Fast Failure Detection of OpenFlow Channels

Daisuke Kotani
Kyoto University
Yoshida-Honmachi, Sakyo, Kyoto, Japan
kotani@net.ist.i.kyoto-u.ac.jp

Yasuo Okabe
Kyoto University
Yoshida-Honmachi, Sakyo, Kyoto, Japan
okabe@i.kyoto-u.ac.jp

ABSTRACT

We propose a mechanism to detect OpenFlow channel failures quickly in switches and controllers where multiple controllers are running. In OpenFlow networks, it is important to maintain OpenFlow channels between controllers and switches are up and to detect channel failures immediately, so that messages to notify events such as port down and modification of flow tables are always delivered to the other side surely and quickly. Exchanging keep-alive messages frequently is undesirable for controllers because the controllers should handle many keep-alive messages from switches. This would be a significant overhead when the rate of other messages than keep-alive ones is low, because the controllers are forced to handle many keep-alive messages although such keep-alive messages do not affect network control directly. Our proposed mechanism adaptively sends keep-alive messages to detect quickly that a message is not reached to the other side, instead of checking whether a channel is up. A controller shares a message received from a switch with other controllers in a timely manner, and the controller regards a channel has been unavailable if the message notified by other controllers has not arrived via the channel. A switch sends keep-alive messages to all channels just after sending an important asynchronous message such as a port status message, and regards all channels have been gone down if the switch does not receive any response. The evaluation shows that our proposed mechanism reduces failure detection delay to timeout until receiving a response, and that overhead on latency is negligible.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations

AINTEC'15 November 18–20,2015,Bangkok, Thailand.

General Terms

Management

Keywords

Software-Defined Networking, OpenFlow, Failure Detection

1. INTRODUCTION

Computer networks mainly consist of two planes, the data plane that forwards packets, and the control plane that decides where to forward packets. To continue to deliver packets to destinations, networks should reroute traffic quickly at the time of failures; failures occurred in the data plane must be detected and notified to the control plane 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 Bidirectional Forwarding Detection (BFD)[3].

We hardly need to care for connectivity between the data plane and the control plane in traditional networking devices because these planes are in the same device, but things are different with Software-Defined Networking. Software-Defined Networking[11] is an architecture to control a network with a global network view by a centralized controller for simple and flexible network control, and OpenFlow[9] is one of major protocols used for communication between controllers and switches. In OpenFlow, network events including failures are detected at switches (the data plane) and notified to external controllers (the control plane). The controllers calculate new packet forwarding rules and update them in the switches if necessary. A switch and a controller communicate with each other through a TCP connection (called an OpenFlow channel) via multiple links and devices, and such links and devices are also a part of the control plane. When a failure occurs to the data plane, a failure may also occur to the control plane because sometimes the control plane and the data plane are not physically separated, use the same power source, pathway, etc. An OpenFlow channel may also be disconnected temporarily at the same time with a

data plane failure, and a failure in the data plane is not notified to the controller in a timely manner. As a result, there may be a significant delay until new rules are installed in the switches. Therefore, detection of OpenFlow channel failures is very important.

Reliability in OpenFlow controllers has been studied in various perspective, such as high performance for large scale networks[16, 1], controller fault tolerance[2, 7, 6], and path recovery from switches to a controller via the data plane[13], but no work tries to detect an OpenFlow channel failure directly. OpenFlow protocols define keep-alive messages (echo request/reply) 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 the message processing performance in the controllers is not high.

To reduce the number of the keep-alive messages while detecting a channel failure quickly when necessary, we propose a mechanism to detect an OpenFlow channel failure in switches and controllers when network events that should be notified to controllers occur. The reason to maintain OpenFlow channels is to be always ready to deliver important messages in real-time. The proposed mechanism quickly detects that an OpenFlow message does not arrive at the other side of the channel, instead of checking a channel is always up.

We assume that, for redundancy, multiple controllers are running and a switch has channels to two or more controllers. From a switch's perspective, a message by events in the data plane such as port down is sent over all channels at the same time, and the message arrives at the 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 the 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 avoids using the failed channels, and the opposite switch can receive the message faster.

In the switch side, a switch sends a keep-alive message (echo request) to all channels just after sending a message by a event occurred in the data plane. 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 within a certain period (timeout), the switch regards all channels are failed, and enters the fail secure or standalone mode according to the switch configuration.

