividing elephants into many mice, but without further elephant flow detection requirements by handling the hard time-out based flow removal feature of OpenFlow switches. The divided mice flows are routed via random egress port.

In [83], the authors proposed the middle box design for chunked video exchange. This chunked video splitting scheme is tested over three nodes of TEIN network (Korea, Malaysia, and Thailand). The splitting ratio is considered based on round-trip-time values (eg. Path1: 125 ms, Path2: 105 ms, then splitting ratio is 1:2). The limitation of this scheme is splitting ratio cannot be adapted dynamically according to delay variation. Network condition such as delay, jitter and packet loss are not measured in proposed scheme. The additional buffer at the receiver end makes the packet resequencing overhead.

In [7], the proposed scheme split the incoming traffic into bursts of packets (flowlet) along the paths using Weighted Round-Robin (WRR) scheduler based on capacity of path. It uses Dijkstra algorithm to select the path for each flowlet. This

approach also introduced the delay difference concept of available paths. If this delay difference higher than the reordering threshold, a flow-reordering rule is set up at the receiving edge switch. This approach needs additional overhead at the receiver switch to reorder the sequence of packets. Moreover, to split the flowlet level, additional modification of OpenFlow switches are required.

RepFlow [92] duplicates mice flows to decrease head-of-line blocking likelihood and eliminate mice flow's flow completion time (FCT) without modifying TCP. In addition to their original transmitter, each transmitter establishes a TCP connection and sends the same packets through both links. The main drawback is the poor usage of available bandwidth due to the elephant flow collision.

In [58], the multipath forwarding mechanism is proposed using elephant flow detection method. The elephant flow detection makes use of a shim layer integrated in the end-host that monitors TCP socket buffers. The shim layer identifies and marks the flow as an elephant when the number of bytes in the buffer exceeds a predefined rate threshold over a given time window. Once elephant flow is detected, it applies weight multipath routing using OpenFlow group table feature. The weight of path is dynamically computed based on link load ratio. To get link load ratio, it collects statistics from each OpenFlow switch by polling them at fixed intervals. The per-table, per-flow and per-port statistics are gathered from all the connected OpenFlow switches.

A local rerouting approach is introduced by authors in [44], taking the nature of flow into consideration whether elephants or mice. In the event of congestion, elephant flows only reroute while the mice flows are allowed to continue. In addition, redirection takes place locally (or one hop before) at the congestion point rather than redirecting flows from the sender through the available multi - pathways. The SDN controller from all connected OpenFlow switches collects per-table, flow and port statistics to calculate the shortest path between any pair of end hosts, which is least loaded. In addition the controller recognises and classifies the elephant flow with the LAN priority bits (set as 001-elephant and 000- mice) if the flow size is 100KB or more. If one of the link loads exceeds the congestion limit of 75 percent at any time, the controller reroutes one or more elephant flows through the link to an alternative path by installing new flow rules.

In [43], the authors presented the SDN based load-balanced multipath algorithm. The central controller to collect the traffic load data for all network links to make globally decisions for optimized routing for the proposed algorithm. When a new

flow come to the network, the switch that first encounters the flow will inform the controller. The controller will enumerate all possible paths that can carry the flow, compare the load of the bottleneck links of those paths, and then select the one with the minimum load. The controller informs all of the switches along the route after the route is selected and updates their flow tables so that subsequent flow packets are forwarded along the same route.

In [54], the authors presented a load balancer for the fat-tree network with multipath support. They implement a dynamic load balancing routing algorithm in the load balancer. This algorithm is adaptive to network traffic, and schedules flows by examining current available bandwidth on all alternative links. In this design, it calculates the available bandwidth indirectly by collecting the transmitted bytes on each port from switches periodically.

The authors in [72] discussed a multipath routing with load balancing and admission control in SDN. Load balancing is to balance the network traffic through each path. It collects the flow statistics periodically from all switches. Load balancer schedules the network traffic according to updated traffic statistics on each switch. It selects the path that has the link with highest bandwidth availability or with the lowest traffic load. Admission control is to control or reject the amount of traffic that injected into the network when the traffic flow exceeds the capacity or a certain threshold to avoid congestion. It uses threshold at 80% of the maximum capacity link for admission control system.

In [25], the authors proposed the SDN controller design to allow QoS to deliver multimedia over OpenFlow networks. the incoming traffic is grouped into two as data flow and multimedia flows to support QoS, where QoS guaranteed routes dynamically provide multimedia streams and the data streams continue to stay on their traditional shortest path. In every one second, OpenQoS controller regularly collects the bandwidth available to each link. In this approach, although delay metric is considered in proposed framework, the authors left as an open issue. The actual verification results based on only bandwidth utilization cost.

In [13], the authors described a method called OFMPC for isolated transmission of short flows and long flows by scheduling them. Short flows are assigned to low delay paths and long flows are assigned to high throughput path by using bandwidth allocation model. OFMPC have to arrange half of the available paths to be low delay paths (LDP) and other half to high throughput paths (HTP) in network topology. For

short flows, OFMPC assigns them on LDPs and allocates enough bandwidth for them by preinstall flow tables based on server-level load balancing. For long flows, OFMPC specifies them on HTPs, distributes enough bandwidth for the flow, and then serve more long flows until the available bandwidth of HTPs is fully utilized. OFMPC routes remaining long flows on LDPs to make full use of the residual bandwidth of LDPs. The OFMPC have to maintain several variables: flow address information, remaining flow size and demanded bandwidth. The controller updates these variables in a polling period.

OpenE2EQoS [55] proposed a statistically rerouting algorithm for multimedia traffic. To evaluate the statistics of the link utilization, the SDN controller monitors each link usage in network. Instead of shifting multimedia traffic, where a link is found as congested, The congested link routes low priority packets (such as delay tolerant traffic or best effort traffic) by mitigating congestion by predicting the necessary bandwidth of the QoS. The packets with the lowest priority are statistically distributed according to the amount of bandwidth available for each route. The original route continues to remain for multimedia traffic. Limitation of this proposed method is that congestion of links is monitored by collecting statistics on switch port which are not suitable for scalability.

In [27], when the specific switch buffer occupancy exceeds a certain threshold the congestion point is avoided by rerouting traffic flow. In this case, controller gets statistics of ports and queues at regular intervals of time by polling OpenFlow messages. As the controller has the complete view of the network topology, it runs shortest path algorithm for all the flows in the network. Consequently, it can divert some flows to lightly loaded paths to reduce the buffer pressure at the bottleneck switch.

Ming-Tsung Kao et al. [45] detailed the flow based scheduling scheme for link congestion avoidance in Software-Defined Networking (SDN). This scheme monitors and detect the congestion event by collecting per port switch statistics periodically (every 3 seconds). When the congestion event is detected, the ongoing flow is rerouted to another available shortest path which is calculated using Dijkstra's algorithm.

In [97], the authors described the scheme that supports QoS for video streaming. Scalable Video Coding (SVC) encodes a video into a base layer and one or more enhancement layers. The video in the base layer should be streamed without any packet loss or minimized delay variation, regarded as level-1 QoS flows, and the video in the enhancement layer can be regarded as either level-2 QoS flows (if capacity is available)

or best-effort flows. In this scheme, base layer packets and enhancement layer packets of video bit streams are treated separately as two levels of QoS flows level-1 and level-2. During video streaming, if the shortest path does not satisfy the delay variation (jitter) constraint, the level-1 QoS flows have the first priority to be rerouted to a calculated feasible path based on the available bandwidth of this path, and the level-2 QoS flows will stay on the shortest path. However, if there is no path that does not satisfy the available bandwidth, the level-1 QoS will stay on the shortest path while the level-2 QoS will be rerouted to this path. This scheme is only purposed for video streaming application.

In [18], the authors proposed load distribution mechanism to assign traffic based on weight of paths. Controller finds all the disjoint paths for a given pair of nodes. Then, the controller demands the ingress switch to distribute the traffic based on the status of the discovered disjoint paths in terms of hop count using modified Dijkstra algorithm. The limitation of this approach is that it does not consider other critical network conditions such as delay, jitter, and packet loss and bandwidth utilization. The shortest path in terms of hop count cannot be optimal path for ununiformed traffic.

Yuan-Liang et al. [50] detailed a priority-based flow table updating strategy, which is based on the priority field in a flow entry, to resolve the interrupted flow transmission problem. This scheme is to avoid packet loss or oscillation caused by changing paths of flows. It uses the OpenFlow protocol to retrieve load statistics from switches and detect the load-balance status. If link loads in the data center network are imbalanced, some flows are rerouted to light-loaded path. This method also has scalability problem cause of retrieving load statistics from all switches.

CAMOR [78] proposed a technique that brings together intelligent multipath routing and congestion detection mechanisms for ideal route selection in all available routes in network. As a congestion parameter, the load of each link involved in routing mechanisms. It has proposed a flow based differentiation method to make better use of the bandwidth and resources and to distribute traffic equally along multiple routes. This proposed method uses the popular Dijkstra algorithm to calculate the multiple paths between two endpoints at equal cost and then all the available paths were distributed evenly by using the multipathing technique.

The authors in [64] proposed the accumulative load aware flow-scheduling scheme. In this method, the accumulative load is the weighted summation of data flows going through a link. Each data flow installed on a link consumes a certain amount of

link's bandwidth. Thus a link which has a lot of data flows should has high cost to limit installing additional flows which is potential of congestion. In the other way, a link currently transfer small amount of data will have lower cost and therefore have chance to carry more data. When packet_in event occurs, it calculates shortest paths between source and destination by using Dijkstra algorithm. The cost of links are calculated from accumulative load. Link cost values can be updated after flow entries installation.

SynRace [76] detailed a path latency measurement method with multipath transmission. It selects least congested path based on measuring latency. SynRace's solution leverages the interdependence of data rate and latency. SynRace can use a path's latency as an indicator for its load and its (likely) unused data rate. Latency is measured by replicating initial SYN packet of every TCP flow over all paths in network. This method decides the least congested path whose SYN probe arrives firstly. The limitation of SynRace is that it can be applied for only TCP traffic and does not work well for UDP.

Table 2.1 Methods of SDN flow scheduling

Method Name	Description
Hedera [5]	Edge based elephant flow detection & per flow statistics
OpenQoS [25]	Link utilization threshold
OpenE2EQoS [55]	Byte counts at port
OFMPC [13]	Link utilization threshold
Li et al [21]	sFlow analyzer to detect elephant flow & link utilization
Chenhui et al [90]	Differentiate type of service (ToS) field in packet header & link load threshold
OFLoad [86]	End host based elephant flow detection, link load threshold & hop count
FreeWay [88]	End host based elephant flow detection & dynamic queue size
Yu et al [54]	Byte counts at port
Maris Fajar et al [72]	Link utilization threshold
SynRace [76]	Measure latency to find least congested path

Some summarized related works are described in Table 2.1, which combine traffic measurement and management techniques to avoid congestion problem. Most of the proposed works use OpenFlow-based collecting statistics techniques to detect the elephant flow and to compute least-congested path. [21] uses sFlow analyzer to detect

elephant flow and compute link utilization to avoid congested path. The proposed work in [76] finds the optimized path based on end-to-end latency measurement using TCP SYN packet.

2.7. Chapter Summary

Because of such a limitation of traditional Ethernet network caused by Spanning Tree Protocol, the SDN infrastructure has been evolved. To avoid network congestion which is caused by using single shortest path, there are different objectives and techniques in Traffic Engineering (TE). To improve network performance and quality of service (QoS), TE plays as a critical application. It manages the network traffic by measuring and analysis of traffic and network conditions. Multipath distribution is a common solution of traffic engineering approach. Instead of using the single best path, multipath scheme can avoid the congested path by rerouting the traffic to uncongested path. Unlike the traditional networks, Software-Defined-Network (SDN) has many advantages to support dynamic multipath forwarding due to its special characteristics, such as separation of control and data planes, global centralized control, programmability of network behavior, software-based traffic analysis, dynamic flow updating and flow abstraction.

Various multipath transmission methods have been proposed in literature review. Although these methods can optimize the high transmission capacity availability and latency, their limitations may degrade network performance. Contrasts in characteristics of each method prompt distinctive points of advantages and limitations. According Table 2.1, most of the studies on flow re-routing except [76] have focused on bandwidth utilization (current load of the network); however, they do not consider network latency (end-to-end delay) problem. When the network has high latency, data transmission time will take a long time. Long data transmission time causes bottlenecks in the network nodes. But [76] uses the least end-to-end latency path for elephant flow. This approach only works for TCP flows and not for UDP flows. Therefore, more efficient research works are still needed for SDN traffic engineering. In the next section, the background theory will be described in detail.

CHAPTER 3
BACKGROUND THEORY

This chapter presents the basic concepts and components of SDN infrastructure and ONOS controller [65]. It also classifies the elephant flow detection methods and components of multipath forwarding mechanisms in SDN.

Any traditional network device (switch/router) comprises three logical layer: application layer, control layer and data layer. Application layer consists of switching/routing protocols by providing network administrator using CLI. Control layer uses the switching/routing information from protocols to generate the data forwarding table in data layer. In data layer, data packets are forwarded according to the data forwarding table. The traditional network architecture is facing the limitation with three layer coupling because it makes difficult to behave the network innovations and to add new network function in the underlying data layer. Because of traditional network's limitation, application layer and control layer are moved to external entity which is called controller (such as server). Control layer configures the data layer through data forwarding rules. In general, the key differences are:

- In traditional network, each switch/router is needed to configure protocol to compute data forwarding rules while in SDN, controller computes the data forwarding rules.
- In traditional network, each switch/router uses a proprietary communication to manage forwarding rules in data layer while in SDN, the controller communicates with the switch/router via OpenFlow protocol in order to manage data forwarding rules in each device.
- In traditional network, changing the network policy is complex when a new network application is deployed. In SDN, policy management can be done easily through application layer.
- In traditional network, if a new network device is wanted to add or remove, the adjacent switches will be needed to reconfigure. In SDN, this can be done by controller software.

Nowadays, SDN is widely adapted in significant IT industries, for example, Cloud Computing, Datacenters, Server Providers, and Campus Networks. Today's Campus Network (CN) (see in Figure 3.1) requires support for multiple devices,

mobility, security and variants of application demands. The campus's network administrators manage all network devices within campus through Network Management System (NMS). If the network administrators need to change the policies, (eg. access policy, security policy, usage policy and QoS policy) and to add the new network device, they have to modify the configuration of associate devices via CLI. It consumes the significant amount of time and human resources. The consequences of this results make the static network behavior and difficult to update users' requirement. Moreover, it is difficult to innovate the network when the research students deploy the new network application. Therefore, CN has evolved with more complexity in management and technology to support growing demands. With the growth size in CN, it requires a very complex network architecture. SDN enabled Campus Network (see in Figure 3.2) can manage the packet-level visibility through OpenFlow protocol periodically. Based on centralization and global network-wide view, the optimized network policies can be made by using applications in controller. Moreover, it can easily monitor and configure the network devices through OpenFlow protocol dynamically. Network applications (eg. Firewall, Load-balancer, VLAN management, etc) can be deployed in controller by using API (Application Programming Interface).

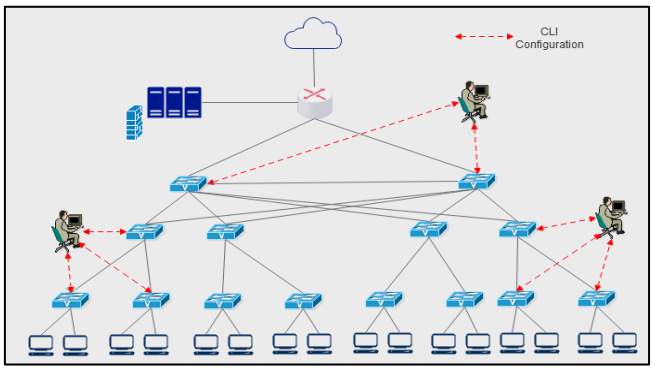

Figure 3.1 Traditional Campus Network

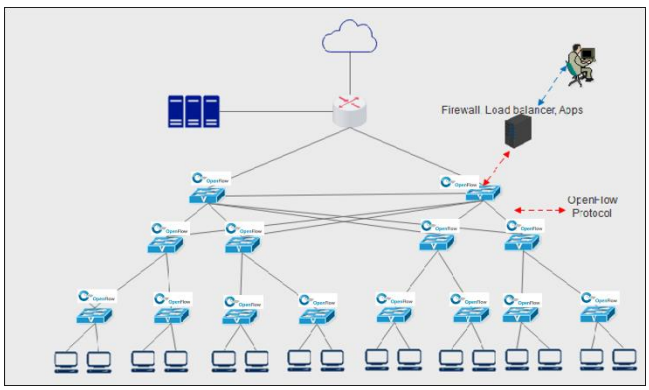

Figure 3.2 SDN Campus Network

3.1. Architecture of Software-Defined Network

The SDN architecture is based on the idea of (logically) centralized Network Intelligence in software based SDN controllers that maintain a global network view. The network thus appears as a single logical switch for applications and policy engines. Through SDN, the vendor-independent control over the whole network is gained from a single logical point by companies and network operators [66]. Figure 3.3 provides an overview of the Software-Defined-Networking architecture. The architectural components of SDN are defined and explained in the following list:

- **SDN application (SDN App):** Depending on controller platform, SDN applications are programs that write their network requirements and desired network behaviors to the SDN controller using so called Northbound Interface. The program is based on higher levels of programming languages. A greater degree of abstracted network control is introduced in the applications and is easy to create and test [36, 67].
- **SDN controller:** The SDN controller [37] is a platform for easier communication of SDN data paths by the application layer. It offers a narrow view of the network as well. A SDN controller consists of one or more agents, the SDN Control Logic, and the CDPI (Control-Data-Plane-Interface) drivers, respectively. Its definition as a logically centralized entity does not prescribe or preclude details about the implementation, for example, multi-controller federation, hierarchical connectibility, interface communication between controllers or network resources virtualization or slicing [77].
- **SDN datapath:** The SDN data path is a platform for network devices to advertise the controller for their data transmission and processing capabilities. It is responsible for the handling of traffic, but it has followed the decision from the application layer. The data traffic is transmitted by its forwarding engine (the flow table in OpenFlow). OpenFlow [66] will be discussed in next section in details. This makes it easier to operate network devices vendor-independently.
- **SDN southbound interface (SBI):** The SDN SBI is the interface that provides at least:
  (i) Programmatic control of all forwarding operations
  (ii) Advertisement of capabilities
  (iii) Statistics reporting
  (iv) Event notification between an SDN Controller and an SDN datapath.

- **SDN northbound interfaces (NBI)**: The SDN NBIs are interfaces between SDN applications and SDN controllers that generally provide an overview of the network and enable the network behavior and requirements to be immediately expressed. This can happen at any abstract (latitude) level and in various functionality sets [77].

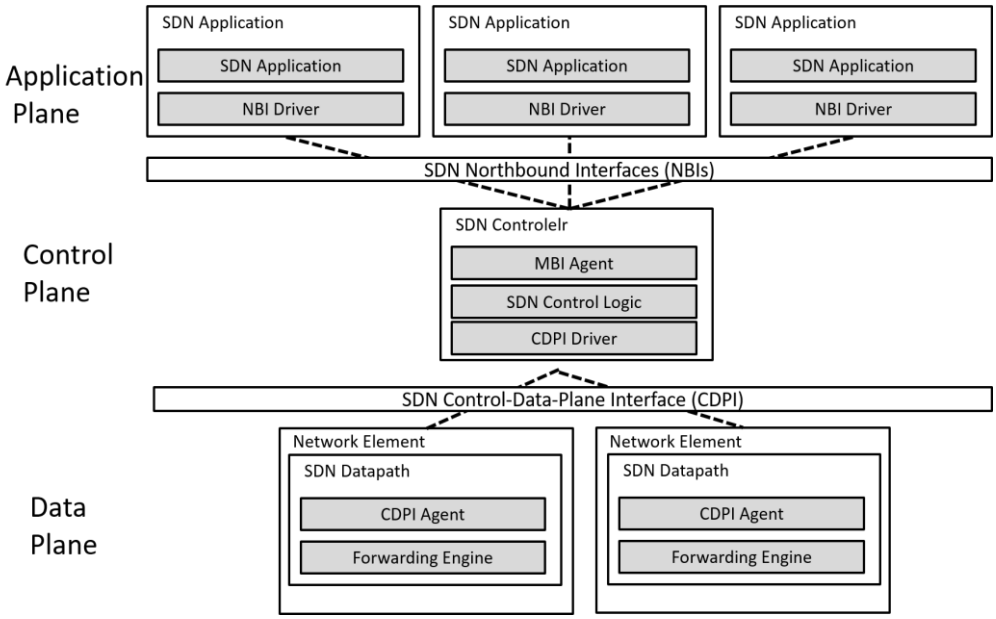

Figure 3.3 Main Components of SDN Architecture [66]

3.2. OpenFlow

One of the standard southbound interfaces in SDN is OpenFlow [66] that allows hardware control via a central controller. Many suppliers, such as HP, NEC, NetGear and IBM, provide network switches that can work with OpenFlow protocol.

The OpenFlow enabled switch is the fundamental element in the data layer. The flow tables are used instead of the MAC table to forward the traffic. When the packet reaches the switch, it tries to prioritize the header information by matching the flow table entry. If it matches, the flow entry action will be carried out. The packet will otherwise be dropped or sent to the controller on the basis of the flow entry setup. Figure 3.4 shows the principal components of an OpenFlow switch. It includes three main components [66]:

(i) OpenFlow switch consists of a flow table and a group table, which looks up and transmits the packet. Each flow table involves the key flow entry components. The key flow entry components are as follows:

- Match fields: This field sets the filter for the packets to be processed by this flow entry. (Packet header, ingress port, metadata and other).
- Priority: Sets a priority when certain flow entries overlap to specify the packet flow item.
- Counters: These are some records that keep the number of flow entry packets and bytes processed. It also keeps the lifetime from being installed on the switch of the flow entry.
- Instructions: Set of measures to be applied to the flow entry packets.
- Timeout: Specifies the time to expire the flow entry. Two kinds of timeouts are available:
    - Hard timeout specifies the maximum time since the SDN controller installed the flow entry to the switch.
    - The idle timeout defines the maximum time between two consecutive flow-equivalent packets.
    - Both timeouts can simultaneously be installed to determine when the flow entry is removed from the switch.
- Cookie: Unique opacity value chosen to identify the flow entry by the SDN controller. This allows the controller, when modifying or removing flow entries, to filter specific flow entries.
- Flags: These fields define how to manage the flow entries on the switch. For example, if the switch sends the removed flow message to the controller, including counter data, after the flow is expired, it can be defined.

(ii) **OpenFlow channel** connects an external controller to switches for signaling and for programming purposes. The switch communicates with the controller and the controller manages the switch via the OpenFlow protocol.

(iii) **OpenFlow protocol** allows the controller to add, update, and delete flow entries in flow tables, both reactively (in response to packets) and proactively. Each flow table in the switch contains a set of flow entries; each flow entry consists of match fields (The fields from packets used to match with flow entries are shown in Table 3.1), counters, and a set of instructions.

Table 3.1 Matching Fields in OpenFlow Table

Ingress Port	Ether Src	Ether Dst	Ether Type	VLAN ID	VLAN Priority	IP Src	IP Dst	IP Proto	ToS	TCP Src	TCP Dst

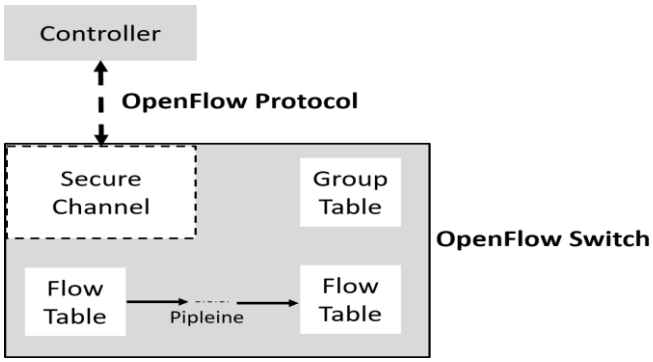

Figure 3.4 Main components of an OpenFlow switch [66]

3.2.1. Open vSwitches

There are a lot of implementations for virtual switches. Linux bridge is a kernel Ethernet switch to help filter and shape traffic. KVM [49] uses the Linux Bridge to connect virtual machines on the same server and also external network connectivity. Open vSwitch (OVS) [66] operates as both an Ethernet switch and an OpenFlow switch.

Open vSwitch (OVS) is an open source software switch widely used for manufacturing applications and is supported by a variety of Linux systems. It has also been supported to the operating systems of Microsoft Windows. OVS is typically deployed on the host and works with hypervisor (e.g., KVM), container systems (e.g. Docker [24]), and connects virtual machines and containers and may use the server host's physical network. OVS also acts as an OpenFlow switch by exporting an external interface using the OpenFlow protocol in addition to the standard Ethernet switch. OVS can send packets to the SDN controller through OpenFlow and receive flow table upgrades from the controller during execution. OVS uses the OVSDB management protocol to export another external interface. This allows the configuration of a switch to be read, switches can be created or deleted and switch settings change (e.g. ports added or deleted and service quality configured). Both OpenFlow and OVSDB provides secure communication using the Transport Layer Security (TLS) protocol. Open vSwitch provides various different protocols (such as Virtual Extensible Local Area Network (VXLAN) to isolate different tenants within cloud environments, Generic Routing Encapsulation (GRE), and Virtual Local Area Network (VLAN)). Three components comprise an Open vSwitch implementation [66] as shown in Figure 3.5:

(i) **Slow path (ovs-vswitchd):** The ovs-vswitchd process implements the slow path in the user space. It includes the transmission logic and an interface of OpenFlow. It keeps the flow table as well. With the ovs-ofctl command or with OpenFlow the external SDN controller can manipulate the flow table manually. The SDN controller can monitor flows, obtain flow statistics and send packets to the switch using OpenFlow.

