Stability and Robustness of Control Planes in OpenFlow Networks

Daisuke Kotani

Abstract

Computer networks are essential for our daily life, and the use cases are expanding from connecting individually owned physical computers to cloud computing, Internet of Things, etc. To meet requirements for new use cases, computer networks should have flexibility of network control mechanisms, that is, network control mechanisms should be able to be improved when needed, as well as the networks are operated stably and robustly to avoid a huge negative impact on the society. However, it is hard for people other than network device vendors to develop and deploy network control mechanisms because traditional network devices tightly couple control software with physical devices.

Software Defined Networking (SDN) is changing this situation in private networks like campus, enterprise and data center networks. In SDN, network control software is decoupled from network devices, and the devices provide an open and common API to centrally control from external computers (controllers) how the devices forward packets. SDN enables network managers to meet their needs by developing and deploying their own network control software with the API on the devices. OpenFlow is attracted as a future promising API, and the devices that support OpenFlow feature are called OpenFlow switches. OpenFlow abstracts packet forwarding rules on each device as flow entries, each of which consists of a tuple of header fields and their values as a condition to match packets, actions applied to matched packets, and statistics. Packets that match flow entries are processed as specified by actions. The flow entries are installed by controllers, and packets that miss flow entries are encapsulated into Packet-In messages, which is one of OpenFlow messages, and the Packet-In messages are sent to controllers through OpenFlow channels over TCP unless otherwise specified.

SDN based on OpenFlow can increase flexibility of introducing new network control mechanisms, but has issues in terms of stability and robustness, such as fault tolerance, protection of network devices against unexpected traffic, and avoidance of a high load by control software. In SDN and OpenFlow, mechanisms to increase stability and robustness

must retain flexibility of network control, while the mechanisms in traditional networks are designed under particular specifications. Processing for network control is almost centralized in controllers in SDN and OpenFlow, whereas control software is distributed across network devices in traditional networks. This thesis discusses how we can address these issues.

The first issue is in processing at controllers. Controllers centrally handle all events occurring at a network, therefore loads on controllers easily increase when more events occur and processing of handling each event causes a high load. Multicast control is one of the cases where we need to care for this point. The multicast control in a private network requires that the network update multicast trees on-demand according to a state of senders and receivers. In addition, considering typical applications of multicast like video streaming, the network should restore delivery of multicast packets quickly at the time of failures. Traditional network technology cannot meet both at the same time, and OpenFlow based approaches have potential to meet. However, if a controller controls multicast in a simple way, that is, the controller recalculates and reinstalls multicast trees when a set of senders and receivers is changed or a failure occurs on the network, a load on the controller would increase because tree computation takes much time. Furthermore, the controller should take care of low performance on modification of flow entries in switches for fast restoration of delivery of multicast packets.

This thesis proposes to precompute redundant multicast trees that covers all switches where receivers may be connected, and install a part of each tree that is needed to connect a sender to actual receivers. All multicast trees are labeled, and a switch that packets come in attaches a label of a multicast tree that should be used for delivery of multicast packets. With this method, the controller reduces the processing time to add or remove a receiver on existing multicast groups and to recover multicast trees from failures, and needs not to replace many flow entries in a switch at the time of a failure.

The second issue is in an OpenFlow channel between a controller and a switch. A failure of the channel should be detected immediately to keep a network controllable. However, exchanging keep-alive messages frequently is inadequate because controllers should handle many keep-alive messages but message processing performance in the controllers is not necessarily high.

Instead of checking that a channel is always available, this thesis proposes to detect a failure when a message that should be delivered to the other end is generated since no message means no data must be updated at both controllers and switches. Controllers share and compare arrival of messages from switches, and find failed channels when arrival of messages is inconsistent. Switches send a keep-alive message just after sending important messages such as port down and important Packet-In messages. As a result, the switches and the controllers can detect a failure of a channel immediately when needed, and can take appropriate actions.

The third issue is a protection of switches against many unknown packets. When a switch receives many packets that miss flow entries before corresponding flow entries are installed, a CPU on the switch, which often has poor performance, would be overloaded due to generating many Packet-In messages, and the switch would become uncontrollable. The switch is hard to filter out all of such packets because some of them should be important for network control and should not be filtered out.

This thesis proposes a mechanism on switches to selectively filter out such packets while surely passing important packets to the controllers as Packet-In messages. Header fields that the controllers use for network control are specified to the switches by the controllers in advance. A switch records values of the header fields specified by the controllers, and filters out Packet-In messages including packets that match values recorded. This can reduce the processing at the CPU on the switch without huge negative effect on network control.

Contents

1 Introduction **10**

1.1 Overview of Software Defined Networking and OpenFlow 13

1.2 Issues concerning Centralized Control 14

1.2.1 Performance of Controllers and Switches 14

1.2.2 Failure Detection of OpenFlow Channels 15

1.3 An Issue that Concerns Decoupling Network Control from Network Devices 17

1.3.1 Protection of Switches against Many Unknown Packets 17

1.4 Organization of This Thesis 18

2 Software Defined Networking and OpenFlow **21**

2.1 Software Defined Networking 21

2.2 OpenFlow 23

2.3 Research on SDN and OpenFlow from the Viewpoint of Stability and Robustness 25

3 Management of Multicast Trees in OpenFlow Controllers **30**

3.1 Introduction 30

3.2 Related Work 34

3.2.1 Control and Packet Delivery of Multicast in Traditional Networks 34

3.2.2 Algorithms to Compute Multicast Trees 35

3.2.3 Multicast and Failure Recovery in OpenFlow 35

3.3 Tree Management Method 36

3.3.1 Tree Update when Multicast Group Membership Changes 37

3.3.2 Failure Recovery 41

3.3.3 Multicast Group Membership and Tree Databases 42

3.4 Implementation of Prototype System 43

3.5 Evaluation 45
3.5.1 Processing Time of Controllers for Update of Multicast Group Membership and Failure Recovery 46
3.5.2 Recovery Time of the Multicast Tree 49
3.6 Discussion 50
3.6.1 Processing Time in Controllers 50
3.6.2 Failure Recovery 52
3.6.3 Limitations and Overheads 53
3.6.4 Comparison of Signaling and Available Trees with Existing Multicast Control Mechanisms 54
3.7 Concluding Remark 55

4 Fast Failure Detection of OpenFlow Channels 56
4.1 Introduction 56
4.2 Related Work 58
4.2.1 Failure Detector in Distributed Systems 58
4.2.2 Failure Detection and Recovery in OpenFlow Networks 60
4.3 Types of OpenFlow Channel Failure 60
4.4 Detection of OpenFlow Channel Failure 61
4.4.1 Detection at Switches 62
4.4.2 Detection at Controllers 63
4.4.3 Sequence ID in Messages for Duplicate Filtering and Reordering . 64
4.5 Implementation of a Prototype System as an OpenFlow Channel Proxy . 65
4.6 Evaluation 66
4.6.1 Failure Detection Delay 67
4.6.2 Overhead in Throughput and Latency 68
4.7 Discussion 69
4.8 Concluding Remark 71

5 Protection of OpenFlow Switches against Many Unknown Packets 72
5.1 Introduction 72
5.2 Related Work 75
5.2.1 Protection of Control Planes in Traditional Network Devices . . . 75
5.2.2 DoS Attacks 75
5.2.3 Extending Capability in OpenFlow Switches 76

5.3 Proposed Mechanism . . . 77
5.3.1 Important Packet-In Messages for Network Control . . . 77
5.3.2 Overview of Proposed Mechanism . . . 78
5.3.3 Pending Flow Rules . . . 79
5.3.4 Pending Flow Tables . . . 80
5.3.5 An Example of Pending Flow Rules and Tables . . . 81
5.4 Implementation of a Prototype Switch . . . 83
5.5 Evaluation . . . 85
5.5.1 Use Cases and Pending Flow Rules . . . 85
5.5.2 Loads in Switches . . . 86
5.5.3 Processing Time of Actions in Switches . . . 90
5.6 Discussion . . . 90
5.6.1 Effects of Our Proposed Mechanism . . . 91
5.6.2 Overhead by Our Proposed Mechanism . . . 93
5.6.3 Similarity with OpenFlow Mechanisms and Possibility for Hardware Implementation . . . 94
5.6.4 Drawbacks of Our Proposed Mechanism . . . 95
5.6.5 Attacks and Mitigations . . . 96
5.7 Concluding Remark . . . 97

6 Conclusion **98**
6.1 Summary . . . 98
6.2 Future Directions . . . 100

List of Figures

1.1 Conceptual diagram of SDN architecture 13
1.2 Overview of OpenFlow 14

2.1 Traditional network architecture 22
2.2 SDN architecture 22
2.3 OpenFlow architecture 23
2.4 Multiple controller support in OpenFlow 25
2.5 Overview of research topics on SDN and OpenFlow and their stability and robustness issues 26

3.1 An example of processing multiple trees 41
3.2 An overview of our controller design 44
3.3 Three-layer hierarchical topology for evaluation 46
3.4 Processing time to add a receiver 47
3.5 Processing time to remove a receiver 48
3.6 Processing time to recover from a link failure 49
3.7 Network topology for evaluation of packet loss duration during a failure recovery 49
3.8 Maximum packet loss duration when a link between Root Switch and Core Switch 1 is down 50

4.1 A key idea to detect a failure at controllers 63
4.2 Overview of prototype OpenFlow channel proxy 66
4.3 Connections for evaluation 67

5.1 Problems in OpenFlow switches when a large number of packets are suddenly sent 73
5.2 Conceptual diagram for the proposed mechanism 78

5.3 A design of our prototype switch based on Open vSwitch 83
5.4 Load evaluation setting 88
5.5 CPU load in the switch 88

List of Tables

2.1 An example of a flow table 24

4.1 Specifications of PCs used for evaluation 67
4.2 Keep-alive intervals and timeouts 68
4.3 Average failure detection delay 69
4.4 Average throughput and latency 70

5.1 An example of a packet, pending flow rules, pending flow tables (in part) 82
5.2 Number of pending flow rules in typical use cases 86
5.3 Specifications of PCs used for evaluation 88
5.4 Number of Packet-In messages and pending flow entries per second (min/average/max) 89
5.5 Execution time by components in switches 91

Chapter 1

Introduction

Computer networks and the Internet are essential for our daily life. People connect their devices, not only their personal computers but also their smart phones, to the Internet from anywhere by wired LANs, Wi-Fi, LTE, etc., and use various services over the Internet. They also connect sensors and electric home appliances to the Internet for collecting and analyzing data, and operating such devices remotely.

To support current and emerging use cases of computer networks, a control mechanism of networks should have flexibility of introducing new protocols, new methods for network control and management automation functions that are suitable for the new use cases in a timely manner. Such flexibility is important especially in private networks where hosts are directly attached like campus and enterprise networks. From a security perspective, for example, networks should authenticate an owner of connected hosts to track users who illegally use the networks, monitor traffic to find malicious flows, dynamically isolate hosts infected by malware, and discard Denial of Service (DoS) traffic at ingress routers and switches. Some functions will be implemented into middleboxes installed in somewhere on networks, and network control mechanisms need to automatically configure packet forwarding rules to deliver packets through appropriate middleboxes. From a perspective of people and companies who operate services over the networks, they need to connect computing resources, which they rent from cloud service providers, to their networks on-demand to start using such resources seamlessly. Cloud service providers must properly isolate communication between users. From a perspective of network service providers, they need to automate rerouting traffic to avoid congestion because a traffic pattern suddenly changes due to events such as TV shows and update of system design in application service providers. Network managers should be able to develop and deploy new protocols, control mechanisms, and automation mechanisms into their networks to meet users' and their needs like above quickly.

At the same time, networks should be stable and robust enough not to become unavailable in any condition because there is a huge negative impact on users' life and business when networks are down. Networks should restore their connectivity immediately when a fault occurs in the networks. A number of hosts are connected or disconnected at different timing, and networks should be scalable to frequent connection or disconnection of a number of hosts. Hosts sometimes send traffic that network managers do not expect, and networks should handle such traffic without causing any trouble.

A traditional approach to construct a network can provide robust and stable networks, but inflexible to modify network control or management mechanisms. Originally, all network devices such as IP routers both processed packets and controlled networks only in software. Then, hard-wired engines such as ASICs that process packets at high speed became essential to handle traffic that dramatically increased. Now, almost all network devices are equipped with the hard-wired engines that process packets for low energy consumption and high port density, and software on the devices is in charge of network control including calculating routes to forward packets and installing packet forwarding rules in the hard-wired engines for packet processing. Network device vendors design and manufacture both software and hardware of the devices in accordance with specifications standardized by open community like IETF [1] or defined by them. People involved in the advancement of network technology like the vendors, network operators and researchers make an effort for a long time so that the networks are operated stably in any situation, such as maturing specifications and software by using them, and offloading more complex packet processing on hard-wired engines like FPGAs and Network Processors [9, 50] as well as ASICs for performance. From a network managers' view, however, it takes much time to introduce new control mechanisms due to such standardization and implementation process. When a new control mechanism or a modification of an existing control mechanism is required, network managers should standardize it over a long time, or ask the vendors to implement it and wait until the implementation has been completed. In addition, it sometimes needs additional time to replace or upgrade their devices.

A new approach, Software Defined Networking (SDN) [38, 62], is proposed to increase flexibility of network control mechanisms in a private network that is managed by a single administrator, such as a campus, enterprise, and data center network. SDN introduces two paradigms into network control; separation of network control and packet forwarding from network devices, and centralized network control. To decouple network control from network devices, the devices expose an API to control packet forwarding rules, rather than parameters for control software to calculate routes, to filter out packets, to authenticate and authorize hosts, etc. Control software on external computers, called controllers, centrally makes decisions how to handle packets such as forwarding to destinations or discarding, and sets forwarding rules to the devices through the API on the devices. With SDN, network managers can implement their network control software to meet their needs by themselves using the API, and centralized control makes network mangers easy to develop and deploy the software and to collect a state of their networks.

OpenFlow [54] is one of protocols that are attracted as future promising APIs on the devices. OpenFlow abstracts packet forwarding rules in a simple way; each rule, referred as a flow entry, consists of a condition to match packets specified by a tuple of header fields and their values, actions applied to packets like outputting to ports, and statistics. OpenFlow supported network devices (OpenFlow switches) store flow entries to flow tables, and report events to controllers, such as port down and unknown packets. The controllers are responsible for managing flow entries. The controllers and the switches communicate with each other through OpenFlow channels over TCP connections.

Although network managers can implement network control mechanisms with OpenFlow, issues regarding stability and robustness remain. OpenFlow switches and protocols are hard to change for network managers, thus the switches and the protocols must have mechanisms to make networks stable and robust. Such mechanisms must be designed without sacrificing flexibility of network control, and this is against the strategies in traditional networks that network devices are equipped with matured software and specialized hardware for specific processing. In addition, centralized controllers are designed and implemented not to be overloaded. Furthermore, channels between controllers and switches must be maintained to keep a network controllable, though controllers and switches are physically separated in most cases. This thesis discusses mechanisms for mitigation and avoidance of stability and robustness issues in OpenFlow networks.

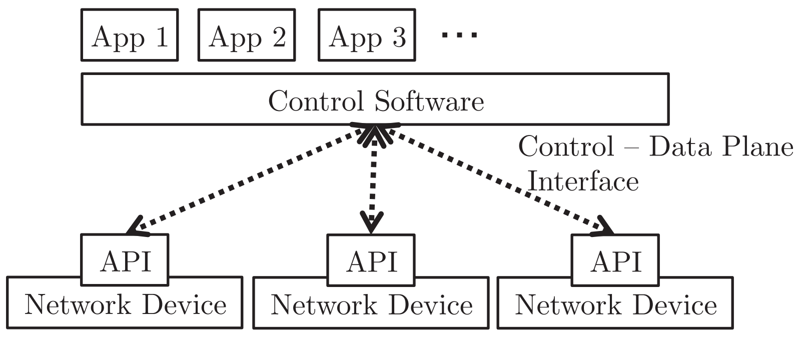


Figure 1.1: Conceptual diagram of SDN architecture

1.1 Overview of Software Defined Networking and OpenFlow

Software Defined Networking (SDN) is a new network control architecture for networks that adapt to changing needs rapidly and readily. The target of SDN is a private network where hosts are directly attached and that is managed by a single administrator, for example, a campus, enterprise or data center network.

Traditional networks cannot achieve the goal of SDN because network control software is coupled on network devices and the software is hard to change. As a result, network managers are forced to ask network device vendors to implement new network control mechanisms to meet their needs when they introduce a new network service or configuration automation mechanism to their networks.

Figure 1.1 shows a conceptual diagram of the SDN architecture. In SDN, network control software is decoupled from network devices, and the software running on other computers centrally controls all network devices in a network through open and common APIs on the devices. The role of the devices in SDN is to forward packets at high speed, and the devices are called a data plane. A system for network control including control software, applications that extend the control software, and networks connecting network devices, the control software and the applications is called a control plane. The APIs are defined for interaction between control and data planes, which is called Control - Data Plane Interface or Southbound Interface. With proper APIs to flexibly control packet forwarding at the data plane, network managers can develop control software and applications on it for their networks by themselves.

OpenFlow [54], which is endorsed by an industrial forum [2] is one of protocols for Control - Data Plane Interface, and currently attracts attention from both network man-

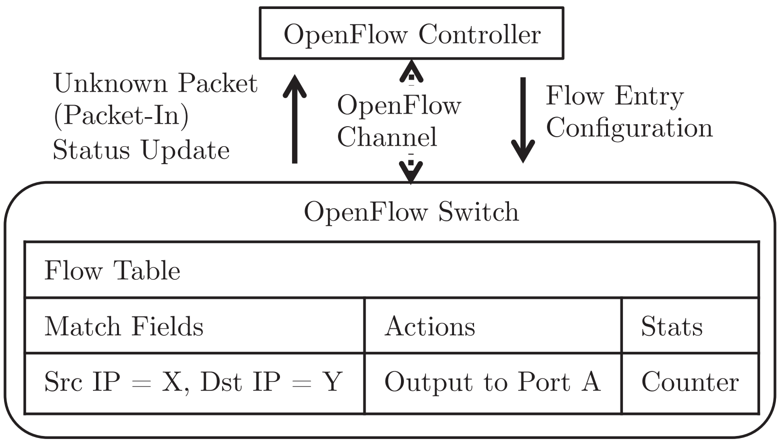


Figure 1.2: Overview of OpenFlow

agers and network device vendors. Figure 1.2 shows an overview of OpenFlow.

OpenFlow switches forward packets according to the forwarding rules (flow entries) stored in flow tables. A flow entry consists of match fields, actions, and statistics. The match fields are conditions to match packets, and are defined by a tuple of header fields and their values. Wildcards are supported in the fields. The actions are applied to packets that match the entry, such as outputting to specific ports, and rewriting header fields. The statistics include packet counter, time since an entry is installed, etc. In Fig. 1.2, OpenFlow switch has a flow entry that matches packets of X in the source IP address and Y in the destination IP address, and such packets are outputted to Port A.

The flow entries and other configurations are set by controllers. Events occurring at switches and packets that miss flow tables are reported to controllers. Messages to notify such missed packets are called Packet-In messages, and the Packet-In messages include the missed packets that cause a switch to send the messages.

1.2 Issues concerning Centralized Control

1.2.1 Performance of Controllers and Switches

OpenFlow controllers manage all OpenFlow switches in a network. This means that, more events occur at switches as the size of a network grows, and the controllers must handle more messages that notify the events. If a processing of each event causes a high load on a controller, the controller becomes busy and hard to control the network. It is also important that any change in response to the events should be applied to the network quickly so that hosts can communicate with each other smoothly.

To handle many messages without high loads on controllers, the controllers are expected to process each message very quickly. The processing usually includes time-consuming tasks such as path computation between switches. A key to decrease processing time is to reduce and optimize such time-consuming tasks.

Multicast control is one of the cases where we need to care for this point. The multicast control in a private network requires that a network update multicast trees on-demand according to a state of senders and receivers. In addition, considering typical applications of multicast like video streaming, the network should restore delivery of multicast packets quickly at the time of failures. Traditional network technologies cannot achieve both, and OpenFlow based approaches have potential to achieve.

However, controlling multicast in OpenFlow can easily increase loads on controllers. When a receiver joins in or leaves from a multicast group, the controllers should replace multicast trees to include or exclude a switch where the receiver is attached. It is hard to precompute the trees in private networks because a set of receivers is frequently updated due to frequent connection and disconnection of hosts. Thus, the controllers should install trees on demand, but frequent tree computation increases loads on the controllers.

Furthermore, recovery of multicast trees from failures is also problematic. Considering applications of multicast such as video streaming, delivery of multicast packets should be restored immediately at the time of failures. The controllers should reroute each multicast tree, for example by recomputing all the trees, but it takes much time. In addition, modification of flow entries to the switches should be minimized because such modification is slow especially in hardware switches, which include hard-wired engines specialized to packet forwarding. Huang et al. [34] reported that hardware switches can modify flow entries at only 40 flows per second, that is, modification of one flow entry takes around 25 milliseconds on average. This slow performance should increase the time until delivery of multicast packets is restored, especially when many flow entries need to be modified in some switches. The controllers should also take care of this point.

In summary, recomputation and reinstallation are significant burdens to controllers and switches, and we need a mechanism to alleviate the burdens.

1.2.2 Failure Detection of OpenFlow Channels

To provide network connectivity, networks should update their configuration quickly according to their state. When a fault is detected at a switch, the event should be notified to controllers immediately, and the controllers should modify flow entries to reroute packets that get through the fault point. When a user requests the controllers to change a configuration of the network such as setting up a new path and updating packet filtering rules, the controllers should update flow entries quickly.

To exchange messages between controllers and switches in a timely manner, OpenFlow channels should be maintained. In the case where a channel is unavailable, a controller and a switch at both ends of the channel should detect it immediately, especially when some messages are sent to the other end. Then, they should take appropriate actions such as avoiding failed channels for transmission of messages or rerouting packets so that packets do not get through uncontrollable switches.

A simple way to detect a channel failure quickly is to exchange keep-alive messages in a short interval, but this approach is inappropriate due to centralized controllers. Each controller should check the state of a channel to each switch. Although other messages than keep-alive ones should be handled at one of controllers and controllers can distribute loads to handle the messages, the keep-alive messages should be exchanged at every channel and load-balancing of the keep-alive messages is impossible. Therefore, the controllers need to process many keep-alive messages. In addition, message processing performance in a controller is not necessarily high. According to a report by Shalimov et al. [71], the performance varies according to controller frameworks. In the context of SDN and OpenFlow, the message processing performance is not the only measure of the controller frameworks; other measures such as productivity of controllers are also important for development and deployment of new services rapidly. Therefore, it is desirable that the controllers with low message processing performance can also detect failures quickly.

Fault sometimes occur both in the data plane and in the control plane at the same time, including a control network that connects controllers and switches. In this case, a message to notify a fault must arrive at one of controllers, but delivery of the message will be delayed due to the fault in the control plane. Ideally, such situation merely occur because the data plane and the control plane are totally separated, and faults in the data plane and in the control plane occur independently. In practice, parts of the data plane and the control plane often share the same infrastructure, such as physical devices, cables, power sources, and a route of cables. When a cause of faults is in the physical infrastructure, both the data plane and the control plane will be affected.

In summary, we need a mechanism to detect a failure of OpenFlow channels quickly with a small amount of messages, especially when faults occur both in the data plane and in the control plane at the same time.

1.3 An Issue that Concerns Decoupling Network Control from Network Devices

1.3.1 Protection of Switches against Many Unknown Packets

Hosts sometimes start to send unexpected traffic without advance notification to a network. For instance, a host starts to send a large amount of traffic such as video streams. Hosts infected by malware may send malicious traffic like a Denial of Service (DoS) attack and a port scan.

The network should handle such unexpected traffic without disturbing traffic sent by other hosts. If controllers were able to preinstall flow entries in switches to process such traffic, the switches could handle such traffic by themselves. In practice, there are many cases where the controllers cannot know such traffic and install the flow entries in advance. One of examples is to deliver multicast packets. When the controllers do not know where a sender is connected, the controllers cannot compute a multicast tree and install flow entries to deliver such packets. In order to learn locations of senders, the controllers need to set switches to send unknown multicast packets to the controllers as Packet-In messages.

Although the controllers try to install flow entries immediately after necessary information to install the flow entries is gathered, there is small delay until the flow entries are installed after a host has started to send the unexpected traffic. During this delay, a switch continues to send Packet-In messages generated by the unexpected traffic. When a large amount of the traffic is sent as Packet-In messages, a CPU on the switch would be overloaded because Packet-In messages are generated at the CPU on the switch, which is known to have poor performance. As a result, control of the switch becomes unstable, such as delay of processing messages from the controllers, and discard of packets sent by hosts other than ones sending a large amount of the unexpected traffic.

Traditional networks tackle this problem by offloading packet processing to hard-wired engines as much as possible, such as to ASICs, Network Processors, and FPGAs [9, 50]. In the context of SDN and OpenFlow, this approach is inappropriate because use of such engines lacks a certain amount of flexibility of network control that OpenFlow tries to provide. Some of the engines are designed for specific purposes, and network managers cannot use them unless intent of network managers matches specifications of the engines. Others provide a certain amount of flexibility, but network managers should learn different programming models than those for development of controllers. In addition, this approach cannot relieve the effect in cases where the controllers need to learn a state of a network from packets and execute something such as detecting new hosts and setting up a new path on a network.

In summary, we need a mechanism to effectively filter out Packet-In messages only by switches while preserving flexibility of network control that OpenFlow tries to provide as much as possible.

1.4 Organization of This Thesis

In this thesis, we propose mechanisms to improve the robustness and the stability of OpenFlow networks.

Chapter 2 introduces architecture of Software-Defined Networking and OpenFlow, as well as works toward stable and robust OpenFlow networks.

Chapter 3 addresses the issue of multicast control as described in Sec. 1.2.1. The controllers handling multicast should: (1) update multicast trees quickly so that each tree covers the latest set of receivers, such as adding a receiver to a tree and removing a receiver from the tree, (2) recover the trees quickly when a failure is detected on a network. In order to make the changes effective in switches quickly, the controllers should also make it smaller the number of flow entries that should be updated.

In our proposed mechanism, the controllers precompute a tree covering all switches where receivers may be connected, and install a part of the tree in switches that is needed to deliver multicast packets to receivers. When a receiver is added or removed, the controllers use a tree that has been precomputed and update a part of the tree installed in switches, instead of computing a new tree. The evaluation shows that our proposed mechanism can decrease the controller's processing time to add and remove a receiver, except the cases where a multicast group is created or deleted.