(ii) **Fast path (data path):** The fast path was initially implemented as kernel module, also known as the data path. The fast path is to transfer the packet. The data path keeps the flow table cached, while Netlink sockets are used by ovs-vswitchd to update the cache table with flows and related actions. Data paths are provided for packets that arrive from physical or virtual Network Interface Cards (NICs). The data path searches its cached flow table to determine how the packet is to be managed. Otherwise, the packet is forwarded to the ovs-vswitchd via the Netlink socket. Upon receipt of the packet, ovs-vswitchd sends the necessary cache table update along with the original packet to the cache table, which will be sent on the basis of a new flow table entry. The data path is therefore handled completely in the subsequent packets in the same flow. If ovs-vswitch can not find a match in its flow table, it sends the packet to the SDN controller, which decides how to process the packet and then sends a flow table update with the original packet and an action, according to the flow table [80] that was newly installed. The packet flow in fast path and slow path are described in Figure 3.6. The cached flow table can also be accessed using the ovs-dpctl command. The cached flow table was originally designed to support microflow caching, i.e. accurate matches on all the header areas of the packet. However, when a large number of short-term connections were installed, this caused performance degradation. This led to the introduction of megaflow caching, in which flows can be combined to create a two level cache. The Data Plane Development Kit (DPDK) was introduced to ensure the data path is implemented in user space, providing the packet processing of high performance. The network throughput result is higher and latency result is reduced [68]. Inter-process communication is done by using a ring data structure in common memory between the two user space processes (ovs-vschitchd and data path).

(iii) **Configuration database (ovsdb-server):** The ovsdb-server keeps a continuous database in which all configurations of the switch will be saved. The OVSDB management protocol exposes an external interface. Furthermore, the command

ovs-vsctl can be used to set up the database. Setting the IP address of the SDN controller, creating or removing switches, adding or removing ports and settings of service quality are examples of such configurations. The ovs-vswitchd process can query the database with OVSDB for switch configurations [80].

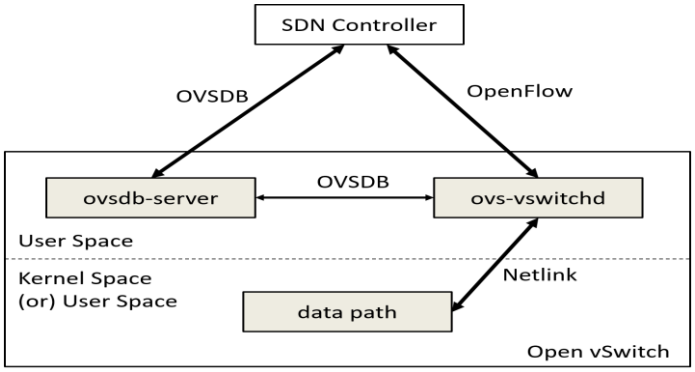

Figure 3.5 Components of Open vSwitch

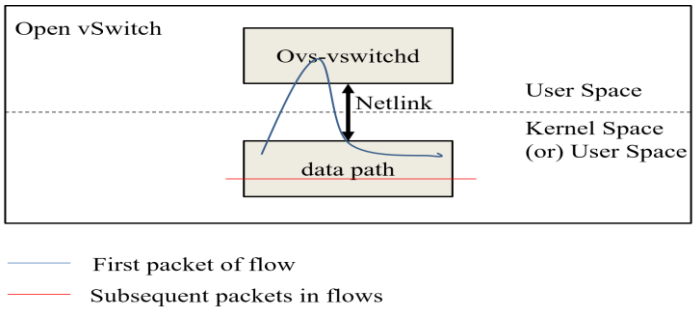

Figure 3.6 Packet Flow in Open vSwitch

3.2.2. Open vSwitch TLS Support

To connect to the SDN controller, OVS may be configured to use TLS. In OVS documentation [66], two certification authorities (CA) are distinguished: the CA for switch and the CA for controller. Each instance of OVS manually produces its own private/ public key pair, but it can use its own private key two ways for an OVS instance to sign certificates: self- signed certificates. This prevents a central authority from being required. The certificate must be manually submitted to the CA and manually supplied to an OVS by the signed certificate. The controller can validate the switch's certificate by using the switch CA's root certificate. Both switches and controller need to have the other's CA root certificate. These switches could either get the CA root certificate from the controller manually or by bootstrap mode [68] (where it accepts the CA root certificate from the controller for the first time).

3.2.3. OpenFlow Protocol

Openflow protocol has three message types:

- Symmetric: Submitted without request by switches or controller;
  (i) Hello: Connection beginning;
  (ii) Echo: Request/Reply, aims at testing connections and resilience.
- Controller to Switch: Controller initiated: may require a response or may not;
  (i) Features: Request/Reply, to get the switch capabilities and to get ports/flow statistics;
  (ii) Packet-Out: Explicit packet transmission instructions;
  (iii)Flow-mod: Remove and add specified flows;
- Asynchronous: Controller or Switches transmit messages driven by event;
  (i) Packet-in: In case of notification of the controller;
  (ii) Port-status: A switch port has changed its state;

Figure 3.7, as described below, shows the workflow of a packet-processing switch of OpenFlow. The switches are OF v1.0-based and usually integrated with the model TCAM single table. However, a single table for the implementation of flow entry rules can create a massive set of rules that severely restricts flow rules and failure to implement wide-scale applications, as TCAM (Ternary Content Addressable Memory) space is an inexpensive and shortage of resources. There are inefficient numerous attributes that can degrade the search and match the speed to be stored in an individual table. OF v1.1 (+) provides a multi-flow mechanism to flexibly and efficiently manage flow. As Figure 3.7 shows, an OpenFlow switch can be mounted on flow tables pipeline. The reduction of the single flow table to greater efficiency and standardized tables can enhance the use of TCAM resources substantially and accelerate the matching process.

In Figure 3.7, the first priority flow in table 0 must match the packet. Two conditions exist, if the packet matches or does not. If the packet relates to a table flow entry, the instructions or measures in the table fields will be applied. The Goto instructions can be used to indicate that the packet will go to another flow table other than the last flow table. This table stops the pipeline process when the flow entry for the match does not indicate a packet to another flow table. When the packet reaches to the end of pipeline processing, the appropriate instruction set is applied to the packet.

If the packet fails to match the flow rule entry of a table called “table-miss” which depends on the setup of OpenFlow table. The unknown packets are forwarded to the controller by default. Another option should be dropped in the packets.

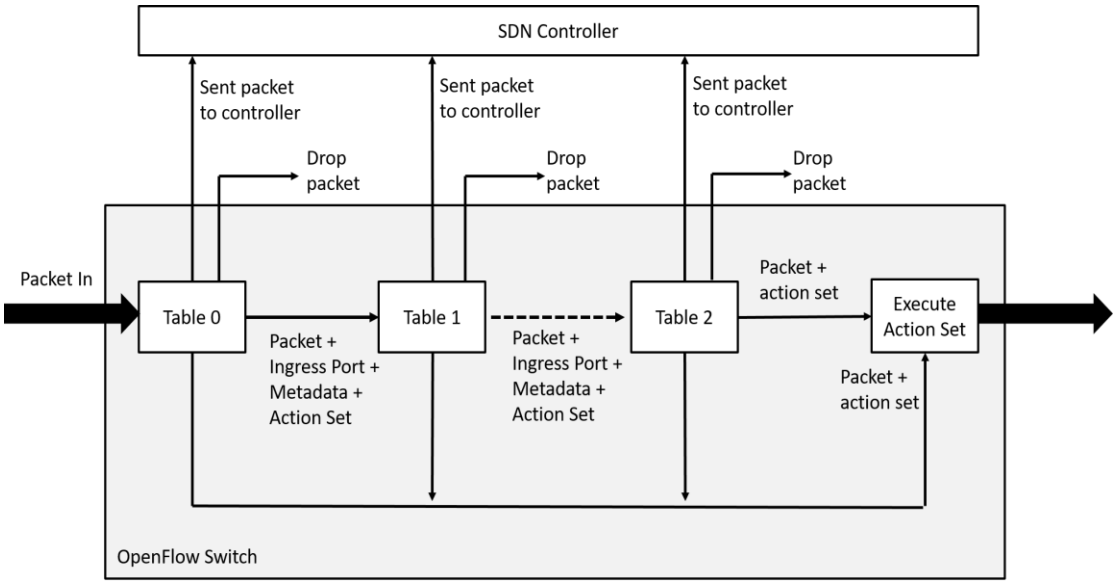

Figure 3.7 Packet flow over multiple pipeline tables under OpenFlow1.1+ [66]

3.3. Reactive, Proactive and Hybrid Flow Installation

The communication between the OpenFlow controller and the OpenFlow switch is possible in reactively, proactively or hybrid mode. The controller can configure the switch in three different ways:

(i) Reactive flow installation: According Figure 3.8, the switch performs a search in the flow tables when a packet arrives from host 1. The switch creates an OpenFlow packet-in packet and sent it to the controller to request instructions as no flow entry is set from the controller. A packet-out packet is sent by the controller to instruct the switch to transfer the packets to the port where host 2 connects. This instantiation model is referred to as reactive flow. The downfall of the model is the latency for the packets to be sent and received by the controller and the high message burden and CPU processing related to each flow being detected.

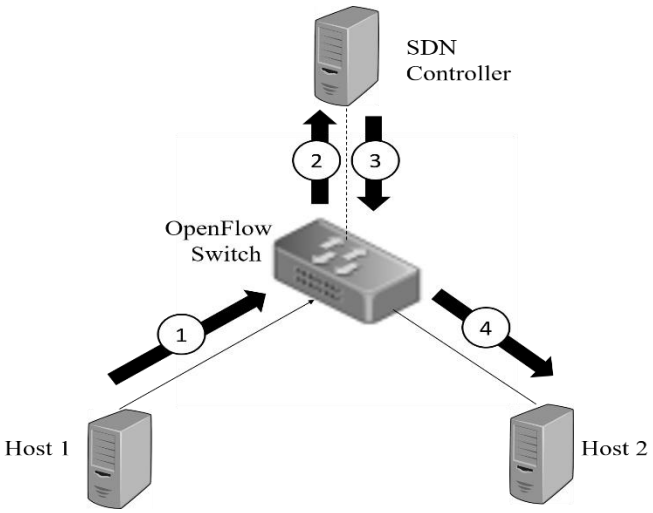

Figure 3.8 Reactive Flow Installation

(ii) Proactive flow installation: In this mode the controller can add the flow tables in advance and match all the flows from host 1 to host 2 as shown in Figure 3.9 rather than reacting to a packet. Proactive flow tables remove latency caused by consultancy on each flow by a controller. However, there is no flexibility when downsizing this model.

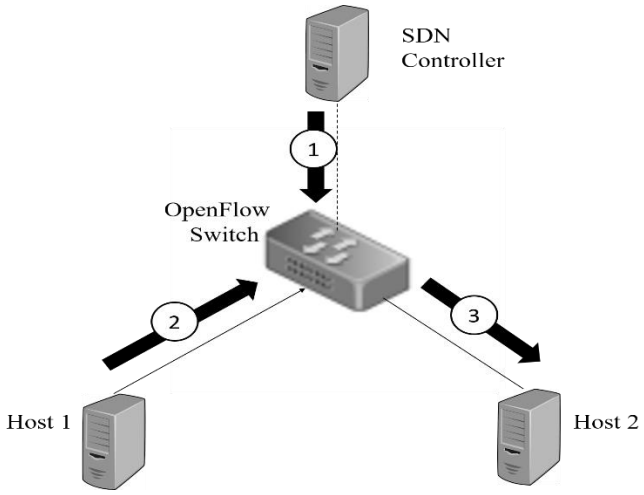

Figure 3.9 Proactive Flow Installation

(iii)Hybrid flow installation: The combined flexibility of reactive instantiation for specific sets and granular traffic control of proactive instantiation would leverage both approaches and still preserve low-latency transmission for the rest of the traffic. For instance, when the first packet arrives from host1 to the switch, it is sent to the controller and the controller inserts a flow entry, which will lead the packets to the host 2 with a finite idle time. If idle time is running out, the switch deletes

flow entries from the table and transfers the next packet to the controller. The most frequent method of deploying network in OpenFlow is hybrid flow instantiation.

3.4. Components in ONOS Controller

The mission of ONOS is "to create an Open Source Network Operating System that allows providers to build actual Software-Defined Networks"[65]. The ONOS Ecosystem consists of the Open Networking Laboratory, organizations including service providers, leading suppliers and other community members, who fund and contribute to the ONOS initiative. In December 2014, Avocet's first release open-sourced. There have been 17 releases to date. Quail version 2.0.0 (January 18, 2019), which focuses on optimizing performance, scalability, availability and building future generation SDN/NFV (Network Function Virtualization) solutions. Hummingbird (September 2016) provides that important improvements have been made in core control function areas and automation and legacy network configuration. In its l release, Kingfisher (June 2017), in terms of Northbound layer, Southbound layer and YANG model tools, among others, ONOS has been enhanced.

ONOS was designed to achieve the following objectives: modularity of the code, separation of concerns, configurability and agnosticism of the protocols. In particular, development principles define that new features, as independent modules should be allowed to be incorporated. There should be specific boundaries among the subsystems in order to achieve this. Furthermore, it is possible to configure a number of features, either on boot or during runtime. Finally, the platform should not be interconnected to specific protocols, but to be easy to implement modules that enable communication depending on different network protocols.

The ONOS-Kernel and core services, as bundles loaded into the Karaf OSGi container, are written in Java. ONOS can run on multiple OS-platforms, since it works on the JVM. The ONOS platform supports different categories of applications, including control, setup and management applications. The functionality levels of ONOS are architectured. It consists of a group of sub- projects, each with an independent source tree. Each subsystem performs and executes a service in application layers, core and southbound protocol. Its main supplementary systems are Device Subsystem (infrastructure inventory), Link Subsystem (infrastructure link inventory management), Host Subsystem (manages end-station hosts inventory and their network locations), Topology Subsystem (manages network diagram view snapshots), Path

Service (Find paths from infrastructure or end station hosts with the latest topology diagram instantaneous), Flow Rule Subsystem (manages inventory and flow metrics of match/action flow rules installed in infrastructure devices), and Packet Subsystem (enables applications to listen to network device data packets and send data packets over the network using one or more network devices) [24].

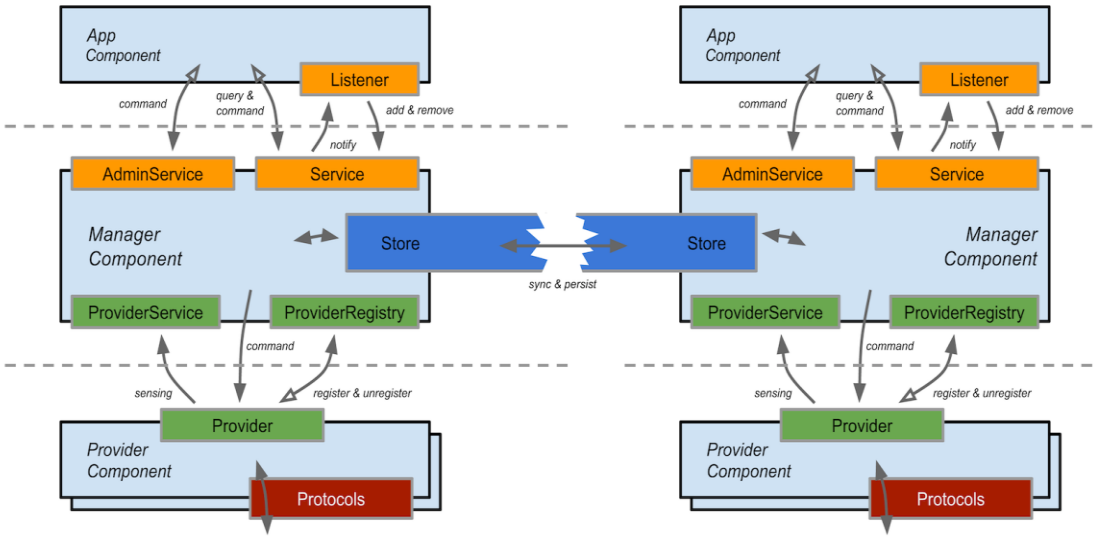

Figure 3.10 ONOS Subsystem Structure [65]

The structure and interconnection of subsystems is represented in Figure 3.10. The ONOS core layer shows the two interfaces, the administration service and service, used to invoke different service components in the core by applications. Each application registers to a core service that provides a unique application identifier for the application. This identification is used by ONOS for monitoring applications ' tasks and objectives, such as intents and flow rules. In a similar fashion, protocol-aware providers interact using several control and setup protocols with the network environment. Furthermore, providers collect data from other subsystems to convert them to specific data for service. A provider is linked to a Provider Identifier (ID). As is the case with the northbound application ID, ONOS also uses this ID, which is assigned to every southbound provider. The Provider ID serves to identify the devices uniquely and to correctly map them. Finally, multiple providers can be linked to a single subsystem from a subsystem perspective. Multiple providers support a device subsystem. The Intent Framework illustrates a subsystem. In the ONOS core, it is a subsystem. Applications can intentionally define their preferences for network control in the form of policy rather than a mechanism [65]. The intent is a model object, which describes the ONOS core application to alter the behavior of the network. The network

resource, constraints, criteria and instructions are described in detail. The framework mainly comprises intent compilers, which translate attempts into installable, network-specific intents, and coordinators who decide how to program the network, including the device- level installation order. The framework includes translation and compilation, support for network change and optimization across purposes and other functions.

In ONOS, support for distributed architecture is also an important design principle. It can be used as a collection of controller servers that coordinate to achieve resilience, fault tolerance and better management of load. There are different problems, as with traditional distributed architectures, to achieve this. One is cluster coordination, which is achieved by including a distribution mechanism in the various subsystems which generate events in the case of ONOS. To ensure availability, ONOS uses methods such as vector clocks, distributed queuing and queue-sharing groups. Vector clocks is an algorithm for generating a partial ordering of events in a distributed system and detecting causality violations [65]. The optimistic replication technique and an anti-entropy mechanism based on gossip-protocol and periodic node testing to ensure consistency between events and host management are used. This mechanism is also used to deal with failure cases (i.e. unavailability of nodes). In addition, ONOS gives the northbound abstraction to the networking graph and the whole network view. This global network information is logically centralized even if it is distributed physically across multiple servers. Each ONOS instance provides a global network vision of network topology and status, such as a switch, ports, links and host information.

ONOS aims to support multiple protocols (OpenFlow, NetConf, etc.), to communicate with a diverse device at the southbound interface, and to expose APIs on the northbound interface to meet the needs of service provider applications and development companies. If ONOS needs to support a new protocol, a new network module could be built into the system via the southbound API as a plugin. ONOS uses the concept of providers like other control systems (i.e. Open DayLight ODL), which hide the complexity of the protocol from other platform components. These providers provide the core layer with all the necessary network element information

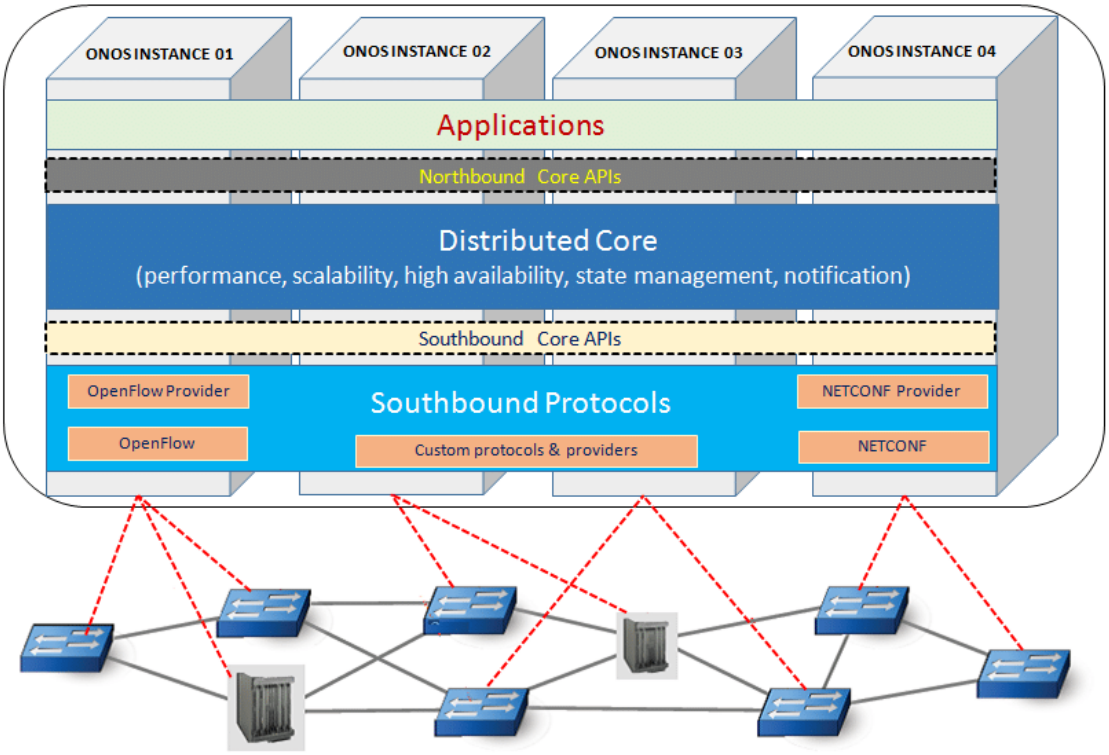

Figure 3.11 ONOS Architecture [29]

3.5. Elephant Flow Detection in SDN

Elephant flows generally transfer large amount of data to the network, which can lead to network congestion and network performance effects. The accurate, timely and cheap detection of elephant flow in the efficient management of the network is therefore crucial. A newly evolved change appears to take advantage of SDN in elephant flow detection.

Typical network environments (e.g. the campus network and data center) have become increasingly complicated, making efficient network performance optimization a big challenge. The phenomenon of elephants and mice suggests that the efficient detection of elephant flows is a good choice to tremendously enhance network performance. Elephant flows can generally transmit much more traffic over the network, even though they are small in number, including ftp, backup, VM migration, etc. And mice flows, like HTTP requests, web search etc, could transfer much less. Thus, elephant flows are more likely to affect network congestion compared to mice flows. It clearly increases network performance by detecting and scheduling the elephant flows with the efficient consideration of network management.

With regards, elephant flow detection, three assessment indicators typically exist for the detection of elephant flow: the precision, timeliness and network overhead. The importance of these three are not the same. High accuracy is the most important for elephant flow detection, since detection errors can result in significantly less network efficiency. Therefore, improving timeliness or reducing the network overhead at the expense of precision is not a good idea.

3.5.1. Polling

A periodic polling of the edge switch is used to ensure traffic change in the network for Hedera [5] and Helios [26]. In theory, the network traffic can be captured to maximize as long as it maintains the high polling frequency. However, this can lead to great overhead monitoring that can affect the transmission of normal flows. Payless [20] proposes an adaptive flow statistics polling approach to reduce the monitoring overhead while ensuring collection accuracy. With varying network traffic, the polling frequency can be adjusted. The polling frequency will be increased when traffic changes rapidly. Another way is to decrease the query frequency.

OpenTM [84] selects the optimum set of edge switches from the perspective of the network as a whole by creating a mathematical model to minimize poll requests from the controller for flow- statistics. Although the periodic polling has been to some extent optimized by PayLess and OpenTM, overhead monitoring caused by this method can not still be ignored.

In [56], it uses a dichotomy-like approach to survey flow statistics step by step, taking advantage of the regularity of the IP address allocation. The controller initially surveys the aggregate flow statistics for several IP ranges actively. The controller compares the results with the previously assumed threshold after receiving the polling result. If the outcome exceeds the pre-defined threshold, it will divide the IP range and proactively polled the IP range of each subdivision again. Continuously, the IP range is constantly restricted until the elephant flow is identified. The overhead monitoring caused by elephant detection can be effectively reduced by this approach.

In [38], to optimize the TM (traffic matrix) estimation problem, it uses SNMP linking counters and SDN counters. The estimated TM indicates that this approach first finds the Talky ToR (Top of Rack) pairs and then the Talky ToR pairs show elephants flow.

3.5.2. Sampling

This technique uses the mathematical statistical method to collect partial packets of the target flow to infer the overall information of the flow statistics. The flow statistics collection overhead can be dramatically reduced. In [2], the sFlow technology provides real-time flow monitoring. sFlow-RT is used to analyze the collected flow data and to determine the elephant flow in the network based on the supposed (or) predefined threshold. First, the SDN controller can determine the flow set in the network that needs to be monitored. sFlow, by contrast, can flexibly adjust the sampling frequency for flow sampling. In [81], it proposes a dynamic adjustment of the sample frequency to reduce the overhead of sampling. The sampling frequency is lowered when the number of packets received is large. Instead, the frequency of sampling is increased. While the overhead from the collection of flow statistics is reduced, the sample technology generally sacrifices the accuracy of the collection. The reason is because the sampling technique performs packet analysis according to the method of mathematical statistics, which could cause major errors. OpenSample [82] improves the analysis of sampled packets by using the TCP packet header sequence number to deduce flow transmission bytes quickly and accurately. The proposed method in [3] compensates the flow packets with less precision of the sampling technique and gives the corresponding flow statistics and then selects the suspected elephant flows; this paper adopts its precise flow statistics to be actively examined and further identifies it using the new flow statistics.