To restore delivery of multicast packets quickly at the time of failures, the controllers prepare redundant trees in advance, and use them for restoration. The controllers compute two or more redundant trees for each multicast group, all of which covers all switches where receivers may be connected. When a set of receivers is updated, the controllers modify the part of each tree that is installed in switches. To avoid duplication of packets at receivers, a switch at the root of the tree chooses a tree that packets traverse. When a failure is detected, the controllers choose a tree that is not affected by the failure, and install a flow entry in the switch at the root of the tree so that a packet traverses one of unaffected trees. The evaluation shows that our proposed mechanism, precomputing and preinstalling trees, decreases the duration when packets do not arrive at receivers at the time of failures.

Chapter 4 addresses the issue of an OpenFlow channel as described in Sec. 1.2.2. OpenFlow channels are important for both switches and controllers. If a channel is unreachable, both ends must detect it immediately; otherwise a part of the network remains uncontrollable. In order to avoid an increase of messages handled by controllers, a small number of messages to check the state of a channel should be exchanged between both ends.

Our proposed mechanism checks arrival of messages instead of checking the channels are always available. When no message is generated at neither side, switches need not to communicate with the controllers, nor to change their configurations including flow tables.

Our proposed mechanism works under assumption that two or more controllers run for performance and redundancy. A switch sends keep-alive messages to all controllers just after sending an important message, such as a notification of port down. A controller shares arrival of messages with other controllers, and detects inconsistency of arrival, that is, some controllers receive a message but others do not. The evaluation shows that our proposed mechanism can detect fast that an OpenFlow channel is unreachable when a message is generated and sent, and overhead in latency introduced by our proposed mechanism is negligible.

Chapter 5 addresses the issue of a switch as described in Sec. 1.3.1. Switches must be in stable operation even when many packets are handled at CPUs on the switches, which is mainly due to missing flow entries or being specified by flow entries. At the same time, the mechanisms to address this problem should not sacrifice flexibility of network control provided by OpenFlow as possible. One simple mechanism of filtering out such packets is to restrict the rate of packets handled at the CPU of the switch in total, but this mechanism has a significant drawback: a small number of important packets for network control are mostly discarded. Another mechanism is to record all flows in switches and pass through only packets whose flows are not recorded, but it is impractical to record all flows because it consumes too much memory by many flows that exist in a network, especially under attacks such as a port scan.

Our proposed mechanism is also based on an idea that a switch records flows, and we add a mechanism to specify header fields that should be recorded. Important packets for network control are ones from which the controllers learn data, and one packet is necessary to learn the same data. For instance, when Ethernet addresses and IP addresses are used to detect new hosts and install new flow entries, the controllers need one of packets that have the same addresses, and other packets are less important and can be discarded.

In our proposed mechanism, the controllers specify header fields that should be recorded to switches as pending flow rules before the switches start to process packets. Pending flow rules also include header fields and their values as a condition to match packets. When a packet misses the flow tables and matches the condition of a pending flow rule, a switch records the values of the header fields specified by the rule to a table called a pending flow table, and sends the packet to the controllers as a Packet-In message. Subsequent packets that have the same values with recorded ones in the pending flow tables are filtered out, or processed by other mechanisms that do not use a CPU of the switch. The evaluation shows that our proposed mechanism decreases CPU utilization of a switch as well as important packets surely arrive at the controllers.

Chapter 6 concludes this thesis.

Chapter 2

Software Defined Networking and OpenFlow

2.1 Software Defined Networking

Software Defined Networking is one of new network control architecture for flexible network control by network managers.

Traditional networks (Fig. 2.1) are inflexible in network control for network managers because control software is tightly coupled with network devices. A network device mainly consists of two parts: a data plane for processing packets at high speed and control software (a control plane) for controlling how to forward packets. Network device vendors rarely disclose interfaces to the data plane, and it is difficult for others to develop and deploy their control software. As a result, the vendors implement many control mechanisms in one control software, and network managers are forced to choose and use some of the mechanisms that are suited to their networks. When a new network control mechanism is required, network managers must standardize it or negotiate with the vendors for implementation.

In addition, it is hard for network managers to implement a network-wide policy and to confirm that the policy is enforced because a state of a network and configurations of network devices are distributed over the devices. Network managers could implement software to automatically configure each device, but control software developed by the vendors does not necessarily have an appropriate interface to configure the devices by the software, nor provide enough state of the devices via its configuration interface.

To enable network managers to develop and deploy their network control mechanisms

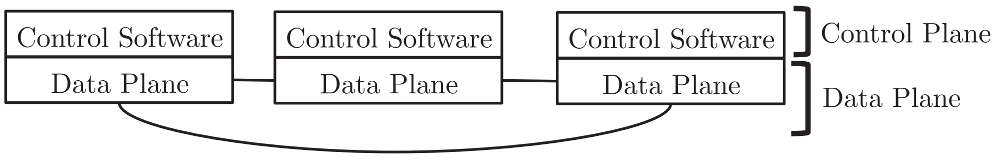


Figure 2.1: Traditional network architecture

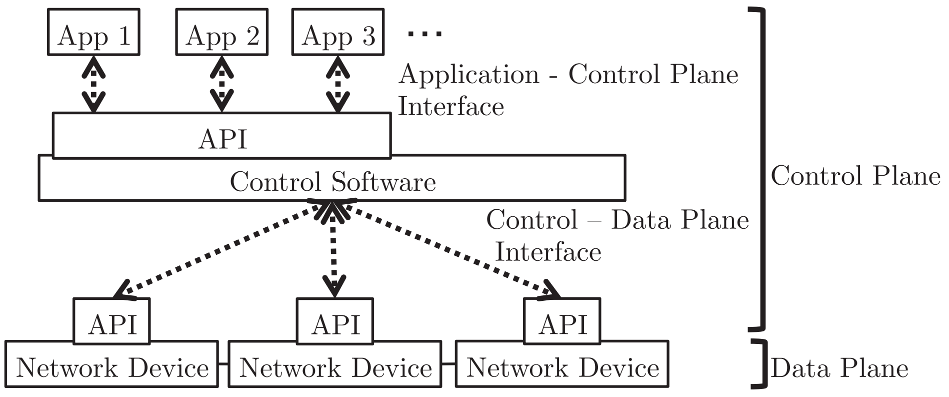


Figure 2.2: SDN architecture

by themselves, Software Defined Networking (Fig. 2.2) tries to decouple control software from network devices and to control a network centrally. Network managers run their control software on external computers called controllers, and the controllers collect a state of an entire network and configure each device over an open and common API for controlling packet forwarding behavior in each device. This API is called Control - Data Plane Interface or Southbound Interface. The devices forward packets at high speed with hard-wired engines such as ASICs or software optimized for packet processing, and packet forwarding rules are installed via the API. The devices also have lightweight software for communication with the controllers. The control software is mainly responsible for maintaining channels to network devices, executing basic control functions like discovering topology, abstracting network topology for applications, enforcing access control from applications to switches, etc.

The control software often provides an API for other software called applications so that the control software can be extended easily, and the API is called Application - Control Plane Interface or Northbound Interface. Core parts of network control mechanisms and policies, such as path computation algorithms and access control policy, are mainly implemented into the applications. The control software merges processing results from multiple applications, and installs them in switches.

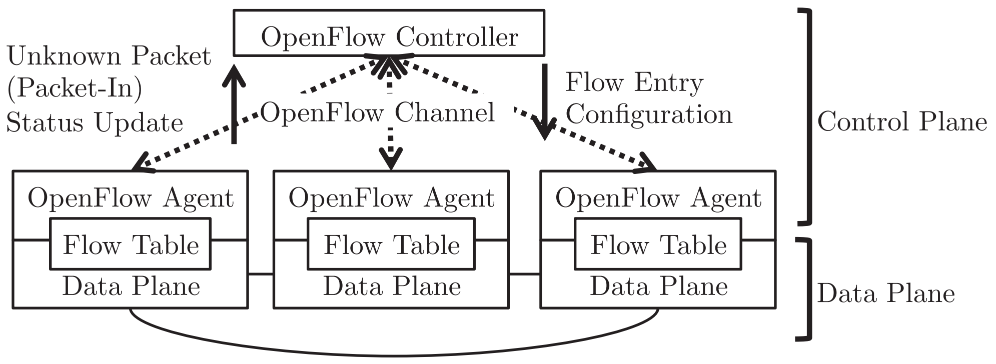


Figure 2.3: OpenFlow architecture

2.2 OpenFlow

OpenFlow [54] attracts attention from both network managers and network device vendors as a southbound interface. Figure 2.3 shows architecture of OpenFlow networks. Each network device (OpenFlow switch) stores packet forwarding rules in Flow Tables, and each entry in the flow tables is called a flow entry. The switches forward packets according to flow entries at high speed in the data plane. OpenFlow controllers install the flow entries in the switches through OpenFlow channels over TCP. Other configurations are also sent from the controllers through the channels. An OpenFlow agent, which is lightweight software on each switch, is responsible for maintaining OpenFlow channels, carrying out state changes indicated by messages from the controllers such as translating flow entries into a format that the data plane uses, and generating messages for notification of state changes like port up and down. If a switch is configured to send packets that miss flow tables to the controllers, the agent generates a Packet-In message including a missed packet and sends it to the controllers.

OpenFlow channels are usually established over a dedicated control network, which is separated from a network that OpenFlow controls (Out-of-band control). OpenFlow specifications also define that OpenFlow channels can be established over the same network that OpenFlow controls (In-band control).

Table 2.1 shows an example of a flow table. The match fields are a condition to match packets, and consist of priority, an input port, and a tuple of header fields. Wildcards are supported in all fields, and the flow entry that has the highest priority is selected when multiple flow entries match a packet. The actions list operations applied to packets that match the entry. The actions include outputting to specific ports or one of ports selected according to a state of a switch like one of ports alive (Fast Failover Group) and round

Match Fields										Actions	Statistics
Priority	In Port	Frame Type	Src Eth Address	Dest Eth Address	Src IP Address	Dest IP Address	Protocol	Src Port	Dest Port		
5	1	0x0800 (IP)	A	B	C	D	6 (TCP)	*	80	Output to Port 2	W bytes, etc
4	1	*	A	B	*	*	*	*	*	Output to Port 3	X bytes, etc
4	2	*	*	E	*	F	*	*	*	Dest Eth Addr to G, Output to Port 3	Y bytes, etc
0	*	*	*	*	*	*	*	*	*	Send to Controller	Z bytes, etc

Table 2.1: An example of a flow table

robin (Select Group), rewriting some header fields, discarding, sending to controllers as Packet-In messages, restricting the rate of packets, etc. The statistics include various statistical data regarding the flow entry, such as the number of packets matched, the sum of length of packets matched, duration until the flow entry expires, etc.

There are two types of methods to install flow entries. One is proactive setup, which is to install flow entries before packets matched with the entries arrive at switches. All necessary data is provided by external systems like cloud controllers. The other is reactive setup. In reactive setup, controllers install flow entries after a Packet-In message including a packet arrives at controllers, and the response time to install flow entries since a packet arrives at a switch is important for communications between hosts. The proactive setup is preferred in terms of loads on controllers because the controllers do not need to process anything per packet, and the reactive setup is better in terms of spaces for storing flow entries in switches because only necessary flow entries would be installed. In both methods, the controllers cannot install flow entries before necessary information is gathered. The controllers learn some information from packets, and flow entries that the controllers need such data to generate must be installed in a reactive way. Typical examples in this case include learning Ethernet addresses, detecting sources of multicast traffic, handling ARP and broadcast packets.

OpenFlow supports that a switch establishes multiple OpenFlow channels to different controllers (Fig. 2.4). A switch has a mode for each channel, master, slave, or equal, and the mode is configured by controllers. Only one master channel is allowed in one switch. A controller can both read and write a state and configuration of the switch only via the master channel, and other channels are forced to be in the slave mode, which means that

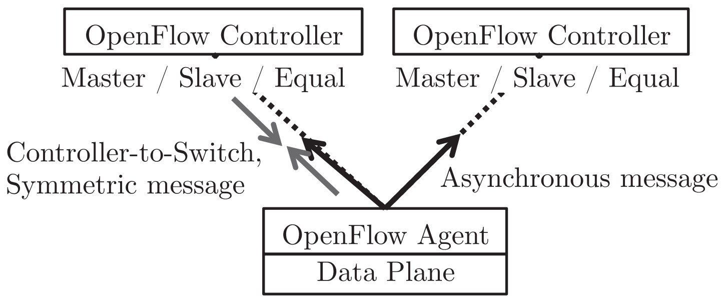


Figure 2.4: Multiple controller support in OpenFlow

controllers can only read a state and configuration of the switch. To read and write via multiple channels at the same time, the modes should be set to equal.

There are three types in OpenFlow messages: controller-to-switch, asynchronous, and symmetric. The controller-to-switch messages are used to get and set a state of a switch by a controller, and the controller expects that the switch respond to some of these messages. A switch informs controllers of a state change with an asynchronous message. Although the OpenFlow protocol does not require a controller to respond to the asynchronous messages, the controller may update a state of the switch in response to some asynchronous messages like port status messages. The symmetric messages are request and reply messages, and request messages can be sent not only from controllers but also from switches. A keep-alive message (echo request/reply) is an example of the symmetric messages.

Figure 2.4 shows how these types of messages are transferred over multiple channels. The asynchronous messages are sent over all channels other than ones where the messages are set to be filtered out, and other messages are sent over one of channels. A reply message is sent over the channel where a request has been received.

2.3 Research on SDN and OpenFlow from the Viewpoint of Stability and Robustness

Figure 2.5 summarizes research topics related to SDN and OpenFlow, and their stability and robustness issues.

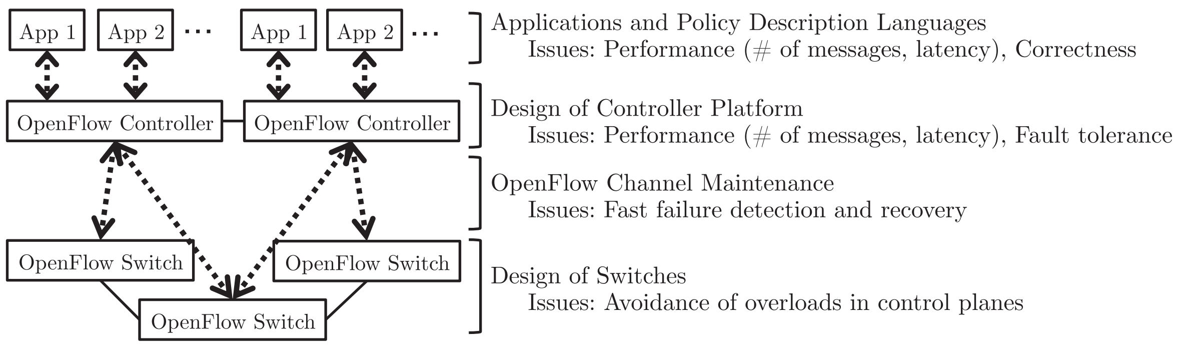

App 1
App 2
App 1
App 2
OpenFlow Controller
OpenFlow Controller
OpenFlow Switch
OpenFlow Switch
OpenFlow Switch
Applications and Policy Description Languages
Issues: Performance (# of messages, latency), Correctness
Design of Controller Platform
Issues: Performance (# of messages, latency), Fault tolerance
OpenFlow Channel Maintenance
Issues: Fast failure detection and recovery
Design of Switches
Issues: Avoidance of overloads in control planes

Figure 2.5: Overview of research topics on SDN and OpenFlow and their stability and robustness issues

Applications and Policy Description Languages

One of research topics in the application area is to explore new network control mechanisms and network services that OpenFlow enables. For example, Zou et al. [96] have proposed an authentication and authorization mechanism for IP multicast. This mechanism installs flow entries to deliver multicast packets only to authorized hosts. Other examples spread over many fields, like network management (access control policy [12], mobility support for virtual machines among multiple data centers [31]), network security (DDoS attack detection [8], cooperation with middleboxes like IDS [14]), and Wide Area Networks (WAN) (application specific traffic engineering and shared middleboxes in Internet Exchange Points (IXP) [28], traffic management for efficient use of circuits [35]).

Important issues related to stability and robustness are lightweight processing in controllers to avoid being overloaded, including decreasing messages handled at controllers, and to help testing and debugging applications to reduce critical bugs.

One approach is to do it in application or environment specific ways. In terms of performance, Sharma et al. [74] show a path restoration mechanism without processing at controllers when a fault is detected. In multicast, controllers need to modify existing multicast trees and to restore delivery of multicast packets from failures quickly. CastFlow [53] has proposed a method to handle modification of multicast trees by precomputing multicast trees from possible senders to receivers and storing them as a list of links included in the trees. Li et al. [48] have proposed a multicast management scheme in datacenter topologies. In terms of correctness, many tools to help to find a cause of bugs are proposed. NetSight [30] is a tool to capture and store all packets to find packets that cause errors. Zeng et al. [95] have proposed a method to generate test packets to find errors. With this approach, the controllers can improve their control applications in detail, but the control applications cannot be used in general; if a use case changes, network managers should use other control applications or adapt them to a new use case.

Rather than developing network control mechanisms in application specific ways, more general ways are also studied, such as description of network policy in a higher level than OpenFlow. For example, intent of network managers or applications is expressed with some domain specific languages, and a run-time system translates the intent into OpenFlow semantics. Frenetic [26] and its python version, Pyretic [58], are examples of such languages, although they are focused on modularity of applications. SDX [28], which introduces SDN in IXP, uses Pyretic to express policies of each participant in IXP, and proposes a method to optimize flow entries installed. FatTire [68] is a language for describing paths and fault-tolerance requirements explicitly. In terms of correctness, existing software testing methods can be used to test applications, such as model checking with symbolic execution [10]. The run-time system can apply these techniques irrespective of use cases, although performance may be below than in application specific ways because of pursuing generality.

Controller Platform

Controller platform intermediates switches and applications, and has various roles for supporting application development and management of applications. Research and software try to abstract and sort out the roles with appropriate APIs to applications.

One of important roles is to provide basic functions that are common to many applications, such as managing OpenFlow channels and finding network topology. NOX [27] and its Python version, POX, show the first prototype for this role. Trema [84] is another OpenFlow controller framework, and helps an entire cycle of development of controllers including software tests and debugging in addition to the basic functions. Floodlight [24] and Ryu [69] provides more functions than basic ones, such as packet parser, interpretation of existing protocols for interoperability with other networks, MAC learning switch that forward packets according to Ethernet addresses, and REST API. Network managers choose functions necessary to their networks, or implement new ones if existing ones do not meet their requirements.

Another important role is to run and manage multiple applications from the point of a network-wide view, such as enforcing access control, combining output from applications into one, and translating physical network topology to logical topology that is appropriate for applications. FlowVisor [75] separates flowspace as slices, which is a collection of packet headers and range of their values, and allocates each slice to individual applications. With FlowVisor, many applications can run independently in a physical OpenFlow network. OpenVirteX [5], RISE [37] and ODENOS [43] also provide a function to convert physical network topology to virtual ones that are appropriate for each application, as well as to slice a network. It is important not only to isolate each application but also to combine output of several applications because one application is merely enough to manage a network. CoVisor [36] enables to combine the output into one by representing relationship between applications with operators proposed.

Regarding stability and robustness, controller platform must handle many messages with low latency to avoid being overloaded and disruption of communication between hosts, and must have fault tolerance. The controller platform is evaluated in various perspectives, such as the maximum number of messages processed for performance [83] and line of codes for productivity. Cbench [76] is one of famous benchmark tools of controllers to measure the number of messages that controllers can process and latency that controllers respond to a message.

To construct high performance and fault tolerant controllers, distributed controller platforms are proposed such as ONIX [44], HyperFlow [82] and ONOS [6]. These controllers store data to a distributed data store, and it is easy to deploy additional controllers when necessary. In addition, how to deploy multiple physical controllers is also an important issue when thinking of fault tolerance. Heller et al. [33] have investigated how many and where controllers should be deployed, considering latency and redundancy with topologies of WAN indexed in Internet Topology Zoo [42].

OpenFlow Channel Maintenance

OpenFlow channels are very important for OpenFlow networks. If channels are failed, networks cannot be controlled.

OpenFlow channels can be established in out-of-band control if controllers and switches can communicate with each other. On the other hand, in in-band control, switches and controllers should setup paths for the channels before establishing the channels. Sharma et al. [72] have proposed a procedure of automatically establishing OpenFlow channels.

Important issues in OpenFlow channels is to detect a failure immediately when a channel is unavailable so that controllers and switches can take appropriate actions such as recovering channels quickly and isolating uncontrollable switches. Sharma et al. [73]

have showed a fast restoration mechanism of OpenFlow channels in in-band control. Other technologies used in traditional networks such as Bidirectional Forwarding Detection (BFD) [39] are also used to detect failures below the network layer.

OpenFlow Switch

OpenFlow switches can apply a limited number of types of actions to packets by themselves, and the actions are defined by OpenFlow Switch Specifications [64]. Actions that are not supported by OpenFlow switches are applied per packet at controllers.

The switches must not send many Packet-In messages to the controllers because CPUs on switches is easily overloaded by generating and sending many Packet-In messages, and a bandwidth of a control path between the data plane and the control plane inside a switch is saturated [16]. Under such situations, it is hard for controllers to operate switches.

Many extensions are proposed as follows to decrease the number of Packet-In messages, that is, to process packets without consulting with the controllers. For example, DevoFlow [16] have proposed an action of cloning an flow entry that is matched with a packet. This action is intended for helping detection and rerouting of elephant flows that have significantly more traffic than other flows. FleXam [78] is a packet sampling extension for security applications. With FleXam, a switch chooses some flows and send some consecutive packets of the selected flows to controllers.

DIFANE [93] has capability to distribute packet forwarding rules from some switches (called authority switches) to others for reducing the work of controllers to install flow entries. OpenState [7] has a similar mechanism of controlling a network with switch-to-switch communications triggered by some packets, and Capone et al. [11] have proposed a mechanism to restore paths without consulting with controllers when a fault is detected.

Protection of OpenFlow switches is also studied. AVANT-GUARD [77] is a mechanism to protect the switches from TCP SYN Flood attacks. Luo et al. [51] have proposed a mechanism to filter out Packet-In messages based on (source, destination) address pairs recorded when sending Packet-In messages.

Chapter 3

Management of Multicast Trees in OpenFlow Controllers

3.1 Introduction

Multicast is an important communication tool to distribute data to a number of hosts simultaneously with lower bandwidth. Examples of multicast applications include video streaming, replication of large data, and distribution of news headlines to many hosts.

As various hosts are connected to IP networks, use of multicast becomes more beneficial in private networks such as campus and enterprise networks, as well as in carrier networks where traditional approaches are targeted. For example, multicast is an efficient way to distribute data to many monitors installed inside buildings and public areas, and to send video to many receivers for teleconferences and remote lectures.

To use multicast in these scenarios, multicast communication must be highly reliable and data loss should be minimized while delivering data in the network. In the network layer, less packet-loss packet delivery is important even at the time of failures, because packet loss imposes much overhead on a sender and many receivers for data recovery, or degrades quality of services provided by applications. For example in video streaming, when frames are lost, video and audio are disturbed for a longer time than the duration of losing frames, and the distortion gets worse when frames are lost bursty [49] such as by failures. Such distortion causes obstruction in teleconferences and remote lectures.

In addition, private networks should handle many changes of multicast group membership because receivers are frequently added or removed such as by turning hosts on and off like monitors, and a set of receivers is not fixed until a teleconference or a remote lecture starts. It is hard to preallocate network resources to deliver multicast packets such as packet forwarding rules, and networks should quickly and dynamically modify multicast trees according to changes of multicast group membership.

Many algorithms are proposed to tackle this problem, such as computing redundant trees for failure recovery [55][91] and computing trees faster to setup multicast trees quickly [80][79]. This problem cannot be solved only by tree computation algorithms. We also need to take into account a system to implement multicast trees in a network using output of these algorithms so that the network can make use of the benefits of these algorithms effectively.

Point to Multipoint (P2MP) MPLS, which is used for reliable delivery of multicast packets in carrier networks, is not designed for use in such frequent changes of multicast group membership [92], and has much overhead on routers by the tree setup procedure in RSVP-TE [4]. A tree manager in P2MP MPLS is proposed to use more flexible tree management policies by computing multicast trees centrally [15]. Since the tree manager recalculates and replaces the trees every time that multicast group membership is changed, the manager cannot handle many changes of multicast group membership because tree calculation takes much time and overhead on routers by tree setup procedures in RSVP-TE remains.

Local repair based mechanisms such as MPLS Fast Reroute [66] can reduce packet loss at the time of a failure by rerouting packets at a switch where a failure is detected, but such mechanisms sometimes force a network to use non-optimal trees as backup. The private networks usually have three layered tree topology (core - aggregation - edge), and redundant routers and switches are deployed with redundant links. For example, edge switches provide links to each room in a floor, and have links to two or more aggregation switches that connect edge switches in a building. Each aggregation switch is connected to two or more core switches that connect aggregation switches on campus. Thus, several optimal multicast trees that can cover different core and aggregation switches are available in many cases, and use of such trees as backup is preferred in terms of bandwidth utilization, latency, etc. Therefore, a multicast management scheme that support both frequent changes of multicast group membership and fast restoration of delivery of multicast packets, or fast failure recovery, using optimal backup trees is expected in private networks, and traditional network technology cannot meet both at the same time.

OpenFlow based multicast control schemes have been proposed [53] [94] [96] to support both frequent changes of multicast group membership and fast failure recovery. A basic concept of these works is to calculate a path or a tree in advance to reduce processing time to modify the trees by changes of multicast group membership or calculate them using powerful computing power in OpenFlow controllers, then to modify flow entries in OpenFlow switches.

In terms of fast failure recovery, the controllers should also take account of the time to modify flow entries in switches, especially in hardware switches. Before flow entries are installed to hardware flow tables using TCAM (ternary content addressable memory) in the ASICs, switch firmware will execute various tasks with their CPU that has poor performance, such as translating flow entries in OpenFlow format into those in format appropriate for design of the ASICs, optimizing entries stored in TCAM, or delaying installation of flow entries for batch processing, etc. Therefore, modification of flow entries in hardware switches is usually not fast. In fact, Huang et al. [34] have reported that installation of one flow entry takes around 25 msec on average in hardware switches, which is much longer than in software switches. When one failure affects several multicast groups, it may take several hundred milliseconds until all multicast trees are recovered, and large packet loss may be observed at receivers. This results in dropping tens of frames in video, for example. The previous work for multicast control schemes using OpenFlow does not consider this performance problem in switches.

In this chapter, we propose a multicast tree management method in an OpenFlow controller (a multicast controller) that provides both fast failure recovery for reliable delivery of multicast packets and quick updates of multicast trees by changes of multicast group membership. A multicast controller is responsible for collecting multicast group membership from switches and management of multicast trees including computation, setup and modification of flow entries installed in switches, and failure recovery.

To shorten the time to process modification of multicast trees by changes of multicast group membership, the multicast controller uses a preplanned approach. When a multicast group appears in the network, which means that the controller learns a sender's IP address and a root switch where a sender is connected and finds one or more receivers in the new group, the controller computes and stores two or more trees for the new group. The trees cover all switches where receivers may be connected, and the trees share less common edges and nodes.

When the controller handles an update of multicast group membership, the controller lists switches whose flow entries must be updated by following each stored tree for the group from the leaf switch where the membership is updated, instead of recalculating trees. The controller stores a state of nodes and edges in each tree to show whether flow entries to forward packets through corresponding switches and links are installed. The controller changes these states while following the trees if necessary, and stops following the tree at the node whose state does not have to be changed. Then, the controller updates flow entries in switches that correspond to nodes the controller has visited while following the trees.

To shorten the time for failure recovery, the controller installs multiple trees in switches for one multicast group at the same time. To avoid duplication of multicast packets at receivers, the controller assigns a unique ID within a multicast group to each tree. The tree ID for packet delivery is embedded into packets at the root switch of the tree. The switches other than the root switch have one flow entry per tree, and forward packets according to the tree ID. When a failure is detected on a network, for each group, the controller checks whether the failed switch or link is included in the tree used for packet delivery. If yes, the controller finds a tree unaffected by the failure from trees that the controller has already installed in switches, and changes the tree ID embedded at the root switch to the new one. In this way, the controller needs to modify only one flow entry in one switch per multicast group, instead of modifying flow entries in many switches to replace trees.

We have implemented our prototype controller using C and Trema [84]. For each multicast group, we use two trees that share fewer edges. We embed a tree ID into a source MAC address. Our evaluation shows that our proposed method can handle the addition and the removal of receivers in a short time, and restore delivery of multicast packets faster than without our proposed method at the time of link failures when we use hardware switches.

This chapter is organized as follows. We describe related work in Sec. 3.2. We explain our proposed method in Sec. 3.3 and our prototype controller in Sec. 3.4. In Sec. 3.5, we present the results of evaluation experiments. We discuss the evaluation results and our proposed method in Sec. 3.6. and give a concluding remark in Sec. 3.7.

3.2 Related Work

3.2.1 Control and Packet Delivery of Multicast in Traditional Networks

In current IP multicast, a host multicasts or broadcasts IGMP [23] messages to join in or leave from multicast groups, and IP routers find receivers by monitoring IGMP messages. Multicast routing is controlled by PIM-SM [21], PIM-DM [3], DVMRP [87] or MOSPF [59]. PIM-SM, DVMRP, and PIM-DM cannot provide fast failure recovery. PIM-SM and DVMRP construct multicast trees using unicast routes, and they cannot reconstruct multicast trees until unicast routes are stabilized when a failure occurs. PIM-DM periodically updates multicast trees by flooding multicast packets and pruning the trees, and the trees are not reconstructed until next periodic flooding occurs. MOSPF distributes multicast group membership information to all routers to compute multicast trees. This behavior is not scalable to changes of multicast group membership.

Point to Multipoint (P2MP) MPLS [92] provides reliable delivery of multicast packets with fast reroute and bandwidth guarantees using MPLS mechanisms for carrier networks. When a failure occurs, the fast reroute function in P2MP MPLS reroutes the traffic at a router where a failure is detected (local repair) to reduce packet loss. Cui et al. [15] proposed to use P2MP MPLS to implement Aggregated Multicast [22] in IP networks, and introduced a tree manager to manage multicast trees such as mapping trees to multicast groups.

P2MP MPLS based solutions are unsuitable when multicast group membership is frequently changed. P2MP MPLS assumes that the membership is rarely changed [92]. To modify a multicast tree in P2MP MPLS, RSVP-TE [4], which is used to setup multicast trees in P2MP MPLS, requires the root router of the tree send a path setup message to the leaf routers, and routers other than the root router must send a response to their upstream routers. Thus, all routers should process many messages as the membership is frequently updated. A multicast extension to LDP (mLDP) [90] may also be used for P2MP MPLS, but mLDP does not provide fast reroute functionality.

There is a great deal of research on reducing data loss in an application layer. RFC3048 [89] recommends some mechanisms like NACK based packet loss detection and recovery, and FEC coding to recover data lost by a few packets. Hasegawa et al. [32] proposed a system to multicast HD quality videos with non-stop service availability in carrier networks by putting buffers in backup servers near receivers and constructing backup trees controlled by separate processes from main trees.

3.2.2 Algorithms to Compute Multicast Trees

A shortest path tree computed by Dijkstra's [18] algorithm is often used in unicast and multicast routing. In some cases, other metrics are important such as total costs and redundancy, and the algorithms are proposed to optimize trees based on such metrics. Médard et al. [55] showed an algorithm to compute redundant trees in arbitrary vertex-redundant or edge redundant graphs. Xue et al. [91] extended Médard's algorithms to compute redundant trees considering costs. Mochizuki et al. [57] showed an algorithm to compute a tree minimizing the number of links included in the tree. Li et al. [47] proposed methods to compute efficient P2MP backup trees in MPLS networks. Our proposed method can use any algorithm to compute trees including above, and can suppress the execution of such algorithms and the number of flow entries the multicast controller should modify in given trees.

Some tree precomputation or caching algorithms are also proposed to compute multicast trees faster. Their purpose is to balance the storage overhead by storing trees precomputed and the time to compute trees [85]. Siew et al. [80] proposed an algorithm to compute a tree by connecting cached trees. Siachalou et al. [79] showed an algorithm to compute a tree constrained by bandwidth. These proposals only consider the algorithms, but processing time for modification of trees, including modifying flow entries, is also an important metric in multicast. In addition, the storage overhead in OpenFlow controllers is negligible because recent PCs have a large size of RAM.

3.2.3 Multicast and Failure Recovery in OpenFlow

There are several works to manage multicast trees using OpenFlow. OFM [94] proposed a mechanism to manage multicast in OpenFlow networks using multiple controllers that control different parts of a network. CastFlow [53] proposed to precompute multicast trees from possible senders to receivers, and to store a list of links included in the trees for fast processing of changes of the membership. Zou et al. [96] used OpenFlow to authenticate receivers when joining in a multicast group. These schemes provide how to manage multicast trees in the controller, but they do not consider how to setup multicast trees to switches in a short time, which is also an important aspect for failure recovery in OpenFlow networks because of slow flow entry setup performance in hardware switches.

Huang et al. [34] reported that one hardware switch took 25 msec on average to modify a flow entry. If a single failure affects multicast trees of several groups, it may take hundreds of milliseconds until the trees used by all multicast groups are recovered, which is much longer than traditional approaches like MPLS fast reroute.

Li et al. [48] proposed to manage multicast groups separately in data centers, but their work can be applied only to multi-rooted tree networks. Capone et al. [11] proposed an algorithm to reroute multicast traffic with zero packet loss by their own extension called OpenState. Although their mechanism cannot work without modification of switches to OpenState capable ones, their algorithm can be used to compute multicast trees in our proposed mechanism. Gyllstrom et al. [29] proposed a similar multicast tree recovery scheme with our proposed mechanism, but their work cannot directly applied to private networks where the membership is frequently changed because they implicitly assumed that a set of receivers is static.

In terms of failure recovery in OpenFlow networks, OpenFlow 1.1 [61] and later define a Fast Failover Group feature to implement local repair. The fast failover group feature is used for rerouting of paths faster [68], but local repair using the fast failover group feature is not suitable for failure recovery of multicast trees. Controllers should prepare for backup multicast trees per switch, and switches should have flow entries for each backup tree. This dramatically increases the number of flow entries in switches. In addition, backup multicast trees created by local repair are sometimes inefficient in terms of bandwidth utilization or latency in networks. For example, if one link between an aggregation switch and an edge switch in a three layered tree network is down and multicast traffic gets through a core switch, it would be better to reroute traffic to another core switch near the root of the tree, otherwise multicast traffic may go through multiple core switches because of a reroute at the aggregation switch where the failed link is connected.

3.3 Tree Management Method

In this section, we explain how to manage multicast trees in a multicast controller.

To calculate and manage multicast trees, the controller must collect network topology, and find where senders and receivers are connected. The controller has a database to store network topology as a graph, a state of multicast trees, and multicast group membership including senders and receivers. Switches send packets that are related to changes of the membership to the controller, such as IGMP messages and packets sent to unknown multicast groups. The switches also send other events to the controller, like port down or up. When the controller receives such events, the controller updates its database, and modifies flow entries in switches if needed.

When the membership is frequently updated, the controller should be overloaded due to calculation of new trees that reflect changes, and some switches should also be busy updating their flow entries. To reduce the loads, we propose that the controller computes and stores trees covering all switches that receivers may be connected to, and uses them to extract switches whose flow entries should be updated to reflect changes. We describe the details in Sec. 3.3.1.

When a failure is notified to the controller, the controller should recalculate the trees that do not include failed links or switches, remove the existing trees from switches to avoid duplicate delivery of multicast packets, and install the new trees in switches. There are several time-consuming tasks such as tree computation, installing, and uninstalling trees in switches. These tasks make the packet loss duration longer. We propose that the controller calculates and installs redundant trees when the membership is changed, and the controller can switch the tree to another for delivering packets at the time of failure. The details are explained in Sec. 3.3.2.

In the following, we use the terms “switch” and “link” to point out physical entities that construct a network, and “node” and “edge” as internal data structure of switch and link in the controller. Each node and edge corresponds to each switch and link. “Root node” or “root switch” represents a node or a switch where a sender is connected, and “leaf node” or “leaf switch” is a node or a switch where a receiver is connected.

3.3.1 Tree Update when Multicast Group Membership Changes

Processing changes of multicast group membership quickly is a key to handle many changes in the controller. This process includes three steps as follows:

1. Update the multicast group membership database in the controller (explained in Sec. 3.3.3), including switches and ports where senders and receivers are connected.
2. Compute how multicast trees must be updated.
3. Update flow entries in switches.

To handle the changes quickly, we need to reduce time-consuming tasks, such as tree computation in the controller, and modification of flow entries in switches.

Our approach is to calculate multiple trees for one multicast group. Each tree covers all nodes where receivers may be connected, and is stored in the tree database in the controller. The trees are calculated when both a sender and one or more receivers appear. The trees are deleted when neither sender nor receiver exists in a group. The controller maintains and modifies the trees as below.

In order for the controller to extract switches whose flow entries should be modified, the controller should know which switches have flow entries for forwarding multicast packets. A simple way to do this is to query switches to send flow entries regarding multicast to the controller, but this brings much overhead such as delay until the controller receives all responses. We store such state in the controller by defining a subtree of each tree. The subtree covers a sender and all receivers, and is installed in switches to deliver packets. Each node and edge in the tree has a flag that indicates whether the node or edge is included in the subtree. The values of the flag means the following state:

On Active State. The corresponding switch or link is in the subtree.

Off Inactive State. The corresponding switch or link is **not** in the subtree.

When a new receiver is added to a multicast group, firstly the controller retrieves the trees used by the group from the database, or calculates the trees and stores them to the database if no tree is computed for the group in advance. Then, for each tree, the controller executes Algorithm 1 to include the new leaf node, where the new receiver is connected, in the subtree, and to list nodes whose corresponding switches must update their flow entries (*update_nodes* in Algorithm 1). The controller follows the tree to the root node from the new leaf node until the controller visits the active node. While following the tree, the controller changes a state of visited nodes and edges to active, and adds the visited nodes to *update_nodes*. When adding a node to *update_nodes*, edges to the parent node and to the child nodes that are active are also saved in the list, and this data is used for constructing new flow entries. Then, the controller fetches edges that are not directly related to the tree, such as edges to a sender and to receivers, from the multicast group membership database in Sec. 3.3.3, and updates flow entries in the switches whose corresponding nodes are in *update_nodes*. The flow entries are updated in the order of *update_nodes* so that packets are transmitted after all switches are ready to deliver. For the trees other than the tree used for packet delivery, the controller does

Algorithm 1 Algorithm to list nodes when adding a new receiver

```
$update\_nodes$ = []
$node$ ← a leaf node where a new receiver is attached
append $node$ to $update\_nodes$
while a state of $node$ is inactive do
    set a state of $node$ to active
    if $node$'s parent node exists then
        set a state of the edge between $node$ and $node$'s parent node to active
        append $node$'s parent node to $update\_nodes$
        $node$ ← $node$'s parent node
    else
        break
    end if
end while
```

not install any flow entry in the root switch, so that the flow entry does not override the flow entry installed by the tree used for packet delivery.

The process of removing a receiver from a multicast group is similar to the process of adding a receiver. Firstly, the controller retrieves the trees used by the group from the database. Then, for each tree, the controller executes Algorithm 2 to list nodes whose corresponding switches must update their flow entries (*update_nodes*). The controller follows the tree from the leaf node where the receiver is connected until the controller finds the node that has other receivers or child nodes that are active. While following the tree, the controller adds visited nodes to *update_nodes*, and sets a state of visited nodes and edges to inactive except the node that the controller stops following the tree, which must be active. Finally, the controller updates or removes flow entries in the switches whose corresponding nodes are in *update_nodes*. The flow entries are updated or removed in the reverse order of *update_nodes* to stop delivering packets first.

Figure 3.1 gives an example of adding a receiver (Receiver 1) to the trees. There are two trees, Tree 1 shown by solid lines, and Tree 2 by dotted lines.

If Receiver 1 is the first receiver in the group, the controller calculates two trees, Tree 1 for packet delivery and Tree 2 for backup. Then, the controller follows nodes of Switch D, B, and A on Tree 1 in order, changes the state of these nodes and edges between them to active, and installs flow entries in these switches. Next, the controller follows nodes

Algorithm 2 Algorithm to list nodes when removing a receiver

```
update_nodes = []
node ← a leaf node where a receiver is left
append node to update_nodes
while no child node of node is active and node has no receiver do
    set a state of node to inactive
    if node's parent node exists then
        set a state of the edge between node and node's parent node to inactive
        append node's parent node to update_nodes
        node ← node's parent node
    else
        break
    end if
end while
```

of Switch D, C, and A on Tree 2 in order, changes the state of these nodes and edges between them to active, and installs flow entries in the switches except Switch A.

When another receiver joins in the group, for example a new receiver is connected to Switch E, the controller follows the trees and installs flow entries like the case of Receiver 1. The controller stops following the tree at the node of Switch B or C because these nodes have already been active. In this case, the controller modifies the flow entries in Switch B and C, and installs new flow entries in Switch E.

When a receiver at Switch E leaves from the group, the controller follows the trees from the node of Switch E to the nodes of Switch B or C because these nodes have active edges to the node of Switch D. Then, the controller removes flow entries in Switch E, changes the state of the node of Switch E and the edge between it and its parents (the nodes of Switch B and C) to inactive, and modifies the flow entries in Switch B and C. When Receiver 1 also leaves, the controller follows the trees until the node of Switch A, removes the flow entries from all switches, and deletes the trees stored in the controller's database because no receiver exists in the group.

In our proposed method, the controller needs to calculate trees only when a new multicast group appears, and reuses them when the controller handles changes of multicast group membership. In addition, the controller does not need to communicate with switches to retrieve the state of the trees because the state is already stored in the con-

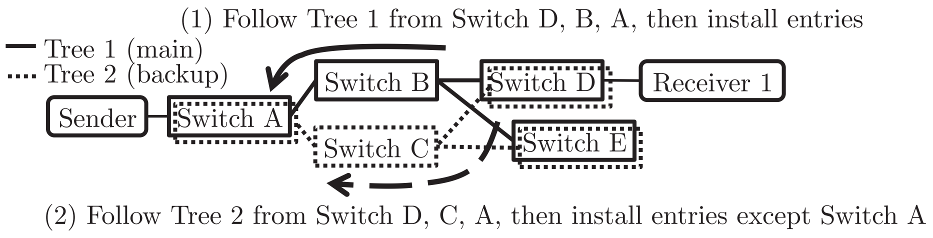

(1) Follow Tree 1 from Switch D, B, A, then install entries
Tree 1 (main)
Tree 2 (backup)
Sender
Switch A
Switch B
Switch C
Switch D
Switch E
Receiver 1
(2) Follow Tree 2 from Switch D, C, A, then install entries except Switch A

Figure 3.1: An example of processing multiple trees

troller. Therefore, the controller can handle modifications to multiple trees quickly, and a few switches communicate with the controller to update their flow entries.

3.3.2 Failure Recovery

To make multicast reliable, the multicast trees must be recovered quickly when a failure occurs in networks. MPLS Fast Reroute (FRR) mechanism works well by setting up backup paths or trees in advance and by rerouting to one of the backup paths at the time of failures. We also use the preplanned approach like MPLS FRR, but MPLS FRR restricts flexibility of backup trees that the controller computes because MPLS FRR requires the parent router of the failed router or link reroute the traffic.

As explained above, in our proposed method, the controller computes multiple trees per multicast group, and installs the subtrees needed to deliver packets to receivers at the same time in the same way of the tree for packet delivery except the root switch.

The network should avoid duplication of multicast packets at receivers and packet loop both in normal state and during failure recovery. The controller assigns a unique ID within a multicast group to each tree. The root switch embeds the tree ID used for packet delivery into packets by rewriting a part of packet headers. Other switches forward packets based on the tree ID, the source IP address and the multicast IP address in header fields. The switches rewrite the header fields in packets to appropriate ones when they send packets to receivers.

When a failure is detected at a switch or other devices, the switch or other devices notify the failure to the controller. For each multicast group, the controller checks whether the tree currently used for packet delivery has become disconnected by the failure, and executes the tree recovery procedure if disconnected.

The tree recovery procedure is as follows. The controller selects one of backup trees unaffected by the failure, and modifies the flow entry in the root switch to embed an ID of the selected backup tree into packets. For example in Fig. 3.1, when a link between Switch B and D is down, the controller replaces an flow entry in Switch A to forward packets through Switch C. If more than one backup tree are unaffected, the controller selects one tree by the criteria that the network operators defined, for example, priority, minimum cost of all edges, and disjointness from trees used by other groups. This criteria is out of scope in this chapter.

By switching a tree at the root switch, we can use any algorithm to compute trees, such as vertex redundant, edge redundant and least disjoint trees. In addition, the controller does not need to update flow entries in switches other than the root switch, and multicast packet delivery through a new tree is started quickly.

3.3.3 Multicast Group Membership and Tree Databases

To setup multicast trees, the multicast controller must find switches and ports where senders and receiver are attached. Such location data is stored in the multicast group membership database for senders (the senders database) and that for receivers (the receivers database) in the controller. The controller also needs to hold tree status, such as active or inactive nodes and edges, and such status is also stored in the tree database.

The databases identify a multicast group by a pair of a sender and a multicast IP address. The controller sometimes needs to look up the database without a sender IP address, for example, a receiver will join in or leave from multicast groups without a sender IP address. In such cases, the controller uses 0.0.0.0 as a sender IP address, and the databases return results appropriately as follows. When the controller queries receivers belonging to a specific multicast group, which means the group has a sender IP address other than 0.0.0.0, the database merges lists of receivers that specify the sender IP address when joining and that do not specify the sender IP address, and returns the merged list.

Senders Database: Each record in the senders database contains a switch ID and a port number where a sender is connected. The controller will look up the senders database in two ways, specifying both a sender and a multicast IP address to retrieve the location of a sender, and specifying only a multicast IP address to fetch a list of senders using a specific multicast IP address. To make these lookups fast, the senders database is indexed by multicast IP addresses using hash tables, and each entry in the hash tables is a list of the records that have the same multicast IP address.

Receivers Database: The records in the receivers database contain a list of receivers. Each receiver is identified by a switch ID and a port number where a receiver is connected, and it includes other data related to state management such as ones required by IGMP. The controller makes two kinds of queries to the receivers database. One is to retrieve one receiver with a sender IP address, a multicast IP address, a switch ID and a port number, and this type of query is often executed to update the state of a specific receiver. The other is to fetch all receivers belonging to the group when updating multicast trees. To handle the latter case quickly, the receivers database is indexed by multicast IP addresses using hash tables, and receivers in each group are stored as a list per multicast group.

Tree Database: The controller must store several data per multicast group in the tree database, such as multiple trees including backup trees, a subtree of each tree by indicating whether each node and edge is active or not, data to show which tree is used for packet delivery. When the controller updates the trees by adding or removing a receiver, the controller must quickly look up the leaf node of each tree where the receiver is attached. When the controller handles failure recovery, the controller must find the failed node or edge in trees quickly to check whether the failed node or edge is active in trees. To meet above points, the tree database manages entries as follows. The top level record is an entry of a multicast group (group entry), which consists of (source IP address, multicast IP address) pair, a list of trees and its IDs (tree entries), and the tree ID used for packet delivery. The group entries are indexed by (source IP address, multicast IP address) pair. Each tree entry consists of a list of nodes and edges, and indexed by nodes. Each node and edge has a state, active or inactive, and pointers to the physical entity such as switch ID and and port number.

3.4 Implementation of Prototype System

We have implemented our prototype multicast controller using C and Trema. Figure 3.2 shows an overview of our design of the OpenFlow controller. Switch Communication, Event Dispatcher and Topology modules are provided as a part of Trema. Switch Communication module manages OpenFlow channels to the switches. The Event Dispatcher forwards messages from switches to pre-configured modules. Topology module collects network topology by sending and receiving LLDP packets from each port in switches. This module also handles switch status related messages like port status changes, and provides topology data to other modules.

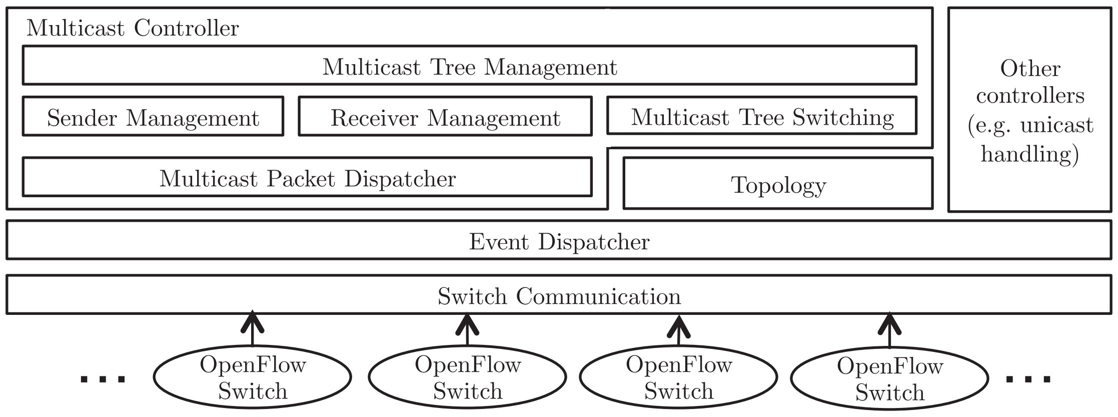


Figure 3.2: An overview of our controller design

The Multicast Controller in Fig. 3.2 is the core of multicast tree management. It consists of Multicast Tree Management module, Sender Management module, Receiver Management module, and Multicast Tree Switching module. The Multicast Packet Dispatcher dispatches packets in Packet-In messages from switches to the appropriate modules, such as IGMP packets to the Receiver Management module, and other packets to the Sender Management module.

The Sender Management module stores and updates the senders database, and provides an interface for the senders database to other modules. This module sets switches to forward unknown multicast packets to the controller. When the controller receives such packets, this module updates the senders database. If a new record is added to the senders database, this module sends a notification to the Multicast Tree Management module. In addition to the parameters described in Sec. 3.3.3, each record in the senders database includes a sender's MAC address for rewriting the source MAC address in packets to the original one when switches output the packets to receivers because a field of the source MAC address is modified to embed the tree IDs into packets.

The Receiver Management module is in charge of the receivers database. To collect multicast group membership, this module acts as a multicast router in IGMP specification [23]. Receivers' data is stored in the receivers database. When the receivers database is updated such as when a new receiver is added, this module notifies it to the Multicast Tree Management module. IGMP requires multicast routers have a state of a receiver, and we include such state into the receivers database.

The Multicast Tree Management module is responsible for the tree database and flow entries related to forwarding multicast packets. This module receives events of sender and receiver changes from the Sender Management and the Receiver Management modules, computes and updates multicast trees including backup trees in the tree database, and modifies flow entries in switches. This module retrieves records from the senders database in the Sender Management module and from the receivers database in the Receiver Management module if necessary. This module also receives and handles requests of changing multicast trees for packet delivery from the Multicast Tree Switching module.

The controller computes two trees that have few common edges by Dijkstra's SPF algorithm. The controller calculates a tree with the original costs of edges, and this tree is used for packet delivery. The backup tree is computed in the graph created by adding the sum of original costs of all edges to the costs of edges included in the tree for packet delivery. A tree ID is embedded in a source MAC address of packets.

The Multicast Tree Switching module handles topology change events. A link down event is notified to the Multicast Tree Switching module via the Topology module. For each multicast group, the Multicast Tree Switching module retrieves trees in the group including backup trees from the tree database in the Multicast Tree Management module, and checks whether the trees are disconnected by the link down. If the tree for packet delivery is disconnected but a backup tree is connected, this module sets the ID of the backup tree to the tree ID for packet delivery, and notifies the Multicast Tree Management module to update a flow entry in the root switch.

3.5 Evaluation

We have evaluated the processing time for changes of multicast group membership and failure recovery in our prototype controller, and duration of packet loss at the time of a failure using hardware switches.

As a system for comparison, we implemented a controller that has the following types of modes for management of multicast trees.

Redundant/No Redundant The controller computes redundant trees for backup or not.

Install/No Install The controller install backup trees in advance or not (This mode exists only in Redundant mode).