3.5.3. End-Host Based

Mahout [22] uses a kernel patch to monitor the flow statistics generated by the host and determines whether an elephant flow is available based on the supposed elephant flow threshold. If there is an elephant, the controller is notified that the central traffic schedule is executed. Mahout proposes that an in-band mechanism used to notify the controller of the elephant flows to reduce communication overhead. The switch will forward the respective packets according to the default flow entry, once the marked elephant flow arrives the switch. Like Mahout, the entire network traffic assessment can be carried out by MicroTE [10]. The network traffic is collected, aggregated and reported to the controller on time by designating the monitoring end host in every ToR.

However, because virtual machines generate a network traffic at the end of a host, the virtual traffic control can not be carried out by the deployment of a kernel patch.

The widespread popularization of virtualization technology often means that virtual machines are used economically in the network. When virtual machines are deployed in end host systems, traffic monitoring does not just mean the normal traffic on the network, but also the traffic of the virtual network. EMC [59] proposes to collect flow statistics with the hypervisor deployed on the end host, depending on monitoring tools like sFlow and NetFlow supported by Open vSwitch (OVS). However, for further analysis the collected data must be forwarded to the centralized flow collector, which can lead to overhead monitoring. The paper [99] uses OVS monitoring on the end host without the use of the ready- to- use monitoring tools. Each OVS monitors all the traffic sent to that end host by all OVS hosts. Using the supposed elephant flow threshold, the OVS monitor detects if an elephant flow exists. If available, it will notify the controller module for the elephant flow detection. VirtMonE [8] proposes two ways to notify the elephant flow information controller for the purpose of further reducing network overhead monitoring. One is like Mahout [22], using the differentiated IP Notification service fields. The second is that the controller can store the information from the elephant flow into the OVSDB by periodically querying the OVSDB. There are many benefits in the detection of elephant flows in the end host. However, specialized monitoring modules for the end host often need to be included. Given the growing size of the network, it is usually expensive to deploy.

3.6. Functional Components of Multipath Forwarding Mechanism in Software-Defined Network

Two main components are the critical part of multi- path transfer. First component is how incoming traffic is divided (traffic divisions) and second component is how traffic is translated into multiple paths. The taxonomy of these two tasks is generalized. The various forwarding mechanisms operate in different ways according to their traffic splitter and track selector functions as described in Figure 3.12. The traffic splitter divides the incoming flow (packet sequence) into path selector components at certain granularity. The path selector is responsible for selecting path based on certain conditions for traffic splitter. Table 3.2 summarizes the current existing TE solutions in SDN which are presented in Chapter 2 with multipath functional components.

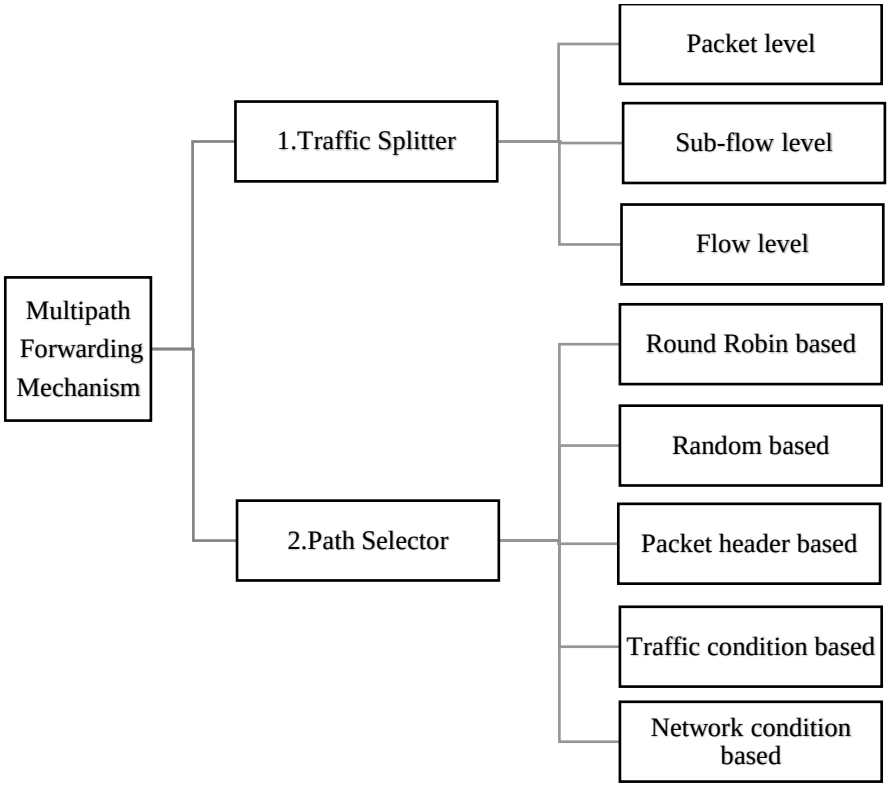

Figure 3.12 Classification of Multipath Forwarding Mechanism Components

3.7. Traffic Splitter Methods

The methods of traffic splitters differ depending on the traffic granularity level. This section describes the traffic splitter classification.

3.7.1. Packet-Level Splitter

The splitter on packets level (see Figure 3.13) divides the incoming stream into the smallest basic unit as a packet and extends over alternative pathways [4, 23, 34 and 71]. A multi- path transmission mechanism with this type of traffic division is known as a multipath transmission based on packets.

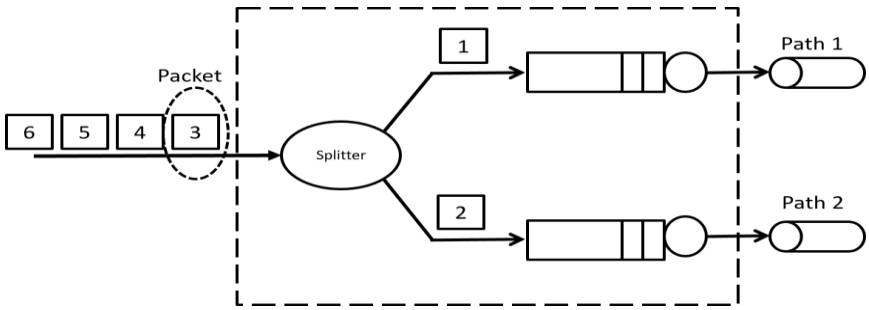

Figure 3.13 Example of Packet Level Splitting

3.7.2. Sub-Flow Level Splitter

Packet-based splitting assigns to each path fast the desired load share. However, divisions on packet granularity can reorder a large number of packets if paths differ in delay. TCP confuses this reordering, which leads to a degraded performance, as a sign of congestion. Even certain UDP- based applications like VoIP [52] are sensitive to rearranging packets. The splitter splits at sub flow level (i.e. the subset of packets in an original flow) or packet burst, which are called flowlets in Figure 3.14 [1, 7, 14, 25, 34, 60, 63, 85 and 96], not switching packets. A multi- path transmission mechanism with this kind of traffic division is called a flowlet- based transmission.

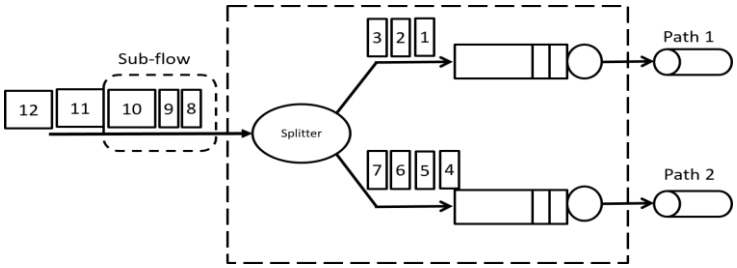

Figure 3.14 Example of Sub-Flow Level Splitting

In splitting conditions, different flow-characteristics, like packet inter-arrival rate and packet arrival rate may be considered depending on the objective of the network policy.

3.7.3. Flow Level Splitter

Dividing by flow as shown in Figure 3.15 pins each flow towards a particular path and avoids packet reshuffling. However, their flows vary greatly in size and rates, and once assigned, the path lasts for life. As a result, flow-based division requires accurate traffic quantities to be assigned to each path and the load must be readjusted rapidly in response to changing demands [4-5, 18-19, 21, 25, 27, 39, 41, 43-45, 50, 54-55, 58, 64, 72, 76, 78, 88, 90, 92, 97]. A transmission mechanism with this kind of traffic division is called a flow-based transmission.

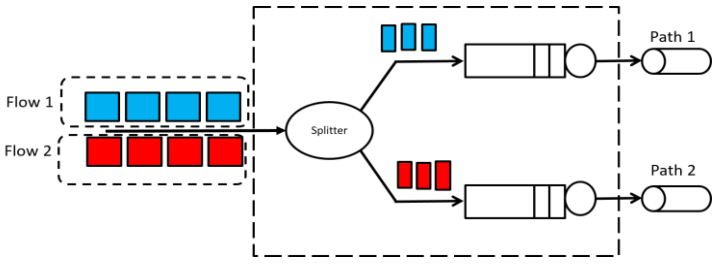

Figure 3.15 Example of Flow Level Splitting

3.8. Path Selector Methods

The path selector selects the right route for splitter traffic. The path selector module must identify the path of each packet if the splitting module splits at the single packet level. The selector will only find the path when a sub-set of packets or flow is reached if the divider splits at either sub- flow or flow level. The path selector is different based on classification type: Round-Robin based, random based, packet header based, traffic condition based and network condition based.

3.8.1. Round-Robin Selector

The Round-Robin selector distributes traffic in a round-robin way through alternative paths. This is the simplest selector system and does not affect traffic or network conditions.

3.8.2. Random-Based Selector

Random selector selects the paths for the random distribution of traffic load. The traffic and network conditions are not considered by this method.

3.8.3. Packet Header Based Selector

Some tuples of header information may be used in packet- based header- based method: source address, destination address, port number, service type, protocol number, etc. Taking into account a source address, service type and protocol number, each flow can be differentiated according to its source, service class and network application type. For example, the selector chooses the low delay path for the traffic of the delay sensitive application (i.e. VoIP) and chooses high throughput path for the traffic of throughput sensitive application (i.e. video streaming).

3.8.4. Traffic Condition Based Selector

In this selector, traffic conditions play a main role in path selection. The critical traffic conditions include traffic rate, traffic load, traffic size, traffic arrival rate and the number of active flows in network, and paths are selected relies on these critical objectives.

3.8.5. Network Condition Based Selector

Various types of network condition such as path delay, jitter, packet loss, available bandwidth and queue length are used to determine the outgoing path according to the goal of routing policy. Shortest-Path-First (SPF) and Least-Loaded-First (LLF) are some of the most well known path selection schemes. In SPF, a path with the lowest cost (generally in terms of hop count) will be selected for an arrived packet. In LLF, a path having the smallest amount of load will be chosen.

3.9. Chapter Summary

This section gives outlines of different existing multipath transmission methods. Each method is depicted as far as its functional components in multipath transmission mechanisms, in terms of traffic splitter and path selector, which plays as critical part in traffic management.

The original traffic flow can be split into a few levels. With an alternative traffic distributor, the multipath transmission methods show the different characteristics, i.e. packet level based traffic splitter enables the multipath transmission method to accomplish load balancing, while at the same time flow level based traffic distributor enables the multipath transmission method to keep up packet ordering. The classification and analysis of the functional components will give a thorough comprehension of different multipath transmission methods.

Comparing to the specific functional components, different cases of multipath selection methods are classified into four different classes, in particular, (i) *Round-Robin based*, (ii) *random based*, (iii) *packet header based*, (iv) *traffic condition based*, and (v) *network condition based* methods. Moreover, the performance evaluation environments are also described in overview of literature review. According to the literature surveys, some methods use only one type of class. On the other hand, some combine one or more classes to be better multipath transmission schemes. The *Round-Robin* and *random based* multipath transmission methods are the simplest methods which have low complexity because they do not incur the operational cost to monitor the traffic and network. However, these two methods cannot work efficiently in high demand network. The packet header based multipath transmission methods may work perfectly for mice flows or latency sensitive flows in network. However, for elephant

flows, it can cause flow collision problem that makes performance degradation. The *traffic condition based* transmission methods require information with respect to traffic load in making a decision on traffic splitter and path selector. Path selection can be controlled precisely in such methods. Also, some methods do not distribute for all flows, they differentiate elephant and mice flows, and only distribute the elephant flows by exploiting some knowledge of traffic condition such as flow rate, traffic load and utilization. On the other hand, network condition based multipath transmission method enables to select path to adapt to network conditions. Based on the near real-time network conditions, the path selector can select the reasonable path conditionally. Most of literature review choose the appropriate path for traffic flow based on link utilization.

Table 3.2 A Brief Overview of Multipath Forwarding Mechanisms in SDN

Traffic Splitter	**Multipath Forwarding Mechanisms**	**Path Selector**					**Performance Evaluation Environment**
		Round-Robin Based	**Random -Based**	**Packet Header Based**	**Traffic Condition Based**	**Network Condition Based**	
Packet level	Fair Forwarding [4]	*			*		Mininet
Sub-flow level	TinyFlow [85]		*	*	*		NS3
	MPTCP		*				Real Testbed
	Suchandra [96]		*		*		Mininet
	Flare [63]					*	NS2
	Multipath Transport [1]	*				*	Mininet
	DifffFlow [14]		*	*	*		Simulator
	Chunked Video [34]					*	OF@TEIN Playground
Flow level	Hedera [5]				*	*	Real Testbed
	Yi Rou et al [18]					*	Mininet
	SDN-based ECMP [19]				*		Mininet
	Congestion Control [27]					*	Mininet
	OpenQoS [25]				*	*	Real Testbed
	Adaptive Routing[[97]					*	Mininet
	Congestion Avoidance [45]					*	Mininet
	RepFlow [51]				*		Mininet
	OpenE2EQoS [55]				*	*	

	Yuan-Liang et al. [50]					*	Estinet
	CAMOR [78]					*	Mininet
	DMSP [39]	*		*	*		Mininet
	Trong Tien et al [64]						Mininet
	L. Bo et al [13]				*	*	Mininet
	Li et al [21]				*	*	
	SynRace [76]					*	Mininet
	Jing Li et al [58]				*		Mininet
	Chenhui et al [90]				*	*	Mininet
	OFLoad [86]				*	*	Prototype Setup
	ECMP [35]			*			
	FreeWay [88]				*	*	Htsim
	Local Reroute [44]				*		Omnet++
	Yu et al [54]				*	*	Mininet
	Maris Fajar et al [72]				*	*	Mininet
	Eric Jo et al [43]				*		Mininet & MiniSSF

CHAPTER 4
THE PROPOSED SYSTEM ARCHITECTURE

This chapter identifies the delay-related problems and motivations, and then proposes Delay-Aware Elephant Flow Rerouting (DAEFR), which can optimize the network throughput, reduce flow completion time and packet loss.

4.1. Problems and Motivations

For a variety of network applications, multipath forwarding methods were applied in different networks and research has been studied over many years on multipath forwarding algorithms. However, as indicated in the literature review, latency is not the focus of most investigations, and QoS is significantly impacted by the transmission of large amount of data through network applications. A major driving force for this work is the demand for network infrastructure to provide lower latency and lower load network services that support QoS-sensitive applications.

4.1.1. Delay-Related Issues

Total packet delay is the time for a packet to be passed, i.e. end-to-end delay in the packet transmission and extra time in the packet reordering recovery. End-to-End delay ($D_{E\text{-}E}$) refers to the time taken for a packet to travel from one source to another network destination. It consists of nodal processing delay (D_N), queue delay (D_Q), transmission delay (D_T) and propagation delay (D_P), i.e. the end-to-end delay: $D_{E\text{-}E} = D_N + D_Q + D_T + D_P$. For multipath routing, end-to-end difference delays between multiple paths can result in greater network congestion and degradation applications' QoS performance. Therefore, before routing traffic load, end-to-end delay of multiple paths is required to estimate in order to be an effective multipath approach.

In flow-based transmission methods, the conditions of the traffic and the network situation play a central role in the transmission mechanisms to transmit traffic efficiently between multiple paths. A large end-to-end delay and a significant delay difference between several paths lead to traffic imbalances. The large delay difference leads to significant network performance degradation, leading to a large D_Q. Flow-based models can prevent reordering packets entirely. Therefore, in estimating the total

packet delay time, it does not need to consider additional packet reordering time. The major disadvantage of the flow-based models is that they are unable to handle variations in flow size distribution, which leads to load imbalance. Flow-based models can cause large packet delay variations affected by overloaded path and, consequently, the large D_Q (causing a large end-to-end delay) on a particular path.

4.2. The Preliminary Studies on Elephant Flow Rerouting

The basic concept [30] is to reroute elephant flow based on average end-to-end delays of parallel paths between source and destination. The sFlow real time analyzer is used for monitoring and detecting elephant flows. In order to access the elephant flow information from rerouting application, the sFlow REST API is called periodically. The new elephant flow event can be defined in proposed rerouting method by comparing the time stamp values of elephant flow events since sFlow REST API provides flow information with time stamp values.

Algorithm 4.1 Flow Rerouting Algorithm based on End-to-End Delay

Input : mice flow, elephant flow
Output :least_delay_path
Initialize: $D_{total} = 0$, least_delay_path = MAX_VALUE
if Elephant Flow exists **then**
Find available shortest path-list between S and D;
for each path p in path-list do
  for each state $i \in 1,2,\dots,N$ **do**
    Measure end-to-end delay d_i, for p;
    $D_{total} \leftarrow D_{total} + d_i$;
  end for
  Calculate average delay $D_{avg}[p] = \frac{D_{total}}{N}$;
  least_delay_path = $min(D_{avg}[p], least\_delay\_path)$;
end for
Install flow rules to least_delay_path;
else
  Use reactive forwarding;
end if

According to Algorithm 4.1, as soon as the elephant flow is found, firstly it finds available shortest path list in terms of hop counts between source S and destination D nodes. Then end-to-end delay (d) of each path from path list are measured by sending out probe packets from the controller. In previous implementation, the number of estimation N is 10 for each path. When the total delay D_{total} is computed for each path, the average delay D_{avg} can be calculated. After comparing the average end-to-end delays of available paths, the elephant flow is shifted to the least delay path to optimize throughput and flow completion time. For TCP traffic flow, the least and second least delay path are selected. In general, three main modules: monitoring and detecting elephant flows, estimating end-to-end delays and flow entry installation are developed for ONOS application. The problem statement of this concept is that it can only be effective for TCP traffic flow because the UDP traffic does not sensitize to latency and it is sensitive to congestion. Therefore, in order to be effective rerouting mechanism for both TCP and UDP flows, the improved work consider not only end-to-end delay but also link load utilization. The next limitation is that the weak point of delay estimation module. The previous delay estimation method can only estimate the delays of available paths in simple topology. Moreover, it can not handle parallel elephant flows which are generated from same source host and destination host. The details of previous delay estimation method is explained in next subsection. In improved work, the delay estimation method is worked well for any symmetric network topology and handled multiple flows from any host to any destination.

The basic concept [31] has been modified to adapt in two-level fat-tree topology and still handle only TCP traffic flow. It has been modified the faked MAC address of probe in measuring end-to-end path delay. For example, assume to measure the delay of the path: S1-S2-S3, both of faked source/destination MAC address of this path will be 11:11:22:22:33:33. Although it could adapt in two-level fat-tree topology, it still has challenges to adapt in n-level fat-tree topology. Therefore, to address this dependency of faked MAC address, the next solution [32] presented the new scheme in faked MAC address creation. In this study, the faked MAC address of probes are created based on hashing of some header values.

$$MAC_{Probe} = MD5(MAC_{src}\ ,\ MAC_{dst}, IP_{src}, IP_{dst}, Port_{src}, Port_{dst}, Path) \quad (4.1)$$

For example, as shown in Figure 4.1, let's assume that elephant flow occurs between source node A and destination node B. The elephant flow information from sFlow is as follows: aa:aa:aa:aa:aa:aa, bb:bb:bb:bb:bb:bb, 10.0.0.1, 10.0.0.2, 60053 and 5001.

Firstly, the possible shortest paths between node A and node B are computed path 1: A-C-B and path2: A-D-B. Therefore, the probe MAC address for path1 (A-C-B) will be hashed aa:aa:aa:aa:aa:aa, bb:bb:bb:bb:bb:bb, 10.0.0.1, 10.0.0.2, 60053, 5001, A-C-B, and path2 (A-D-B) will be hashed aa:aa:aa:aa:aa:aa, bb:bb:bb:bb:bb:bb, 10.0.0.1, 10.0.0.2, 60053, 5001, A-D-B. In delay estimation, all probe packets are handled by the single controller. The problem statement of [32] is that it cannot still handle UDP flow and hash value collision of probe packet. This hash collision problem can degrade the accuracy of end-to-end delay results. So, to be unique probe for each elephant flow, instead of hashing header attributes, unique identifier value is used as a faked MAC address of probes.

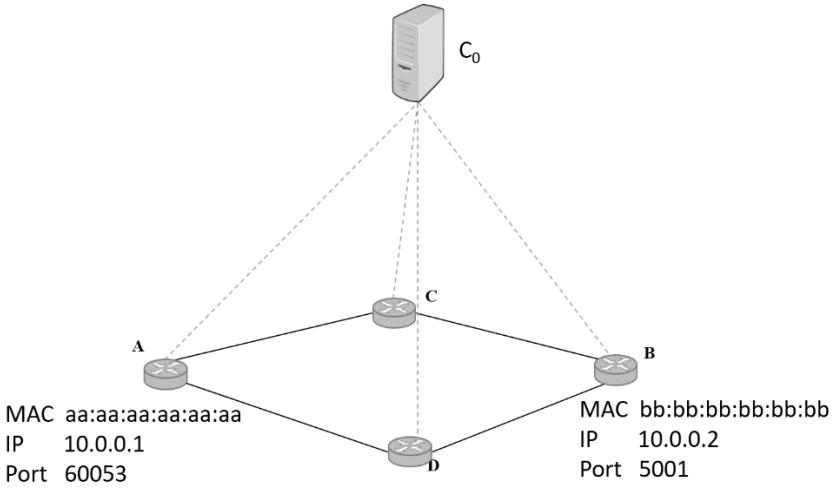

Figure 4.1 Example of Hashing Faked MAC Address

4.3. Architecture of Delay-Aware Elephant Flow Rerouting (DAEFR)

The basic concepts of research work and problem statements are discussed in Section 4.2. To solve these problem statements, Delay-Aware Elephant Flow Rerouting (DAEFR) is proposed (see in Figure 4.2), which can outperform the existing models in solving the delay-related problems. The Figure 4.2 shows the functional block diagram of DAEFR. There are five main components in DAEFR method: (1) detecting elephant flow, (2) computing shortest path, (3) estimating end-to-end delay and link load utilization, (4) calculating least cost path and (5) flow rule installation. The sFlow engine monitors the network continuously and detect elephant flows. As soon as the elephant flow is detected, sFlow generates the event, which has source/destination MAC addresses, IP addresses and port number. The proposed DAEFR method needs

to access the large flow events from sFlow periodically. When DAEFR finds the new elephant event, it computes available shortest paths between source and destination MAC addresses. This available shortest path list becomes the input for estimating end-to-end delay and link load utilization. Then, DAEFR compares the estimated end-to-end delay and current link load for each path and calculates the least cost path. Then, the ongoing traffic flow is rerouted to this least cost path to improve network performance. The DAEFR takes into account of incoming flow size, end-to-end path delay and the link load ratio, which are locally available information, in determining the traffic rerouting vector in flow management module, and thereby properly responding to network condition without additional network overhead.

In the path selector, the end-to-end delay estimation method has been implemented, which does not need to synchronize the network devices. This is suitable for the proposed work since the calculated link load is typically not feasible for finding best route. The end-to-end delay estimation method for multiple flows and adaptive path cost calculation, which are improved from the previous work [30-32], are detailed in the previous section.

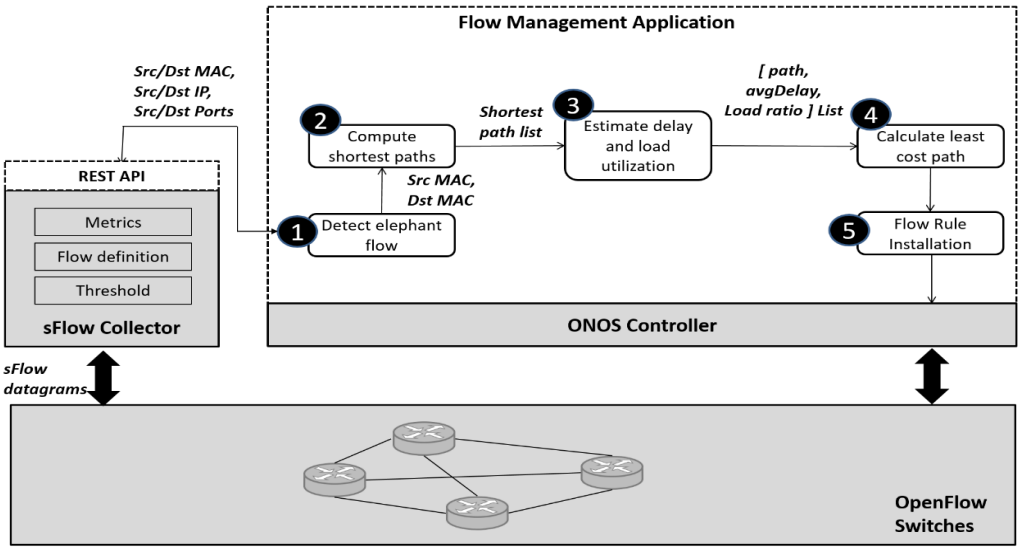

Figure 4.2 Description of Delay-Aware Elephant Flow Rerouting (DAEFR)

4.3.1. Detecting Elephant Flow

Elephant detection can be done in several ways: maintaining and polling per-flow statistics, packet sampling, and end-host based monitoring. Per-flow statistics as used in Hedera [5] has high accuracy at the cost of poor scalability in commodity switches. End-host based monitoring such as Mahout [22] overcomes the scalability issue, though

it has not been widely adopted. In DAEFR design, packet sampling method is used, since it is widely used in practice with mature switch support such as sFlow-RT [85].