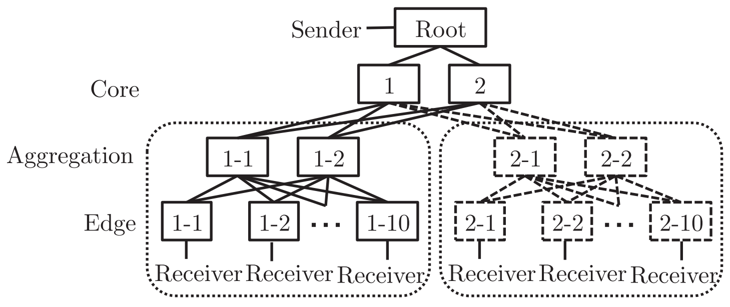


Figure 3.3: Three-layer hierarchical topology for evaluation

Precompute/No Precompute The controller precomputes the trees or not. In the Precompute mode, the controller computes trees covering all nodes and stores them in the tree database when a multicast group is created. In the No Precompute mode, the controller computes trees every time that the controller needs to find new paths or trees, and stores only necessary parts of the trees.

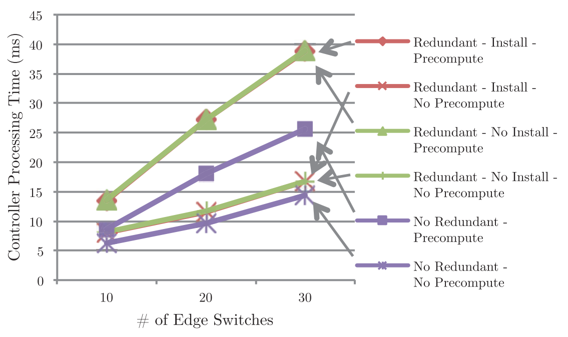
In summary, there are six modes, Redundant - Install - Precompute (our proposed method), Redundant - Install - No Precompute, Redundant - No Install - Precompute (CastFlow [53] approach), Redundant - No Install - No Precompute, No Redundant - Precompute, and No Redundant - No Precompute.

3.5.1 Processing Time of Controllers for Update of Multicast Group Membership and Failure Recovery

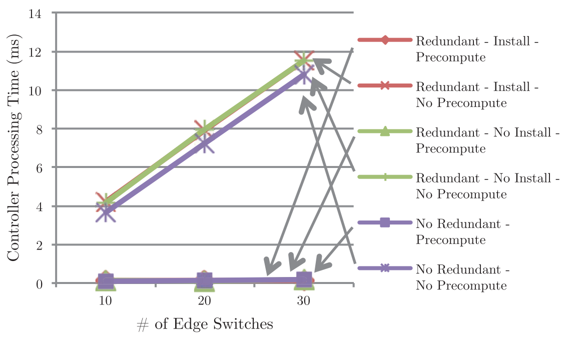
We have evaluated how much our proposed method shortens the processing time to update multicast group membership and the time for failure recovery in a controller. We measured the processing time to add a new receiver, to remove a receiver, and to restore packet delivery from a link failure.

We used three-layered hierarchical topology shown in Fig. 3.3, which is typical for private networks. The topology consists of the root switch and three layers, core, aggregation, edge layers, and they are connected in a redundant way. Each aggregation switch is connected to 10 edge switches, and one receiver is connected to each edge switch. In our evaluation, we use three types of networks that have one, two, and three sets of two aggregation and 10 edge switches, which have 10, 20, and 30 edge switches in total respectively.

We measured the time to add a first receiver and others separately because these processes are different. The processing for a first receiver includes creation of a new



(a) A first receiver



(b) Another receiver

Figure 3.4: Processing time to add a receiver

record in the tree database and tree computation, but the controller does not need to compute trees when adding other receivers. Similarly, we measured the time to remove a last receiver and others separately because the former includes deleting a group entry from the tree database but the latter does not.

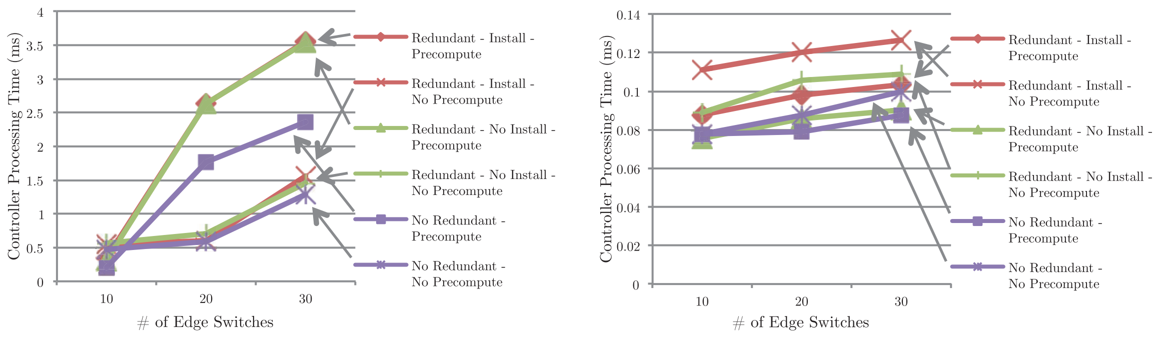
We measured the time to add a first receiver or to remove a last receiver 10 times each. We also measured the time to add or remove other receivers 10 times by adding or removing them one by one. To induce failures many times, we repeatedly set a link between the root switch and a core switch to down and up, then set the link to the other core switch to down and up. We induced failures 10 times per mode.

We recorded the time when the controller received port status messages and Packet-In messages containing IGMP messages, started the process of removing a receiver[1], and sent flow entry modification (flow-mod) messages to switches.

We virtually constructed the network in a PC using the virtual ethernet link (veth) and Open vSwitch [65], which had Xeon X5355 CPU, 8GB RAM and run CentOS 6.4. The controller run on another PC that had Core 2 Duo T7400 CPU, 4GB RAM and run Ubuntu 12.04, and was directly connected to the PC running Open vSwitch at 1Gbps.

Figure 3.4(a) and Fig. 3.4(b) shows the processing time in the controller to add a first and another receiver. Figure 3.5(a) and Fig. 3.5(b) shows the processing time in the controller to remove a last or another receiver. Figure 3.6 shows the processing time to restore packet delivery from a link failure. The x-axis shows the number of edge switches, and the y-axis shows the processing time in milliseconds. The arrows point to the lines corresponding to the modes. The values are on average.

[1]IGMP [23] requires a router delay removing a receiver until the router confirms that no other receiver is left.



(a) A last receiver (b) Another receiver

Figure 3.5: Processing time to remove a receiver

In Fig. 3.4(a), we can see the overhead by computing redundant trees. The time with redundant trees (Redundant - No Install - Precompute) is about 1.5 times longer than the time without redundant trees (No Redundant - Precompute). Another overhead is to precompute trees. For example in “Redundant - Install” cases, the time in the “Precompute” case is about 2 to 3 times longer than the time in the “No Precompute” case. We can see this trend in both “Redundant” and ”No Redundant” cases.

In the case of adding a receiver to an existing multicast group (Fig. 3.4(b)), a trend is changed. In the “Precompute” cases, the processing time is less than 1 ms regardless of the number of switches, although in the “No Precompute” cases it takes more than 3 ms and the time increases as the number of switches increases.

In Fig. 3.5(a), the controller takes more time to remove the last receiver in the “Precompute” cases than in the “No Precompute” cases. Within the “Precompute” cases, it takes about 1.5 times longer time in the “Redundant” cases than in the “No Redundant” case. There is little difference between three “No Precompute” cases. As with the case of adding a first receiver (Fig. 3.4(a)), we can see this trend in both the “Redundant” and “No Redundant” cases.

According to Fig. 3.5(b), there seems to take a little longer time to remove a receiver when trees are computed in advance. Overall, the values are small compared to the cases of adding receivers and removing the last receiver.

Figure 3.6 shows the processing time to restore packet delivery from a link failure, and it includes only cases where trees are precomputed because the procedures are not changed whether trees are precomputed or not.

In Fig. 3.6, the time in “No Redundant - No Install” is increased in proportion to the number of edge switches. The time in the other cases, which computes redundant

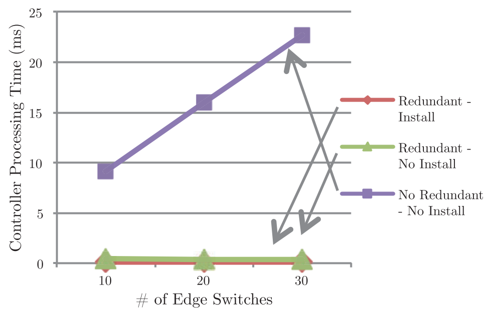


Figure 3.6: Processing time to recover from a link failure

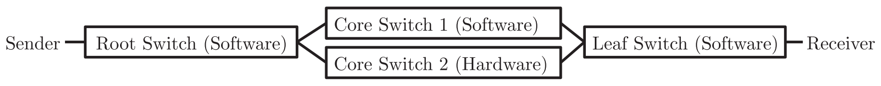


Figure 3.7: Network topology for evaluation of packet loss duration during a failure recovery

trees in advance, is almost the same regardless of the number of edge switches, and much shorter than in “No Redundant” case.

3.5.2 Recovery Time of the Multicast Tree

For fast failure recovery, it is also important that switches install new flow entries quickly. We have measured the duration that receivers do not receive multicast packets when a failure occurs in a network using a hardware switch.

We used network topology in Fig. 3.7. The Root Switch, Core Switch 1 and Leaf Switch were software switches, and the Core Switch 2 was a hardware switch. The sender was connected to the root switch, and the receiver was in the leaf switch. To induce a failure where installation of flow entries to the hardware switch is necessary, we shut down a port on the root switch to the Core Switch 1. This makes the controller installing flow entries to the Core Switch 2 when backup trees are not installed in advance, but no installation to the Core Switch 2 is necessary when backup trees are installed in advance. We induced failures 10 times in total.

A sender transmitted packets to each multicast group with incrementing sequence numbers in the packets every millisecond. The receiver monitored sequence numbers and intervals of packet arrivals, and recorded the intervals when the number in a packet was jumped to two or more bigger value at once.

We created one to ten multicast groups. In all the groups, the root switch was the

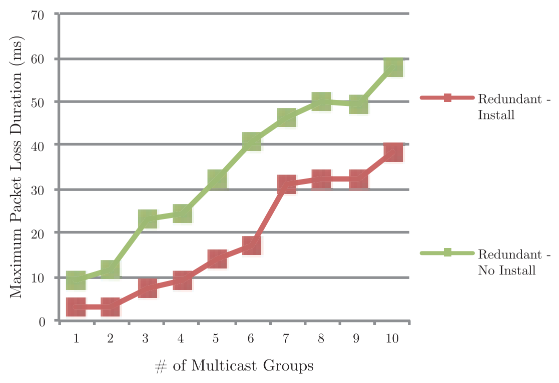


Figure 3.8: Maximum packet loss duration when a link between Root Switch and Core Switch 1 is down

Root Switch, and the receiver was connected to the Leaf Switch. In each failure, we regard the maximum packet loss duration as the largest value of packet loss duration among the groups.

We configured software switches on one PC, which had Intel Atom C2358 CPU and 4 GB RAM and run Ubuntu 14.04.2 and Open vSwitch 2.0.1. The hardware switch was NEC PF5240. The sender and receiver PCs were almost the same as the controller in Sec. 3.5.1, but the size of RAM was 1 GB in the sender PC and 512 MB in the receiver PC. Both were connected to switches at 1 Gbps.

Figure 3.8 shows maximum packet loss duration when a failure occurs. The x-axis shows the number of multicast groups, and the y-axis the maximum packet loss duration on average. As with Fig. 3.6, Fig. 3.8 includes only cases where trees are precomputed. The "No Redundant" case is not included because it would take longer than in the "Redundant - No Install" case due to calculating trees.

The values were unstable when there were more multicast groups, especially seven to ten groups. We believe this is due to a jitter added by software switches.

3.6 Discussion

3.6.1 Processing Time in Controllers

The results of the processing time in the controller show that the tree precomputation approach can dramatically shorten the time to add a receiver except a first receiver. This reduction is because, as we have expected, a controller uses the same tree once the controller calculates a tree, instead of calculating trees every time that a receiver joins in a multicast group. The controller can add a receiver in $O(N)$ with the tree precomputation approach, but tree computation requires $O(N^2)$ for example in Dijkstra's algorithm, where N is the number of switches. We cannot see a large difference between cases where redundant trees are installed or not when adding a receiver.

When a first receiver joins in, the controller performs more processes in the tree precomputation approach, including computing a tree, allocating memory space for a new entry in the databases, etc. This is the reason why the controller in the tree precomputation approach takes longer time to add a first receiver. When the controller computes redundant trees, it takes more time to compute additional trees. As with the case of adding another receiver, there seems little overhead in the processing time by installing redundant trees.

Considering many multicast applications such as video streaming and distribution of data to many hosts, taking more time in the cases of a first receiver should not be a significant problem. In such applications, a controller frequently performs the processing to add a receiver to an existing multicast group to include more hosts in multicast trees, rather than to a new multicast group for delivering different data.

As for the processing time to remove a last receiver from a multicast group (Fig. 3.5(a)), the controller takes longer with the tree precomputation approach than without precomputing trees. There is also a difference whether the controller uses redundant trees or not. This is due to the time to delete trees from the tree database. The controller deletes all trees used by the group when the group has no receiver. If the trees are precomputed, the controller should deallocate memory space for all nodes and edges one by one, but if not, the controller deallocates memory space only for the path from the root node to the left node. There is a room to optimize this procedure, such as allocating and deallocating memory space for entries in the tree database at once.

As to removing another receiver, the controller takes slightly longer without the tree precomputation approach than with this approach. The reason is that the controller deallocates memory space for unnecessary nodes and edges only when the controller precomputes trees.

In summary, our proposed method greatly increases the controller's capacity to handle addition or removal of receivers to or from existing multicast groups, which is one of important characteristics in private networks. It takes a slightly longer time to create or remove a group entry in the tree database, but it is not a significant problem because it merely occurs in practice compared to the addition or removal of receivers.

3.6.2 Failure Recovery

Our proposed method can also reduce the time to recover multicast trees. The processing time for failure recovery (Fig. 3.6) increases linearly in the "No Redundant - No Install" case as the network size increases. The processing time in other two cases ("Redundant - Install" and "Redundant - No Install") is greatly shorter than "No Redundant - No Install" case, and hardly increase as the network grows. This is because the controller recalculates a tree only if no redundant or backup tree is available.

There is little difference between "Redundant - Install" and "Redundant - No Install" in terms of the processing time in the controllers, but there is a large difference when we consider the number of flow entries that should be installed in hardware switches (Fig. 3.8). The larger the number of flow entries the controller requires to install in hardware switches, more time it takes until delivery of multicast packets in all multicast groups are restored. Although the values in Fig. 3.8 are slightly smaller than those reported [34], we can see a trend that software switches handles flow modifications faster than hardware switches. We believe the difference between values reported in [34] and us would be due to difference in implementation of switches.

When a redundant or backup tree is available and installed, the controller needs to replace a flow entry only in the root switch, therefore the number of flow entries modified at each switch is reduced, especially at the core of networks like core switches. Our proposed method cannot work effectively when many multicast groups share the same root switch, but we can avoid this by allocating senders to different root switches, using a software switch as a root switch, etc.

Considering video streaming as an application of multicast, it would be better to minimize frame loss, which corresponds to packet loss in the network layer, because losing one frame disturbs several frames [49]. Our proposed method can reduce the possibility of frame loss because our proposed method can restore packet delivery quickly. Let us assume that a frame rate of a video is 30, which is close to values used in television. In this case, frames are sent at the interval of about 33 msec. According to the results of our evaluation (Sec. 3.5.2), one frame should be lost without our proposed method if there are more than five multicast groups, but our proposed method can increase the number of multicast groups to seven or more. In addition, as we have mentioned, the packet loss duration can be reduced when root switches are not shared among multicast groups.

3.6.3 Limitations and Overheads

In real situations, there are some factors to make the packet loss duration longer. One factor is a failure detection delay. In our experiments, we set interfaces to down, and the delay in a switch was small. If switches or other devices cannot detect failures immediately, the packet loss duration would increase due to the detection delay.

The other factor is latency. We connected each switch and the PCs directly and all hosts were in our lab, the latency among switches and the controller was very small. When the latency becomes high, the packet loss duration would be long because packets are rerouted at the root switch. Our proposed method, "Redundant - Install - Precompute," is beneficial in such cases because the processing time in the controller is shorter and the latency among switches and the controller equally increases the packet loss duration in all cases. In real environments such as campus or enterprise networks, the latency would be a little higher than our experiment, but the time to compute trees or to install flow entries in switches would be longer than the latency because switches are located in a small area. Therefore, the problem caused by the latency would not be significant.

One of the overheads in our proposed method is the memory usage. When a controller precomputes and stores multiple trees, the controller uses more memory space than the cases where trees are not precomputed or the controller stores one tree per multicast group. In this perspective, our proposed method uses the largest memory space in all cases that we have evaluated. Although a total size of the tree database depends on the tree size and the number of multicast groups, we think this problem is not significant because tens of gigabytes of memory are available in current PCs. For example, in our proposed method, the data size of each node and edge on a tree is less than 20 bytes, and more than 53 million nodes and edges in total can be stored in one GB. In addition, when a multicast group has more receivers, trees would cover most of switches, and the overhead becomes small.

Another overhead is the increase of the number of flow entries that switches store due to backup trees. This overhead can be negligible in private networks that have hierarchical tree topology and two redundant switches in each layer. In such networks, when a failure occurs, the traffic is generally rerouted to another switch in the same layer. In the case of multicast using OpenFlow without our proposed method, flow entries are removed from switches, and installed to other switches where multicast packets are rerouted. In other words, flow entries for backup trees are installed at the time of failures, and a capacity of each switch, such as the number of flow entries that a switch can install, must be designed with consideration of backup trees. In this perspective, our proposed method just installs flow entries that are needed to reroute packets before a failure occurs, and switches do not require extra capacity to support our proposed method. If more than two redundant switches are installed, the overhead becomes larger in our proposed method than cases without our proposed method.

3.6.4 Comparison of Signaling and Available Trees with Existing Multicast Control Mechanisms

One of the problems in MPLS based approaches is that all switches or routers on a tree must handle messages to modify the tree because of the specifications of a signaling protocol, RSVP-TE. This means that switches or routers must process messages to change some parts of the tree even when they do not have to change their state. By calculating the difference of multicast trees centrally, only switches that have to modify its flow entries need to process messages from the controller. Furthermore, switches near the core of a network do not have to modify its flow entries frequently if the group has many receivers and its tree covers many parts of the network. These characteristics would greatly decrease loads of switches.

There is a tradeoff where to reroute multicast packets, or how to construct backup trees before a failure occurs. One choice is to reroute packets at a switch where a failure is detected, called local repair, and this approach is used by MPLS Fast Reroute and the fast failover group feature in OpenFlow 1.1 and later. Mechanisms based on local repair can dramatically reduce packet loss when latency between routers and switches is long such as in WAN, but trees used by local repair after failures are detected are not necessarily efficient in terms of metrics such as the number of flow entries used by backup trees (backup trees or paths must be created from each switch), bandwidth usage, and the number of routers and switches that a tree covers (rerouted packets may go through additional links to reach to receivers via backup paths).

Another choice is to reroute packets at routers or switches near the root router or switch. This choice has an advantage that backup trees can be created with various metrics, such as minimizing routers or switches that packets go through, and avoid wasting bandwidth caused by rerouting packets. A drawback in this choice is the increase of packet loss at the time of failures because packets during delivery on a network are lost, and there seems no way to notify of failures a router or a switch that reroute packets.

The former is negligible in private networks because latency between switches is small. The latter can be solved by centrally managing multicast trees.

Although rerouting packets at a switch other than the root switch solves some problems in terms of bandwidth usage and the number of switches, there still has a problem that the number of flow entries used by backup increases, as with the local repair approach. If we can prepare for a backup tree that cover the failures in any switch, we can reroute packets at the root switch, and only one backup tree is needed to recover multicast trees. The overhead caused by one backup tree will be acceptable as we have explained above.

3.7 Concluding Remark

In this chapter, we propose a method to manage multicast trees in an OpenFlow controller in private networks, which supports both dynamic multicast group membership changes and fast failure recovery that takes into account processing time in controllers and slow performance for modification of flow entries in hardware switches. Our key idea is that a controller computes and saves trees covering all switches in advance, and updates a state of each node or edge in a saved tree when a receiver is added or removed. This reduces the number of tree computation times. For failure recovery, our proposed method enables setting up multiple trees including backup trees in advance, which are computed by any algorithm, and switching to another tree for packet delivery with minimum modification of flow entries.

Our evaluation using our prototype controller shows our proposed method can reduce the processing time in the controller to add or remove a receiver and to recover trees from failures, and we also show that packet loss duration due to a failure is also reduced in a physical network. We also show that there is little difference in the controller processing time whether redundant trees are installed in advance or not, but the difference becomes large in hardware switches due to the number of flow entries to be modified. Although our proposed method has some disadvantages, these disadvantages would not be significant in practice because these rarely occurs, PCs running a controller program usually have enough computing resources to mitigate the impact of these problems, and the characteristics of private networks.

Chapter 4

Fast Failure Detection of OpenFlow Channels

4.1 Introduction

To continue to deliver packets to destinations, it is important that the control plane and the data plane are connected and can exchange messages when needed. Networks should reroute traffic quickly at the time of failures; the data plane must detect failures and notify the control plane of them immediately, and the control plane must update packet forwarding rules quickly if necessary. The first and important step is to detect failures, and some mechanisms are proposed to make the detection faster like BFD [39].

We hardly need to care for connectivity between the data plane and the control plane in traditional network devices because both planes are in the same device, but things are different with Software Defined Networking and OpenFlow. In OpenFlow, switches (the data plane) detect occurrance of events including failures and notify controllers (the control plane) of the events through OpenFlow channels, then the controllers calculate new flow entries and update them in switches if necessary. To communicate over OpenFlow channels, links and network devices that connect controllers and switches should be up, and such links and network devices also consists of the control plane. When a failure occurs to the data plane, a failure may also occur to the control plane because the control plane and the data plane are sometimes not physically separated, use the same power source, pathway, etc. Thus, an OpenFlow channel may also be disconnected temporarily at the same time with the failure in the data plane, and the controller is not notified of the failure in a timely manner. As a result, there may be significant delay until new flow entries are installed in the switches. Therefore, fast failure detection of OpenFlow channels is very important.

OpenFlow protocols define keep-alive messages (echo request/reply) for failure detection and multi-controller support for fault tolerance. It is not desirable to exchange keep-alive messages between switches and multiple controllers in a short interval because the controllers should respond to many keep-alive messages from many switches. This is a significant overhead when the rate of messages other than keep-alive ones is low and controllers' performance for processing messages is not high.

To reduce the number of keep-alive messages while detecting a channel failure quickly when necessary, we propose a mechanism to detect a failure of OpenFlow channels in switches and controllers when switches and controllers should notify the other side of the events occuring. The proposed mechanism quickly detects that an OpenFlow message does not arrive at the other side of the channel, instead of frequently checking a channel is up. Considering that the reason to maintain OpenFlow channels is to be always ready to deliver important messages with less delay, detecting non-arrival of messages is enough to keep a network controllable.

We assume that, for redundancy, multiple controllers run and a switch establishes channels to two or more controllers. From a switch's perspective, a message by an event in the data plane such as port down is sent over all channels at the same time, and the message arrives at controllers if at least one channel is not failed. Therefore, the switch should detect that no controller receives the message. From a controller's perspective, a controller can detect that a message does not arrive at a switch by a synchronous message that uses a request-reply pattern, for example, a message that changes a state or a configuration of a switch. In this case, the controller sets a timeout for receiving a reply message from the switch and waits until the timeout has expired, then the controller tries to use another channel. If the controller can detect channel failures by other means, the controller can avoid using the failed channels for sending request messages, and the switch can receive the messages with less delay.

In the switch side, a switch sends a keep-alive message (echo request) to all channels just after sending a message by an important event such as port down. If the switch receives a reply message (echo reply) from one or more channels, the switch expects that all the previous messages including one by the event have arrived at one or more controllers. When the switch does not receive the reply message for a certain period of time (timeout), the switch suspects all channels are failed, and enters the fail secure or standalone mode according to the configuration of the switch.

In the controller side, controllers detect channel failures by sharing received messages among them. When a controller receives a message that is sent to multiple channels such as a port status message, the controller notifies other controllers of the arrival of the message. A controller that receives the notification suspects the channel to the switch from the controller has been lost when the controller has not received or does not receive the notified message from the channel to the switch for a certain period of time (timeout).

We have implemented the proposed mechanism in an OpenFlow channel proxy. We have evaluated failure detection delay and overhead in latency and throughput, and show that the failure detection delay becomes shorter than using the standard keep-alive mechanism, and overhead in latency is negligible.

This chapter is organized as follows. Section 4.2 introduces work for failure detection and recovery in general and in OpenFlow networks. Section 4.3 describes a failure pattern of OpenFlow channels, and our proposed mechanisms to detect failures is shown in Sec. 4.4. Section 4.5 shows a design of an OpenFlow channel proxy and implementation of our proposed mechanism to it. Evaluation of our proposed mechanism is in Sec. 4.6, and we discuss our proposed mechanism and the results of the evaluation in Sec. 4.7. Section 4.8 concludes this chapter.

4.2 Related Work

4.2.1 Failure Detector in Distributed Systems

Failure detection has been studied in the field of distributed systems because it is essential to construct the systems. Failure detection in distributed systems is a problem that, given a finite set of processes, finds a set of processes that are suspected to have crashed, misbehaved, etc.

A failure of transmitting OpenFlow messages are categorized as the omission failure. The omission failure is that a process did not receive some messages sent by another process [19]. A *correct* process needs to be live (not crashed) and to be able to send and receive messages with the other *correct* processes.

Sergent et al. [70] gives a summary and comparison of implementation techniques for failure detection: heartbeat, interrogation, algorithm-specific and application-specific heartbeat. Heartbeat [40] is a technique that every process q periodically broadcasts a message "I am alive", and a process p suspects that the process q has crashed when the process p does not receive messages from the process q for a certain period of time. Interrogation is a technique that a process q periodically sends a request message "Are you alive?" to a process p and waits for a reply message "I am alive" from the process p. An algorithm-specific technique is that the failure detection is executed only when a process p running an algorithm sends a critical message m_p to a process q. The process p waits for a response m_q to the message m_p, and the failure detector suspects that the process q has crashed when the process p does not receive m_q for a certain period of time, like the interrogation technique. An application-specific heartbeat technique is a combination of the algorithm-specific and heartbeat techniques. The failure detection is executed only when a process p sends a critical message m_p to a process q, and the heartbeat is executed until the process p receives a response m_q from the process q.