Since the architecture of sFlow-RT is based on collector and agents (see in Figure 4.3), sflow agents are needed to embed in network devices (eg. Open vSwitch) to be monitored as shown in Figure 4.3. The sFlow standard is used in the switches/ routers using a separate Application-Specific-IC (ASIC), enabling traffic at the wire speeds on all interfaces to be continuously monitored at application level. SFlow and OpenFlow combined offer an integrated flow monitoring system that enables the OpenFlow controller to define the flows that will be monitored by sFlow. In addition, sFlow metrics can be used to control the forwarding behavior of the switches through a SDN application as feedback. The sFlow agency is a software system that runs within a device as shown in Figure 4.5 as part of the network administration software. It flows into sFlow datagrams samples and interface counters, which are sent to a sFlow collector via the network. Sampling of packets is in particular carried out by the switching/routing ASICS, giving wire-speed performance. The condition for each sample packet is also recorded for the forwarding / routing entries.

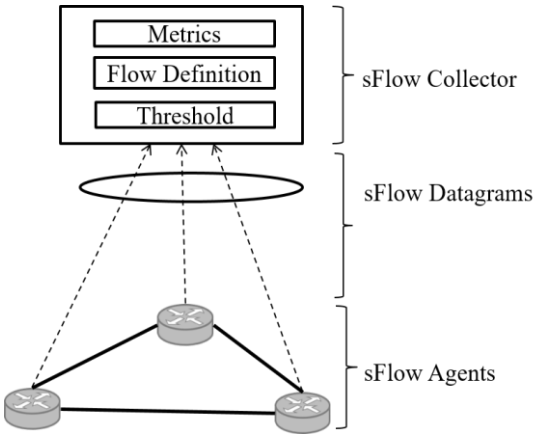

Figure 4.3 sFlow Infrastructure

```
SetFlow('tcpflow',
        {keys :'macsource,macdestination,
               ipsource,ipdestination,
               tcpsourceport,tcpdestinationport,
               link:inputifindex,link:outputinfindex',
         value:'bytes'});
```

```
SetFlow('udpflow',
        {keys :'macsource,macdestination,
               ipsource,ipdestination,
               udpsourceport,udpdestinationport,
               link:inputifindex,link:outputinfindex',
         value:'bytes'});
```

Figure 4.4 Flow Definitions

Two threshold values according to Table 4.1 and two flow definitions are implemented in sFlow collector (see in Figure 4.4). A flow is defined using name, keys and value. Every sFlow agents send the continuous streams of sFlow datagrams to collector. The sFlow datagram contains encapsulation and header information of sampled packet from individual flows. The collector computes the flow rate of sFlow datagrams in every second. Flows have to be defined in order to get them detected as elephant flows. sFlow is set for large flow detection by defining a large flow as a flow consuming 10% of the link bandwidth for one second as shown in the table in Table 4.1. According to the table, for example, the link speed given is 10Mbit/s, so a threshold of 1Mbit/s (10% of the link bandwidth) has been defined in DAEFR application. This threshold is set so that any flow that consumes 10% or more of the link bandwidth is recorded as a large flow. If the flow rate exceeds the predefined threshold, the collector generates the elephant flow event and converts header information into metrics based on flow definition.

Table 4.1 Threshold Values for Elephant Flows

Link Speed	Elephant Flow	Sampling Rate	Polling Interval
10 Mbps	1 Mbps	1-in-10	10 seconds
100 Mbps	10 Mbps	1-in-100	10 seconds
1 Gbps	100 Mbps	1-in-1000	10 seconds
10 Gbps	1 Gbps	1-in-10,000	10 seconds
100 Gbps	10 Gbps	1-in-100,000	10 seconds

In proposed architecture, there are two flow definitions, as it is needed to detect TCP and UDP elephant flows. According to the flow definition (see in Figure. 4.4), the output information of elephant flow includes source and destination MAC addresses, IP addresses, TCP/UDP port numbers and the names associated with the ports of a link. The sFlow-RT sits in the control plane of the SDN stack. It converts the received datagram into actionable metrics or summary statistics based on the flows defined by the user. Any language that supports HTTP request messages (Perl, Python, Java, Java script etc.) can be used to retrieve metrics from sFlow-RT. sFlow-RT statistics can be retrieved in JSON format. JSON encoded text based results are easy to read and widely supported by programming tools. Following URL is used to retrieve JSON encoded

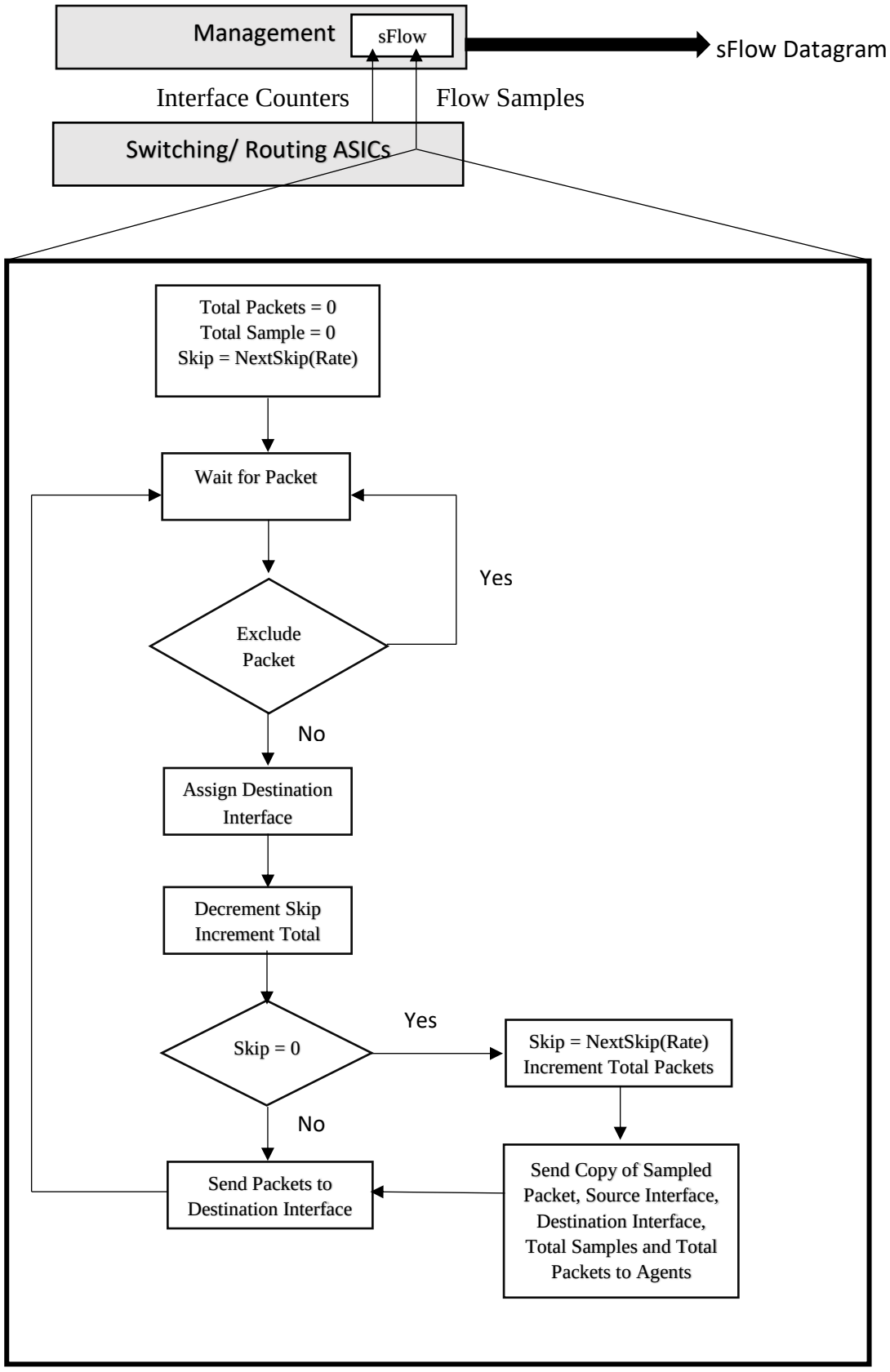

Figure 4.5 sFlow Agent Embedded in Switch/Router

metrics from sFlow-RT: http://server:8008/metric/agents/metrics/json where server is the host running sFlow-RT, agents are semicolon separated list of host addresses or names, or ALL to include all hosts and metrics are comma separated list of metrics to retrieve.

In order to access the elephant flow information from elephant flow rerouting application, the sFlow-RT REST API: /events/json which is used to filter the threshold exceed events, is called periodically. In the DAEFR method, the REST API calling interval is set 1 second. The new elephant flow event can be defined in rerouting application by comparing the time stamp values of elephant flow events since sFlow REST API provides flow information with timestamp values. The output metrics are represented by JSON format which consisting of attribute-value pairs. According to the flow definition (see in Figure. 4.4), the output information of elephant flow includes source and destination MAC addresses, IP addresses, TCP/UDP port numbers and the names associated with the ports of a link and the value means the flow rate will be calculated in bytes.

4.3.2. Computing Shortest Paths

To find available shortest paths between source/destination pair where elephant flow happens, ONOS [65] controller provides DijkstraGraphSearch as a module. Its primary usage is in TopologyManager and flow management application invokes it associated with source/destination MAC addresses whenever elephant flow is detected.

4.3.3. Estimating End-to-End Delay

After computing available shortest paths between the source and destination hosts, the end-to-end path delay and link utilization are needed to measure for these paths. The proposed method of measuring delay (see in Figure 4.6) is based on sending a probe packet from controller, and then sent to source switch which will forward it to the destination switch along the path, and finally return back to the controller. The other delay from controller to both source and destination switches are calculated separately each other.

End-to-end delay estimation comprises three main tasks: (i) probe packet creation, (ii) delay estimation and (iii) probe packet processing.

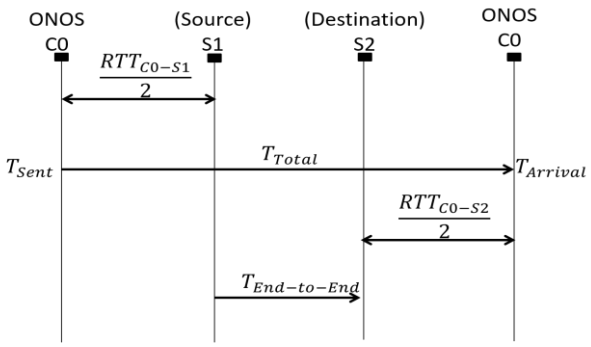

Figure 4.6 Timing Diagram of Delay Estimation Method

(i) Probe Packet Creation

In probe packet creation, each probe includes two parts (see in Figure 4.7): header and payload. The header field includes faked source/destination (src/dst) MAC addresses and Ethernet type value (0x8888). Here, the faked src/dst MAC addresses for probes are generated by creating unique identifier value. Instead of traditional packet encapsulation, the time stamp (probe packet sent time) value is encapsulated in payload field of probe. The network footprint of proposed method is quite low. Indeed, The probe's Ethernet frame uses 22 Byte packets to determine the link latency instead of ICMP's 64 Bytes. It thus uses 65% less bandwidth.

6 bytes	6 bytes	2 bytes	8 bytes
Source Mac	Destination MAC	Type	Payload

Figure 4.7 Probe Packet Frame

(ii) End-to-End Delay Estimation

After probe packet creation, as shown in Figure 4.8, the first probe (P1) with faked MAC address is sent from controller to source switch, through the path and back to the controller. The controller has to keep the faked source MAC address of this probe in order to match the probe when it is sent back. When the controller receives the probe back, the probe sent time is extracted from payload. The total time (T_{total}) can be computed by differentiating probe sent time (T_{sent}) from probe receive time ($T_{receive}$) as shown in Equation (4.2).

$$T_{total} = T_{receive} - T_{sent} \quad (4.2)$$

Since T_{total} contains the one-way-delay from controller to source node ($OWD_{C0\text{-}Source}$), time taken from source node to destination ($T_{end\text{-}to\text{-}end}$) node and destination node to controller ($OWD_{C0\text{-}Destination}$). Therefore, $T_{end\text{-}to\text{-}end}$ can be derived as follows:

$$T_{total} = OWD_{C0-Source} + T_{end-to-end} + OWD_{c0-Destination}$$
$$T_{end-to-end} = T_{total} - OWD_{C0-Source} - OWD_{C0-Destination} \quad (4.3)$$

Where the one-way-delays for $OWD_{C0\text{-}Source}$ and $OWD_{C0\text{-}Destination}$ is given as:

$$OWD_{C0-Source} = \frac{RTT_{C0-Source}}{2};\ OWD_{C0-Destination} = \frac{RTT_{C0-Destination}}{2} \quad (4.4)$$

Where ($RTT_{C0\text{-}Source}$) is the round-trip-time between controller and source switch. The next probe packet (P2) is sent out from controller to source switch. As there is no matched rule in source switch for this probe, the source switch send it back to controller. The $RTT_{C0\text{-}Source}$ can be retrieved from this probe P2. The other probe (P3) is also sent out to destination switch in similar way to measure the round-trip-time between controller and destination switch $RTT_{C0\text{-}Destination}$.

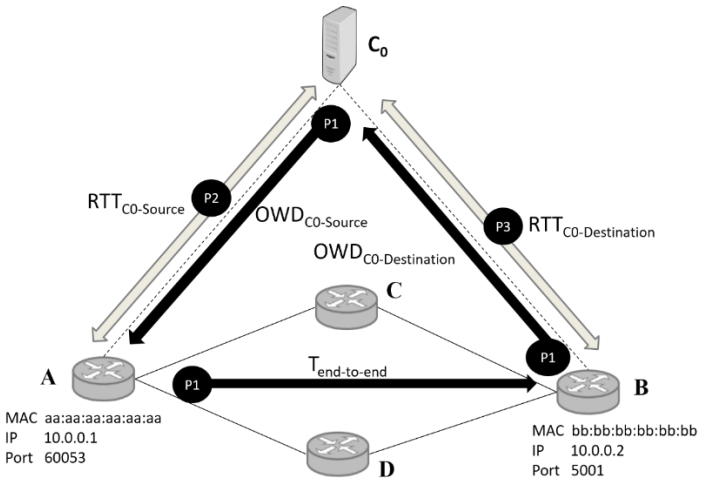

Figure 4.8 Delay Measurement Scenario

(iii) Probe Packet Processing

There are two types of packet types in probe packet processing.

(i) *Packet_Out*: Message sent from the controller to a switch containing a data packet to be forwarded through one or more ports.

(ii) *Packet_In*: Message sent from a switch to the controller when encountering an unknown packet (i.e there is no corresponding entry in the switch's flow table).

Firstly, the probe is created as mentioned in previous subsection and encapsulated in payload of OpenFlow Packet_Out message. This Packet_Out message is sent out to source switch of the path. When the packet arrive at the source switch, it's header fields (source MAC, destination MAC, Ethernet type) against source switch's flow table. If it matches, the probe's destination MAC address is changed to next hop MAC address and send out via exit interface. Here, these flow rules for probes are installed before probe is sent. If the probe is not matched with flow rules (when the probe reaches to destination switch), the probe is sent back to controller port via OpenFlow Packet_In message. The flowcharts (Figure 4.9) below explain how packets are processed at controller and at switch level.

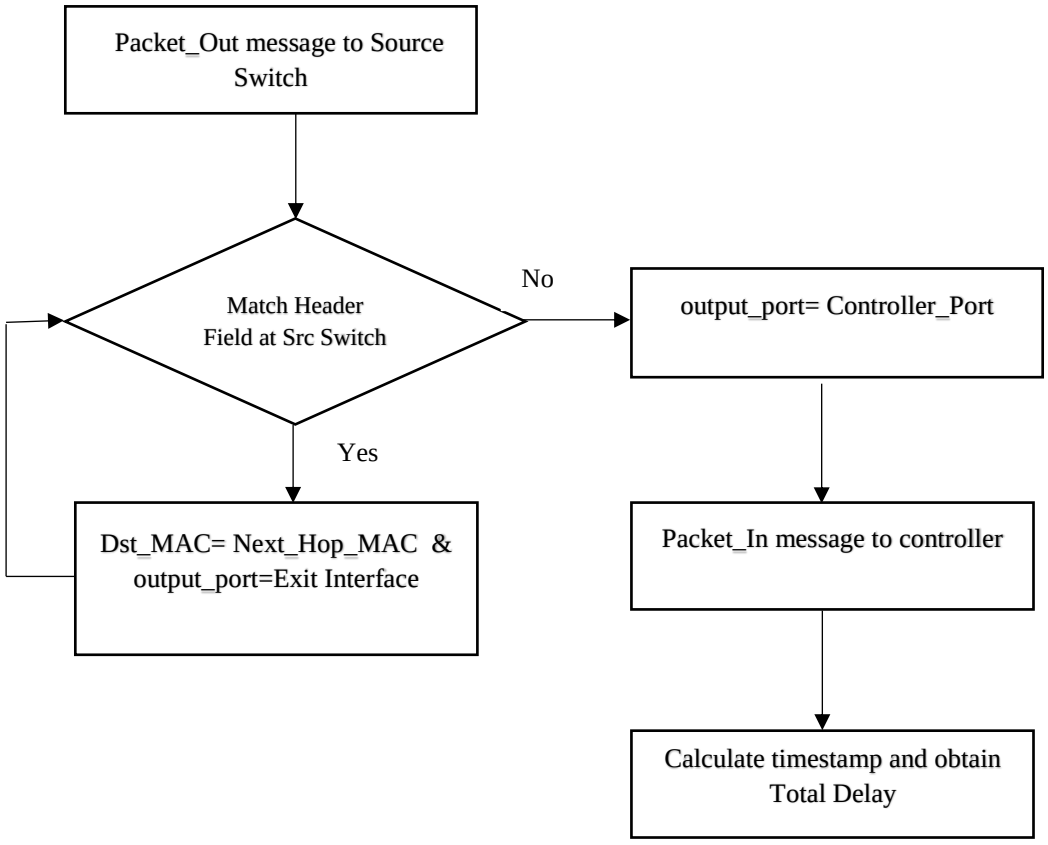

Figure 4.9 Flow Chart Explaining Process for Probe Packet

The Packet_In Event handler allows then to know the time of arrival of packet to the controller. Thus, the total delay is able to know. In this way, it took the probe message to go through all the intermediate switches along the path. By matching source MAC address, destination MAC address and Ethernet type, allow to calculate total delay for an end-to-end path. Similarity, other Total delay for other paths can be easily calculated just by modifying source and destination MAC address to math with their end to end path.

4.3.4. Estimating Link Utilization

The purpose of this module is to query, consolidate and store the statistics from all OpenFlow switches. These statistics are then used by the Least Cost Path Computation module as one of metric to compute the load on various links. The key idea behind the proposed DAEFR is to reroute the packets of elephant flow along several least cost paths based on two metrics: end-to-end delay and link load. Hence the load on the links along the possible paths needs to be determined. This module collects statistics from each OpenFlow switch by polling them at fixed intervals. The per-table, per-flow and per-port statistics are gathered from all the connected OpenFlow switches in network.

In DAEFR, traffic load statistics of each links for specific path is computed to estimate link load utilization. Total sent bytes (sent bytes) and total receive bytes (receive bytes) of specific port which is associated with the device and port for link represent the total transmission bytes (or) the current bandwidth usage. The calculation of current total bytes (λ) for source port and destination port of each link can be derived as follows:

$$\lambda = src\_port\_statistics.bytesReceived + src\_port\_statistics.bytesSent + dst\_port\_statistics.bytesReceived + dst\_port\_statistics.bytesSent \quad (4.5)$$

By applying Equation (4.5) repeatedly until there is no next link in path, the total bytes of specific path can be calculated. Assume the network topology is represented by a connected graph G = (V, E), where V is the set of nodes and E represents the set of directed edges. Q is given source destination pairs (s (q), d (q)) for q = 1, 2, ..., Q. Let Pq represent the set of all available paths between the source-destination pair q, with $|Pq|$ = Nq. Assume that for any path $i \in P_q$ has a number of M_i subpaths and each subpath $j \in M_i$ has a number of L_{ij} links. Denoting with λ_{kji} and μ_{kji}

as the link load and link capacity on link k of subpath j of path i, respectively, the Link Utilization (LU) is given by Equation (4.6).

$$LU_{kji} = \frac{\lambda_{kji}}{\mu_{kji}} \tag{4.6}$$

The link load is reported by polling the respective switch port using standard OF mechanisms.

For example, Figure 4.10 describes the actual OpenFlow table in edge switch of fat-tree topology. Control flows and data flows can be classified by "Action" field of flow rules. According to Figure 4.10, "actions=CONTROLLER:65535" represents control data and "actions=output:" represents data flows. The control data in flow table is LLDP (Link Layer Discovery Protocol, 0x88cc), BDDP (Broadcast Domain Discovery Protocol 0x8942), (Address Resolution Protocol) ARP and IP. When the port statistics of switch port "s3001-eth2" is computed, it collects "n_bytes" (number of bytes) which is associated with this port number. According Figure 4.10, total number of bytes on port "s3001-eth2" is 10384928 (10384708+220).

Figure 4.10 OpenFlow Table in Switch

4.3.5. Calculating Least Cost Path

In order to compute the best path, the proposed DAEFR method uses end-to-end path delay and link load utilization. To calculate least cost path, the previous work [30] chooses the least minimum delay and second least minimum delay among available shortest paths for only TCP traffic flow. The DAEFR can handle for both TCP and UDP traffic types. The cost c_k of each path $p_k \in P$ can be computed as follows in Equation (4.8):

$$T_k = (LU)_k + d_k \tag{4.7}$$

$$c_k = \frac{\lambda_k}{T_k} = \frac{\lambda_k}{(LU)_k + d_k} = \frac{\lambda_k}{(\frac{\lambda_k}{\mu_k} + d_k)} \tag{4.8}$$

Where T_k is the total transfer time, d_k is average delay of path p_k, $(LU)_k$ is the total link load utilization cost of $L_k \in p_k$.

Some numerical values are assigned to Equation 4.8 by changing parameters such as number of links, latency and link load as shown in Table 4.2, 4.3 and 4.4. These tables show that path cost values are exponentially increasing as the parameter values are increased. The cost represents the actual transmission rate for each path. To reroute the elephant flow to the uncongested path, firstly we need to know which path is currently being over utilized. According to path cost values, the over utilized path can be defined when the path cost value is greater the path capacity (cost > capacity). Therefore, the least cost path will be the best path for elephant flows.

Table 4.2 Changing Number of Links

Datapath	Number of Link	tx_bytes	rx_bytes	Link Capacity (bps)	Latency (ms)	Path Cost (kBps)
2	1	5484325	5484325	75000000	120	91404.5
3	2	5484325	5484325	75000000	120	182807.3
4	3	5484325	5484325	75000000	120	274208.2
5	4	5484325	5484325	75000000	120	365607.4
6	5	5484325	5484325	75000000	120	457004.8
7	6	5484325	5484325	75000000	120	548400.4
8	7	5484325	5484325	75000000	120	639794.3
9	8	5484325	5484325	75000000	120	731186.3
10	9	5484325	5484325	75000000	120	822576.6
11	10	5484325	5484325	75000000	120	913965.1
12	11	5484325	5484325	75000000	120	1005351.8
13	12	5484325	5484325	75000000	120	1096736.7
14	13	5484325	5484325	75000000	120	1188119.8
15	14	5484325	5484325	75000000	120	1279501.2
16	15	5484325	5484325	75000000	120	1370880.8
17	16	5484325	5484325	75000000	120	1462258.6
18	17	5484325	5484325	75000000	120	1553634.6
19	18	5484325	5484325	75000000	120	1645008.8
20	19	5484325	5484325	75000000	120	1736381.3
21	20	5484325	5484325	75000000	120	1827751.9
22	21	5484325	5484325	75000000	120	1919120.8
23	22	5484325	5484325	75000000	120	2010487.9
24	23	5484325	5484325	75000000	120	2101853.2
25	24	5484325	5484325	75000000	120	2193216.8
26	25	5484325	5484325	75000000	120	2284578.6

Table 4.3 Changing Number of Links and Latency

Datapath	Number of Link	tx_bytes	rx_bytes	Link Capacity (bps)	Latency (ms)	Path Cost (kBps)
2	1	5484325	5484325	75000000	105	104462.2
3	2	5484325	5484325	75000000	110	199427.9
4	3	5484325	5484325	75000000	115	286135.8
5	4	5484325	5484325	75000000	120	365618.1
6	5	5484325	5484325	75000000	125	438741.9
7	6	5484325	5484325	75000000	130	506240.8
8	7	5484325	5484325	75000000	135	568739.9
9	8	5484325	5484325	75000000	140	626774.8
10	9	5484325	5484325	75000000	145	680807.3
11	10	5484325	5484325	75000000	150	731237.6
12	11	5484325	5484325	75000000	155	778414.4
13	12	5484325	5484325	75000000	160	822642.7
14	13	5484325	5484325	75000000	165	864190.5
15	14	5484325	5484325	75000000	170	903294.4
16	15	5484325	5484325	75000000	175	940163.7
17	16	5484325	5484325	75000000	180	974984.8
18	17	5484325	5484325	75000000	185	1007923.6

Table 4.4 Changing Number of Link Load and Latency

Datapath	Number of Link	tx_bytes	rx_bytes	Link Capacity (b/s)	Latency (ms)	Path Cost (kBps)
5	4	5000	5000	75000000	120	333.3
5	4	70535	70535	75000000	125	4514.2
5	4	136070	136070	75000000	130	8373.5
5	4	201605	201605	75000000	135	11947.0
5	4	267140	267140	75000000	140	15265.1
5	4	332675	332675	75000000	145	18354.5
5	4	398210	398210	75000000	150	21237.9
5	4	463745	463745	75000000	155	23935.2
5	4	529280	529280	75000000	160	26464.0
5	4	594815	594815	75000000	165	28839.5
5	4	660350	660350	75000000	170	31075.3
5	4	725885	725885	75000000	175	33183.3
5	4	791420	791420	75000000	180	35174.2
5	4	856955	856955	75000000	185	37057.5
5	4	922490	922490	75000000	190	38841.6
5	4	988025	988025	75000000	195	40534.3
5	4	1053560	1053560	75000000	200	42142.4

4.3.6. Flow Rule Installation (or) Rerouting

After choosing the least cost path among available paths, the new flow entries are injected to respective devices through this path by using FlowRuleService which is provided from ONOS controller. The traffic selection fields of each flow entry are source MAC address, destination MAC address, protocol type and TCP/UDP ports. When the traffic flow rate does not exceed the threshold, the route decision and flow entries are made by using reactive forwarding method. As soon as the sFlow-RT analyzer detects elephant flow, the route decision and new flow entries are made by proposed flow management application (DAEFR). The old entries, which are injected from reactive forwarding, will be removed automatically after 10 seconds, which is defined in idle-timeout. In flow rule installation module, two main contributions are added in order to improve the performance of DAEFR algorithm:

(i) First, the flow rules placement in different tables, and

(ii) Second, avoiding unnecessary flow rule installation.