OpenFlow protocols define messages for implementing the interrogation technique, an echo request and echo reply message. To detect failures with a small number of messages, a simple interrogation technique is inappropriate because the interval of the request messages needs to be small to detect the failures with small delay, that is, the number of the messages increases. Therefore, the algorithm-specific technique will be required, but how to design failure detection mechanisms with making use of OpenFlow protocols is unclear.

The failure detection of OpenFlow channels is executed between a switch and multiple controllers, that is, multiple processes (controllers) can cooperate with each other outside a failure detector for failure detection. The failure detection between controllers is handled by other mechanisms, and the mechanisms are usually provided as a part of controller platform. This is a slightly different model than the model in traditional failure detection in distributed systems. It is worth considering that the failure detector uses this difference to improve the failure detection mechanisms of OpenFlow channels.

When we think of multiple controllers as one controller, we can model OpenFlow channels on multiple channels over two processes, OpenFlow switch and one virtual controller. This model is similar to Stream Control Transmission Protocol (SCTP) [81] and Multipath TCP [25] where there are two or more channels (connections or associations) between endpoints. Unlike OpenFlow that floods some messages to multiple channels, they hardly send the same data over multiple channels at once, and they check the liveness of each channel separately. OpenFlow protocol has possibility to improve the failure detection of each channel by making use of this difference.

4.2.2 Failure Detection and Recovery in OpenFlow Networks

The failure detection and recovery in the data plane is one of the main issues for failure tolerance of OpenFlow networks. Controllers periodically check the liveness of many links and switches in the data plane, and reroute the traffic if necessary. Kempf et al. [41] proposed a scalable topology monitoring architecture by implementing a monitoring function in switches. Sharma et al. [74] added a recovery action in switches to recover from failures in the data plane within 50 msec, which meets a carrier-grade requirement.

There are works to make in-band control practical, including failure detection and recovery for paths between switches and controllers. Sharma et al. [73] proposed a mechanism to recover in-band OpenFlow channels from failures using fast failover actions. Their evaluation used BFD [39] to detect a path failure between a switch and a controller, instead of detecting a channel failure.

In out-of-band control, OpenFlow channels are established over control networks that use traditional network technology, such as Link Aggregation for link level redundancy, and BFD [39] and Ethernet OAM for path monitoring. These technologies improve the reliability of control networks, but controllers and switches should also have a failure detection mechanism because other parts of the control plane that cannot be monitored by these technologies will go wrong.

Fault management of controllers is also important because we cannot control OpenFlow networks without controllers. One of the purposes of distributed controllers [44, 82, 6] is to make controllers fault tolerant. The controllers monitor each other to check whether each controller is running, and failover to other controllers running when one of controllers has been down. Kuroki et al. [45] shows the procedure to detect and recover from controller failures by monitoring controllers each other in a very short interval. Unlike these works, we try to detect failures of OpenFlow channels between the switches and the controllers. An OpenFlow channel is failed not only because a controller and a switch become down, but also because networking devices that connect switches to controllers are failed, cables that connect such devices are cut, etc.

4.3 Types of OpenFlow Channel Failure

In this section, we describe patterns of OpenFlow channel failures and which patterns controllers and switches should detect.

An OpenFlow channel becomes useless due to various reasons. One is a failure in the control plane, such as controllers, switches, or network devices in the control plane become down, or a cable is cut. The other is large delay in delivering messages. From the viewpoint of controlling the data plane, the controllers and the switches avoid using such useless channels, and we do not distinguish between failures and large delay.

We assume that multiple controllers run and each switch has channels to more than one controller for redundancy. We categorize failures of OpenFlow channels in a switch as follows.

All failure All channels are failed, and no channel to any controller is available.

Partial failure Some, not all, channels are failed.

Switches need to detect the All failure immediately. When detected, the switches must enter a fail secure or standalone mode in OpenFlow specifications [64] according to the configuration of the switches. Switches do not have to detect the Partial failure because switches mainly generate asynchronous messages, which are flooded to multiple channels, and at least one controller can receive and handle the asynchronous messages.

Controllers need to detect both the All and Partial failures immediately. In the Partial failure, the controllers should avoid using failed channels. In the All failure, the controllers should stop using a switch that the controllers cannot control, and reroute the traffic so that the traffic does not get through the switch.

However, detecting the All failure in controllers is almost impossible without exchanging request-reply messages like echo request/reply messages at all channels, because the controllers cannot get any clue from switches to start checking a state of a channel, such as sending an echo request message and starting a timer. Unlike switches, the controllers may have more chances to exchange request-reply messages, such as sending barrier request messages to confirm that previous messages are processed. In this thesis, the controllers will detect the All failure by setting a proper timeout until receiving a reply messages and by trying to send a request message to all channels one by one.

4.4 Detection of OpenFlow Channel Failure

We can easily assume that a failure of an OpenFlow channel can be detected by exchanging keep-alive (echo request/reply) messages. If each switch sends an echo request message in a short interval, a controller will suffer from responding to many echo request messages sent by many switches. In addition, load balancing with multiple controllers is impossible for keep-alive messages because each switch should check availability of each controller.

It is not desirable to exchange many keep-alive messages when the rate of other messages are low. Controllers are usually designed to insert flow entries into switches proactively so that controllers and switches do not need to communicate with each other frequently. According to a study by Shalimov et al. [71], message processing performance in some controllers is not so high. This is because some controllers place high importance on productivity of controller development rather than performance, and use a scripting language with inefficient interpreters. If a controller needs to handle many keep-alive messages, such controllers cannot be used due to many keep-alive messages although such controllers have enough performance to handle other messages.

Our approach is that, instead of checking an OpenFlow channel directly, a switch and a controller detect that a message has not arrived at the other side. When neither controller nor switch generate a message, the liveness of an OpenFlow channel does not matter because switches can forward packets only with existing configurations.

4.4.1 Detection at Switches

A switch mainly sends asynchronous messages, and does not expect a controller to respond to the messages. Therefore, we cannot set a timeout until receiving a reply, and the switch cannot confirm that a message has arrived at one or more controllers. An exception is symmetric messages including echo request/reply messages, and we use them for failure detection.

Each switch holds one state to indicate a state of channels as follows.

Active One or more channels are available.

Checking The switch is executing a keep-alive procedure.

Inactive All channels are unavailable.

When the channel state is active or inactive, a switch sends echo request messages through all channels and sets the channel state to checking just after sending an important asynchronous message such as a Port Status message, a Role Status message, and a selected Packet-In message. When the switch receives echo reply messages from one or

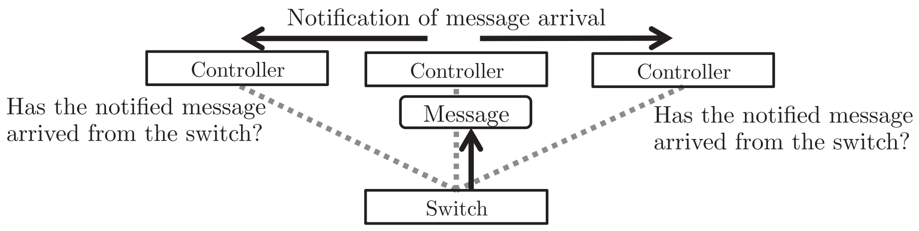

Notification of message arrival
Controller
Controller
Controller
Has the notified message arrived from the switch?
Message
Has the notified message arrived from the switch?
Switch

Figure 4.1: A key idea to detect a failure at controllers

more channels, the switch sets the channel state to active, and regards that the previous message has arrived at least one controller because TCP delivers data in order. If the switch does not receive the reply until the timeout is expired, the switch suspects that it loses contact with all controllers, and sets the channel state to inactive.

By triggering a keep-alive procedure just after sending an asynchronous message, we can reduce failure detection delay to a timeout value of the echo reply messages. We can avoid that a controller is busy to handle many echo request messages by sending them only when the channel state is not checking.

4.4.2 Detection at Controllers

A limitation in controllers is that a message initiated by the controllers is sent through one channel, not multiple channels, at the same time. This means that a controller should try to exchange request/reply messages over one channel, and when the timeout has expired, another controller should try to exchange the same messages over another channel. This is a time-wasted behavior, and we want to reduce the number of trials.

One of triggers when a controller sends a request message is an asynchronous message from a switch. For example, a controller receives a message that informs a port becomes down, the controller will update flow entries to reroute traffic. By sharing what messages arrive at which controllers among controllers by informing arrival of messages each other illustated by Fig. 4.1, the controllers can use such state of the arrival for failure detection of each channel, and the controllers can avoid trials over channels that might be failed.

The detail is as follows. Each controller has a per-switch state that records the maximum sequence ID in messages that the controller has received, which is called the Latest Sequence ID (see Sec. 4.4.3 for the sequence ID). Each controller holds a state of each channel, active, checking, or inactive as with the switches. The controllers prefer active channels for sending messages.

When a controller receives a message from a switch and a sequence ID in the received message is more than the Latest Sequence ID, the controller notifies other controllers of the message including the sequence ID and an ID of the switch (datapath ID), and sets the Latest Sequence ID to the sequence ID in the message received.

When a controller receives a notification from another controller and a sequence ID in the notification is more than the Latest Sequence ID in the state of the corresponding switch, the controller sets a channel state to the switch to checking, starts a timer, and sets the Latest Sequence ID to the received one. If the channel state has already been set to checking, the controller does not start the timer. When the controller receives a message from the channel and the message has the same or large sequence ID with the notification, the controller cancels the timer, and returns the channel state to active. When the timer has expired, which means that the controller cannot receive the message from the switch, the controller suspects that the channel to the switch has a problem, and set the channel state to inactive.

When a sequence ID in a message received from another controller is less than the Latest Sequence ID, such message is ignored.

We can obtain the followings about the state of OpenFlow channels. If a channel from a controller to a switch is active, the controller knows its channel is available. If a channel from a controller to a switch is inactive but another controller has an active channel to the switch, the controller knows at least one channel from another controller is available. If the latter situation occurs, the controller will send messages to the switch via another controller. The selection of another controller depends on the design of controllers. For example, some controllers elect a backup controller in advance, or other controllers run a leader election algorithm among controllers that have active channels.

4.4.3 Sequence ID in Messages for Duplicate Filtering and Reordering

A switch floods asynchronous messages to all channels, and the messages arrive at controllers at different timing. For example, when some channels are unavailable for a short period of time and a port in a switch becomes down and up, some controllers will receive a port down message after other controllers receive a port up message. It is difficult that controllers reorder messages according to the time when events occur.

This is so called out-of-order problem. OpenFlow specifications [64] considers this problem only in the case of role transition of OpenFlow channels, but the same problem also occurs in other asynchronous messages.

We can easily solve this problem by assigning a switch-local sequence ID to each asynchronous message. A switch assigns the same sequence ID for messages that point to the same event such as a change of port status. Controllers should ignore messages whose sequence IDs are less than the Latest Sequence ID.

4.5 Implementation of a Prototype System as an OpenFlow Channel Proxy

We have implemented the proposed mechanism in an OpenFlow channel proxy. The OpenFlow channel proxy is our own implementation by Python and Tornado library[1]. The proxy forwards OpenFlow messages and can intercept the messages.

Figure 4.2 shows an overview of the design of our prototype OpenFlow channel proxy. The prototype proxy consists of two parts. A switch-side proxy (Switch Proxy in Fig. 4.2) runs near or on a switch. It accepts a connection from a switch, and connects to controller-side proxies. A controller-side proxy (Controller Proxy in Fig. 4.2) runs near or on a controller. It accepts connections from Switch Proxies, and exchanges messages with other Controller Proxies. One of Controller Proxies connects to an OpenFlow controller.

In order that the prototype proxy is transparent to an OpenFlow channel, the proxy encapsulates OpenFlow messages in our own format and exchanges the encapsulated messages between Switch Proxies and Controller Proxies. The black solid lines in Fig. 4.2 show flows of messages in our own format, and the gray dotted lines show flows of OpenFlow messages in the original format. The headers of our format consists of a sequence ID in Sec. 4.4.3, a datapath ID, and a message type such as echo request/reply used in Sec. 4.4.1 for failure detection, notification for arrival of a message, and request for sending a message to a switch in Sec. 4.4.2.

When a Switch Proxy receives an OpenFlow message from a switch, the Encap module assigns a sequence ID to the message, encapsulates it into our own format, and sends to Controller Proxies. At the same time, the Failure Detector module monitors messages received from the switch, and sends an echo request message in our own format to Controller Proxies when necessary. The Failure Detector closes the channel to the switch if

[1]http://www.tornadoweb.org/

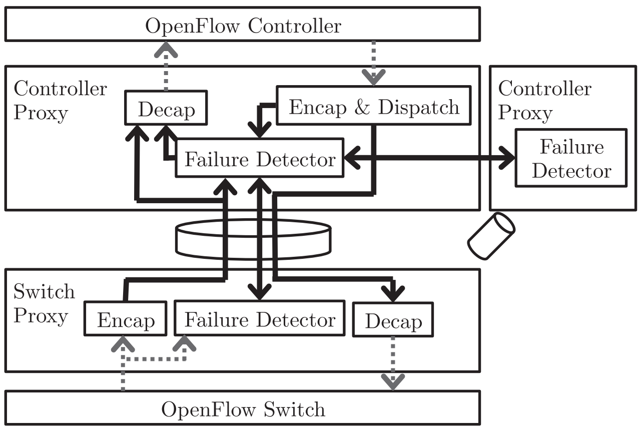


Figure 4.2: Overview of prototype OpenFlow channel proxy

the channel state becomes inactive. The Decap module filters out duplicate messages, decapsulates messages, and sends OpenFlow messages to the switch.

In a Controller Proxy, the Failure Detector module has two roles. One is to respond to echo request messages from the Failure Detector in Switch Proxies. The other is to share messages received from Switch Proxies with other Controller Proxies for monitoring connections to Switch Proxies as described in Sec. 4.4.2. The Encap & Dispatch module assigns a sequence ID to a message received from the controller, and encapsulates the message into our own format. Then, the module sends the message to a Switch Proxy if a connection state to the Switch Proxy is active, or to other Controller Proxies if the connection state is checking or inactive. We use a publish-subscribe pattern for communication between Controller Proxies, and use ØMQ[2] for implementation. The Decap module works for duplicate filtering, decapsulation, and sending OpenFlow messages.

4.6 Evaluation

We have measured how much failure detection delay becomes short, and the overhead caused by our proposed mechanism in terms of message processing throughput in controllers and latency of delivering messages between controllers and switches.

An environment for the evaluation is shown in Fig. 4.3. One Switch Proxy and two Controller Proxies (Main and Sub) are connected via a Packet Filter. An OpenFlow Switch is connected to the Switch Proxy, and the Controller Proxy (Main) establishes a

[2]http://zeromq.org/

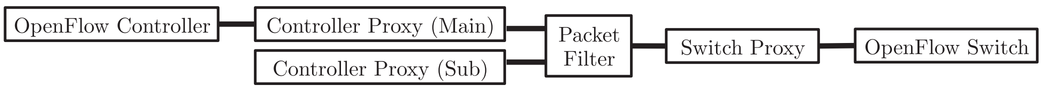


Figure 4.3: Connections for evaluation

Table 4.1: Specifications of PCs used for evaluation

	CPU	RAM	OS
Controller	Intel Core 2 Duo T7400	4 GB	Ubuntu 12.04
Controller/Switch Proxy	Intel Core 2 Duo T7400	512 MB	Ubuntu 14.04.2
Switch	Intel Xeon X5255	8 GB	CentOS 6.2
Packet Filter	Intel Atom C2358	4 GB	Ubuntu 14.04.2

channel to the OpenFlow Controller. Specifications of PCs are summarized in Tab. 4.1. The Packet Filter uses Open vSwitch 2.0.1 as a software switch. All links are directly connected at 1Gbps, and the latency between them is less than 1 msec. To induce channel failures between the Switch Proxy and the Controller Proxies, we install rules in the Packet Filter to discard packets from the Switch Proxy to Controller Proxies.

4.6.1 Failure Detection Delay

To confirm that our proposed mechanism shortens failure detection delay, we have measured the time to detect a channel failure between the Switch Proxy and a Controller Proxy. Intervals of sending an echo request message and timeout values until receiving an echo reply message are summarized in Tab. 4.2. Table 4.2 also includes default values of the intervals and the timeout values in some controllers and switches[3]. A Controller Proxy sets one second as a timeout value until receiving the same message from a Switch Proxy after the Controller Proxy receives a notification from another Controller Proxy.

The procedure of the measurement is as follows. A measurement server collects and monitors logs from the proxies, and controls the switch and the rules in the Packet Filter.

1. The Switch Proxy intercepts an echo reply message, and notifies the measurement server of the message.

2. One second later, the measurement server installs rules in the Packet Filter to discard packets from the Switch Proxy to Controller Proxies.

[3]We cannot find the keep-alive procedure in Ryu by inspecting the source code of Ryu and by executing Ryu.

Table 4.2: Keep-alive intervals and timeouts

		Interval	Timeout
Switch	For evaluation	5 sec	5 sec
	Open vSwitch 2.3.1	5 sec	5 sec
	NEC PF5240	3 sec	9 sec
Controller	For evaluation	60 sec	2 sec
	Trema 0.4.6	60 sec	2 sec
	Ryu	None	None
	ONOS	20 sec	10 sec
	Floodlight	2 sec	30 sec

3. The Packet Filter notifies the measurement server when the installation is completed.

4. The measurement server asks the switch to send a port status message.

5. The proxies detect channel failures between them, and notify the measurement server of the failures. When the controller or the switch closes a channel, the proxies notify the measurement server of the termination of the channel.

A measurement server appends timestamp to all logs. We regard the detection delay as duration between 4 and 5, and executed the above procedure 10 times.

For the evaluation of the Partial failure at the Controller Proxies, the Packet Filter discards only packets from the Switch Proxy to the Controller Proxy (Main). For the evaluation of the All failure at the Switch Proxy, the Packet Filter discards packets from the Switch Proxy to both the Controller Proxies.

The results are summarized in Tab. 4.3. Without our proposed mechanism, the delay includes the duration until next transmission of an echo request message in addition to the timeout value. We can see that the delay is reduced to the timeout value with our proposed mechanism.

4.6.2 Overhead in Throughput and Latency

We have evaluated overhead caused by our proposed mechanism in terms of message processing performance in controllers and latency on delivering messages. We run cbench [76] that fakes one switch on the switch, and a cbench controller in Trema [84] at the controller. An RTT between two Controller Proxies is about 0.2 msec.

Table 4.3: Average failure detection delay

		Delay
Switch	With our proposed mechanism	5.0 sec
	Without our proposed mechanism	9.0 sec
Controller	With our proposed mechanism	1.0 sec
	Without our proposed mechanism	61.0 sec

Cbench transmits one Packet-In message at the same time in a latency mode. When cbench receives a flow_mod message that corresponds to the Packet-In message, the cbench sends another Packet-In message. In a throughput mode, cbench sends multiple Packet-In messages at the same time, and counts the number of flow_mod messages that correspond to the Packet-In messages.

Table 4.4 shows the throughput and latency on average of 10 seconds. Failure Detection Disabled means we disabled our proposed mechanism. No Channel Failure means both connections between Switch Proxy and Controller Proxies are available. In Failure cases, a connection to a Controller Proxy mentioned is failed.

In terms of latency, the case “Failure: Controller Proxy (Main)” took 1 msec longer than other cases. This is because the messages were transferred via Switch Proxy - Controller Proxy (Sub) - Controller Proxy (Main). In other cases, the messages were transferred only via Switch Proxy and Controller Proxy (Main).

In terms of throughput as a whole system, we measured 25 percent decrease for the “No Channel Failure” case from the “Failure Detection Disabled” case, and 5 to 10 percent decrease for Failure cases. In the “No Channel Failure” case, Controller Proxy (Main) received 1.5 times more messages than in the “Failure Detection Disabled” case from Switch Proxy, Controller, and Controller Proxy (Sub). The overhead caused by components for our failure detection mechanism can be measured by comparing “Failure Detection Disabled” case and Failure cases, and the overhead seems 10 to 15 percent.

4.7 Discussion

We have confirmed that our proposed mechanism can reduce failure detection delay to the timeout value because our proposed mechanism starts checking a state of a channel immediately. By properly reducing the timeout, we can detect a channel failure within acceptable delay, regardless of intervals of exchanging keep-alive messages.

Table 4.4: Average throughput and latency

	Throughput	Latency
Failure Detection Disabled (Controller Proxy (Main) only)	2491.5 flows/sec	1.2 msec
No Channel Failure	1845.0 flows/sec	1.2 msec
Failure: Controller Proxy (Main)	2097.6 flows/sec	2.1 msec
Failure: Controller Proxy (Sub)	2201.7 flows/sec	1.3 msec

We cannot see any overhead in latency when the Switch Proxy and the Controller Proxy (Main) connected to a controller directly establish a connection. When a connection between the Switch Proxy and the Controller Proxy (Main) is failed, the latency is increased because messages are transferred via the Controller Proxy (Sub). This route adds two kinds of delay to the overall latency, encoding and decoding messages in the Controller Proxy (Sub), and latency between two Controller Proxies. If the latency between the Switch Proxy and the Controller Proxies and the processing time in the controller is longer than the latency between the Controller Proxies, this overhead is negligible.

In terms of the overhead in throughput, an increase of the number of received messages at the Controller Proxy (Main) reduces the overall throughput. However, the difference between the "No Channel Failure" and "Failure: Controller Proxy (Sub)" cases shows that the Controller Proxy (Main) takes less time to process additional messages notified by the Controller Proxy (Sub) in the "No Channel Failure" case. This is because the difference is around 10 to 15 percent, although the number of received messages is 1.5 times larger.

One of problems to shorten an interval of exchanging keep-alive messages is the number of messages that controllers process. If switches and controllers try to shorten failure detection delay to the timeout value without our proposed mechanism, the switches and the controllers should always exchange keep-alive messages. When the switches generate messages other than ones for the keep-alive mechanism at a low rate, our proposed mechanism works well without significantly increasing loads on controllers. This is because our proposed mechanism generates additional messages only when a message is sent or received and additinal messages by our proposed mechanism can be handled lighter than other messages.

Our proposed mechanism is not suitable to networks where the switches and the controllers generate messages at high rate, sometimes more messages than keep-alive messages. In this situation, we will need to address the problem in a different approach like an heartbeat technique.

Controllers exchange a lot of messages with other controllers to detect the Partial failure if switches establish channels to many controllers. It is better to reduce the number of controllers where switches connect. If this solution is impossible, the controllers may be able to reduce the number of messages exchanged between them by notifying other controllers only when a controller has not received notification from other controllers like Trickle algorithm [46].

We do not note anything about the design of controllers. Some controllers like ONOS [6] have a function to elect a master controller for each switch, or other controllers may have another coordination function. We assume that some parameters for using our proposed mechanism, such as which controllers have channels to a switch, a state of each channel, or which channel controllers use for sending request messages, are provided by controller platforms. The controllers will run their coordination algorithm to choose such parameters.

4.8 Concluding Remark

We propose a mechanism to detect failures of OpenFlow channels quickly with fewer messages, compared to a simple and periodic keep-alive mechanism. It is important to detect channel failures both in switches and in controllers. Switches need to detect only the All failure, and the switches send a keep-alive message to each channel just after sending important asynchronous messages. Controllers can detect which channel is suspected to be unavailable by sharing which messages have arrived at which controllers. To filter out duplicate messages and reorder messages, we assign a switch-local sequence ID to each asynchronous message at switches. Our evaluation shows that failure detection delay is reduced to the timeout value and the overhead in latency is negligible. The overhead in throughput cannot be negligible, but this is due to the increased number of messages at controllers, and such messages can be processed lighter than other messages.

Chapter 5

Protection of OpenFlow Switches against Many Unknown Packets

5.1 Introduction

Such network devices as Ethernet switches commonly forward most packets at high speed by ASICs, and its control software runs on low performance CPUs. Some functions in network devices, which are designed under an assumption that they are rarely used, are implemented and executed in the control software; however, in many cases these functions are executed frequently. Hosts sometimes send an unexpected amount of traffic, and some of such traffic may be handled at these functions. Some networks need to use these functions to meet requirements for the networks, and many hosts use these functions at the same time. In these cases, the control plane in the network devices becomes overloaded, operation becomes unwieldy, and network devices and operators often suffer from high loads caused by such traffic.

Let me assume in the context of SDN and OpenFlow that a host suddenly starts to send too many packets without advance notice, and switches are configured to send Packet-In messages including packets that miss any flow entry to controllers, as illustrated by Fig. 5.1. There is a small time lag in switches between sending a Packet-In message and adding new flow entries that correspond to it. A switch sends Packet-In messages for all missed packets to controllers during this time lag. As a result, a switch may be overloaded by generating and sending many Packet-In messages, delaying the handling of OpenFlow messages from the controllers. If the data plane restricts the rate of packets that go to a CPU of the switch to reduce a load on the CPU, packets from one host (Host

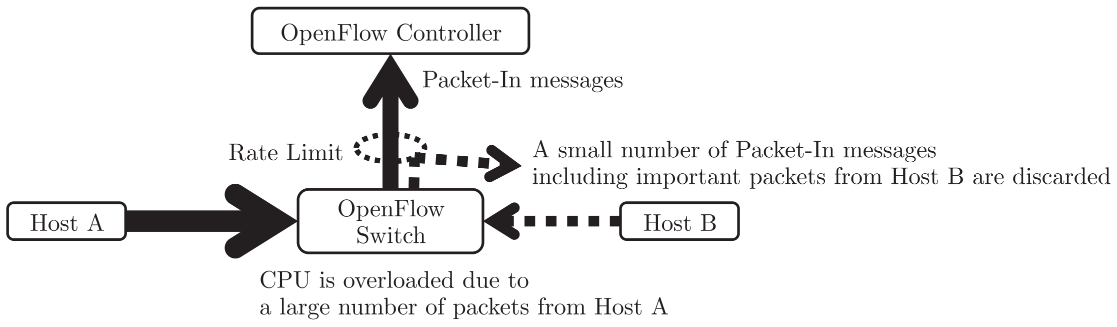

OpenFlow Controller
Packet-In messages
Rate Limit
A small number of Packet-In messages
including important packets from Host B are discarded
Host A
OpenFlow
Switch
Host B
CPU is overloaded due to
a large number of packets from Host A

Figure 5.1: Problems in OpenFlow switches when a large number of packets are suddenly sent

A in Fig. 5.1) may occupy the low bandwidth between the data plane and the CPU, and the data plane may discard packets from other hosts (Host B in Fig. 5.1), which include important packets for network control. As a result, the controllers would be failed to learn necessary network state.