Firstly, the flow rules that are generated from DAEFR application are mainly categorized into two: flow rules from delay measurement function and flow rules from rerouting function. As mentioned in previous subsection 4.3.3, delay measuring method is based on probing. The flow rules for probes are needed to installed proactively to pass through the path. Figure 4.11 represents the scenario of flow rule placement (e.g. switch S1). In Figure 4.11, the red color box highlights the flow rules for probe packets and yellow color box highlights the flow rules for rerouting. If all of these flow rules are placed in one table (Table 0), it makes the unnecessary flow matching time for rerouting. Therefore, in the proposed approach, the rerouting flow rules are defined as main rules because the elephant flow rerouting timely is studied as an important fact for network performance in previous chapter. Therefore, the flow rules for rerouting are placed in "OpenFlow Table 0" and the flow rules for probes are placed in "OpenFlow Table 1".

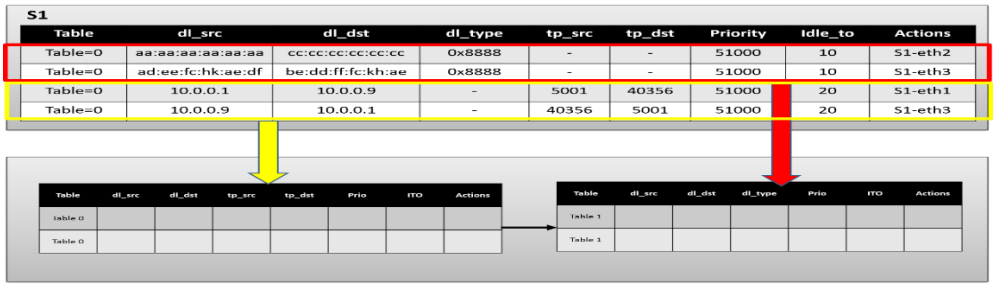

Figure 4.11 Scenario of Flow Rule Placement

Secondly, the proposed approach added the condition between finding least cost path and flow rule installation functions (see in Figure 4.12). Sometimes, the elephant flow may already be on an optimal path. For this event, the new flow rule installation is unnecessary and even makes an increase in packet loss rate due to flow rule modification. This condition is to check whether the flow existing path is equal to the least cost path. If it is equal, the flow rerouting action does not need to take because the flow is already taken on the best path. If not, the elephant flow is needed to reroute to the least cost path. In this way, the DAEFR can avoid some packet loss rate because of unnecessary flow rule modification.

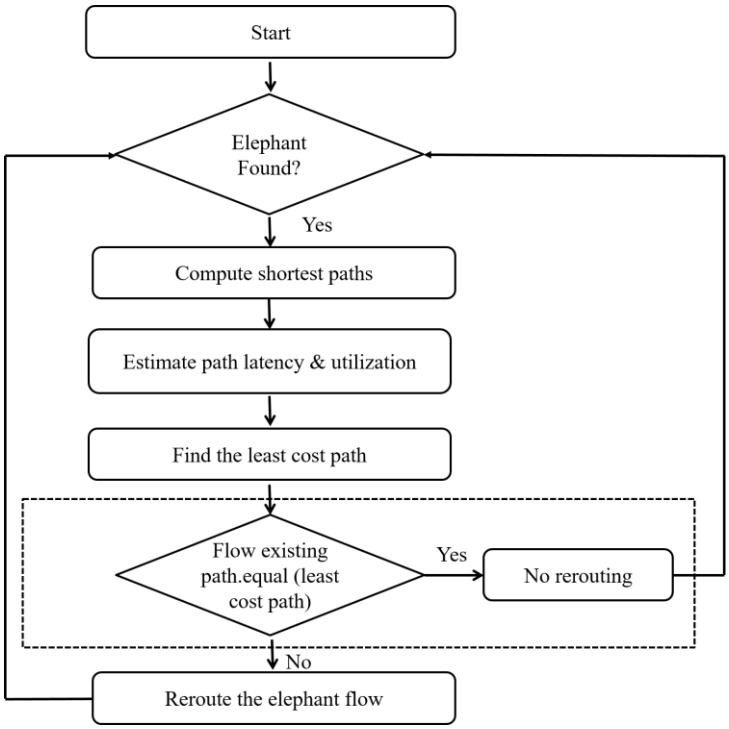

Figure 4.12 Avoiding Unnecessary Flow Rule Installation

4.4. Chapter Summary

In summary, this chapter describes the detailed components of proposed work and mentions the basic concepts of research work and problem statement. The next chapter will present the design implementation of testbed and experimental results evaluation.

CHAPTER 5
EXPERIMENTAL RESULTS EVALUATION

In this chapter, the experimental results are evaluated and compared with existing methods under various network and traffic conditions.

5.1 Performance Measurement and Expectations

The proposed DAEFR method is compared with three methods: (i) reactive forwarding [12], (ii) ECMP [35] and (iii) Hedera [5] in terms of performance. Following parameters are used for evaluating the performances of different schemes.

(i) Network Throughput

It is the rate of successful message delivery over a network. Throughput is measured in Bps (bytes per second) or bps (bits per second). If more data transferred, higher throughput result.

$$\text{Throughput} = \frac{\text{Amount of Data Transferred}}{\text{Time Taken to Transfer Data}} \tag{5.1}$$

(ii) Flow Completion Time (FCT)

Flow completion time (FCT) is the time difference between the time when the first packet of a flow leaves the source and the time when the last packet of the same flow arrives at the destination (in seconds). Ideally, FCT is as small as possible in order to get better performance. Excessive FCT creates bottlenecks that prevent data from filling the network pipe, thus decreasing effective bandwidth.

$$\text{Flow Completion Time} = \frac{\text{Amount of Data Transferred}}{\text{Throughput}} \tag{5.2}$$

(iii) Packet Loss

When accessing the internet or any network, small units of data called packets are sent and received. When one or more of these packets fails to reach its intended destination, this is called packet loss. Packet loss can be caused by network congestion and security threats. Packet loss can be calculated as following equation:

$$\text{Packet Loss} = \text{Number of Sent Packets} - \text{Number of Received Packets} \tag{5.3}$$

Then, packet loss ratio can be calculated using the following formula:

$$\text{Packet Loss Ratio} = \frac{\text{Number of Lost Packets}}{\text{Number of Received Packets}} \quad (5.4)$$

Results are expected to be better for DAEFR scheme because in this scheme flows have best path to reroute among multiple available paths in network. It believed to perform better than reactive forwarding (fwd), ECMP and Hedera because of better flow scheduling in network with the benefits of mice and elephant flows differentiation. Reactive forwarding (fwd) is expected to perform worst because it is a hop count based routing method and it uses single random path to transfer data.

5.2 Experimental Testbed Design

In order to assess the performance of the proposed method, the experimental testbed has to be designed. As the proposed system is implemented using ONOS controller, it runs tree topologies in order to compare and evaluate the results with or without the proposed method. In order to have deterministic and low-cost environments to test, a virtual testbed was created that can run on two machines and do not require additional effort to be maintained and operated. The selected network emulator is Mininet [62]. The Figure 5.1 illustrates the experimental testbed design running on two machines.

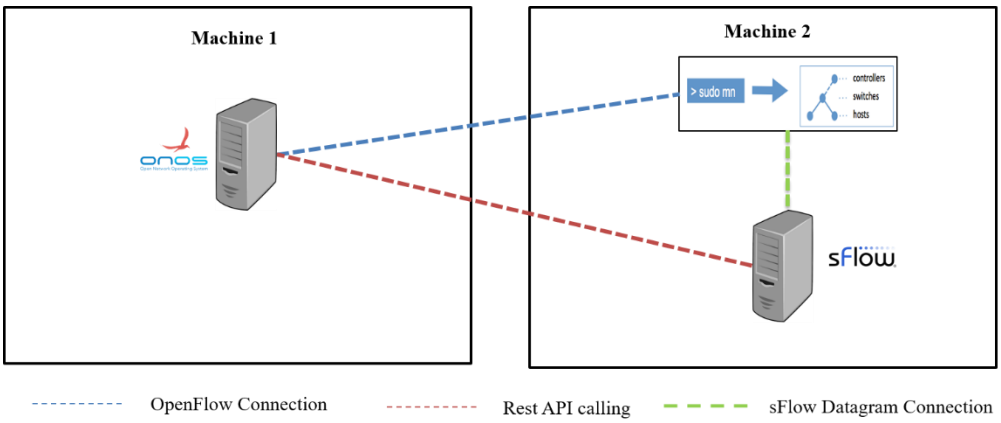

Figure 5.1 Experimental Testbed Design

In Figure 5.1, ONOS controller is running on Machine 1. The sFlow analyzer and Mininet topology are running on Machine 2. The machine 1 and machine 2 can be connected via wired or wireless. Mininet emulator is used to create virtual SDN network and sFlow analyzer is used to monitor and detect the elephant flow in the network. Therefore, sFlow datagram connection is to carry the sample packets between sFlow analyzer and the network. The elephant flow information from sFlow analyzer

can be retrieved by calling REST API from ONOS application. To discover the topology and install the flow rules to switches, there is OpenFlow connection between controller and Mininet. For the experiment testbed, Table 5.1 describes two machines specification and Table 5.2 describes the software versions, which are used in testbed.

Table 5.1 Description of Machines

Parameters	Description
CPU	Core TM i5-5200U CPU @2.20GHz
RAM	8 GB
HDD	1 TB
OS	Linux Ubuntu Desktop 14.04LTS

Table 5.2 Description of Software

Parameters	Description
ONOS	1.8 (Ibis)
sFlow-rt	2.0-r1121
Mininet	2.2.1
Open vSwitch	2.9.2
OpenFlow	1.3

5.3 Evaluation Scenario for Delay Estimation

Before the evaluation of the proposed DAEFR method, the proposed end-to-end delay estimation method is evaluated. The results are compared with Ping which is a tool to check the availability and responsiveness of network nodes. It uses Internet Control Message Protocol (ICMP). The result of Ping is working in source device and the proposed delay estimation method is working in controller only.

5.3.1 Scenario: Delay Variation between Controller and Switches

The proposed end-to-end delay estimation method is evaluated by using delay variation between controller and switches. The objective of this scenario is to know the effects of delay variation between controller and switches on delay estimation method. In this scenario, delay variation is created by different media: wired and wireless as shown in Figure 5.2. The wired media has low delay variation. The wireless media has high delay variation. The experimental testbed linear topology is as shown in Figure

5.3. For each variation, the parameter setting is used as in Table 5.3. The link delay level is ranging from 1 to 20 ms for each link. The average delay result will change according to number of links in path. All of link speed is 100 Mbps. The running duration is 480 seconds. The estimation interval is 0.5 seconds. The number of probes for each interval is 10.

Table 5.3 Parameter Setting for Delay Estimation

Parameter	Value
Link Delay Levels	0ms , 1ms, 10ms, 20 ms
Link Speed	100 Mbps
Duration	480 seconds
Estimation Interval	0.5 seconds
Number of Probes	10

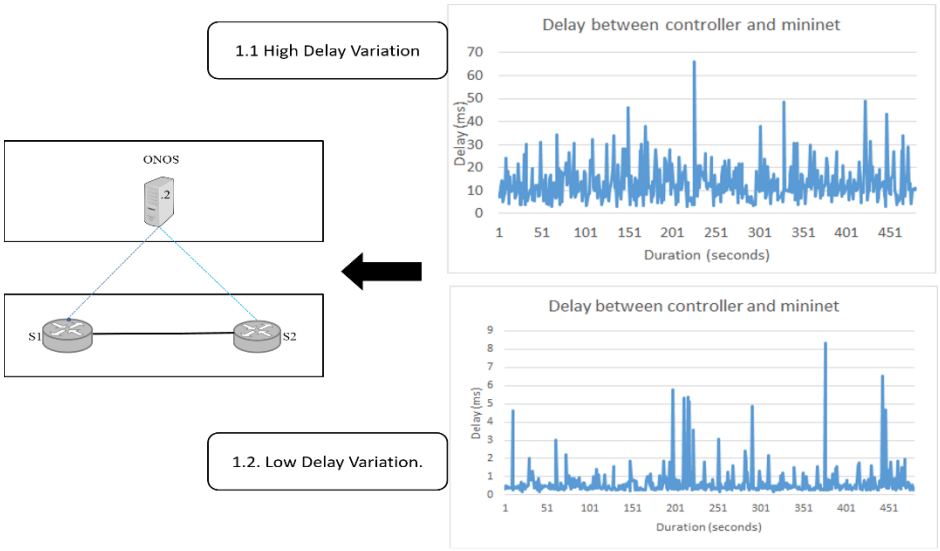

Figure 5.2 Different Delay Variation

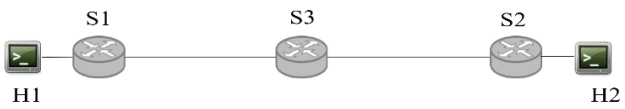

Figure 5.3 Linear Topology

According to Figure 5.4, the delay result graphs are described in terms of link delay 0 ms, 1 ms, 10 ms and 20 ms in high delay variation environment. The Figure

5.4(a) shows that the proposed method has the average delay 1 ms and maximum delay 7.8 ms while the Ping result has 0 ms. The Figure 5.4(b) describes that the proposed method has the average delay 3 ms and the maximum delay 5.8 ms while the Ping result has 2 ms. The Figure 5.4(c) shows that the proposed method has the average delay nearly 24 ms and the maximum delay 45 ms while the Ping result has 20 ms.

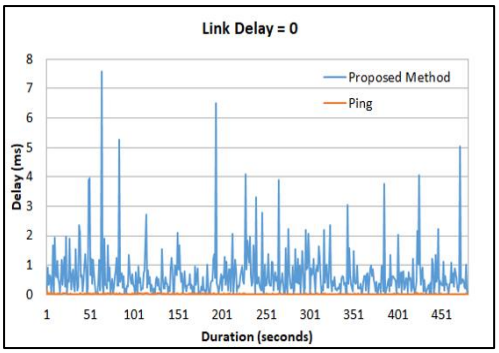

Figure 5.4(a) Average Path Delay Results when Link Delay = 0 ms

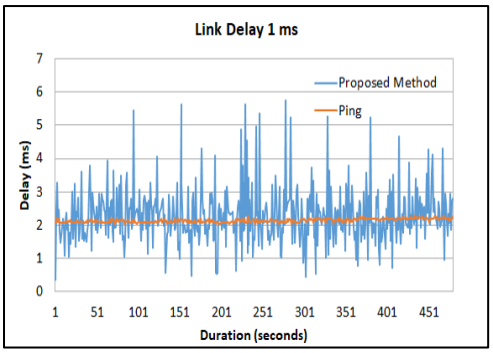

Figure 5.4(b) Average Path Delay Results when Link Delay =1ms

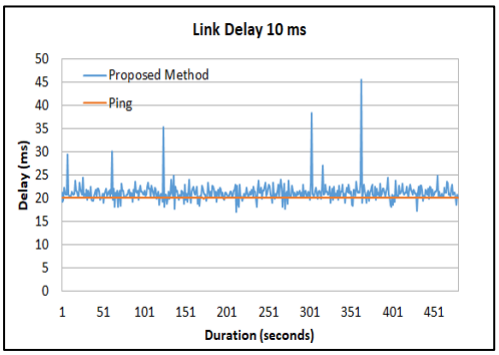

Figure 5.4(c) Average Path Delay Results when Link Delay =10ms

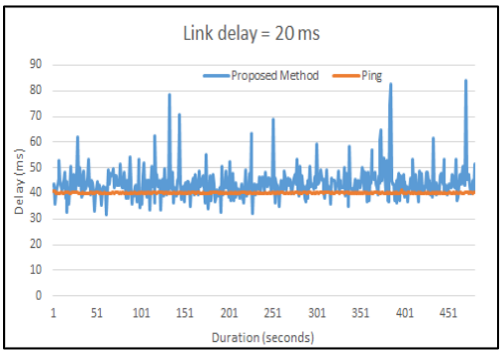

Figure 5.4(d) Average Path Delay Results when Link Delay = 20 ms

Figure 5.4 High Delay Variation between Controller and Switches

The Figure 5.4(d) describes that the proposed method has the average delay nearly 50 ms and the maximum delay 83 ms while the Ping result has 40 ms. Therefore the results of the proposed delay estimation method in high delay variation between controller and switches are nearly 20% less accuracy than Ping's results. The Figure 5.5(a) proves that the proposed method has the average delay nearly 0.6 ms and the maximum delay is 3ms. Figure 5.5(b) shows that the proposed method has the average

delay nearly 3.5 ms and the maximum delay 6 ms. Figure 5.5(c) shows that the proposed method has the average delay nearly 20.2 ms. Figure 5.5(d) shows that the proposed method has the average delay nearly 42 ms and the maximum delay 50 ms. According to Figure 5.5(d) the proposed delay estimation method in low delay variation between controller and switches are 4.7% less accuracy than Ping's results. Moreover, in Figure 5.5(d), the more link delay in network, the better accuracy results. Based on these delay variation results, the proposed method got better accuracy in low delay variation between controller and switches. High delay variation makes the less accuracy in estimation. Therefore, all of DAEFR's evaluations in next subsection is based on low delay variation environment.

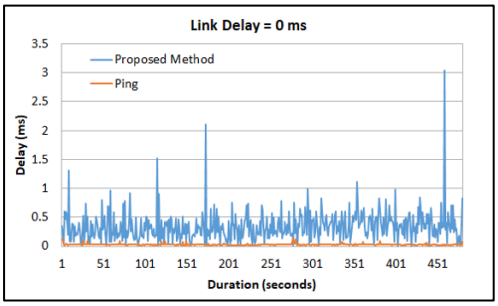

Figure 5.5(a) Average Path Delay Results when Link Delay = 0 ms

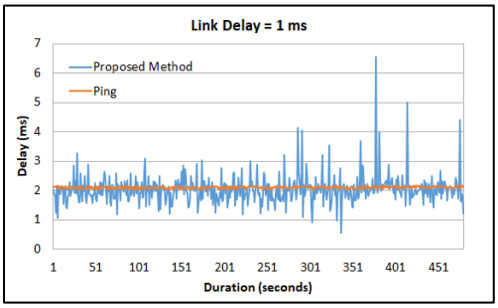

Figure 5.5(b) Average Path Delay Results when Link Delay = 1 ms

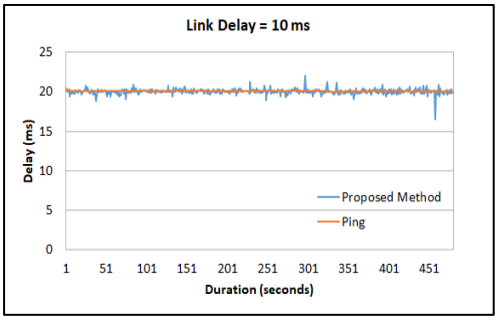

Figure 5.5(c) Average Path Delay Results when Link Delay = 10 ms

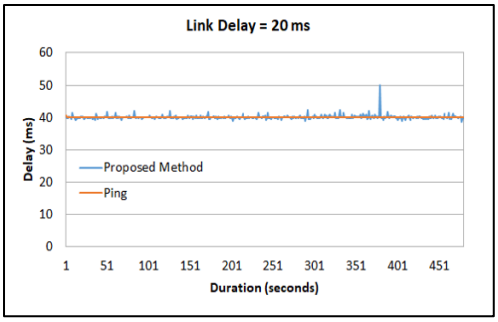

Figure 5.5(d) Average Path Delay Results when Link Delay = 20 ms

Figure 5.5 Low Delay Variation between Controller and Switches

5.4 Evaluation Scenarios for DAEFR

The emulated network for testing is created in Mininet using python API. The topology used for testing and requirements are discussed in previous chapter.

Measurements are done in both TCP and UDP traffic. Iperf [41], FTP and HTTP are used for generating traffic. Different scenarios and results discussion are described in next subsection. The results are calculated based on the average value of 4 running results.

5.4.1 Scenario 1: Changing Different Topology Parameters

For scenario 1, in Table 5.4, the test uses two different bandwidth parameters [200,100,300] Mbps and [100,50,200] Mbps. The objective of scenario 1 is to prove that DAEFR's performance is better than other compared methods when link bandwidth includes >100 Mbps and < 100 Mbps. The link delay is ranging from 20ms to 130ms and delay difference among paths is 30ms to 80ms. In Table 5.5, TCP and UDP random traffic using iPerf are generated. We generate 1 to 12 TCP and UDP elephant flows and each flow size is 1.4 GB. It is the traffic generator used to simulate the flows from client. This tool is used to create requests to the servers; it also used to capture performance parameters: network throughput and FCT using TCP and UDP traffics.

Followings are the commands used to generate traffic:

To start the server in tcp,

$ *iperf -s -i 10*

To start elephant flow in tcp:

$ *iperf -c 10.0.0.1 –n 1.4G -i 10*

To start the server in udp,

$ *iperf -s -u -i 10*

To start elephant flow in udp:

$ *iperf -c 10.0.0.1 -u -i 10 -b 100M -t 100*

Table 5.4 Topology Setting for Scenario 1

Parameter	Value
Link Speed (Mbps)	[200,100,300] and [100,50,200]
Latency	10~130 ms
Latency Diff	30~ 80 ms

Table 5.5 Testing Parameters for Scenario 1

Parameter	Value
Traffic Generator	iPerf
Elephant Flow	1.4 GB
Traffic Type	TCP, UDP
Traffic Pattern	Random

According to Figure 5.6 and Figure 5.8, the proposed DAEFR improves throughput results for TCP, 41% and 44% than reactive forwarding, 25% and 19% than ECMP, and 17% and 26% than Hedera. For high bandwidth environment, DAEFR has a better throughput improvement for ECMP than Hedera. For low bandwidth environment, the throughput improvement of DAEFR outperform Hedera rather than ECMP. However, DAEFR has the same throughput improvement for reactive forwarding in both environments.

Figure 5.7 and Figure 5.9 proves that DAEFR reduces FCT results in 72% and 79% than reactive forwarding, 25% and 26% than ECMP and 24% and 25% than Hedera. For flow completion time, DAEFR's FCT reduction is greater in low bandwidth environment rather than in high bandwidth because maximum throughput in high bandwidth testing is 95 Mbps and in low bandwidth is 48 Mbps.