A traditional method to mitigate this problem is to offload more packet processing onto hard-wired offload engines like ASICs, Network Processors and FPGAs [9, 50]. However, this choice is unsuitable for SDN based on OpenFlow because programming these engines is very different from developing control software, and switches does not necessarily have such engines. Thus, the flexibility provided by OpenFlow is limited by introducing such engines. Many OpenFlow extensions try to expand use cases of OpenFlow [13, 60, 77, 78, 86, 88, 93], but it is unpractical that a switch supports all the extensions for all the potential cases. For this reason, we believe that the switches need a mechanism that protects the control plane, especially a CPU on the switch, while preserving flexibility provided by OpenFlow.

Another method is to classify normal and abnormal packets and apply restrictions to abnormal packets when most of packets are abnormal. This method is used for the control plane protection in traditional network devices [20] and Denial of Service (DoS) protection [52]. A key of this method is how to classify packets. The traditinal network devices distinguish packets of control protocols such as routing protocols from all packets. The DoS protection requires more sophisticated classification mechanisms to identify DoS related packets. In SDN and OpenFlow, this classification mechanisms should have flexibility of configuration to allow various control mechanisms in OpenFlow controllers.

In this chapter, we propose a mechanism to classify packets included in Packet-In messages into important packets and others so that the switches surely send important

Packet-In messages for network control and apply restrictions on less important Packet-In messages. The restrictions are selected by network managers, like filtering out Packet-In messages or greatly limiting bandwidth consumption for Packet-In messages.

The most important work of controllers is to control networks. This work includes learning data necessary for network control, inserting and deleting flow entries quickly, etc. To continue this work, switches must not discard important packets for network control from which the controllers learn necessary data, while the switches discard many packets that bring overloads in the control plane, especially in the switches with low CPU performance.

Although switches include an entire packet in a Packet-In message, controllers use only a part of header fields in the packet, and it depends on design of controllers that which header fields are necessary to learn data. If two or more packets have the same values in necessary header fields, one packet is enough for the controllers to learn values, and the others are less important.

A basic approach of the proposed mechanism is simple: switches record values of header fields when they send Packet-In messages, and apply predefined restrictions on subsequent packets that match recorded entries for a certain period of time. To avoid an explosion of the number of entries the switches record, controllers set the switches the header fields that the controllers use before the switches start to process packets. The switches record only values of the header fields that the controllers have specified, and other fields are set to be wildcard.

The proposed mechanism consists of two parts: Pending Flow Rules and Pending Flow Tables. A pending flow rule is an entry to specify the header fields that controllers use, and each rule consists of a condition to match and a list of header fields to record. The pending flow rules are assumed to be set before switches start to process packets. A pending flow table is a list of entries where switches record values of header fields in packets included in Packet-In messages, and we call its entries pending flow entries. One pending flow table per pending flow rule is used. A pending flow entry consists of a condition to match, and the values of the header fields specified by the pending flow rule are copies from packets to the condition.

This chapter is organized as follows. Section 5.2 introduces related work. We explain the proposed mechanism in Sec. 5.3, and its implementation to Open vSwitch by extending OpenFlow's standard mechanisms in Sec. 5.4. In Sec. 5.5, we show how it can be used in various cases, and describe experiments to show that it dramatically reduces

CPU utilization in switches while forwarding important Packet-In messages to controllers. Section 5.6 discusses our proposed mechanism, and Sec. 5.7 concludes this chapter.

5.2 Related Work

This section surveys related work for protection of control planes in traditional network devices, and DoS attacks where a situation we assume are categorized, and expanding capability of OpenFlow switches.

5.2.1 Protection of Control Planes in Traditional Network Devices

Traditional network devices have a similar problem that the control plane is overwhelmed by many undesired or malicious traffic, such as packets that are addressed to routers and that cause ICMP errors. RFC 6192 by Dugal et al. [20] summarizes approaches to protect the control plane in traditional network devices as follows.

The identification of legitimate traffic is done manually by network managers. They carefully examine protocols and IP addresses used for network control and management, such as OSPF, BGP, SSH and NTP. Restrictions are mainly implemented as traffic policing. The policy filter allocates more bandwidth to legitimate traffic and low bandwidth to other traffic. Traffic that routers obviously do not need to handle are configured to be discarded.

OpenFlow networks need to have similar restriction mechanisms, but the identification must be done in a different way. Traffic that goes to the control plane in OpenFlow networks are not limited to packets addressed to routers or controllers, that is, some packets that should go to other hosts will also be intercepted and sent to controllers as Packet-In messages for setting up paths, learning a state of a network, etc. In addition, some packets processed at ASICs in traditional network devices will also be handled at OpenFlow controllers. The mechanisms must classify such packets into important ones for network control or less important ones.

5.2.2 DoS Attacks

There are lots of works to prevent, detect, and filter Denial of Service attacks on the Internet [56, 67], like bandwidth consumption attacks. A problem of generating a lot of Packet-In messages in OpenFlow switches is a similar situation with bandwidth consumption attacks because the problem consumes most of bandwidth from the data plane to the CPU of a switch.

For detecting DoS attacks and identifying DoS related packets, sophisticated algorithms against bandwidth consumption attacks like ACC [52] need to count packets that pass through links in some way before installing a rule to filter out DoS related packets. We cannot prevent the switches from overwhelming Packet-In messages by these algorithms. When a lot of Packet-In messages are being generated, both the CPU of a switch and the channel between the data plane and the CPU of a switch are overloaded, and a switch has almost no room to execute actions for counting and filtering out packets. Therefore, we need a way to set a rule to filter out many Packet-In messages to the data plane without counting any of them.

5.2.3 Extending Capability in OpenFlow Switches

To reduce loads on the CPU of switches, more and more packets should be processed at the data plane, and extending capability of the data plane to process packets contributes to it. Many extensions are proposed, such as Information-Centric Networking support [13, 86], flexible sampling actions [78, 88], and multiple output ports in one action [60]. These extensions are usually designed for such specific use cases as access policy enforcement and traffic monitoring, but it is impractical to design, implement, and deploy new switch functions every time a new use case or extension emerges.

DevoFlow [16] shows that the switches have meager performance for installing flow entries, and proposes a clone flag in the flow entries mainly for elephant flow detection. The clone flag shows that, when a packet matches the flow entry, the switch creates and installs a new flow entry by copying the same actions as the matched flow entry and the values of all the header fields in the packet to the match fields. Luo et al. [51] have proposed for protection of the control plane by recording specific (source, destination) address pairs of packets included in Packet-In messages and by filtering out Packet-In messages that match recorded values. A serious limitation in DevoFlow [16] and work by Luo et al. [51] is that the number of entries generated by these mechanisms is rapidly increased and space to store these entries in switches like TCAM or RAM overflow easily, for example, by port scanning, although controllers may not need to some values recorded. Therefore, we need a generic control plane protection mechanism from many Packet-In messages as a last resort.

AVANT-GUARD [77], which is also a work for protection of the control plane, mainly from TCP SYN flooding attacks by a SYN cookies approach [17]. In real networks, hosts often send such packets other than TCP as UDP and ARP, and such protocols can also overload the control plane.

OpenFlow Switch Specification 1.3 [63] and later have a mechanism called Meter that limits the rate of packets in a group of flows. We can use Meter as one of the restrictions to limit the rate of Packet-In messages, but applying Meter to all Packet-In messages causes a problem that Packet-In messages that are important for network control are discarded.

5.3 Proposed Mechanism

In this section, we discuss which packets can be filtered out, then propose a mechanism to filter out packets that can be filtered out with an example.

5.3.1 Important Packet-In Messages for Network Control

To versatileley reduce loads on switches that are caused by too many Packet-In messages, we discuss which Packet-In messages are candidates for being filtered out by the switches. This classification must be done without asking controllers per packet.

The most important role of the controllers is to control networks like insertion and deletion of flow entries, and switches should not discard packets that include important data for control like ones used for flow entries. From this point of view, if the controllers extract and store the same data from several packets, the controllers only need one of such packets, and the switches can discard the others. The switches can also discard other packets that include only unnecessary header fields. When designing a mechanism on the switches to process Packet-In messages in this way, we need to consider the following points that current OpenFlow specifications [1] do not handle.

The first is that the switches must send a Packet-In message including a packet that arrives first rather than ones that arrive second or later, which have the same values in their header fields, to the controllers so that the controllers can obtain necessary data as soon as possible. OpenFlow expects that the controllers set flow entries quickly when a packet matches no flow entry so that subsequent packets can be processed only at the

[1]The latest version is 1.5.1 [64] at the time of writing.

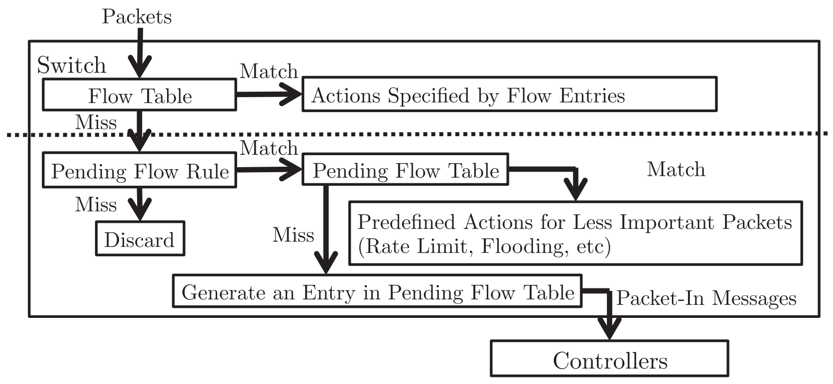

Packets
Switch
Flow Table
Match
Actions Specified by Flow Entries
Miss
Pending Flow Rule
Match
Pending Flow Table
Match
Miss
Discard
Miss
Predefined Actions for Less Important Packets (Rate Limit, Flooding, etc)
Generate an Entry in Pending Flow Table
Packet-In Messages
Controllers

Figure 5.2: Conceptual diagram for the proposed mechanism

data plane. If modification of flow entries is delayed, subsequent packets are also sent as Packet-In messages, and hosts cannot start communicating quickly. This is undesired behavior.

The second one is that each controller must use different set of header fields in packets, which depends on controller design. This means that many controllers use some, not all, the header fields. For example, if a controller correlates an Ethernet address with a switch and a port, it needs to learn only a source Ethernet address from a packet. Source and destination IP addresses are adequate for a load balancing function using IP addresses.

In the following section, we explain our proposed mechanism that filters out Packet-In messages including packets that have the same values in header fields with previous Packet-In messages, while considering the header fields to which the controllers use.

5.3.2 Overview of Proposed Mechanism

Figure 5.2 shows a conceptual diagram of an overview of our proposed mechanism. Our extensions are shown below the dotted line. We have add two components to switches: Pending Flow Table and Pending Flow Rule. First, a switch looks up entries that match a packet in the flow table, then in the pending flow rule, and finally in the pending flow table of the matched rule. The switch discards packets that do not match both the flow table and the pending flow rule. Packet-In messages are generated when packets that do not match the flow table, match the pending flow rule, and do not match the pending flow table of the matched rule.

A switch records values of header fields in packets into pending flow tables when the switch sends Packet-In messages including the packets to controllers. The switch regards packets that match pending flow entries as less important ones for network control, and apply the predefined actions to them to avoid generating Packet-In messages including them.

The pending flow rules specify header fields whose values should be recorded in the pending flow tables. One pending flow rule includes a list of the header fields whose values are recorded and a pending flow table with entries generated by the rule. The header fields in the list include those which controllers use, and the controllers install pending flow rules before a switch starts to process packets. If a pending flow rule matches a packet but the switch cannot find any pending flow entry in the associated table, the switch creates an pending flow entry from the matched rule and the packet, and installs it into the table of the matched rule.

The predefined actions, which are applied to less important packets, must be executed at the data plane to reduce the amount of packets handled by the CPU of switches. Network managers choose the actions based on their policies. If they want to minimize packet loss, for example, a packet flooding action or an action to severely limit the bandwidth of Packet-In messages will be appropriate. If they are not concerned with a small amount of packet loss, they will select a discard action.

5.3.3 Pending Flow Rules

With pending flow rules, controllers inform switches of header fields that are important for network control, and the switches avoid explosively increasing the pending flow entries by not recording values in less important header fields. Each rule consists of the following elements.

Priority Priority value of the rule.

Match Fields List of header fields and their values for a match condition to packets.

Clone Fields List of header fields whose values are recorded in the pending flow table.

Timeout for Table Timeout value set to pending flow entries created by the rule.

Pending Flow Table List of entries where values of the header fields are recorded by the rule.

Actions List of actions to apply to packets matched with entries in the pending flow table of the rule.

Timeout for Rule Timeout value of the rule.

Statistics Packet counters, duration until expiration, etc.

The match fields and the priority values closely resemble those of flow entries in OpenFlow. The match fields include values of header fields that matched packets have, and a wildcard is allowed in each field. If a packet matches multiple rules, a rule that has the highest priority among the rules is applied.

The clone fields are header fields whose values must be recorded in switches. The clone fields include all the header fields that are not set to be wildcards in the match fields of the rule, in addition to some header fields that are wildcards in the match fields. When a switch creates a new pending flow entry in the table associated with the matched rule, values of header fields listed in the clone fields are copied from the matched packet to the match fields of the new entry, and the other fields are set to be wildcards.

There are two timeout values. Timeout for Rule should be long or infinite because the main attributes of the rule, the match fields and the clone fields, are determined only by controller design, and they should be static. Timeout for Table should be small as explained in Sec. 5.3.4.

The statistics also resemble ones of flow entries. The statistics provide information for controllers to determine whether to execute some functions related to managemenet of the rules, such as evicting the rules from switches when space for storing the rules in switches becomes full.

5.3.4 Pending Flow Tables

A pending flow table is a list of pending flow entries to store values in header fields of packets that a switch has already included in Packet-In messages sent to controllers. Since we regard the packets that match the pending flow entries as less important ones for network control, we expect that network managers pre-configure actions for these packets, called Predefined Actions in Fig. 5.2, so that the switch sends fewer of them to the controllers so much. The following are examples of predefined actions: limiting the rate of Packet-In messages including these packets, flooding these packets to a network, and discarding them. We assume that these actions are executed only at the data plane to prevent switches from being overloaded.

A pending flow entry resembles the flow entries, and consists of the following elements.

Match Fields List of header fields, its values, and an input port number that matched packets have. A wildcard is allowed in each field.

Timeout Timeout value until removal of an entry after the entry is inserted.

Statistics Packet counters, duration until expiration, etc.

The contents of the match fields are generated according to the parent pending flow rule. Values of header fields included in the clone fields of the parent rule are copied to the match fields in the entry from the matched packet. The header fields are set to be wildcards if the clone fields in the parent rule do not include them.

A Timeout value should be small. Packet-In messages including important packets may be lost due to other reasons that we are not trying to solve, such as queue overflow, even when the data plane filters out less important packets by the proposed mechanism. In general, hosts resend lost packets at an interval of one or more seconds if necessary. To avoid situations where controllers do not receive important packets for a long time, we need to assure that the controllers can receive important packets retransmitted by the hosts without being filtered out by the pending flow tables. In addition, once the controllers successfully install the new flow entries generated by the Packet-In messages, pending flow entries whose match fields overlap with the match fields of the new flow entries are not used any more. Therefore, the pending flow entries must be deleted shortly so that retransmitted packets do not match any entry, and unused entries are not left for a long time.

The statistics are the same as the flow entries and used for the same purposes as the pending flow rules.

Entries are added when a packet matches a pending flow rule but does not match any entry in the pending flow table of the matched rule, and entries are removed when expired.

5.3.5 An Example of Pending Flow Rules and Tables

Table 5.1 shows an example of pending flow rules and a pending flow entry generated by a packets and the rules. The rules are designed for a controller that manages hosts by IP addresses, and a switch forwards IPv4 packets by IPv4 addresses and other packets by Ethernet addresses.

Table 5.1: An example of a packet, pending flow rules, pending flow tables (in part)

		Priority	In Port	Frame Type	Src Eth Address	Dest Eth Address	Src IP Address	Dest IP Address	Actions	Timeout
Packet		-	1	0x0800	00:00:5e:00:53:01	00:00:5e:00:53:02	192.0.2.1	203.0.113.1	-	-
Rule 1	Match	2	*	0x0800	*	*	*	*	-	∞
	Clone	-	Yes	Yes	Yes	Yes	Yes	Yes	Packet-In with Rate Limit	1 sec
	Table	-	1	0x0800	00:00:5e:00:53:01	00:00:5e:00:53:02	192.0.2.1	203.0.113.1	Packet-In with Rate Limit	1 sec
Rule 2	Match	1	*	*	*	*	*	*	-	∞
	Clone	-	Yes	No	Yes	Yes	No	No	Packet-In with Rate Limit	1 sec
	Table	-	-	-	-	-	-	-	-	-

Rule 1 is for controllers to learn hosts' IP addresses, Ethernet addresses, and input port numbers. A switch records an input port number, a Frame Type (0x0800, IPv4), and source and destination Ethernet and IP addresses in the Rule 1's pending flow table. A controller sets flow entries using these parameters to forward packets by IP addresses. Rule 2 is for other packets and does not record values of IPv4 related header fields (Frame Type and IP addresses). These rules do not automatically expire, and the pending flow entries expire one second after insertion (Timeout of Rule - Clone in Tab. 5.1).

When Packet in Tab. 5.1 arrives at a switch and no flow entry matches it, the switch looks up the pending flow rules, and both Rules 1 and 2 match the Packet. The switch chooses Rule 1 because of higher priority than Rule 2. Then, the switch creates a pending flow entry for Rule 1 by copying values of header fields marked "YES" in "Rule 1 - Clone" from the Packet and the Timeout for Table from Rule 1, and sends a Packet-In message including the Packet without any restriction. When the switch receives subsequent packets that have the same Ethernet and IP addresses as the Packet within 1 second after insertion of the Rule 1's pending flow entry, the switch limits the rate of generating and sending Packet-In messages including such packets without adding new pending flow entries.

When packets other than IPv4 ones miss any flow entry, the switch records their source and destination Ethernet addresses in the Rule 2's pending flow table. The rate of generating Packet-In messages including subsequent packets that have the same source and destination Ethernet addresses is restricted for 1 second by the Rule 2's pending flow table.

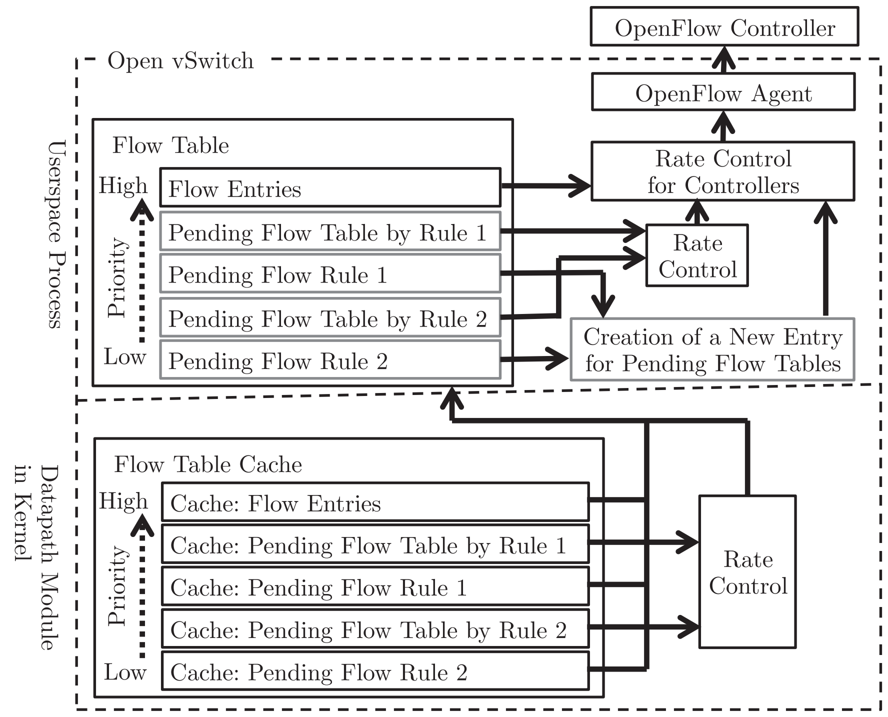

OpenFlow Controller
Open vSwitch
OpenFlow Agent
Userspace Process
Flow Table
High
Flow Entries
Rate Control for Controllers
Pending Flow Table by Rule 1
Rate Control
Priority
Pending Flow Rule 1
Pending Flow Table by Rule 2
Creation of a New Entry for Pending Flow Tables
Low
Pending Flow Rule 2
Datapath Module in Kernel
Flow Table Cache
High
Cache: Flow Entries
Cache: Pending Flow Table by Rule 1
Rate Control
Priority
Cache: Pending Flow Rule 1
Cache: Pending Flow Table by Rule 2
Low
Cache: Pending Flow Rule 2

Figure 5.3: A design of our prototype switch based on Open vSwitch

5.4 Implementation of a Prototype Switch

We implemented our proposed mechanism in Open vSwitch (commit 4ca808d) [65]. Figure 5.3 shows an overview of architecture of Open vSwitch and our modifications (marked in gray). Open vSwitch consists of two parts: a datapath module and a userspace process. The datapath module handles packets in the kernel using the flow table cache. Packets that miss the flow table cache are passed to and handled by the userspace process. The userspace process also manages the flow table cache in the datapath module, and manages OpenFlow channels to controllers (OpenFlow Agent). Packets that miss both the flow table cache in the datapath module and the flow table in the userspace process are included in Packet-In messages at the OpenFlow Agent.

We translated the two components we added in Fig. 5.2, Pending Flow Rules and Pending Flow Tables, into one flow table to avoid increasing the matching overhead introduced by the these components. Each flow entry corresponds to an entry in the pending flow rules or tables. The flow entries for the pending flow tables have actions to limit the rate of the packets and to send to controllers, which we have selected as the predefined actions for less important packets. The flow entries for the pending flow rules have a new action we added to insert a new pending flow entry, and an action to send a

Packet-In message to controllers.

To get the same results as switches that search for the flow tables, the pending flow rules, and the pending flow tables in the order (indicated by arrows in Fig. 5.2), we assign different priority to each component. We set the highest priority to the flow entries by the standard OpenFlow, and assign lower priority to sets of the pending flow rule and table. The sets of a pending flow table and rule are arranged in the order of the priority of the pending flow rules. Within the set, higher priority is assigned to the table, and the rule has lower priority.

In Fig. 5.3, Pending Flow Rule 1 has higher priority than Rule 2. In the flow table, the flow entries for the pending flow table of Rule 1 have higher priority than the flow entry for Rule 1. Similarly, priority is assigned to the flow entries related to Rule 2.

The new action we added is to insert a new flow entry as an entry of the pending flow table associated to the rule that matches a packet. The new action has the following parameters: the clone fields explained in Sec. 5.3.3, timeout and priority values for the new flow entries, and a Meter ID for limiting the rate of packets that match the new flow entries. When this action is executed, a switch installs a new flow entry for the pending flow table with the priority and the timeout values in the action, the match fields whose values are copied from a packet according to the clone fields, and the predefined actions, which is to send Packet-In messages in the matched packets to controllers through Meter specified by the Meter ID in our case.

Controllers can set pending flow rules like flow entries. A flow entry works as a pending flow rule if it has the new action we added, lower priority than the flow entries for the pending flow table of the rule, and higher priority than the pending flow tables of the rules with lower priority.

The prototype switch uses a rate limiting mechanism in two ways. One limits the rate of the Packet-In messages in total, which is shown as Rate Control for Controllers in Fig. 5.3. All Packet-In messages sent by a switch pass through this mechanism. In original Open vSwitch, this mechanism has a queue of Packet-In messages per port to prevent a queue of Packet-In messages from being overwhelmed by many packets from one port. We modified it to use only one queue because the proposed mechanism provides a very similar function in a more generic way. The other limits the rate of packets that match pending flow entries; this is shown as the Rate Control in Fig. 5.3. There are two Rate Controls for the pending flow tables: one in the datapath module and the other in the userspace process. Packets that match the pending flow tables in the flow entry cache go through the Rate Control in the datapath module. Other packets are processed at the userspace process, and go through the Rate Control in the userspace process if necessary.

We implemented our own simple rate limiting mechanism for Rate Control because Meter was not implemented in Open vSwitch when we implemented the prototype switch. Our simple rate limiting mechanism can be replaced with Meter.

5.5 Evaluation

We have evaluated applicability in typical uses cases, how much loads on CPUs of switches are reduced, and analysis of execution time of each component in our prototype switch to see how our proposed mechanism changes the processing of switches.

5.5.1 Use Cases and Pending Flow Rules

Controllers determine how OpenFlow networks forward packets, but they often see Ethernet addresses to implement Ethernet switching, IP addresses to implement IP routing, and TCP and UDP port numbers for Network Address and Port Translation (NAPT). We can use our proposed mechanism in these scenarios as follows, and the number of pending flow rules required is summarized in Tab. 5.2.

Ethernet Switching: A controller that offers an Ethernet switching function must learn an association of a port where a host is connected and an Ethernet address. It sets the flow entries that have input port numbers and source and destination Ethernet addresses in the match fields to forward packets between known hosts in the data plane. In this case, we use only one pending flow rule; the controller sets a pending flow rule that records an input port number and source and destination Ethernet addresses.