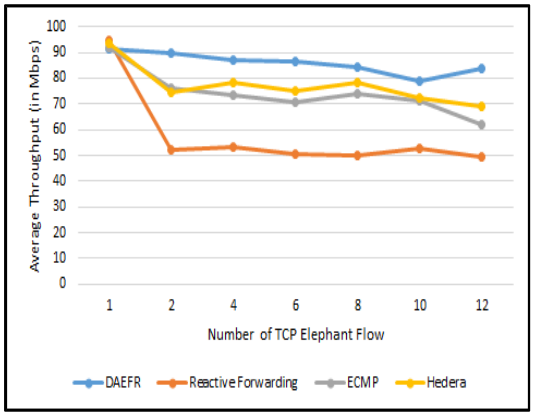

Figure 5.6 [200,100,300] Mbps, Average TCP Throughput Results

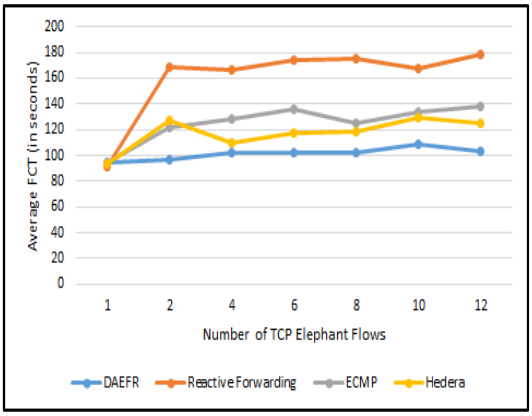

Figure 5.7 [200,100,300] Mbps, Average FCT Results for TCP

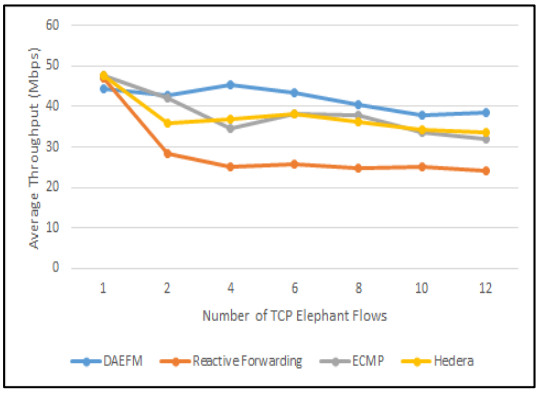

Figure 5.8 [100,50,200] Mbps, Average TCP Throughput Results

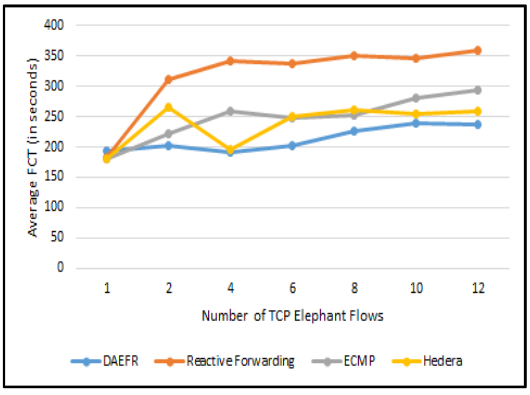

Figure 5.9 [100,50,200] Mbps, Average FCT Results for TCP

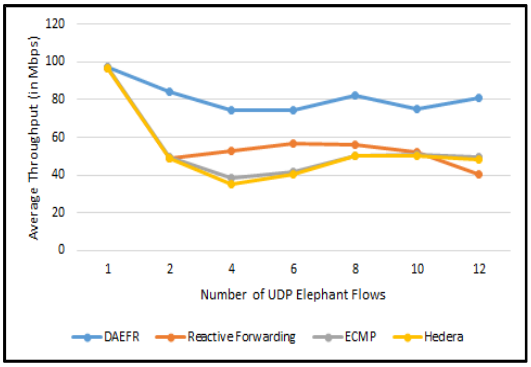

Figure 5.10 [200,100,300] Mbps, Average Throughput for UDP

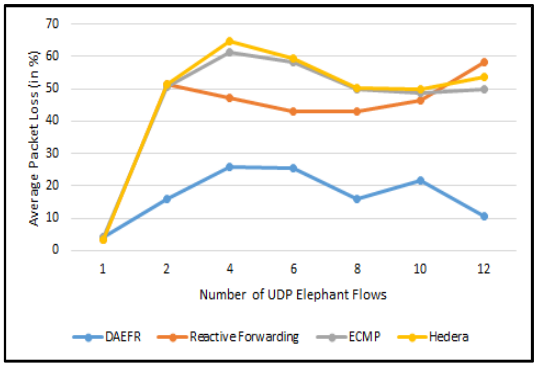

Figure 5.11 [200,100,300] Mbps, Packet Loss Ratio for UDP

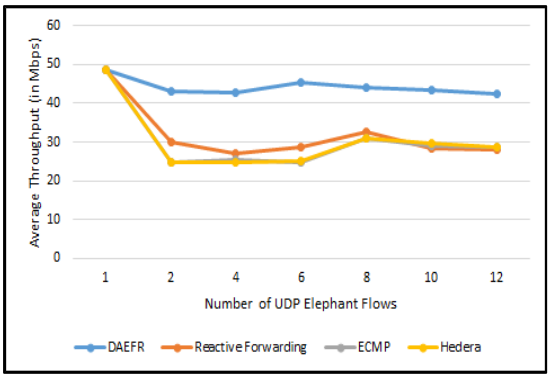

Figure 5.12 [100, 50, 200] Mbps, Average UDP Throughput Results

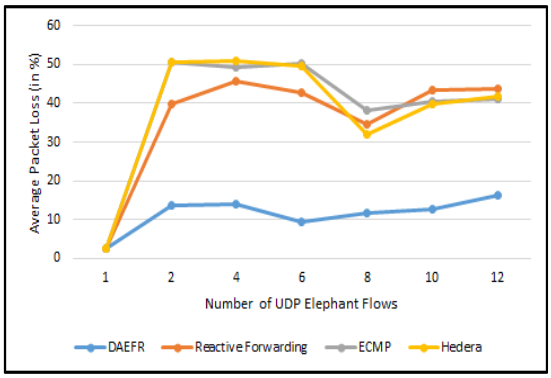

Figure 5.13 [100, 50, 200] Mbps, Packet Loss Ratio for UDP

Figure 5.10 and Figure 5.12 describes that DAEFR improves UDP throughput results, 49% and 36% than reactive forwarding, 48% and 45% than ECMP and 52% and 44% than Hedera. Therefore, DAEFR has been studied that it outperforms UDP throughput result in high bandwidth environment.

According to Figure 5.11, high bandwidth environment, DAEFR's packet loss ratio is 16, reactive forwarding is 51, ECMP is 50.7 and Hedera is 51. According to Figure 5.13, low bandwidth environment, DAEFR's packet loss ratio is 9, reactive forwarding is 52, ECMP is 50.7 and Hedera is 49. According these packet loss results, DAEFR's packet loss ratio is nearly five times less than other three methods.

5.4.2 Scenario 2: Changing Different Traffic Generating Tools

In scenario 2, Table 5.6, DAEFR's throughput results are evaluated using different tools (iPerf, HTTP and FTP). The goal of this scenario is to verify that DAEFR's performance is not depend on only iPerf tool. Each elephant flow size has 1.4 GB and generates 4 flows. The core link speed is 200 Mbps, aggregation link speed is 100 Mbps and edge link speed is 300 Mbps. The latencies of available paths between any source and destination is ranging from 10 to 130 ms and latency difference between path is 30 to 80 ms.

Table 5.6 Parameter Setting for Scenario 2

Parameter	Value
Traffic Generator	iPerf, HTTP, FTP
File Size	1.4 G
Number of Elephant Flows	4
Link Speed	200,100, 300 Mbps
Latency	10~130 ms
Latency Diff	30~ 80 ms

HTTP stands for Hyper Text Transfer Protocol. It is an application layer protocol that allows web-based applications to communicate and exchange data. It is a TCP/IP based protocol. It is used to deliver contents like - images, videos, audios, Documents etc. and if the two computers wants to communicate and exchange data then the client and server usually inform of a request response cycle they both must speak the HTTP communication Protocol. The client is usually a computer that makes request and the server is the one that serves by responding to the request. Following commands are the HTTP server and clients.

To run HTTP server in host H1 (10.0.0.1) on port 80,

$ *python –m SimpleHTTPServer 80*

To run HTTP client in host in order to get video file (zootopia.mp4) from H1 server,

$ *wget http://10.0.0.1:80/zootopia.mp4*

FTP stands for File Transfer Protocol. It is also a TCP/IP based protocol. It's also one of the oldest protocols in use today, and is a convenient way to move files

around. An FTP server offers access to a directory, with sub-directories. Users connect to these servers with an FTP client, a piece of software that lets you download files from the server, as well as upload files to it. Following commands are the FTP server and clients.

To run FTP server in host H1 (10.0.0.1),

$ *inetd &*

To run FTP client in host in order to get image file (eg. ubuntu.iso) from H1 server,

$ *ftp 10.0.0.1*

For scenario 2, Figure 5.14 shows that 38%, 12% and 10% throughput improvement as compared with reactive forwarding, ECMP and Hedera for iPerf traffic. For HTTP traffic, 37%, 20% and 22% throughput improvement and 26%, 8% and 10% throughput improvement as compared with reactive forwarding, ECMP and Hedera for FTP traffic.

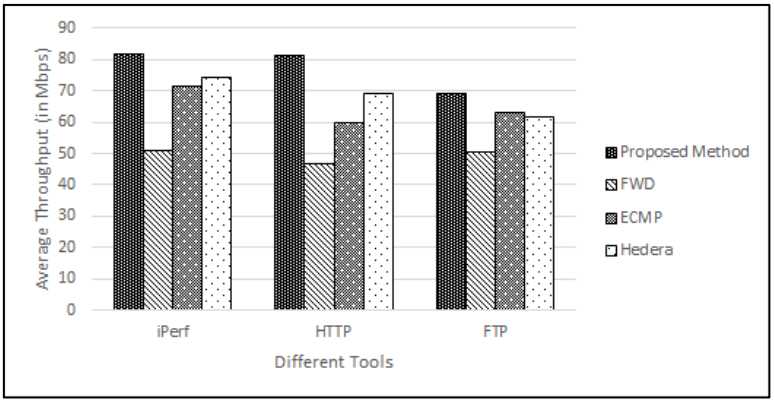

Figure 5.14 Average Throughput Results using Different Tools

According to this graph, although DAEFR improves throughput than other methods, it has better throughput results for iPerf and HTTP traffic. According to throughput graphs, all of four methods decrease throughput while increasing number of elephant flows. However, the proposed DAEFR method still improves than other three methods. This is because DAEFR detects the elephant flow using sampling, DAEFR reroutes the elephant flow based on end-to-end latency and link utilization while reactive forwarding and ECMP do not focus on network condition and Hedera only depends on link utilization.

5.4.3 Scenario 3: Changing Different Amount of Data Transfer

In scenario 3, according to Table 5.7, the throughput results of DAEFR are evaluated using two different amount of data transfer: 1.4 GB and 860 MB. The objective of scenario 3 is to test whether the proposed DAEFR still outperform as compared with other methods while transferring different amount of data. The evaluated traffic type is FTP and HTTP. The configured link bandwidth of core is 200 Mbps, aggregation is 100 Mbps and edge is 300 Mbps. The latency of available paths between any source and destination pair is from 10 ms to 130 ms. The latency difference between each path will be 30 to 80 ms.

Table 5.7 Parameter Setting for Scenario 3

Parameter	Value
Traffic Generator	FTP, HTTP
File Size	1.4 GB and 860 MB
Link Speed	200, 100, 300 Mbps
Latency	10~130 ms
Latency Diff	30~ 80 ms

Figure 5.15 shows the average throughput results per elephant flow for FTP traffic. According to this graph, for DAEFR, the result of transferred amount of data (860MB) is better 11% than amount of data (1.4GB). For reactive forwarding method, the result of transferred amount of data (860MB) is better 8.8% than amount of data (1.4GB). For ECMP, the result of transferred amount of data (860MB) is better 3.6% than amount of data (1.4GB). For Hedera, the result of transferred amount of data (860MB) is better 14.7% than amount of data (1.4GB). Figure 5.16 describes the throughput results for HTTP traffic. In HTTP graph, DAEFR's transferring data (860MB) is higher 5% throughput than transferring 1.4GB. Reactive forwarding's data transferred 860MB is higher 7.4% throughput than transferring 1.4GB. ECMP's data transferred 860MB is higher 17.6% throughput than transferring 1.4GB. Hedera's data transferred 860MB is higher 11.3% throughput than transferring 1.4GB. Both FTP and HTTP results of transferring 860MB is better than the results of transferring 1GB. According to these results of FTP, Hedera has the most significant result among other methods while transferring different amount. For HTTP, ECMP has the most significant result among other methods. These outcomes verify that DAEFR still

improves throughput results although the throughput of all methods decrease while transferring large amount data.

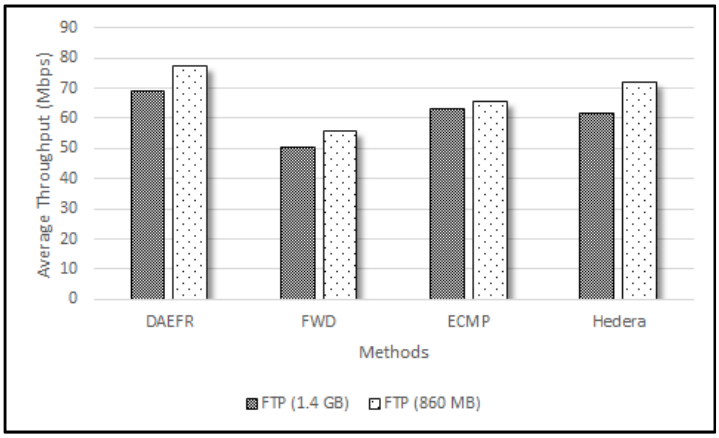

Figure 5.15 Throughput Results for FTP Traffic using Different Amount of Data Transfer

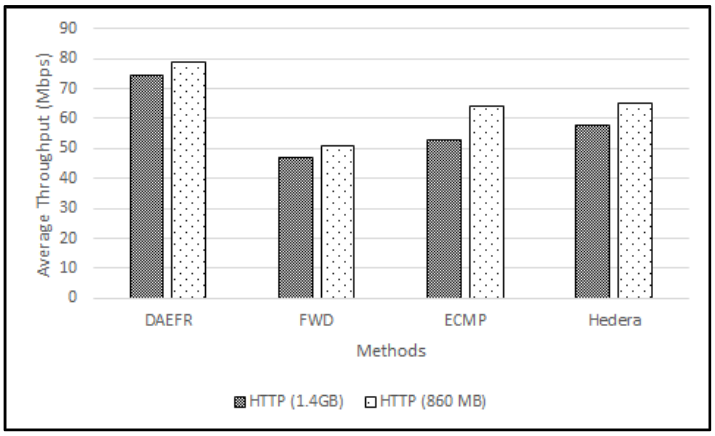

Figure 5.16 Throughput Results for HTTP Traffic using Different Amount of Data Transfer

5.4.4 Scenario 4: Testing in Low Latency Environment

In Table 5.8, the amount of latency and latency difference between available paths is changed. These latency values are less than previous scenarios. The objectives of this scenario is to prove that DAEFR's performance is still higher than other methods not only high latency environment but also low latency environment. In Table 5.8, this evaluation uses FTP and HTTP traffic. Each file size is 860 MB. Link bandwidth of core is 200 Mbps, aggregation is 100 Mbps and edge is 300 Mbps. The latency of each path will be between 10 ms and 60 ms. The latency difference is ranging from 10 to 50 ms. Figure 5.17 proves that DAEFR improves the throughput 39.4% than reactive forwarding, 32.6% than ECMP and 22.4% than Hedera for HTTP traffic. For FTP

traffic, DAEFR improves the throughput 32% than reactive forwarding, 17% than ECMP and 16% than Hedera. Therefore, these results verify that DAEFR method outperform as compared with other methods in both environments.

Table 5.8 Parameter Setting for Scenario 4

Parameter	**Value**
Traffic Generator	FTP, HTTP
File Size	860 MB
Link Speed	200, 100, 300 Mbps
Latency	10~60 ms
Latency Diff	10~50 ms

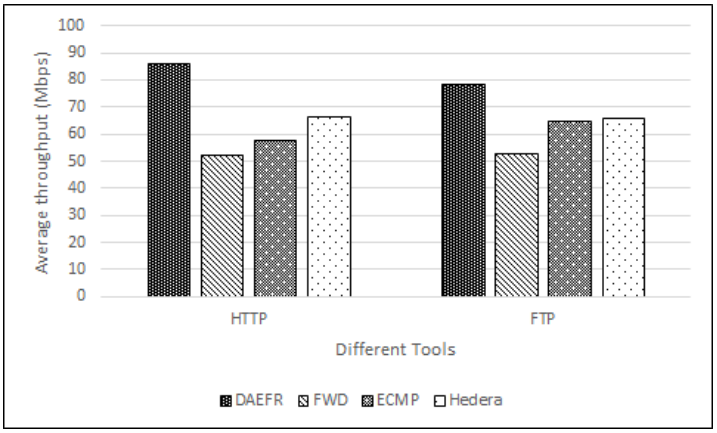

Figure 5.17 Throughput Results in Low Latency Environment

5.4.5 Scenario 5: Packet Loss Measurement using VLC

The objective of this scenario is to test the packet loss effect due to rerouting using video streaming. This scenario is running in simple topology as shown in Figure 5.18. Simple topology includes six open vswitches (S1~S6) and four hosts (H1~H4). There exists four available paths between source and destination.

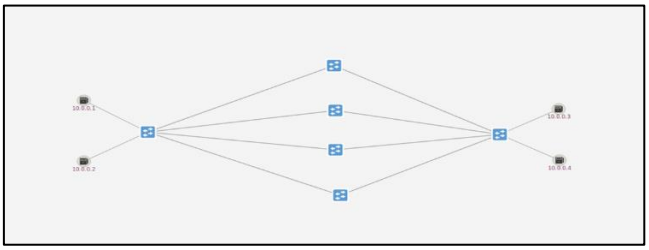

Figure 5.18 Simple Topology

Table 5.9 Parameter Setting for Scenario 5

Parameter	Value
Link Speed	10 Mbps
Latency	10~45 ms
Latency Diff	5~35 ms

As described in Table 5.9, each link speed is 10 Mbps and latency is ranging from 10 ms to 45 ms. Latency difference between source and destination is 5 ms to 35 ms. H1(10.0.0.1)~H2(10.0.0.2) serve as VLC servers and H3(10.0.0.3)~H4(10.0.0.4) serve as VLC clients. VLC servers are streaming MP4 (.mp4) video file type to clients using HTTP protocol nearly at the same time. The video file size is 85.6 MB and duration is 9 minutes and 56 seconds as shown in Table 5.10 and 5.11.

Table 5.10 File Information

Parameter	Value
File Type	MP4 file (.mp4)
File Size	85.6 MB
Video Length	00:09:56
Resolution	1280x720

Table 5.11 VLC Setting

Parameter	Value
Video Codec	H264
Bit Rate	128 kbps
Protocol Type	HTTP

HTTP uses TCP connection. Each byte of data sent in a TCP connection has an associated sequence number. This is indicated on the sequence number field of the TCP header. When the receiving socket detects an incoming segment of data, it uses the acknowledgement number in the TCP header to indicate receipt. After sending a packet of data, the sender will start a retransmission timer of variable length. If it does not receive an acknowledgment before the timer expires, the sender will assume the segment has been lost and will retransmit it. The TCP retransmission mechanism ensures that data is reliably sent from end to end. If retransmissions are detected in a TCP connection, it is logical to assume that packet loss has occurred on the network somewhere between client and server.

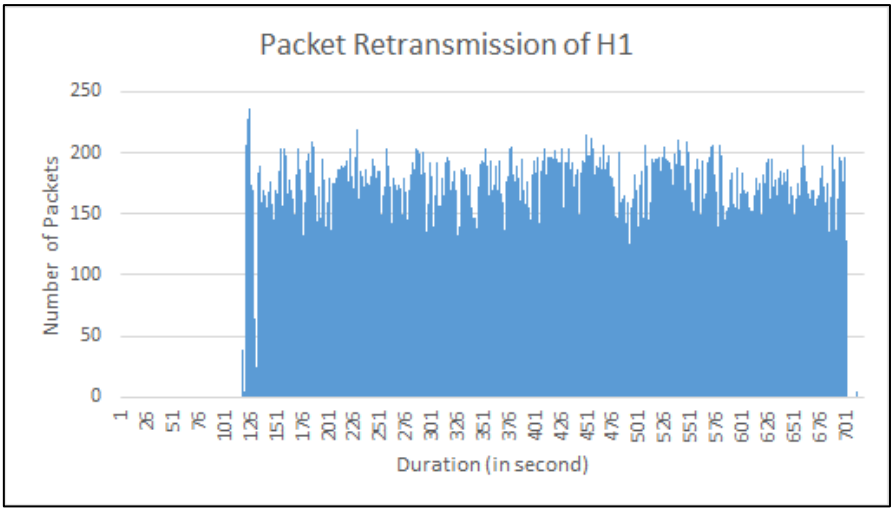

Figure 5.19 Packet Retransmission of HTTP Server (H1)

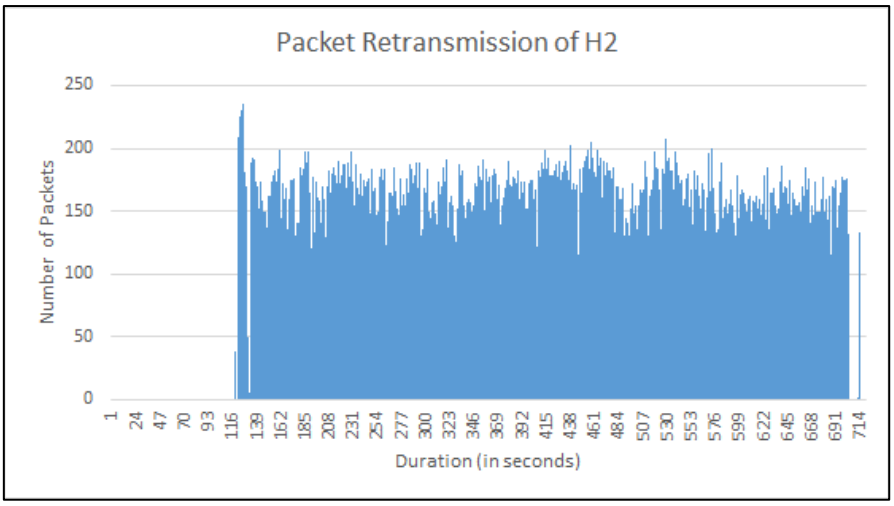

Figure 5.20 Packet Retransmission of HTTP Server (H2)

The packet retransmission is captured by Wireshark [89]. Figure 5.19 and Figure 5.20 describe the amount of retransmitted packets while running the proposed DAEFR method. In both graphs, the maximum number of retransmitted packets is under 250 and average result is under 200. According to Equation 6.4, the maximum packet loss ratio will be 0.004. Therefore, DAEFR's rerouting affects very few packet losses (nearly zero) for HTTP video streaming.

5.4.6 Scenario 6: Running in Leaf-Spine Topology

The objective of this scenario is to prove that DAEFR can adapt in different tree topology such as leaf-spine topology. Leaf-spine topology is based on two-tier

topology, which is mostly used in data center infrastructure. The leaf layer includes switches to provide connectivity of end devices. The spine layer provides connectivity of leaf switches. In leaf-spine testbed topology in Figure 5.21, it uses 8 open vswitches and 8 end hosts. Bandwidth of up links in this network topology are set 100 Mbps and down link is 200 Mbps. The latency of each path is ranging from 20.6 ms to 160.4 ms. The latency difference between available paths is from 20 ms to 140 ms as shown in Table 5.12.

Table 5.12 Parameter Setting for Scenario 6

Parameter	Value
Link Speed	Up Link: 100 Mbps, Down Link: 200 Mbps
Latency	20.6~160.4 ms
Latency Diff	20~140 ms

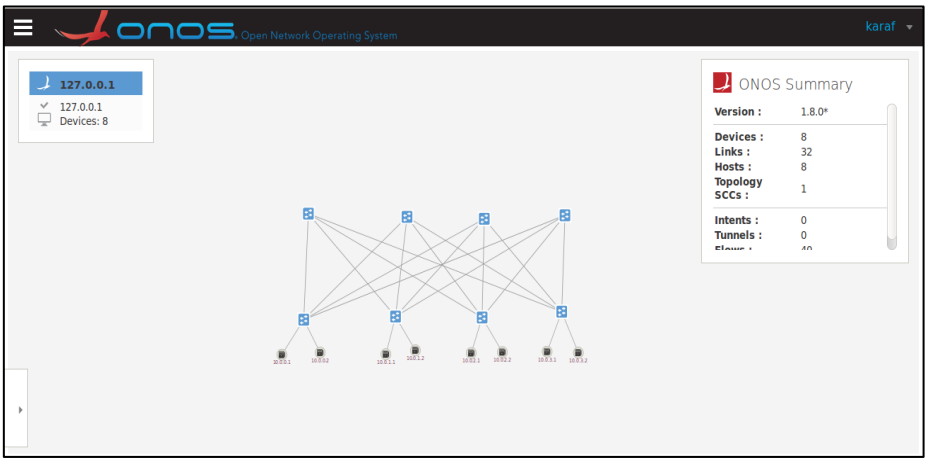

Figure 5.21 Leaf and Spine Topology

The previous scenarios from 1 to 4 is evaluated using k=4, fat-tree topology. In scenario 6, two to eight TCP elephant flows are generated using iPerf. Each elephant flow has 1.4 GB. Generating eight TCP elephant flow is the worst case. Because every end host serves as server and client. The DAEFR's results are compared with reactive forwarding method since the compared method, Hedera, can only adapt in fat-tree topology. Figure 5.22 describes the average throughput result in leaf-spine topology. The proposed DAEFR improves throughput 54% as compared with reactive forwarding. In Figure 5.23, DAEFR reduces FCT 55.4% rather than reactive forwarding method. For both throughput and FCT graphs, although reactive forwarding degrades the results gradually as increasing in number of flows, the proposed DAEFR

still improves. According to this verification, the proposed DAEFR can prove that it can optimize the network performance not only in fat-tree topology but also in leaf-spine topology.

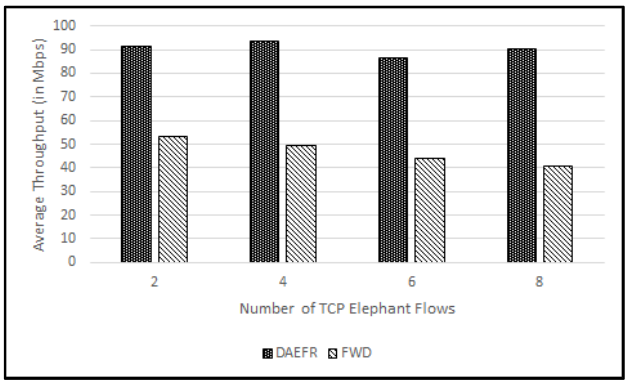

Figure 5.22 Average Throughput Result in Leaf-Spine Topology

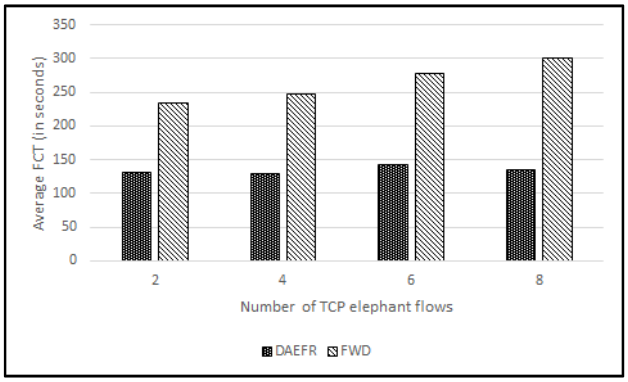

Figure 5.23 Average FCT Result in Leaf-Spine Topology

5.4.7 Scenario 7: Testing Single Server and Multiple Clients

The main objective of this scenario is to test the proposed DAEFR in congested situation. The testbed scenario is fat-tree topology as shown in Figure 5.24. Link bandwidth between edge switches and hosts is 200 Mbps, between aggregation and edge switches is 50 Mbps, and between core and aggregation switches is 100 Mbps, respectively. Each elephant flow size is 1 GB. H1 runs as iPerf Server and H5-H12 are iPerf clients. The number of TCP elephant flows from clients to single server are two to eight flows.