IP Routing: A controller offering IP routing must see IP addresses in packets to associate an IP address with an Ethernet address, and it must handle ARP packets in IPv4 and Neighbor Discovery packets in IPv6. The controller must also provide an Ethernet switching function to forward packets other than IP. This example represents cases where a switch sees header fields of protocols in multiple layers, and we use priorities. In IP Routing case, five pending flow rules are used. We set two rules with the highest priority, which match ARP and Neighbor Discovery packets. These rules copy values in header fields of ARP and ICMPv6 for Neighbor Discovery to the match fields of new pending flow entries. The second highest priority is assigned to two rules that match other

Table 5.2: Number of pending flow rules in typical use cases

	# of Rules
Ethernet Switching	1
IP Routing	4 + 1 (Ethernet Switching)
NAPT	1 per source IP prefix + 5 (IP Routing)

IPv4 and IPv6 packets. These rules set both source and destination IP and Ethernet addresses to the match fields in new pending flow entries. The lowest priority is assigned to one rule that matches other Ethernet frames, and the rule is the same as the case of Ethernet Switching.

NAPT: NAPT is an example of functions where a controller sees header fields in the transport layer. Controllers with the NAPT function set one pending flow rule with the highest priority per source IP address prefix to which the NAPT function is applied. The rule matches all packets whose source IP addresses are in the range to apply the NAPT function, and the clone fields of the rule include an IP protocol number, source and destination ports, and IP and Ethernet addresses. Lower priority is assigned to the rules that handle IP and Ethernet packets as explained for IP Routing. A similar discussion can be applied to other cases, such as a server load balancing function.

5.5.2 Loads in Switches

With our prototype switch, we have evaluated how much the proposed mechanism reduces a load on a switch, how many important Packet-In messages a controller can receive, and how many pending flow entries a switch has.

To measure the above points, we simultaneously sent two kinds of UDP packets to the switch. One emulated where many packets were sent from a host without any advance notice, and called High Rate Packets. The other emulated where a small number of packets were sent from several hosts as usual called Low Rate Packets.

To measure a load caused to the switch until new flow entries are installed, a controller used in this evaluation did not install any flow entry, and the switch continued to send Packet-In messages. This is against a normal situation where new flow entries are installed soon and subsequent packets are processed at the data plane, but this evaluation scenario makes us measure the load before flow entries are installed clearly. We monitored the CPU and memory utilization, the number of pending flow entries, and the traffic to the controller every second by a process running on the switch. We also counted the number of Packet-In messages by the High and Low Rate Packets separately at the controller.

We assume that a controller forwards packets by IP addresses, and we compare three cases for pending flow rules as follows.

Rule - Host The clone fields of the rule installed include only Ethernet and IP addresses.

Rule - Flow The clone fields of the rule installed include all header fields including Ethernet and IP addresses, and UDP port numbers. The UDP port numbers are not necessary for the controller.

No Rule No rule is installed. This case represents the load before flow entries to filter out High Rate Packets are not installed in the case where the flow entries are not installed immediately.

We limit the total rate of Packet-In messages to 100 per second, and limit the rate of Packet-In messages that include packets that match pending flow entries to 50 per second by the predefined actions in the proposed mechanism. The timeout value in the pending flow entries is 1 second, and the pending flow rules are not expired.

High Rate Packets consist of packets with the same source and destination IP addresses and different source and destination UDP ports, and almost all of these packets are less important for network control because the controller does not see UDP port numbers. Most of these packets match the pending flow rule in Rule - Flow, and the pending flow entries of Rule - Host. To observe effects on the loads on the switch by packet rates, we change the rate of High Rate Packets to 1000, 2000, 3000, 4000, and 5000 packets per second. These rates exceed the rate limitation of Packet-In messages to the controller in the switch. Packets in the Low Rate Packets have the same source IP address but different destination IP addresses; these packets are considered important for network control. We send five Low Rate Packets per second. Both High and Low Rate Packets are sent for 30 seconds, and each packet had 128 bytes. We monitored loads on the switch and the Packet-In messages at the controller for 31 seconds including the start and end of transmitting the packets, and omitted the first and last seconds from the measurement logs. The results are averages of 29 seconds.

Figure 5.4 shows an evaluation setting. We used four PCs, two for packet generators, one for our prototype switch, and one for an OpenFlow controller. Specifications of these PCs are summarized in Tab. 5.3. High and Low Rate Packets were sent from

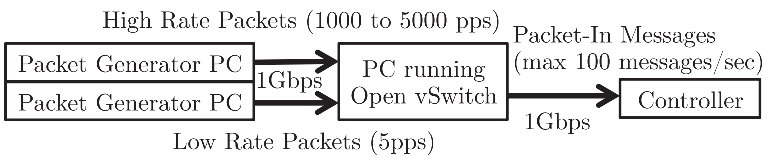


Figure 5.4: Load evaluation setting

Table 5.3: Specifications of PCs used for evaluation

	CPU	RAM	OS
Packet Generator	Intel Core 2 Duo T7400	512 MB	Ubuntu 12.04
Controller	Intel Core 2 Duo T7400	4 GB	Ubuntu 12.04
Switch	Intel Xeon X5255 (down to 1.99 GHz, 1 core)	8 GB	CentOS 6.2

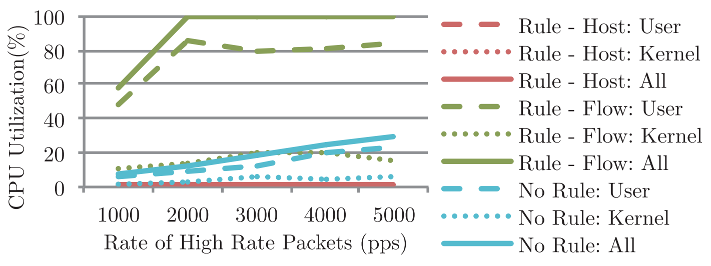


Figure 5.5: CPU load in the switch

different packet generators. Two packet generators and the controller were connected to the switch at 1 Gbps. The maximum length of the queue from the datapath module to the userspace process was 1024 packets. We used Trema [84] and C to implement a controller monitoring Packet-In messages.

Results

Figure 5.5 shows average CPU utilization in the switch. The x-axis shows the rate of packets in High Rate Packets, and the y-axis is the CPU utilization by percentage. *User* denotes those in the userspace process, and *Kernel* means those in the datapath module. We have confirmed that CPU utilization of monitoring process was less than one percent.

When the proposed mechanism is properly configured, the CPU utilization and its increasing ratio to the packet rate in both the userspace process and the datapath module are much lower in the switch with the proposed mechanism (Rule - Host) than without it (No Rule). In Rule - Flow, the CPU utilization is almost 100 percent when the rates of High Rate Packets are 2000 or more, and the figures of the Rule - Flow do not show

Table 5.4: Number of Packet-In messages and pending flow entries per second (min/average/max)

Rate of High Rate Packets		1000	2000	3000	4000	5000
Rule Host	High Rate Packets	30 / 34.4 / 36	32 / 33.5 / 37	30 / 33.8 / 40	29 / 34.3 / 38	31 / 34.1 / 36
	Low Rate Packets	4 / 5.0 / 5	5 / 5.0 / 5	4 / 5.0 / 6	5 / 5.0 / 5	3 / 4.9 / 6
	Pending Flow Entries	6 / 6.0 / 7	6 / 6.0 / 7	6 / 6.3 / 7	6 / 6.0 / 7	6 / 6.0 / 7
Rule Flow	High Rate Packets	99 / 99.9 / 100	98 / 99.6 / 100	100 / 100 / 100	98 / 99.7 / 100	100 / 100 / 100
	Low Rate Packets	0 / 0.1 / 1	0 / 0.4 / 2	0 / 0.0 / 0	0 / 0.3 / 2	0 / 0.0 / 0
	Pending Flow Entries	1006 / 1006.1 / 1008	2007 / 2007.6 / 2010	3013 / 3013.8 / 3018	4013 / 4016.2 / 4022	5021 / 5024.3 / 5035
No Rule	High Rate Packets	100 / 100 / 100	97 / 99.2 / 100	99 / 100 / 100	100 / 100 / 100	100 / 100 / 100
	Low Rate Packets	0 / 0.0 / 0	0 / 0.8 / 3	0 / 0.0 / 1	0 / 0.0 / 0	0 / 0.0 / 0
	Pending Flow Entries	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0

the loads on the switches precisely; they just show the ratio of the CPU utilization of the userspace process and the datapath module. The memory usage averaged 353 MB and did not differ among the rates of the High Rate Packets.

Table 5.4 shows the number of Packet-In messages per second received by the controller, and that of pending flow entries per second at the switch. *High Rate Packets* and *Low Rate Packets* show packets included in the Packet-In messages are from High or Low Rate Packets, and *Pending Flow Entries* is the number of pending flow entries.

In Rule - Flow and No Rule, which do not use the proposed mechanism properly, the packets from the High Rate Packets fill the queue of Packet-In messages to the controller in the switch, and most of packets in the Low Rate Packets are discarded. In Rule - Host, the switch can filter out packets with a few pending flow entries. Some of the maximum number of Low Rate Packets in Rule - Host exceed five, which we sent from the packet generator as Low Rate Packets, and this should be due to a small jitter caused by the switch and the controller.

The traffic between the controller and the switch is very small compared to link speeds in current servers and switches. The traffic from the controller to the switch was around 10 kbps to 50 kbps, and in the opposite direction, about 70 kbps in Rule - Host, and around 180 kbps in Rule - Flow and No Rule.

5.5.3 Processing Time of Actions in Switches

We have also measured execution time of each action in switches to evaluate the overhead introduced by our proposed mechanism. Major modification in our proposed mechanism includes creating an pending flow entry when a packet arrives at a switch, and limiting the rate of packets that match the entries. To evaluate how these modifications affect the time for packet processing, we measured the execution time related to the limitation of the rate of packets and sending Packet-In messages.

In the datapath module, we measured and compared the time to queue a packet to the userspace process (Queue to the Userspace Process), the time to limit the rate of packets (Rate Limit), and the time to output a packet to a port (Output Port) for reference. When the proposed mechanism is used, packets that match the pending flow entries are processed by “Rate Limit” and some packets that pass “Rate Limit” proceed to “Queue to the Userspace Process.” Without the proposed mechanism, the packets are just processed by “Queue to the Userspace Process.”

In the userspace process, we measured the time to create a flow entry for the pending flow tables (Set Flow Entry for Pending Flow Table) and to send a Packet-In message. The process of sending a Packet-In message consists of three components. First, all packets that should be included in Packet-In messages are queued (Enqueue Packet-In messages). Second, all the packets are dequeued and Packet-In messages including them are sent immediately to controllers if a switch gets tokens from the rate limitation mechanism for all Packet-In messages, and the packets overflowed are queued in another line (Dequeue Packet-In Messages and Send). Finally, the packets are sent as a part of Packet-In messages when they get the tokens, or are discarded if they fail to get the tokens (Dequeue Waiting Packet-In Messages and Send).

We used the same PCs as Sec. 5.5.2. We modified the code of the prototype switch to measure the execution time per component. We executed each component 1000 times and calculated the average time (Tab. 5.5). The execution time for other components are not included in the results, such as looking up flow tables.

5.6 Discussion

This section discusses whether our proposed mechanism is really effective, overheads on a switch caused by our proposed mechanism, possibility to implement in hardware, drawbacks of our proposed mechanism including situations where our proposed mechanism

Table 5.5: Execution time by components in switches

Component		Execution Time in nsec (min/average/max)
Datapath	Queue to Userspace Process	3305 / 3511.9 / 6939
	Rate Limitation	75 / 201.9 / 511
	Output to Physical Port	487 / 690.8 / 4045
Userspace	Enqueue Packet-In Messages	2174 / 2440.4 / 5995
	Dequeue Packet-In Messages and Send	12621 / 13505.4 / 38641
	Dequeue Waiting Packet-In Messages and Send	10061 / 11663.9 / 20857
	Set Flow Entry for Pending Flow Table	11968 / 12617.0 / 19536

does not work well and disadvantage from the view of users, possible attack and mitigation to our proposed mechanism, and relationship with existing attack like DDoS and SYN floods.

5.6.1 Effects of Our Proposed Mechanism

As we have expected, the results of our experiments show that the CPU load by the userspace process on the switches using the proposed mechanism properly, which use the rule that records only Ethernet and IP addresses, is dramatically reduced, and the benefits rise as the rate of the High Rate Packets is increased. This is because most of the High Rate Packets match pending flow entries and are filtered out at the datapath module, and the userspace process handles fewer packets than in switches without the proposed mechanism. Although the rate of packets in our experiment are much smaller than the maximum rate in the Gigabit Ethernet, about 1.5 million pps, we infer from the results that these trends are the same because more packets are discarded by the datapath module if a packet rate is increased.

The results of load evaluation (Sec. 5.5.2) show that it is hard to reduce the load by the userspace process on the switches with approaches that do not install a rule to filter out High Rate Packets immediately. The load of No Rule in Fig. 5.5 indicates the load of the switches that process multiple packets in the userspace process before installing a rule to filter out such packets, like sending Packet-In messages to the controller and counting packets for DoS detection. The load of the userspace process in No Rule increases more than with our proposed mechanism as the rate of High Rate Packets increases, and the switches would become overloaded before the rule for filtering out such packets is installed if a lot of packets go to the userspace process. When the switches install the rule to filter out High Rate Packets immediately like our proposed mechanism, the load of the switches would not increase so much like Rule - Host in Fig. 5.5.

In addition to the decrease of the load on the switches by the userspace process, the switches with our proposed mechanism can surely send Low Rate Packets to the controller and reduce the number of Packet-In messages by filtering out High Rate Packets selectively. No Rule and Rule - Flow in Tab. 5.4 indicates that High Rate Packets fill queues to send Packet-In messages to the controller, and there is no room to send Packet-In messages including Low Rate Packets. Rule - Host in Tab. 5.4 shows that our proposed mechanism can filter out only High Rate Packets, and the switches do not need to discard Low Rate Packets.

It depends on networks that how much Low Rate Packets are discarded in practice without our proposed mechanism, such as the number of hosts and the behavior of hosts. Low Rate Packets are discarded when both High Rate Packets and Low Rate Packets are sent simultaneously. According to Tab. 5.4, almost all Low Rate Packets were discarded at a queue from the datapath to the userspace process in our prototype switch because the queue was filled due to High Rate Packets.

Our proposed mechanism can reduce the duration that the queue is filled by High Rate Packets and Low Rate Packets are discarded. High Rate Packets are not filtered out until a flow entry to filter out such packets is installed. Without our proposed mechanism, the following procedure is required to install a flow entry to filter out such packets: (1) A packet goes to the CPU of a switch from the data plane, and the software on the switch generates and sends a Packet-In message. (2) The controller sends flow entries that should be installed to the switch. (3) The switch installs the flow entries sent by the controller in the data plane. Switches with our proposed mechanism installs an entry to filter out High Rate Packets at Step 1, and Low Rate Packets received after Step 1 can be passed to the controller without the effect of High Rate Packets.

According to our experiments (Tab. 5.5), the time to send a Packet-In message and to installs a flow entry to filter out High Rate Packets is in the order of microseconds. It is hard to estimate that how much it takes until a controller sends a response to install flow entries because it depends on controller implementation, but the time is in the order of milliseconds or tens of milliseconds according to Tootoonchian et al. [83] and Chap. 3. Thus, in general, we can reduce the time when a switch discards Low Rate Packets by the response time of the controller.

When many pending flow entries are created like the Rule - Flow case, the proposed mechanism works badly; almost no Low Rate Packets arrives at the controller, and a load on a switch is the highest among the three cases. This is due to a process of matching packets with a number of pending flow entries. It is strongly recommended that controllers exclude unnecessary header fields in the clone fields of the rules to keep the number of pending flow entries small.

5.6.2 Overhead by Our Proposed Mechanism

Regarding the overhead of the execution time at the datapath, the rate limitation mechanism introduces small overhead, but the time to send packets to the userspace process is much larger than this overhead. Therefore we can reduce the CPU utilization on the data plane by limiting the rate of packets that go to the userspace process.

In the userspace process, the execution time to create and install a flow entry for the pending flow table is almost the same as the time to send a Packet-In message immediately. If packets that generate and send Packet-In messages arrive at a higher rate than the limited rate, additional large execution time is needed due to queuing the packets.

Although the execution time by the proposed mechanism for the first packet is twice or more than without it, it can discard many packets at the data plane and reduce the execution rate of the process to send Packet-In messages. In total, the execution time by the userspace process is much smaller with the proposed mechanism than without it when Packet-In messages are arrived at high rate. Even though our evaluation on this view is specific to the Open vSwitch, we believe that a comparison of the execution time in the userspace process can be applied to other switches like hardware switches, because they send Packet-In messages from their software OpenFlow agents.

Another overhead introduced by the proposed mechanism is the number of pending flow entries. If they are large, large memory and TCAM space will be occupied by pending flow tables. We do not believe that this concern is significant. The pending flow rules set the header fields used for the pending flow tables based on the header fields that controllers see, and the rules should be much less than the flow entries. Both the size of the pending flow tables and the flow entries are determined by the number of different values in the header fields that the controllers see, such as IP addresses of connected hosts, and the flows sent by the hosts. The pending flow entries are soon expired, and we can regard that the controllers replace the pending flow entries with the flow entries. Therefore, additional TCAM or memory space required by the pending flow tables is very small, and we can ignore this overhead unless the controllers use some flow entry compression algorithms.

Another important point is the flexibility of controlling the network. We have showed that the controllers can use the proposed mechanism in typical use cases, Ethernet switching, IP routing, and NAPT with a few pending flow rules. Using examples of the pending flow rules, we have also showed that constructing pending flow rules is a similar process as designing how controllers use the match fields in the flow tables. Therefore, we believe that the proposed mechanism hardly sacrifice the flexibility of controlling the network that OpenFlow provides.

5.6.3 Similarity with OpenFlow Mechanisms and Possibility for Hardware Implementation

The proposed mechanism should be implemented not only in software switches but also in hardware switches. The design of the proposed mechanism, which has high affinity with the flow tables in OpenFlow, simplifies its implementation in hardware switches.

The matching procedures in the pending flow rules and tables are almost the same with that in the flow tables. Both use priority, an input port, and the match fields for looking up their entries. Therefore, no additional overhead is introduced by reusing a flow table lookup mechanism in the OpenFlow switches. In addition, the switches do not need to lookup multiple tables by translating the pending flow rules and tables into flow entries, as we did in our prototype switch.

We do not propose any specific mechanism to process packets that match the pending flow entries, and the responsibility for selecting this mechanism, the Predefined Actions in Sec. 5.3.4, falls on the network managers. If they select the actions and instructions from existing OpenFlow mechanisms, OpenFlow switches including both hardware and software switches would be able to execute the actions at the data plane. For example, the prototype switch uses a combination of limiting the rate of matched packets and sending Packet-In messages including the packets to the controller, which can be replaced with a Meter instruction and the Output action to the controller in OpenFlow. In Sec. 5.3.4, we gave another example where packets are flooded in the network. In this case, we can use a list of Output actions or the Group action that sends packets to all ports in a group except the input port. The network managers may prefer other actions, not limited to above.

Implementation of the pending flow rules is slightly complicated. The pending flow rules require switches to create and install a new pending flow entry, including copying values of header fields specified by the clone fields from packets. It is difficult to execute this process by reusing existing OpenFlow mechanisms in the data plane, but traditional network devices have similar functions that create a new forwarding entry using values in packets, such as MAC address learning. We can implement actions for the pending flow rules by using these mechanisms.

The proposed mechanism is still beneficial if switches need to process an action for the pending flow rules at software OpenFlow agents in the switches. After the action for the pending flow rules is executed, subsequent packets that match the new pending flow entry are filtered out in the data plane, and the load on the OpenFlow agents is reduced.

5.6.4 Drawbacks of Our Proposed Mechanism

The proposed mechanism works well when the data plane in switches can parse packets and extract values of header fields used in the match fields of the pending flow rules and tables, and when the data plane can apply the actions to packets that match the pending flow entries. In other words, there might be some cases where the controllers use some header fields, but the data plane cannot parse them and extract the values. Some OpenFlow switches support a part of the header fields in the OpenFlow specifications because supporting all header fields in the match fields is not mandatory in the specifications, for example, VLAN ID in 802.1Q headers, payloads in ARP, and ICMPv6 type and codes do not have to be supported. We can add code to support protocols if we can write it for switches, but most hardware switches do not allow users to modify their firmware in practice.

Although OpenFlow and the proposed mechanism do not define how to handle packets that include header fields that switches cannot parse, in some cases the switches should send such packets to controllers, for example, ARP packets. In this case, since we cannot mitigate the overloads in the switches by the proposed mechanism, we have no choice but to set the switches to send all Packet-In messages including packets of such protocols and to limit the total rate of Packet-In messages.

From the users' view, networks using the proposed mechanism will discard some packets in the beginning of such operations as connecting a new host to networks or sending a new flow. On a network side, these operations generate a new pending flow entry. For example, when a controller sets the flow entries per host and the pending flow rules whose clone fields include source and destination Ethernet and IP addresses, and a host starts to establish several TCP sessions to the same host at the same time; the second or later TCP SYN packets may be lost because of a filtering mechanism by the pending flow tables. This should not be a big problem because Ethernet and IP networks do not assure that networks deliver packets to destinations, and hosts retransmit packets if necessary until the hosts receive reply packets. If a certain kind of packet should not be discarded, we can use other mechanisms, like AVANT-GUARD [77] with our proposed mechanism, to provide special care for such packets.

5.6.5 Attacks and Mitigations

A malicious host can attack networks using OpenFlow and the proposed mechanism by exploiting it. If such attackers know header fields that controllers use to forward packets, they can send packets that have different values in the header fields used by the controllers. In this case, the processing for the pending flow rules are executed frequently. As a result, the CPU load on the switches is increased if the switches are implemented to execute actions for the pending flow rules in their CPUs, and the number of pending flow entries is explosively increased, like "Rule - Flow" case in the evaluation.

It is hard to distinguish these packets from others and to discard them without additional detection systems, but some mitigation exists for such attacks. Regarding the CPU load by processing the pending flow rules, controlling resources consumed by processing the rules helps keep a load low in switches at the cost of discarding packets without processing by the pending flow rules. Temporarily disabling the proposed mechanism might work well to reduce a load on switches when the switches frequently process packets that match entries in the pending flow rules.

To keep the number of pending flow entries low, switches may have to evict some entries by cache algorithms. When no packet that increases the switch load arrives, it might not necessary to protect the switches from the overload, and the pending flow entries are merely matched with the packets; the Least Recently Used (LRU) algorithm may work well to evict the entries in this case. When switches are under attacks that explosively increase the number of pending flow entries, attackers send packets whose header fields have different values for efficient attacks, and such packets do not match any pending flow entry. The simple First In First Out (FIFO) algorithm may adequately mitigate these situations.

Finally, we discuss the relationship between the proposed mechanism and existing attacks, especially DDoS and TCP SYN floods. In these attacks, hosts send many packets to a certain network without any advance notice, and preventing switches from such packets is one of our motivations if the switches are configured to send Packet-In messages including such packets to the controllers.

It depends on controllers whether DDoS or TCP SYN flood packets can be filtered out by the proposed mechanism. When they see port numbers in packets, the clone fields in the pending flow rules include port numbers as well as IP and Ethernet addresses in the packets. In this case, the same situations happen with attacks that exploit the pending flow rules and tables, and we cannot use the proposed mechanism to protect the switches. When controllers are designed to use wildcards in the flow entries and the packets used by the attacks only have different values in the header fields that are set to be wildcards in flow entries, we can effectively filter out such packets with the proposed mechanism and wildcards because the clone fields in the pending flow rules do not include such header fields.

5.7 Concluding Remark

In this chapter, we propose a new mechanism to protect the control plane, especially a software part in switches, from packets that bring excessive loads to it. In terms of a control plane protection, we need a mechanism to filter out less important packets for network control to keep loads on switches low in addition to extend the variety of actions that OpenFlow switches can execute.

A key of the proposed mechanism is to record values of some header fields in the switches, and the switches apply some filtering actions (predefined actions) to packets that match recorded values. The controllers use a part of the header fields in the packets, and the switches record values of only header fields that the controllers use. The controller sets such header fields to the switches as “Pending Flow Rules”, and values in packets are recorded in “Pending Flow Tables.” We provide how to use the proposed mechanism in several use cases. We also provide the evaluation results using our prototype switch that show the proposed mechanism can reduce loads on switches and that important Packet-In messages are allowed to pass through. With the proposed mechanism, we can control how packets that fail to match any flow entry should be handled to maintain low loads on the switches, like OpenFlow uses flow tables for offloading packet forwarding to hard-wired offload engines such as ASICs.

Chapter 6

Conclusion

6.1 Summary

Software Defined Networking, SDN, will be essential network control architecture for computer networks in the future, and OpenFlow plays an important role for SDN as a Control - Data Plane Interface. SDN and OpenFlow can provide flexibility of network control, and enable network managers to program how their networks forward packets in a way that they want when they need. On the other hand, networks should be operated stably and robustly, including failure tolerance, resistance to unexpected traffic, and avoiding a high load on control software, because of an negative impact on users' life and business when networks are unavailable. This thesis discusses stability and robustness issues in OpenFlow networks while preserving flexibility of network control.

Chapter 3 addresses the issue of processing at controllers for management of multicast trees. Controllers centrally manage all switches in a network, therefore any event occurring at any switch such as port down should be handled at the controllers. In the case of multicast control, the controllers should recalculate and reinstall multicast trees when a new receiver joins in a multicast group, a receiver leaves from a multicast group, or a failure occurs at somewhere on the trees. It would take much time for controllers to execute such tasks, especially tree computation, and loads on the controllers increase easily. In addition, restoration of delivery of multicast packets should be completed quickly when a failure occurs; otherwise quality of services provided by applications using multicast would be degraded. To make the restoration faster, the controllers should take care of slow performance for modification of flow entries in switches.

A method to manage multicast trees in Chap. 3 proposes the followings to solve such problems. To reduce tree computations, the controllers precompute trees covering all switches where receivers may be connected, and install a part of each tree, which covers all switches that receivers are actually connected, in a network. This removes the task of computing trees when modification to existing trees is needed, which often occurs in campus and enterprise networks. To make the restoration of delivery of multicast packets faster, the controllers label and preinstall multiple redundant trees, and set a root switch where multicast packets come into a network to select one of the trees to use for delivery of multicast packets. When a failure occurs, the controllers need to check whether each tree is disconnected by the failure, and set the root switches to select another tree. This removes the task of computing trees at the time of failures, and there is no need to replace many flow entries at the core of a network. Our proposed method can be used with any algorithm to compute trees.

Chapter 4 addresses the issue of maintaining OpenFlow channels. OpenFlow channels are very important to control OpenFlow networks. When a channel is unavailable, both a switch and a controller at ends of the channel should detect immediately and take appropriate actions. Controllers will need to manage many channels, therefore the controllers prefer to use a small number of messages exchanged with switches for failure detection.

We propose a mechanism to detect a failure of an OpenFlow channel when availability of the channel is important, that is, a message is generated and sent to the other end. Controllers detect a failure by comparing the state of arrival of messages with other controllers. If a message that should arrive does not arrive, a controller suspects a channel is failed. Switches detect a failure by sending keep-alive messages and monitoring reply messages just after sending an important OpenFlow message. This mechanism can make an interval of periodic keep-alive messages longer because availability check is executed when necessary, and the controllers need not to handle many keep-alive messages.

Chapter 5 addresses the issue of protecting OpenFlow switches from many unknown packets. OpenFlow switches send packets that miss flow tables to the controllers by including them into Packet-In messages unless otherwise specified. Then, the controllers usually install flow entries that correspond to the Packet-In messages immediately. When a lot of such packets suddenly come to a switch without advance notice, a CPU of the switch would be overloaded because Packet-In messages are usually generated at the CPU and installation of flow entries always has small delay. To prevent being overloaded, Packet-In messages should be selectively discarded so that the CPU is not overloaded while important packets for network control are not discarded.

We propose an extension of switches to mitigate this problem, which consists of pending flow rules and tables. Pending flow rules are intent of controllers that which header fields are used for network control, and the rules are configured to switches in advance by controllers. The switches record to pending flow tables the values of header fields in packets included into Packet-In messages, and packets that match pending flow tables are filtered out before generating Packet-In messages. This mechanism effectively filters out packets that the controllers would output the same results, while surely passes packets that the controllers would output the different results.

This thesis discusses stability and robustness issues in all parts of the control plane in OpenFlow networks. We hope our proposals contribute to enhance stability and robustness of OpenFlow networks, and more networks are constructed based on the SDN architecture.

6.2 Future Directions

As the concept of SDN and OpenFlow is widely recognized, other issues arise in computer networking research, design and operation. One is software-based middleboxes, also referred as Network Functions Virtualization (NFV), such as firewalls, WAN optimizers, and intrusion detection systems for rapidly enhancing capability of these boxes and get these boxes on demand like cloud computing. Another is server-like network devices, which run mostly Linux on traditional network devices. These devices make it easier to automate configurations of network devices, and it meets a part of expectation to SDN and OpenFlow. Integration of OpenFlow and such trends will be needed in the near future.

SDN and OpenFlow is beneficial only in a single domain, which means a network that is operated by one entity. Extending flexibility of SDN and OpenFlow to inter-domain environments with a practical abstraction of each network will be necessary to resolve issues that are currently significant on the Internet like security and traffic explosion.

References

[1] Internet Engineering Task Force. https://www.ietf.org/.

[2] Open Networking Foundation. https://www.opennetworking.org/.

[3] Adams, A., Nicholas, J. and Siadak, W. Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised), RFC 3973, IETF (2005).

[4] Aggarwal, R., Papadimitriou, D. and Yasukawa, S. Extensions to Resource Reservation Protocol - Traffic Engineering (RSVP-TE) for Point-to-Multipoint TE Label Switched Paths (LSPs), RFC 4875, IETF (2007).

[5] Al-Shabibi, A., De Leenheer, M., Gerola, M., Koshibe, A., Parulkar, G., Salvadori, E. and Snow, B. OpenVirteX: Make Your Virtual SDNs Programmable, HotSDN '14, ACM, pp. 25–30 (2014).

[6] Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W. and Parulkar, G. ONOS: Towards an Open, Distributed SDN OS, HotSDN '14, ACM, pp. 1–6 (2014).

[7] Bianchi, G., Bonola, M., Capone, A. and Cascone, C. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch, *SIGCOMM Comput. Commun. Rev.*, Vol. 44, No. 2, pp. 44–51 (2014).

[8] Braga, R., Mota, E. and Passito, A. Lightweight DDoS flooding attack detection using NOX/OpenFlow, LCN '10, IEEE, pp. 408–415 (2010).

[9] Bux, W., Denzel, W. E., Engbersen, T., Herkersdorf, A. and Luijten, R. P. Technologies and Building Blocks for Fast Packet Forwarding, *IEEE Commun. Mag.*, Vol. 39, No. 1, pp. 70–77 (2001).

[10] Canini, M., Venzano, D., Perešíni, P., Kostić, D. and Rexford, J. A NICE Way to Test OpenFlow Applications, NSDI '12, USENIX, pp. 127–140 (2012).

[11] Capone, A., Cascone, C., Nguyen, A. and Sanso, B. Detour Planning for Fast and Reliable Failure Recovery in SDN with OpenState, DRCN '15, IEEE, pp. 25–32 (2015).

[12] Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N. and Shenker, S. Ethane: Taking Control of the Enterprise, *SIGCOMM Comput. Commun. Rev.*, Vol. 37, No. 4, pp. 1–12 (2007).

[13] Chang, D., Kwak, M., Choi, N., Kwon, T. and Choi, Y. C-flow: An efficient content delivery framework with OpenFlow, ICOIN '14, IEEE, pp. 270–275 (2014).

[14] Chung, C.-J., Khatkar, P., Xing, T., Lee, J. and Huang, D. NICE: Network Intrusion Detection and Countermeasure Selection in Virtual Network Systems, *IEEE Trans. Dependable and Secure Comput.*, Vol. 10, No. 4, pp. 198–211 (2013).

[15] Cui, J.-H., Faloutsos, M. and Gerla, M. An Architecture for Scalable, Efficient, and Fast Fault-Tolerant Multicast Provisioning, *IEEE Netw.*, Vol. 18, No. 2, pp. 26–34 (2004).

[16] Curtis, A. R., Mogul, J. C., Tourrilhes, J., Yalagandula, P., Sharma, P. and Banerjee, S. DevoFlow: Scaling Flow Management for High-Performance Networks, *SIGCOMM Comput. Commun. Rev.*, Vol. 41, No. 4, pp. 254–265 (2011).

[17] D. J. Bernstein SYN cookies. http://cr.yp.to/syncookies.html.

[18] Dijkstra, E. W. A Note on Two Problems in Connexion with Graphs, *Numerische Mathematik*, Vol. 1, pp. 269–271 (1959).

[19] Dolev, D., Friedman, R., Keidar, I. and Malkhi, D. Failure Detectors in Omission Failure Environments, Technical report, Cornell University (1996).

[20] Dugal, D., Pignataro, C. and Dunn, R. Protecting the Router Control Plane, RFC 6192, IETF (2011).

[21] Estrin, D., Farinacci, D., Helmy, A., Thaler, D., Deering, S., Handley, M., Jacobson, V., Liu, C., Sharma, P. and Wei, L. Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification, RFC 2117, IETF (1997).

[22] Fei, A., Cui, J., Gerla, M. and Faloutsos, M. Aggregated Multicast: an Approach to Reduce Multicast State, GLOBECOM '01, Vol. 3, IEEE, pp. 1595–1599 (2001).

[23] Fenner, W. Internet Group Management Protocol, Version 2, RFC 2236, IETF (1997).

[24] Floodlight, http://www.projectfloodlight.org/floodlight/.

[25] Ford, A., Raiciu, C., Handley, M. and Bonaventure, O. TCP Extensions for Multipath Operation with Multiple Addresses, RFC 6824, IETF (2013).

[26] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A. and Walker, D. Frenetic: A Network Programming Language, *SIGPLAN Not.*, Vol. 46, No. 9, pp. 279–291 (2011).

[27] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N. and Shenker, S. NOX: Towards an Operating System for Networks, *SIGCOMM Comput. Commun. Rev.*, Vol. 38, No. 3, pp. 105–110 (2008).

[28] Gupta, A., Vanbever, L., Shahbaz, M., Donovan, S. P., Schlinker, B., Feamster, N., Rexford, J., Shenker, S., Clark, R. and Katz-Bassett, E. SDX: A Software Defined Internet Exchange, *SIGCOMM Comput. Commun. Rev.*, Vol. 44, No. 4, pp. 551–562 (2014).

[29] Gyllstrom, D., Braga, N. and Kurose, J. Recovery from Link Failures in a Smart Grid Communication Network using OpenFlow, SmartGridComm '14, IEEE, pp. 254–259 (2014).

[30] Handigol, N., Heller, B., Jeyakumar, V., Mazières, D. and McKeown, N. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks, NSDI '14, USENIX, pp. 71–85 (2014).

[31] Hao, F., Lakshman, T. V., Mukherjee, S. and Song, H. Enhancing Dynamic Cloud-based Services using Network Virtualization, *SIGCOMM Comput. Commun. Rev.*, Vol. 40, No. 1, pp. 67–74 (2010).

[32] Hasegawa, T., Kamimura, K., Hoshino, H. and Ano, S. IP-based HDTV broadcasting system architecture with non-stop service availability, GLOBECOM '05, Vol. 1, IEEE, pp. 337 – 342 (2005).

[33] Heller, B., Sherwood, R. and McKeown, N. The Controller Placement Problem, HotSDN '12, ACM, pp. 7–12 (2012).

[34] Huang, D. Y., Yocum, K. and Snoeren, A. C. High-fidelity Switch Models for Software-defined Network Emulation, HotSDN '13, ACM, pp. 43–48 (2013).

[35] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S. and Vahdat, A. B4: Experience with a Globally-deployed Software Defined Wan, *SIGCOMM Comput. Commun. Rev.*, Vol. 43, No. 4, pp. 3–14 (2013).

[36] Jin, X., Gossels, J., Rexford, J. and Walker, D. CoVisor: A Compositional Hypervisor for Software-Defined Networks, NSDI'15, USENIX, pp. 87–101 (2015).

[37] Kanaumi, Y., Saito, S., Kawai, E., Ishii, S., Kobayashi, K. and Shimojo, S. RISE: A Wide-Area Hybrid OpenFlow Network Testbed, *IEICE Trans. Commun.*, Vol. 96, No. 1, pp. 108–118 (2013).

[38] Kate, G. TR10: Software-Defined Networking, *MIT Techn. Rev.* (2009).

[39] Katz, D. and Ward, D. Bidirectional Forwarding Detection (BFD), RFC 5880, IETF (2010).

[40] Kawazoe Aguilera, M., Chen, W. and Toueg, S. Heartbeat: A timeout-free failure detector for quiescent reliable communication, *Distributed Algorithms* (Mavronicolas, M. and Tsigas, P.(eds.)), Lecture Notes in Comput. Sci., Vol. 1320, Springer Berlin / Heidelberg, pp. 126–140 (1997).

[41] Kempf, J., Bellagamba, E., Kern, A., Jocha, D., Takacs, A. and Skoldstrom, P. Scalable fault management for OpenFlow, ICC '12, IEEE, pp. 6606–6610 (2012).

[42] Knight, S., Nguyen, H., Falkner, N., Bowden, R. and Roughan, M. The Internet Topology Zoo, *IEEE J. Sel. Areas Commun.*, Vol. 29, No. 9, pp. 1765–1775 (2011).

[43] Koide, T., Ashida, Y. and Shimonishi, H. A study on network abstraction model in SDN control platform and its evaluation, *Technical report of IEICE. CQ*, Vol. 113, No. 7, pp. 41–46 (2013).

[44] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T. and Shenker, S. Onix: A Distributed

Control Platform for Large-scale Production Networks, OSDI '10, USENIX, pp. 1–6 (2010).

[45] Kuroki, K., Fukushima, M. and Hayashi, M. Redundancy Method for Highly Available OpenFlow Controller, *International Journal on Advances in Internet Technology*, Vol. 7, No. 1, pp. 114–123 (2014).

[46] Levis, P., Patel, N., Culler, D. and Shenker, S. Trickle: A Self-regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks, NSDI '04, USENIX (2004).

[47] Li, G., Wang, D. and Doverspike, R. Efficient Distributed MPLS P2MP Fast Reroute, INFOCOM '06, IEEE, pp. 1 –11 (2006).

[48] Li, X. and Freedman, M. J. Scaling IP Multicast on Datacenter Topologies, CoNEXT '13, ACM, pp. 61–72 (2013).

[49] Liang, Y., Apostolopoulos, J. and Girod, B. Analysis of Packet Loss for Compressed Video: Effect of Burst Losses and Correlation Between Error Frames, *IEEE Trans. Circuits Syst. Video Technol.*, Vol. 18, No. 7, pp. 861–874 (2008).

[50] Lockwood, J. W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R. and Luo, J. NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing, MSE'07, IEEE, pp. 160–161 (2007).

[51] Luo, T., Tan, H.-P., Quan, P., Law, Y. W. and Jin, J. Enhancing Responsiveness and Scalability for OpenFlow Networks via Control-Message Quenching, ICTC '12, IEEE, pp. 348–353 (2012).

[52] Mahajan, R., Bellovin, S. M., Floyd, S., Ioannidis, J., Paxson, V. and Shenker, S. Controlling High Bandwidth Aggregates in the Network, *SIGCOMM Comput. Commun. Rev.*, Vol. 32, No. 3, pp. 62–73 (2002).

[53] Marcondes, C., Santos, T., Godoy, A., Viel, C. and Teixeira, C. CastFlow: Clean-Slate Multicast Approach using In-Advance Path Processing in Programmable Networks, ISCC '12, IEEE, pp. 94–101 (2012).

[54] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S. and Turner, J. OpenFlow: Enabling Innovation in Campus Networks, *SIGCOMM Comput. Commun. Rev.*, Vol. 38, pp. 69–74 (2008).

[55] Médard, M., Finn, S. G. and Barry, R. A. Redundant trees for preplanned recovery in arbitrary vertex-redundant or edge-redundant graphs, *IEEE/ACM Trans. Netw.*, Vol. 7, pp. 641–652 (1999).

[56] Mirkovic, J. and Reiher, P. A Taxonomy of DDoS Attack and DDoS Defense Mechanisms, *SIGCOMM Comput. Commun. Rev.*, Vol. 34, No. 2, pp. 39–53 (2004).

[57] Mochizuki, K., Shimizu, M. and Yasukawa, S. Multicast Tree Algorithm Minimizing the Number of Fast Reroute Protection Links for P2MP-TE Networks, GLOBECOM '06, IEEE, pp. 1 –5 (2006).

[58] Monsanto, C., Reich, J., Foster, N., Rexford, J. and Walker, D. Composing Software Defined Networks, NSDI '13, Lombard, IL, USENIX, pp. 1–13 (2013).

[59] Moy, J. Multicast Extensions to OSPF, RFC 1584, IETF (1994).

[60] Nakagawa, Y., Hyoudou, K. and Shimizu, T. A Management Method of IP Multicast in Overlay Networks Using Openflow, HotSDN '12, ACM, pp. 91–96 (2012).

[61] Open Networking Foundation OpenFlow Switch Specification, Version 1.1.0 Implemented (2011). http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[62] Open Networking Foundation. Software-Defined Networking: The New Norm for Networks (2012). https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

[63] Open Networking Foundation OpenFlow Switch Specification 1.3.4 (2014). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.4.pdf.

[64] Open Networking Foundation. OpenFlow Switch Specification Version 1.5.1 (2015). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf.

[65] Open vSwitch, http://openvswitch.org/.

[66] Pan, P., Swallow, G. and Atlas, A. Fast Reroute Extensions to RSVP-TE for LSP Tunnels, RFC 4090, IETF (2005).

[67] Peng, T., Leckie, C. and Ramamohanarao, K. Survey of Network-based Defense Mechanisms Countering the DoS and DDoS Problems, *ACM Comput. Surv.*, Vol. 39, No. 1 (2007).

[68] Reitblatt, M., Canini, M., Guha, A. and Foster, N. FatTire: Declarative Fault Tolerance for Software-defined Networks, HotSDN '13, ACM, pp. 109–114 (2013).

[69] Ryu SDN Controller, http://osrg.github.io/ryu/.

[70] Sergent, N., Defago, X. and Schiper, A. Impact of a Failure Detection Mechanism on the Performance of Consensus, PRDC '01, IEEE, pp. 137–145 (2001).

[71] Shalimov, A., Zuikov, D., Zimarina, D., Pashkov, V. and Smeliansky, R. Advanced Study of SDN/OpenFlow Controllers, CEE-SECR '13, ACM, pp. 1:1–1:6 (2013).

[72] Sharma, S., Staessens, D., Colle, D., Pickavet, M. and Demeester, P. Automatic bootstrapping of OpenFlow networks, LANMAN '13, IEEE, pp. 1–6 (2013).

[73] Sharma, S., Staessens, D., Colle, D., Pickavet, M. and Demeester, P. Fast failure recovery for in-band OpenFlow networks, DRCN '13, IEEE, pp. 52–59 (2013).

[74] Sharma, S., Staessens, D., Colle, D., Pickavet, M. and Demeester, P. OpenFlow: Meeting carrier-grade recovery requirements, *Comput. Commun.*, Vol. 36, No. 6, pp. 656 – 665 (2013).

[75] Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N. and Parulkar, G. Can the Production Network Be the Testbed?, OSDI'10, USENIX, pp. 1–6 (2010).

[76] Sherwood, R. and Yap, K.-K. Cbench: Controller Benchmarker (2011). http://www.openflow.org/wk/index.php/Oflops.

[77] Shin, S., Yegneswaran, V., Porras, P. and Gu, G. AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-defined Networks, CCS '13, ACM, pp. 413–424 (2013).

[78] Shirali-Shahreza, S. and Ganjali, Y. Efficient Implementation of Security Applications in OpenFlow Controller with FleXam, HOTI '13, IEEE, pp. 49–54 (2013).

[79] Siachalou, S. and Georgiadis, L. Algorithms for Precomputing Constrained Widest Paths and Multicast Trees, *IEEE/ACM Trans. Netw.*, Vol. 13, No. 5, pp. 1174–1187 (2005).

[80] Siew, D. C. K. and Gang, F. Tree-Caching for Multicast Connections with End-to-End Delay Constraint, *IEICE Trans. Commun.*, Vol. 84, No. 4, pp. 1030–1040 (2001).

[81] Stewart, R. R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H. J., Taylor, T., Rytina, I., Kalla, M., Zhang, L. and Paxson, V. Stream Control Transmission Protocol, RFC 2960, IETF (2000).

[82] Tootoonchian, A. and Ganjali, Y. HyperFlow: A Distributed Control Plane for OpenFlow, INM/WREN '10, USENIX, pp. 3–3 (2010).

[83] Tootoonchian, A., Gorbunov, S., Ganjali, Y., Casado, M. and Sherwood, R. On Controller Performance in Software-Defined Networks, Hot-ICE'12, USENIX (2012).

[84] Trema: Full-Stack OpenFlow Framework for Ruby/C, http://trema.github.com/trema/.

[85] Tseng, C.-J. and Chen, C.-H. Combining Precomputation and On-Demand Routing for Multicast QoS Routing, ICC '04, Vol. 2, IEEE, pp. 1171–1176 (2004).

[86] Veltri, L., Morabito, G., Salsano, S., Blefari-Melazzi, N. and Detti, A. Supporting Information-Centric Functionality in Software Defined Networks, ICC '12, IEEE, pp. 6645–6650 (2012).

[87] Waitzman, D., Partridge, C. and Deering, S. E. Distance Vector Multicast Routing Protocol, RFC 1075, IETF (1988).

[88] Wette, P. and Karl, H. Which Flows Are Hiding Behind My Wildcard Rule?: Adding Packet Sampling to Openflow, *SIGCOMM Comput. Commun. Rev.*, Vol. 43, No. 4, pp. 541–542 (2013).

[89] Whetten, B., Vicisano, L., Kermode, R., Handley, M., Floyd, S. and Luby, M. Reliable Multicast Transport Building Blocks for One-to-Many Bulk-Data Transfer, RFC 3048, IETF (2001).

[90] Wijnands, I., Minei, I., Kompella, K. and Thomas, B. Label Distribution Protocol Extensions for Point-to-Multipoint and Multipoint-to-Multipoint Label Switched Paths, RFC 6388, IETF (2011).

[91] Xue, G., Chen, L. and Thulasiraman, K. Quality-of-Service and Quality-of-Protection Issues in Preplanned Recovery Schemes Using Redundant Trees, *IEEE J. Sel. Areas Commun.*, Vol. 21, No. 8, pp. 1332 – 1345 (2003).

[92] Yasukawa, S. Signaling Requirements for Point-to-Multipoint Traffic-Engineered MPLS Label Switched Paths (LSPs), RFC 4461, IETF (2006).

[93] Yu, M., Rexford, J., Freedman, M. J. and Wang, J. Scalable Flow-Based Networking with DIFANE, *SIGCOMM Comput. Commun. Rev.*, Vol. 40, No. 4, pp. 351–362 (2010).

[94] Yu, Y., Zhen, Q., Xin, L. and Shanzhi, C. OFM: A Novel Multicast Mechanism Based on OpenFlow, *Advances in information Sciences and Service Sciences*, Vol. 4, No. 9, pp. 278–286 (2012).

[95] Zeng, H., Kazemian, P., Varghese, G. and McKeown, N. Automatic Test Packet Generation, CoNEXT '12, ACM, pp. 241–252 (2012).

[96] Zou, J., Shou, G., Guo, Z. and Hu, Y. Design and implementation of secure multicast based on SDN, IC-BNMT '13, IEEE, pp. 124–128 (2013).

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Yasuo Okabe. He gave me a lot of valuable guidance and helpful discussion on my research, and many opportunities to broaden my knowledge and research skills as well as experience of working in international communities. I would also like to appreciate Professor Michihiko Minoh and Professor Hajime Kita for serving on my doctoral committee. They gave me many valuable comments to improve this thesis.

I would also like to thank Dr. Hideyuki Shimonishi and Dr. Kazuya Suzuki for a precious opportunity to take part in Software Defined Network and OpenFlow research in NEC Cloud System Research Laboratories. They gave me valuable comments even after I have completed an internship at NEC.

I want to thank all past and present members of Okabe Laboratory, especially Associate Professor Shuichi Miyazaki and Mr. Yoshiharu Tsuzaki. I received a great deal of inspiration from discussions with these members, and they always help me to conduct my research, such as setting up and configuring servers and network devices. I also wish to deeply thank Ms. Kazumi Sakai for her jobs as a secretary of Okabe Laboratory. I was helped by her outstanding ability to process administrative duties.

Finally, I would like to thank my family for tremendous supports to me.

List of Publications by the Author

Journal Papers

- Daisuke Kotani, Kazuya Suzuki and Hideyuki Shimonishi. A Multicast Tree Management Method Supporting Fast Failure Recovery and Dynamic Group Membership Changes in OpenFlow Networks. *Journal of Information Processing*, Vol. 24, No. 2, March 2016 (To appear).

- Daisuke Kotani and Yasuo Okabe. A Packet-In Message Filtering Mechanism for Protection of Control Plane in OpenFlow Switches. *IEICE Transactions on Information and Systems*, Vol. E99-D, No. 3, March 2016 (To appear).

International Conference Papers

- Daisuke Kotani and Yasuo Okabe. Fast Failure Detection of OpenFlow Channels. *The 11th Asian Internet Engineering Conference (AINTEC 2015)*, November 2015.

- Daisuke Kotani and Yasuo Okabe. A Packet-In Message Filtering Mechanism for Protection of Control Plane in OpenFlow Networks. *The 10th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2014)*, pp.29-40, October 2014.

- Yoshiharu Tsuzaki, Ryosuke Matsumoto, Daisuke Kotani, Shuichi Miyazaki and Yasuo Okabe. A Mail Transfer System Selectively Restricting a Huge Amount of E-mails. *Workshop on Resilient Internet based Systems (REIS 2013)*, December, 2013.

- Daisuke Kotani and Yasuo Okabe. PacketIn Message Control for Reducing CPU Load and Control Traffic in OpenFlow Switches. *European Workshop on Software*

Defined Networks (EWSDN), pp.42-47, October 2012.

- Daisuke Kotani, Kazuya Suzuki and Hideyuki Shimonishi. A design and implementation of OpenFlow Controller handling IP multicast with Fast Tree Switching. *The 12th IEEE/IPSJ International Symposium on Applications and the Internet (SAINT 2012)*, pp.60-67, July 2012.

- Hideyuki Shimonishi, Yasuhito Takamiya, Yasunobu Chiba, Kazushi Sugyo, Youichi Hatano, Kentaro Sonoda, Kazuya Suzuki, Daisuke Kotani and Ippei Akiyoshi. Programmable Network Using OpenFlow for Network Researches and Experiments (Invited Paper). *The 6th International Conference on Mobile Computing and Uniquitous Networking (ICMU 2012)*, May 2012.

- Daisuke Kotani, Satoshi Nakamura and Katsumi Tanaka. Supporting Sharing of Browsing Information and Search Results in Mobile Collaborative Searches. *The 12th international conference on Web Information Systems Engineering (WISE 2011)*, pp.298-305, October 2011.

International Conference Posters

- Daisuke Kotani and Yasuo Okabe. A Packet-In Message Filter in OpenFlow Switches for Reducing Control Messages. *AsiaFI 2013 Summer School*, August 2013.

Domestic Conference and Technical Reports

- Daisuke Kotani and Yasuo Okabe. A Packet-In Message Filter in OpenFlow Switches Considering Wildcarded Headers. *IEICE Technical Reports*, Vol. 113, No. 443, IA2013-86, pp. 43-48, February 2014 (In Japanese).

- Yoshiharu Tsuzaki, Ryosuke Matsumoto, Daisuke Kotani, Shuichi Miyazaki and Yasuo Okabe. An Efficient Judging Method of a Large Amount of E-mails Having the Same Sender and Recipient Addresses. *IEICE Technical Reports*, Vol. 113, No. 443, IA2013-87, pp. 61-66, February 2014 (In Japanese).

- 津崎善晴, 松本亮介, 小谷大祐, 宮崎修一, 岡部寿男. 電子メールの大量送信を選択的に制限する中継システム. 情報処理学会関西支部 支部大会, E-23, 2013 年 9 月.

- Daisuke Kotani and Yasuo Okabe. A Method of Controlling Packet-In Messages in OpenFlow Switches for Reducing Loads of Switches and Control Networks. *IEICE Technical Reports*, Vol. 112, No. 212, IA2012-17, pp. 31-36, September 2012 (In Japanese).
- 小谷大祐, 中村聡史, 田中克己. モバイル協調検索におけるユーザ間の情報共有支援. 第3回データ工学と情報マネジメントに関するフォーラム (DEIM Forum 2011), A6-6, 2011年2月.