According to Figure 5.25, the throughput results of all methods are decreasing as increasing in number of TCP elephant flows. More TCP elephant flows to single server causes more competitive resource utilization. However, DAEFR still improves 42% throughput results than FWD, 30% than ECMP and 15% than Hedera.

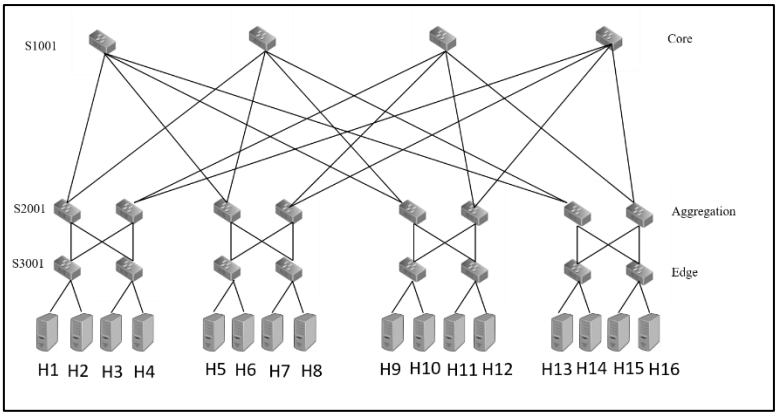

Figure 5.24 Testbed Topology

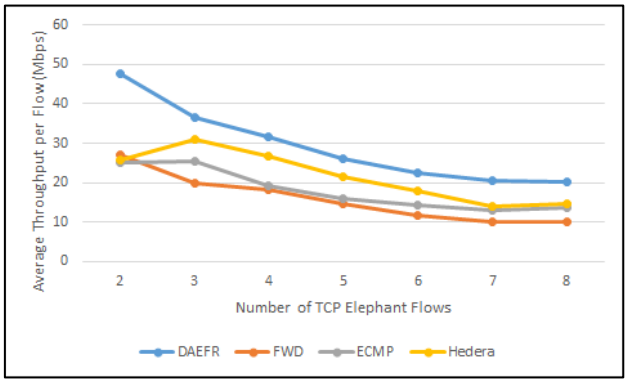

Figure 5.25 Average Throughput Results in Single Server and Multiple Clients

5.5 Chapter Summary

Since an effective traffic scheduling method for elephant flows is critical to efficiently utilize multiple available paths for large amount of data transfer such as VM migration and data backup, Delay-Aware Elephant Flow Rerouting method has been proposed, DAEFR, which aims to improve network throughput and to minimize the flow completion time among different end-to-end delays by using locally available information such as link load ratio and latency. By doing so, the risk of packet reordering is minimized, without incurring additional network overhead. Rerouting the elephant flow to least cost path can maintain not only a small end-to-end delay but also the packet reordering recovery time. In order to verify the performance of DAEFR, the comparative performance among DAEFR and the current existing flow scheduling methods are verified by analysis and simulations under various scenarios.

CHAPTER 6
CONCLUSION AND FUTURE WORK

In this dissertation, the work done which tackled the congestion and performance problems in tree network topologies. In Chapter 2, the limitations of current network infrastructure and the benefits of Software-Defined-Network (SDN) are highlighted. The details of literature review of current existing methods of traffic engineering are mentioned in Chapter 2. The current methods' contributions and their limitations are detailed. According to the literature review, most of the research work focuses on link load metric to handle the congestion problem due to elephant flows. There are very few proposed works, which consider the end-to-end delay metric to tackle congestion. Then, the Software-Defined Network (SDN) infrastructure and the basic theory of the proposed method are described in Chapter 3. To address the congestion problem due to the effect of elephant flow collision in network, the proposed Delay-Aware Elephant Flow Rerouting (DAEFR) architecture and its modules explanation are presented in Chapter 4. Moreover, the limitations and overcoming of previous studies to improve DAEFR are also discussed in Chapter 4. Finally, the proposed DAEFR method has been evaluated using many scenarios in Chapter 5. The evaluation scenario for end-to-end delay estimation module also includes in Chapter 5. The results of evaluations are measured in terms of throughput, flow-completion-time (FCT) and packet loss. The results are compared with reactive forwarding and Equal-Cost-Muli-Path (ECMP). The results of DAEFR are also compared with the current popular flow scheduling method, Hedera. Finally, the DAEFR proves that the link delay and link load consideration in flow rerouting improve the performance rather than only link load consideration.

6.1. Recommendations for Future Work

In the DAEFR architecture, the combination idea of the packet sampling method, which is to differentiate the elephant flow, and the elephant flow rerouting method, which is based on least cost path, is proposed. The least cost or best path is calculated based on end-to-end delay estimation and current link load. The end-to-end delay estimation of each path uses probe packets. The probes are Ethernet packets and the packet sent time is encapsulated in payload field. The delay estimation module is triggered only as soon as the elephant flow detected in network. Therefore, there is

very low overhead for probes compared with Ping because probe packet size is 24 bytes and ping packet is 196 byte. Therefore, it uses less bandwidth than Ping. Moreover, DAEFR does not need any hardware modification for rerouting elephant flows.

Since an effective multipath forwarding method is important to efficiently utilize multiple available paths for large amount of data transmission and real-time applications which are sensitive to QoS performance, a Delay-Aware Elephant Flow Rerouting approach has been proposed, DAEFR, which aims to minimize the suffering from effect of end-to-end path delays and to maximize the network throughput by using locally available information such as end-to-end path delay, ongoing flow size and link load utilization. By doing so, the network congestion problem can be reduced and thus the risk of performance degradation problem is minimized, without incurring additional network overhead. Since the proposed multipath forwarding method is based on flow-based forwarding, the risk of packet reordering is likely small and, no extra time is required for the packet reordering recovery process.

6.2. Summary of Contributions

The flow rerouting method proposed in this dissertation uses the end-to-end delay estimation method. The delay estimation module has been triggered only when the elephant flow is detected. This way can reduce the probes overhead because the probes are not sent every time. However, the processing time of elephant flow rerouting will be increased because least cost path calculation needs to wait the probes to get the current end-to-end delay and link load. In order to reduce the processing time for flow rerouting, the delay measurement module is needed to activate every time. Therefore, the trade-off between the overhead of probe packets and delay measurement module separation needs to investigate. As described in Table 7.1, DAEFR works well in low delay variation between controller and OVS switches. Therefore, it should also be investigated to work well in both low and high delay variation environment. Moreover, DAEFR is implemented and evaluated in layer 2 network. It should also work in layer 3 network to be more effective rerouting method. If DAEFR will be implemented in layer 3 network, the delay estimation method should be extended because the probes are based on layer 2 packets. However, due to the requirement of Mininet emulator for modeling SDN

infrastructure, the proposed work is encouraged to research and evaluate on the realistic SDN testbed. The proposed work should be extended to work in non-tree network topology although it can adapt in any level of tree network topology.

Table 6.1 DAEFR's Recommendations

Recommended	Not Recommended
-When there is low delay variation between control plane and data plane	-When there is high delay variation between control plane and data plane
-When the link delay is greater than the delay between control plane and data plane	-When the link delay is smaller than the delay between control plane and data plane
-only work in layer 2	-does not work in layer 3
-Tree topology	-Non-tree topology

6.3. Conclusion

In this chapter, the proposed solution is summarized that enhance the network performance and congestion by the use of SDN infrastructure. SDN can be used to be an intelligent traffic engineering solution with the centralization of global view and dynamic flow rule installation nature. The proposed DAEFR, dynamic flow rerouting scheme for fat-tree network, is presented which differentiates elephant flows and re-scheduling to least cost path for both TCP and UDP traffic. Therefore, DAEFR can be applied for transferring huge files through HTTP or FTP. Moreover, it can be applied for applications such as streaming media applications (eg. movies), online gaming and Voice over IP (VoIP). The proposed DAEFR is suitable for infrastructures with user-driven/application-driven network capacity and utilization demands (such as campus networks and datacenters). Making use of SDN infrastructure and sFlow engine, the proposed approach can detect and re-reschedule TCP/UDP elephant flows using end-to-end path delay and bandwidth utilization, while mice flows are transmitted via reactive forwarding method. As the verification results, the proposed DAEFR method improves average throughput and FCT for elephant flows in comparison with Hedera, ECMP and reactive forwarding.

Author's Publications

[p1] H. T. Zaw, A. H. Maw, "Large Flow Detection and Delay Measuring for Multipath Routing over SDN", In 15th International Conference on Advanced Information Technologies (ICCA), pp 379-383, February 2017.

[p2] H. T. Zaw, A. H. Maw, "Elephant Flow Detection And Delay-Aware Flow Rerouting In Software-Defined Network", In 9th International Conference on Information Technology and Electrical Engineering (ICITEE), IEEE, pp. 1-6, October 2017.

[p3] H. T. Zaw, A. H. Maw, "Delay Controlled Elephant Flow Rerouting in Software Defined Network", In 1st International Conference on Advanced Information Technologies (ICAIT), pp. 52-57, November 2017.

[p4] H. T. Zaw, A. H. Maw, "PDFR: Path delay based flow rerouting in software-defined networks", In 16th International Conference on Advanced Information Technologies (ICCA), pp. 363-367, February 2018.

[p5] H. T. Zaw, A. H. Maw, "Traffic Management with Elephant Flow Detection in Software-Defined Networks", International Journal of Electrical and Computer Engineering (IJECE), Vol. 9, No. 4 (Part II), pp 3203-3211, August 2019.

Bibliography

[1] F. Adrian, J.P. Vasseur, and J. Ash, “A path computation element (PCE)-based architecture”, No. RFC 4655. 2006.

[2] M. Afaq, S. Rehman and W. C. Song, “Large flows detection, marking, and mitigation based on sFlow standard in SDN”, 18(2), pp.189-198, February 2015.

[3] Y. Afek, A. Bremler-Barr, F. S. Landau and L. Schiff, “Sampling and large flow detection in SDN”, In ACM SIGCOMM Computer Communication Review, ACM, Vol. 45, No. 4, pp. 345-346, August 2015.

[4] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, “A roadmap for traffic engineering in SDN-OpenFlow networks”, Computer Network, Vol. 71, pp. 1-30, Oct. 2014.

[5] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, “Hedera: Dynamic Flow Scheduling for Data Center Networks,” In Proc. NSDI, Vol 10, No. 2010, April 2010.

[6] J.R. Ballard, I. Rae, and A. Akella, “Extensible and Scalable Network Monitoring Using OpenSAFE”, In INM/WREN, April 2010.

[7] D. Banfi, M. Olivier, J. Guillaume, S. Lukas, and R. Holz, "Endpoint-transparent multipath transport with software-defined networks", In Local Computer Networks (LCN), 2016 IEEE 41st Conference on, IEEE, pp. 307-315, 2016.

[8] S. Bashir and N. Ahmed, “VirtMonE: Efficient detection of elephant flows in virtualized data centers”, In 2015 International Telecommunication Networks and Applications Conference (ITNAC), pp. 280-285, November 2015.

[9] T. Benson, A. Akella, and D. Maltz, “Network traffic characteristics of data centers in the wild”, In Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, ACM, pp. 267-280, 2010.

[10] T. Benson, A. Anand, A. Akella, and M. Zhang, ``MicroTE: Fine grained traffic engineering for data centers," In Proceedings of the Seventh Conference on emerging Networking EXperiments and Technologies

(CoNEXT), pp. 1-12, December 2011.

[11] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow et al., “Onos: towards an open, distributed sdn os," in Proceedings of the third workshop on Hot topics in software defined networking. ACM, pp. 1-6, 2014.

[12] A.Bianco, P. Giaccone, R. Mashayekhi, M. Ullio and V. Vercellone, "Scalability of ONOS reactive forwarding applications in ISP networks" Computer Communications 102 (2017), pp. 130-138, 2017.

[13] L. Bo, C. Ming, H. Chao, H. Hui, and X. Bo, "Optimizing the resource utilization of datacenter networks with OpenFlow", China Communications 13, No. 3, pp. 1-11, 2016.

[14] F. Carpio, A. Engelmann and A. Jukan, 2016, “DiffFlow: Differentiating short and long flows for load balancing in data center networks”, In 2016 IEEE Global Communications Conference (GLOBECOM), pp. 1-6, December 2016.

[15] S. Chakraborty and C. Chen, "A low-latency multipath routing without elephant flow detection for data centers", In High Performance Switching and Routing (HPSR), 2016 IEEE 17th International Conference on, IEEE, pp. 49-54, 2016.

[16] M. Chen, Y. Qian, S. Mao, W. Tang and X. Yang, “Software-defined mobile networks security”, Mobile Networks and Applications, 21(5), pp.729-743, 2016.

[17] M. Chen, V. C. M. Leung, S. Mao, and M. Li, ``Cross-layer and path priority scheduling based real-time video communications over wireless sensor networks," in Proceeding of IEEE 67th Vehicle Technology Conference (VTC), Singapore, pp. 2873-2877, May 2008.

[18] Y. R. Chiang, C. H. Ke, Y. S. Yu, Y. S. Chen, and C. J. Pan, "A multipath transmission scheme for the improvement of throughput over SDN", In 2017 International Conference on Applied System Innovation (ICASI), IEEE, pp. 1247-1250, 2017.

[19] M. Chiesa, G. Kindler, and M. Schapira, “Traffic engineering with equal-cost- multipath: An algorithmic perspective”, in Proc. IEEE INFOCOM, pp. 779-792, April 2017.

[20] S.R. Chowdhury, M. F. Bari, R. Ahmed and R. Boutaba, “Payless: A low cost network monitoring framework for software defined networks. In Network Operations and Management Symposium (NOMS), IEEE, pp. 1-9, May 2014.

[21] L. Cong, and W. Yong-Hao. "Strategy of Data Manage Center Network Traffic Scheduling Based on SDN." In Intelligent Transportation, Big Data & Smart City (ICITBS), 2016 International Conference on, pp. 29-34. IEEE, 2016.

[22] A. Curtis, W. Kim, and P. Yalagandula, “Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection,” in Infocom, Vol. 11, pp. 1629-1637, 2011.

[23] A. Dixit, P. Prakash, and R. R. Kompella, "On the efficacy of fine-grained traffic splitting protocols in data center networks", In ACM SIGCOMM Computer Communication Review, ACM, Vol. 41, No. 4, pp. 430-431, 2011.

[24] Docker, Available: https://www.docker.com/

[25] H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp,``OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end quality of service over software-defined networks," In Proceedings of The 2012 Asia Pacific Signal and Information Processing Association Annual Summit and Conference, IEEE, pp. 1-8, December 2012.

[26] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, “ Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers”, In SIGCOMM, 41(4), pp.339-350, 2011.

[27] M. Gholami and B. Akbari, "Congestion control in software defined data center networks through flow rerouting", In Electrical Engineering (ICEE), 2015 23rd Iranian Conference on, IEEE, pp. 654-657, 2015.

[28] S. Ghorbani and B. Godfrey, “Towards correct network virtualization”, in Proceedings of the third workshop on Hot topics in software defined networking. ACM, pp. 109-114, 2014.

[29] V. Goldberg, F. Wohlfart, and D. Raumer, “Datacenter network virtualization in multi-tenant environments”, in DFN-Forum

Kommunikationstechnologien, 2015 [Online], Available:http://www.net.in.tum.de/fileadmin/bibtex/publications/papers/DatacenterNetworkVirtualizationInMulti-TenantEnvironments.pdf.

[30] H.T. Zaw and A. H. Maw, "Elephant flow detection and delay-aware flow rerouting in software-defined network", In 9th International Conference on Information Technology and Electrical Engineering (ICITEE), IEEE, pp. 1-6, 2017.

[31] H.T. Zaw and A.H. Maw, “Delay Controlled Elephant Flow Rerouting in Software Defined Network”, In 1st International Conference on Advanced Information Technologies (ICAIT), pp 52-57, November 2017.

[32] H.T. Zaw and A.H. Maw, “PDFR: Path delay based flow rerouting in software-defined networks”, In 16th International Conference on Advanced Information Technologies (ICCA), pp 363-367, 2018.

[33] G. Han, Y. Dong, H. Guo, L. Shu, and D. Wu, ``Cross-layer optimized routing in wireless sensor networks with duty cycle and energy harvesting," Wireless communications and mobile computing, Vol. 15, No. 16, pp. 1957-1981, November 2015.

[34] S. Hegde, S. G. Koolagudi, and S. Bhattacharya, "Scalable and fair forwarding of elephant and mice traffic in software defined networks", Computer Networks, pp. 330-340, 2015.

[35] C. E. Hopps, “Analysis of an equal-cost multi-path algorithm”, 2000.

[36] HP Open SDN Ecosystem and SDN Applications [online], Available: https://h17007.www1.hpe.com/docs/sdn/4AA5-1871ENW.pdf, 2014.

[37] HP open Ecosystem breaks down barriers to SDN [online], Available: https://www8.hp.com/us/en/hp-news/press-release.html?id=1495044.

[38] Z. Hu and J. Luo, “Cracking network monitoring in DCNs with SDN”, In 2015 IEEE Conference on Computer Communications (INFOCOM), IEEE, pp. 199-207, April 2015.

[39] S. A. Hussain, S. Akbar, and I. Raza. "A dynamic multipath scheduling protocol (DMSP) for full performance isolation of links in software defined networking (SDN)." In Recent Trends in Telecommunications Research (RTTR), Workshop on, IEEE, pp. 1-5, 2017.

[40] IEEE 802.1Q-2011. VLAN Bridges, IEEE 2011.

[41] Iperf [Online], Available: https://iperf.fr.

[42] Java [online], Available: https://www.java.com.

[43] E. Jo, D. Pan, J. Liu, and L. Butler, "A simulation and emulation study of SDN-based multipath routing for fat-tree data center networks", In Simulation Conference (WSC), 2014 Winter, IEEE, pp. 3072-3083, 2014.

[44] R. Kanagevlu and K. M. M. Aung, "SDN controlled local re-routing to reduce congestion in cloud data center", In Cloud Computing Research and Innovation (ICCCRI), 2015 International Conference on, IEEE, pp. 80-88, 2015.

[45] M. T. Kao, B. X. Huang, S. J. Kao and H. W. Tseng, "An effective routing mechanism for link congestion avoidance in software-defined networking", In 2016 International Computer Symposium (ICS), IEEE, pp. 154-158, 2016.

[46] D. Katz, K. Kompella, and D. Yeung, “Traffic engineering (TE) extensions to OSPF version 2”, No. RFC 3630, 2003.

[47] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodol- molky, and S. Uhlig, “Software-defined networking: A comprehensive survey," proceedings of the IEEE, Vol. 103, No. 1, pp. 14-76, 2015.

[48] A. Kumar, M. Sung, J. J. Xu, and J. Wang, “Data streaming algorithms for efficient and accurate estimation of flow size distribution,” In Proceedings of the joint international conference on Measurement and modeling of computer systems, New York, USA, pp. 177–188, 2004.

[49] KVM [online], Available: https://www.linux-kvm.org/page/Main_Page

[50] Y. L. Lan, K. Wang, and Y. H. Hsu, "Dynamic load-balanced path optimization in SDN-based data center networks", In 2016 10th International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP), IEEE, pp. 1-6, 2016.

[51] B. Lantz, B. Heller, and N. 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. ACM, pp. 19, 2010.

[52] M. Laor and L. Gendel, “The effect of packet reordering in a backbone link on application throughput”, IEEE Network, 2002.

[53] K.C. Leung, V. O. K. Li, and D. Yang, “An overview of packet reordering in transmission control protocol (TCP): Problems, solutions, and challenges,” IEEE Trans. Parallel Distributed System, Vol. 18, No. 4, pp. 522–535, April 2007.

[54] Y. Li, and D. Pan, "OpenFlow based load balancing for Fat-Tree networks with multipath support", In Proceeding of 12th IEEE International Conference on Communications (ICC’13), Budapest, Hungary, pp. 1-5. 2013.

[55] T. S. Lin, Y. M. Hsu, S. Y. Kao, and P. W. Chi, "OpenE2EQoS: Meter-based method for end-to-end QoS of multimedia services over SDN", In Personal, Indoor, and Mobile Radio Communications (PIMRC), 2016 IEEE 27th Annual International Symposium on, IEEE, pp. 1-6, 2016.

[56] C. Y. Lin, C. Chen, J. W. Chang and Y. H. Chu, “Elephant flow detection in datacenters using openflow-based hierarchical statistics pulling”, In Global Communications Conference (GLOBECOM), IEEE, pp. 2264-2269, December 2014.

[57] J. Liu, Y. Li, M. Chen, W. Dong, and D. Jin, “Software-defined Internet of Things for smart urban sensing”, IEEE Communations Magazine., Vol. 53, No 8, pp. 55-63, September 2015.

[58] J. Liu, J. Li, G. Shou, Y. Hu, Z. Guo, and W. Dai, "SDN based load balancing mechanism for elephant flow in data center networks", In Wireless Personal Multimedia Communications (WPMC), 2014 International Symposium on, IEEE, pp. 486-490, 2014.

[59] V. Mann, A. Vishnoi and S. Bidkar, “Living on the edge: Monitoring network flows at the edge in cloud data centers”, In 2013 Fifth International Conference on Communication Systems and Networks (COMSNETS), IEEE, pp. 1-9, Janaury 2013.

[60] J. T. Mekkattuparamban, E. Narender and U. Srinivasan, "Dynamic load balancing without packet reordering", U.S. Patent 8,705,366, issued April 22, 2014.

[61] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. Lakshman, “Application-aware data plane processing in sdn," in Proceedings of the third workshop on Hot topics in software defined networking. ACM, pp. 13-18,

2014.

[62] Mininet [online], Available: http://mininet.org.

[63] J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. Curtis, and S. Banerjee, “DevoFlow: Scaling flow management for high-performance networks”, In ACM SIGCOMM Computer Communication Review, Vol. 41, No. 4, pp. 254-265, August 2011.

[64] T. T. Nguyen and D. S. Kim, "Accumulative-load aware routing in software-defined networks", In 2015 IEEE 13th International Conference on Industrial Informatics (INDIN), IEEE, pp. 516-520, 2015.

[65] ONOS Project [online], Available: https://wiki.onosproject.org/ display/ ONOS/System+Components.

[66] Open Networking Foundation (ONF), “OpenFlow switch Specification”, Version 1.3.0, June 25, 2012.

[67] Open Networking Summit [online],
Available: http:// www.opennetsummit.org/ pdf/ 2014/ sdn-idol/ HP-SDN-Idol-Proposal-1.pdf.

[68] Open vSwitch with SSL [Online],
Available:https://github.com/openvswitch/ovs/blob/master/

[69] A. Pathak, M. Zhang, Y. C. Hu, R. Mahajan, and D. Maltz, “Latency inflation with MPLS-based traffic engineering,” in Proc. ACM SIGCOMM Internet Measurement Conference (IMC), Berlin, Germany, pp. 463–472, 2011.

[70] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar et al., “The design and implementation of Open vSwitch,” in Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, ser. NSDI’15. Berkeley, CA, USA: USENIX Association, pp. 117–130, 2015. [Online], Available: http://dl.acm.org/citation.cfm?id=2789770.2789779

[71] S. Prabhavat, H. Nishiyama, N. Ansari, and N. Kato, "Effective delay-controlled load distribution over multipath networks", IEEE transactions on Parallel and distributed systems 22, No. 10, pp. 1730-1741, 2011.

[72] M. F. Ramdhani, S. N. Hertiana, and B. Dirgantara. "Multipath routing with load balancing and admission control in Software-Defined Networking

(SDN)", In Information and Communication Technology (ICoICT), 2016 4th International Conference on, IEEE, pp. 1-6, 2016.

[73] RFC 3954, “Cisco Systems NetFlow Services Export Version 9” [online], Available: http://tools.ietf.org/html/rfc3954.html.

[74] RFC 4655 [Online], Available: http://www.rfc-editor.org/rfc/rfc4655.txt.

[75] Salim, J. Hadi, D. Meyer, and O. Koufopavlou. "SDNRG E. Haleplidis Internet-Draft S. Denazis Intended status: Informational University of Patras Expires: September 4, 2014 K. Pentikousis EICT.", 2014.

[76] A. Schwabe and H. Karl, "Synrace: Decentralized load-adaptive multi-path routing without collecting statistics", In 2015 European Workshop on Software Defined Networks (EWSDN), IEEE, pp. 37-42, 2015.

[77] SDN Architecture Overview [online], Available: https:// www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf, 2013.

[78] S. A. R. Shah, W. Seok, J. Kim, S. Bae, and S. Y. Noh. "CAMOR: Congestion Aware Multipath Optimal Routing Solution by Using Software-Defined Networking", In 2017 International Conference on Platform Technology and Service (PlatCon), IEEE, pp. 1-6, 2017.

[79] S.A.R. Shah, S. Bae, A. Jaikar and S. Y. Noh, “An adaptive load monitoring solution for logically centralized SDN controller”, In 2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS), IEEE, pp. 1-6, 2016.

[80] S. Shanmugalingam, A. Ksentini, and P. Bertin, “DPDK Open vSwitch performance validation with mirroring feature,” in 2016 23rd International Conference on Telecommunications (ICT), IEEE, pp. 1–6, 2016.

[81] S. Shirali-Shahreza and Y. Ganjali, “Traffic statistics collection with FleXam”, In ACM SIGCOMM Computer Communication Review, Vol. 44, No. 4, pp. 117-118, August 2014.

[82] J. Suh, T. T. Kwon, C. Dixon, W. Felter and J. Carter, “Opensample: A low-latency, sampling-based measurement platform for commodity sdn”, In 2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS) , IEEE, pp. 228-237, June 2014.

[83] P. M. Thet, J. W. Kim, and C. Aswakul. "Multi-path chunked video exchanges over OF@ TEIN SDN cloud playground", In ITU Kaleidoscope: ICTs for a Sustainable World (ITU WT), IEEE, pp. 1-7, 2016.

[84] A. Tootoonchian, M. Ghobadi and Y. Ganjali, “OpenTM: traffic matrix estimator for OpenFlow networks”, In International Conference on Passive and Active Network Measurement, Springer, pp. 201-210, April 2010.

[85] Traffic monitoring using sFlow [online], Available: http://www.sflow.org.

[86] R. Trestian, K. Katrinis, and G.M. Muntean. "OFLoad: An OpenFlow-based dynamic load balancing strategy for datacenter networks", IEEE Transactions on Network and Service Management (2017), IEEE, 14(4), pp.792-803, 2017.

[87] Understanding Rapid Spanning Tree Protocol (802.1w), [Online]. Available: https: // www.cisco.com/ c/ en/ us/ support/ docs/ lan-switching/ spanning-tree-protocol/24062-146.html

[88] W. Wang, Y. Sun, K. Zheng, M. A. Kaafar, D. Li, and Z. Li. "Freeway: Adaptively isolating the elephant and mice flows on different transmission paths." In Network Protocols (ICNP), 2014 IEEE 22nd International Conference on, pp. 362-367. IEEE, 2014.

[89] Wireshark [online], Available: https://www.wireshark.org/

[90] C. Xu, B. Chen, P. Fu, and H. Qian. "A Dynamic Resource Allocation Model for Guaranteeing Quality of Service in Software Defined Networking Based Cloud Computing Environment." In International Conference on Cloud Computing and Security, Springer, pp. 206-217, 2015.

[91] H. Xu and L. Baochun, "TinyFlow: Breaking elephants down into mice in data center networks", In Local & Metropolitan Area Networks (LANMAN), 2014 IEEE 20th International Workshop on, IEEE, pp. 1-6, 2014.

[92] H. Xu, and L. B. Li, "RepFlow: Minimizing flow completion times with replicated flows in data centers", In INFOCOM, 2014 Proceedings IEEE, pp. 1581-1589, 2014.

[93] J. Yan, H. Zhang, Q. Shuai, B. Liu, and X. Guo, ``HiQoS: An SDN-based multipath QoS solution," China Communications, Vol. 12, No. 5, pp. 123-133, May 2015.

[94] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang and H.V. Madhyastha, “Flowsense: Monitoring network utilization with zero measurement cost”, In International Conference on Passive and Active Network Measurement, Springer, pp. 31-41, March 2013.

[95] M. Yu, L. Jose and R. Miao, “Software Defined Traffic Measurement with OpenSketch”, In NSDI , Vol. 13, pp. 29-42, April 2013.

[96] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, “Scalable flow-based networking with DIFANE”, ACM SIGCOMM Computer Communication Review, Vol. 41, No. 4, pp. 351-362, October 2011.

[97] T. F. Yu, K. Wang and Y. H. Hsu, "Adaptive routing for video streaming with QoS support over SDN networks", In 2015 International Conference on Information Networking (ICOIN), IEEE, pp. 318-323, 2015.

[98] H. Zhang, X. Guo, J. Yan, B. Liu, and Q. Shuai. "SDN-based ECMP algorithm for data center networks." In Computing, Communications and IT Applications Conference (ComComAp), IEEE, pp. 13-18, 2014.

[99] R. Zhou, "Datacenter network large flow detection and scheduling from the edge", reading & research project, 2014.

APPENDIX A: TESTBED TOPOLOGY SETUP

The experimental testbed topology, k=4 fat-tree topology is described by using Mininet python script. Mininet supports parametrized topologies. Mininet API allows the users to create custom networks depending on the needs, using a few lines of code in Python. In this section fat-tree topology script is presented. Fat-tree topology is a well-known data center network topology, which contains various paths between end hosts, so it can give higher accessible data transmission than a single path tree with a similar number of nodes. It is normally a 3-layer various leveled tree that comprises of switches on the edge, aggregate, and core layers. The k=4, fat-tree topology includes 20 switches and 16 hosts. In Mininet script, firstly the switches and hosts are created as follows:

```
1.  self.pod = k
2.  self.iCoreLayerSwitch = (k/2)**2
3.  self.iAggLayerSwitch = k*k/2
4.  self.iEdgeLayerSwitch = k*k/2
5.  self.density = k/2
6.  self.iHost = self.iEdgeLayerSwitch * self.density
7.
8.  def createCoreLayerSwitch(self, NUMBER):
9.          logger.debug("Create Core Layer")
10.         self._addSwitch(NUMBER, 1, self.CoreSwitchList)
11.
12. def createAggLayerSwitch(self, NUMBER):
13.         logger.debug("Create Agg Layer")
14.         self._addSwitch(NUMBER, 2, self.AggSwitchList)
15.
16. def createEdgeLayerSwitch(self, NUMBER):
17.         logger.debug("Create Edge Layer")
18.         self._addSwitch(NUMBER, 3, self.EdgeSwitchList)
19.
20. def createHost(self, NUMBER):
21.         logger.debug("Create Host")
22.         for x in xrange(1, NUMBER+1):
23.             PREFIX = "h00"
24.     ipaddr="10.0.0."
25.     macaddr="00:00:00:00:00:"
26.     m = "0"
27.             if x >= int(10):
28.                 PREFIX = "h0"
29.         m=""
30.             elif x >= int(100):
31.                 PREFIX = "h"
```

```
32. self.HostList.append(self.addHost(PREFIX +str(x),ip= ipaddr + str(x),mac= ma
    caddr + m +str(x)))
```

According to the above k=4 fat-tree Mininet scripts, number of core layer switch is 4, number of aggregation layer switch is 8, the number of edge switch is 8 and the number of end host is 16. All layer switches create using *addSwitch*( ) API. This API adds a switch to the topology and returns a switch name. The switch names of core layer switches are from s1001 to s1004, aggregation layer switches are from s2001 to s2008 and the edge layer switches are from s3001 to s3008, respectively. All of end hosts are created using *addHost*( ) API which adds a host to the topology and returns a host name. The end host names are starting from h001 to h016. The IP addresses and MAC addresses are 10.0.0.1~16 and 00:00:00:00:00:01~00:00:00:00:00:16 respectively.

Then to perform the emulation, the following command is used:

```
$ sudo mn –custom sflow-rt/extras/sflow.py, fattree.py –topo fattree,4 –link=tc
–controller remote,ip=ONOS_IP
```

At this point, the above commands will be illustrated. By “sudo mn”, the user gets administrator rights on mininet and instruct the mininet to perfrom sflow.py and fattree.py. Here, sflow.py script is used to embed the sFlow agent in Open vSwitches and all switches are monitored by sFlow-RT collector. The fattree.py script is located in the custom folder of mininet and create fattree topology with the corresponding to govern as defined in fattree.py. The command “--link=tc” allows users to set custom link parameters such as bandwidth and delay. Finally, the most important argument is the “--controller remote,ip=ONOS_IP”. With this command, it defines the topology controller which is needed to add the IP address of ONOS machine. This fact may be local host address if the controller is running in local machine.

APPENDIX B: IMPLEMENTATION OF ELEPHANT FLOW DETECTION

In this section, the elephant flow detection script implementation is discussed in details. The elephant flow detection application is embedded application using sFlow-RT's internal JavaScript API. Embedded elephant flow detection application has the following directory structure:

```
sflow-rt/app/elephant/html/index.html
sflow-rt/app/elephant/scripts/
```

When sFlow-RT is started, it will start a thread for each of the files with a .js extension in the scripts directory and will expose the html directory, if it exists, under the path:

```
http://sFlowCollectorIP:8008/app/elephant/html/
```

Information about running scripts can be found under the path:

```
http://sFlowCollectorIP:8008/scripts/json
```

There are three main implementations for elephant flow detection script:

(i) Defining flow

Flows are used to match packets or transactions that share common attributes and compute rate information. A flow called *tcpflow* that captures the source and destination MAC addresses, source and destination IP addresses, TCP ports, interface index of ingress port and egress port and calculates bytes per second for each flow:

```
1. setFlow('tcpflow',
2. {keys:
3. 'macsource, macdestination,
4. ipsource, ipdestination,
5. tcpsourceport, tcpdestinationport,
6. link:inputifindex, link:outputifindex',
7.  value:
8. 'bytes'});
```

A flow called *udpflow* captures the source and destination MAC addresses, source and destination IP addresses, UDP ports, interface index of ingress port and egress port and calculates bytes per second for each flow:

```
1. setFlow('udpflow',
2. {keys:
3. 'macsource,macdestination,
4. ipsource,ipdestination,
5. udpsourceport,udpdestinationport,
6. link:inputifindex,link:outputifindex',
7. value:
8. 'bytes'});
```

Metrics are derived from period sFlow counter records received from sFlow agents and new metrics can be created by defining flows. Therefore, the attributes in keys are also called metrics.

(ii) Defining threshold

Thresholds are applied to metric values. Defining flows create additional metrics and thresholds can be applied to generate a notification when the rate value associated with a flow exceeds the threshold. The proposed approach follows the sFlow standard threshold values as mentioned in Chapter 4. Therefore the elephant flow is defined when the flow rate consumes 10% of link bandwidth. The following function defines a threshold to detect TCP and UDP flows exceeding 10 Mbit/s (at that time the configured link bandwidth is 100 Mbit/s) based on the flow definition above:

```
1.  setThreshold('tcpelephant',
2. {metric:'tcpflow',
3. value: 10000000/8,
4. byFlow:true,
5. timeout:1});
6. setThreshold('udpelephant',
7. {metric:'udpflow',
8. value: 10000000/8,
```

```
9.  byFlow:true,
10. timeout:1});
```

The "*byFlow"* flag indicates that a threshold event should be generated for each individual flow crossing the threshold. If the flag is omitted, a single event will be generated for the largest flow. The *timeout* defines seconds of hysteresis before re-arming threshold, i.e. metric value must be below threshold for timeout seconds.

(iii) Handling threshold event

The events are converted to JSON and logs them. This is a useful way to see the contents of the elephant flow event before deciding how they should be rerouted. Register an event handler function to be notified of each new elephant flow event:

```
1.  setEventHandler(function(evt) {
2.  logInfo(evt.flowKey+":"+evt.thresholdID);
3.  },['tcpelephant','udpelephant']);
```

Each event logs the detailed metrics, which defined in flow key and threshold information. The elephant event can queried from ONOS application by calling *http://sFlowCollectorIP:8008/scripts/event.json* periodically.

APPENDIX C: IMPLEMENTATION OF FLOW MANAGEMENT APPLICATION USING ONOS

In this section, how to implement the flow management application using ONOS controller is described by using some part of Java codes. In this implementation, three main parts: (i) accessing elephant flow information from sFlow, (ii) installing proactive flow rules for probe packets and (iii) processing probes and calculating end-to-end delay are described especially. Since ONOS is based Java programming language [42], DAEFR application is also developed by Java. There are five main components in DAEFR as described in Chapter 3. First of all, in order to perform the flow management as soon as the DAEFR application is activated, the main function of DAEFR is implemented within default activate ( ) function.

```
1.  @Activate
2.      protected void activate() {
3.          log.info("Started");
4.          log.info("Welcome to elephant flow manager");
5.          appId = coreService.registerApplication("org.loadbalancer.app");
6.          packetService.requestPackets(intercept.build(), PacketPriority.CONTROL,
            appId);
7.          packetService.addProcessor(packetProcessor,PacketProcessor.director(1));
8.          timer.schedule(new DetectLargeFlow(), 0, 1000);
9.  }
```

In above codes, the application is registered to ONOS Core Service in order to get an application ID. Then, it requests some packets (which have Ethernet Type: 0x8888) using ONOS Packet Service. The Ethernet Type 0x8888 is the probe packets for measuring end-to-end delay. The requested packets are as follows:

```
TrafficSelector.Builder intercept = DefaultTrafficSelector.builder()
                                        .matchEthType((short)0x8888);
```

This is to handle and get the delay measurement probes from DAEFR application firstly in order to process the delay measurement function. The parameter

PacketPriority.CONTROL means that the highest priority for control traffic. The parameter *PacketProcessor.Director(1)* means that the DAEFR application gets and process the probe packets firstly when other applications are running in parallel at the same time. After that, the timer is started and Detect_Large_Flow function( ) starts working every one second.

(i) Accessing Elephant Flow Information

Detect_Large_Flow function ( ) is detecting elephant flow from sFlow-RT analyzer. This function has responsibility to read the REST API periodically from sFlow according to below codes:

```
1. public void Detect_Elephant_Flow() throws IOException, JSONException {
2.  String url = "http://172.16.33.33:8008/events/json";
3.  InputStream is = new URL(url).openStream();
4.  try {
5.   BufferedReader rd = new BufferedReader(new InputStreamReader(is, Charset.
    forName("UTF-8")));
6.   String jsonText = readAll(rd);
7.   if (jsonText.contains("flowKey")) {
8.    String key = "";
9.    long timestamp = 0;
10.   JSONArray arrayObj = new JSONArray(jsonText);
11.   for (int i = 0; i < arrayObj.length(); i++) {
12.    JSONObject obj = arrayObj.getJSONObject(i);
13.    timestamp = obj.getLong("timestamp");
14.    key = obj.getString("flowKey");
15.    String[] data = key.split(",");
16.    String thrs = obj.getString("thresholdID");
17.   }
18.   if (timestamp > previousTimestamp) {
19.    log.info("Threshold ID: " + thrs);
20.    log.info("flowKey: " + data[0] + "," + data[1] + "," + data[2] + "," + data[3] +
        "," + data[4] + "," + data[5]);
21.    previousTimestamp = timestamp;
```

```
22.     SplitLargeFlow(data, thrs, timestamp);
23.    }
24.   }
25.  }
26. } catch (Exception ex) {
27.  ex.printStackTrace();
28. }
29. finally {
30.  is.close();
31. }
32. }
```

The above codes read the elephant flow event from sFlow-RT which is running on 172.16.33.33:8008 in every one second. From the event, the elephant information can be retrieved as the timestamp, the flow key (contains flow metrics) and thresholdID (e.g. tcpelephant or udpelephant) in json format. As soon as the timestamp value is greater than previous timestamp, these values are passed to ComputeShortestPaths( ) function which is the second component. The ComputeShortestPaths( ) function returns the available shortest paths between source host and destination host and triggers the end-to-end delay estimation module.

(ii) Installing Proactive Flow Rules for Probes

In delay measurement module, before sending out probes, the flow rules for probes are needed to install along the path to pass through. The following AddRuleForProbe function is to install the rules for probes over available paths. The function input parameters are the path which to be measured, the faked source and destination MAC addresses.

```
1.  public void AddRuleForProbe(Path path, String src_probeMac, String dst_probe
    Mac) {
2.   List < Link > linkcol = path.links();
3.   ListIterator < Link > itrlink = linkcol.listIterator();
4.   int i = 0;
5.   String temp_src_mac, temp_dst_mac;
```

```
6.  temp_dst_mac = probeMac;
7.  while (itrlink.hasNext()) {
8.   Link link = itrlink.next();
9.   if (i == 0) {
10.   TrafficSelector.Builder sel = DefaultTrafficSelector.builder();
11.   Criterion cr = Criteria.matchEthType(0x8888);
12.   sel.add(cr);
13.   TrafficSelector pselector1 = sel.build();
14.   Instruction poutput = Instructions.transition(1);
15.   TrafficTreatment.Builder ptreatment = DefaultTrafficTreatment.builder();
16.   TrafficTreatment ptreatment1 = ptreatment
17.    .add(poutput)
18.    .build();
19.   FlowEntry.Builder pflowentry = DefaultFlowEntry.builder();
20.   pflowentry.forDevice(link.src().deviceId());
21.   pflowentry.forTable(0);
22.   pflowentry.withPriority(51000);
23.   pflowentry.withSelector(pselector1);
24.   pflowentry.withTreatment(ptreatment1);
25.   pflowentry.fromApp(appId);
26.   pflowentry.makeTemporary(20);
27.   flowRuleService.applyFlowRules(pflowentry.build());
28.   TrafficSelector.Builder selector = DefaultTrafficSelector.builder();
29.   Criterion c = Criteria.matchEthSrc(MacAddress.valueOf(src_probeMac));
30.   Criterion c2 = Criteria.matchEthType(0x8888);
31.   Criterion c3 = Criteria.matchEthDst(MacAddress.valueOf(dst_probeMac));
32.   selector.add(c);
33.   selector.add(c2);
34.   selector.add(c3);
35.   TrafficSelector selector1 = selector.build();
36.   Port dst_port = deviceService.getPort(link.dst().deviceId(), link.dst().port());
37.   temp_dst_mac = dst_port.annotations().value("portMac");
38.   Instruction output = Instructions.createOutput(link.src().port());
```

```
39.  TrafficTreatment.Builder treatment = DefaultTrafficTreatment.builder();
40.  TrafficTreatment treatment1 = treatment
41.   .add(Instructions.modL2Dst(MacAddress.valueOf(dst_port.annotations().va
     lue("portMac"))))
42.   .add(output)
43.   .build();
44.  FlowEntry.Builder flowentry = DefaultFlowEntry.builder();
45.  flowentry.forDevice(link.src().deviceId());
46.  flowentry.forTable(1);
47.  flowentry.withPriority(51000);
48.  flowentry.withSelector(selector1);
49.  flowentry.withTreatment(treatment1);
50.  flowentry.fromApp(appId);
51.  flowentry.makeTemporary(20);
52.  flowRuleService.applyFlowRules(flowentry.build());
53.  i++;
54. } else {
55.  TrafficSelector.Builder sel = DefaultTrafficSelector.builder();
56.  Criterion cr = Criteria.matchEthType(0x8888);
57.  sel.add(cr);
58.  TrafficSelector pselector1 = sel.build();
59.  Instruction poutput = Instructions.transition(1);
60.  TrafficTreatment.Builder ptreatment = DefaultTrafficTreatment.builder();
61.  TrafficTreatment ptreatment1 = ptreatment
62.   .add(poutput)
63.   .build();
64.  FlowEntry.Builder pflowentry = DefaultFlowEntry.builder();
65.  pflowentry.forDevice(link.src().deviceId());
66.  pflowentry.forTable(0);
67.  pflowentry.withPriority(51000);
68.  pflowentry.withSelector(pselector1);
69.  pflowentry.withTreatment(ptreatment1);
70.  pflowentry.fromApp(appId);
```

```
71.  pflowentry.makeTemporary(20);
72.  flowRuleService.applyFlowRules(pflowentry.build());
73.  TrafficSelector.Builder selector2 = DefaultTrafficSelector.builder();
74.  Criterion cc = Criteria.matchEthSrc(MacAddress.valueOf(probeMac));
75.  Criterion cc1 = Criteria.matchEthType(0x8888);
76.  Criterion cc2 = Criteria.matchEthDst(MacAddress.valueOf(temp_dst_mac));
77.  selector2.add(cc);
78.  selector2.add(cc1);
79.  selector2.add(cc2);
80.  TrafficSelector selector3 = selector2.build();
81.  Port dst_port1 = deviceService.getPort(link.dst().deviceId(), link.dst().port());
82.  temp_dst_mac = dst_port1.annotations().value("portMac");
83.  Instruction output1 = Instructions.createOutput(link.src().port());
84.  TrafficTreatment.Builder treatment2 = DefaultTrafficTreatment.builder();
85.  TrafficTreatment treatment3 = treatment2
86.   .add(Instructions.modL2Dst(MacAddress.valueOf(dst_port1.annotations().val
    ue("portMac"))))
87.   .add(output1)
88.   .build();
89.  FlowEntry.Builder flowentry1 = DefaultFlowEntry.builder();
90.  flowentry1.forDevice(link.src().deviceId());
91.  flowentry1.forTable(1);
92.  flowentry1.withPriority(51000);
93.  flowentry1.withSelector(selector3);
94.  flowentry1.withTreatment(treatment3);
95.  flowentry1.fromApp(appId);
96.  flowentry1.makeTemporary(20);
97.  flowRuleService.applyFlowRules(flowentry1.build());
98.  }
99. }
100.  TrafficSelector.Builder sel = DefaultTrafficSelector.builder();
101.  Criterion cr = Criteria.matchEthType(0x8888);
102.  sel.add(cr);
```

```
103.  TrafficSelector pselector1 = sel.build();
104.  Instruction poutput = Instructions.transition(1);
105.  TrafficTreatment.Builder ptreatment = DefaultTrafficTreatment.builder();
106.  TrafficTreatment ptreatment1 = ptreatment
107.   .add(poutput)
108.   .build();
109.  FlowEntry.Builder pflowentry = DefaultFlowEntry.builder();
110.  pflowentry.forDevice(path.dst().deviceId());
111.  pflowentry.forTable(0);
112.  pflowentry.withPriority(51000);
113.  pflowentry.withSelector(pselector1);
114.  pflowentry.withTreatment(ptreatment1);
115.  pflowentry.fromApp(appId);
116.  pflowentry.makeTemporary(20);
117.  flowRuleService.applyFlowRules(pflowentry.build());
118.  TrafficSelector.Builder selector2 = DefaultTrafficSelector.builder();
119.  Criterion cc = Criteria.matchEthSrc(MacAddress.valueOf(probeMac));
120.  Criterion cc1 = Criteria.matchEthType(0x8888);
121.  Criterion cc2 = Criteria.matchEthDst(MacAddress.valueOf(temp_dst_mac));
122.  selector2.add(cc);
123.  selector2.add(cc1);
124.  selector2.add(cc2);
125.  TrafficSelector selector3 = selector2.build();
126.  TrafficTreatment.Builder treatment3 = DefaultTrafficTreatment.builder()
127.   .setOutput(PortNumber.CONTROLLER);
128.  FlowEntry.Builder flowentry1 = DefaultFlowEntry.builder();
129.  flowentry1.forDevice(path.dst().deviceId());
130.  flowentry1.forTable(1);
131.  flowentry1.withPriority(51000);
132.  flowentry1.withSelector(selector3);
133.  flowentry1.withTreatment(treatment3.build());
134.  flowentry1.fromApp(appId);
135.  flowentry1.makeTemporary(20);
```

```
136.  flowRuleService.applyFlowRules(flowentry1.build());
137.  }
```

The path consists of the collection link list. Each link has source port and destination port until there is no more next link. The flow rule for each source switch of links includes matching fields and instructions. If there is no more next links in list, it assumes the final destination switch of the path and no entries are installed. Here, the important fact is that the destination MAC address of matching fields are changed according to the MAC address of next link destination port. In order to match the probe packet when it comes back to controller, the faked source MAC address and the path are needed to keep in controller until it comes back. After installing flow rules for probes, the probe is sent along the path.

(iii) Processing Probes and Calculating End-to-End Delay

When the probe comes back to controller, the packet sent time can be retrieved from payload of probe packet as shown in following codes.

```
1.  public class DAEFR_PacketProcessor implements PacketProcessor {
2.   @Override
3.   public void process(PacketContext context) {
4.    long receive_time = System.nanoTime();
5.    InboundPacket pkt = context.inPacket();
6.    Ethernet ethPkt = pkt.parsed();
7.    DecimalFormat dft = new DecimalFormat("###.######");
8.    DelayMonitor dm = new DelayMonitor();
9.    String srcmac = ethPkt.getSourceMAC().toString();
10.   String dstmac = ethPkt.getDestinationMAC().toString();
11.   if (context.isHandled()) {
12.    return;
13.   }
14.   if (ethPkt.getEtherType() == (short) 0x8888) {
15.    byte[] payload = ethPkt.getPayload().serialize();
16.    MyProto mp2 = new MyProto();
17.    long ts = mp2.mydeserialize(payload);
```

```
18.      double delay = (receive_time - ts);
19.      String d = dft.format(delay / 1000000);
20.      dm.MatchProbe(srcmac, d);
21.      context.block();
22.     }
23.    }
24.  }
```

The above codes, DAEFR_PacketProcessor intercept all packets, which are directed to the controller. In this module, when the packet arrives, it calculates the packet receive time. Then, the source MAC address and the destination MAC address are retrieved from incoming Ethernet packet. If the Ethernet type of received packet is equaled to 0x8888, the payload of this packet is parsed in order to get the original packet sent time. Then, the total travelling time along the path of this packet can be estimated from packet sent time and received time. Then it invokes the MatchProbe function with source MAC address and total delay time (in milliseconds). The MatchProbe function is responsible to match the source MAC address of received packet with memorized source MAC address in table. If the source MAC address is matched, the end-to-end delay is calculated. Then, this end-to-end delay is the current delay of the path which memorized with source MAC address. In this way, the end-to-end delays of available paths between source and destination host can be estimated and triggers the link load utilization measurement function (mentioned Chapter 3). Then, it calculates the least cost path based average end-to-end delay and load ratio. Finally, the elephant flow is rerouted to the least cost path by installing new flow rules.