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 the arrival of the message to other controllers. When a controller receives a notification of the arrival of a message from other controllers and the controller has not received the notified message from the switch, the controller waits for a certain period (timeout). If the message has not arrived at the controller until the timeout has been expired, the controller regards the channel to the switch from the controller has been lost, and avoids using the channel.

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

Our contributions are summarized as follows:

- We categorize OpenFlow channel failure patterns, and point out that failures of all channels at switches and failures of both all and some channel failures at controllers should be detected.
- We design a fast failure detection mechanism by adaptively sending keep-alive messages and by sharing arrivals of messages among controllers.
- We implement the proposed mechanism into an OpenFlow channel proxy, and show that the proposed mechanism can shorten failure detection delay with negligible overhead on latency.

2. RELATED WORK

2.1 Multi-Controller Support and Message Types in OpenFlow

OpenFlow protocols[10] are used to manage packet forwarding rules stored in a flow table of OpenFlow switches. Each entry in the flow table includes a tuple of values of header fields as a matching condition and actions applied to matched packets.

A connection between a switch and a controller is called an OpenFlow channel, and the channel is established via the data plane (called in-band control) or the control plane that is logically separated from the data plane (called out-of-band control). Each switch that supports OpenFlow Switch Specification 1.2 and later can establish channels to multiple controllers. A channel has a role, MASTER, SLAVE or EQUAL. In

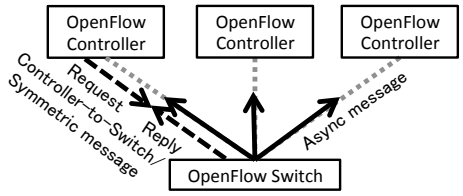

Figure 1: OpenFlow Messages over Multiple Channels

EQUAL, a controller can read and write a state of a switch as well as other controllers. In MASTER, a controller can read and write a state of a switch, and the role of other controllers are set to SLAVE, which allows only to read a state from a switch.

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 (a request message), and the controller expects that the switch responds to these messages (a reply message). A switch informs controllers of a state change with an asynchronous message. Although the OpenFlow protocol does not require a controller to respond to 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 similar to controller-to-switch messages, but a request message can be sent not only from controllers but also from switches. A keep-alive message (echo request/reply) is an example of symmetric messages.

Figure 1 shows how these types of messages are transferred over multiple channels. Asynchronous messages are sent over all channels other than ones where the messages are 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.

2.2 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 devices in the data plane, and reroute the traffic if necessary. Kempf et al.[4] proposed a scalable topology monitoring architecture by implementing a monitoring function on switches. Sharma et al.[14] 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 the path between switches and controllers. Sharma et al.[13] proposed a mechanism to recover in-band OpenFlow channels from failures using fast failover actions. In their evaluation, BFD[3] is used 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 conventional networking technologies. For example, link aggregation (IEEE802.3ad) is used for link level redundancy, and BFD[3] and Ethernet OAM are configured for monitoring a path. These technologies improve the reliability of control networks, but controllers and switches should also have a fast failure detection mechanism, because unrecoverable failures detected by these technologies cannot be notified to controllers and switches in principle.

The fault management of controllers is also important because we cannot control OpenFlow networks without controllers. One of the purposes of distributed controllers[5, 16, 1] is to make controllers fault tolerant. The controllers monitor each other to check whether each controller is running, and failover to running controllers when one of controllers has been down. Kuroki et al.[6] 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 in 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.

3. OPENFLOW CHANNEL FAILURES

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 fault in the control plane, such as controllers, switches, or networking devices in the control plane become down, or a cable is cut. The other is a large delay of 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 faults and large delay.

We assume that, for redundancy, multiple controllers are running and each switch is connected to more than one controller. We can categorize a state of OpenFlow channel failures 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 only the All failure quickly, and switches must enter a preconfigured fail secure or standalone mode according to OpenFlow specifications[10] when detected. Switches do not have to detect the Partial failure, because switches mainly create asyn-

chronous messages, which are flooded to all channels, and at least one controller can receive and handle the asynchronous messages.

Controllers need to detect both the All and Partial failures quickly. 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 request-reply messages like echo request/reply messages, because the controllers cannot get a trigger to start checking a state of a channel like sending an echo request message and starting a timer for a timeout. The controllers may have many chances to start checking a state of a channel, such as confirming that previous messages are processed by barrier request/reply messages. In this paper, the controllers will detect the All failure by setting a proper timeout of receiving a reply messages and by trying to send a request message to all channels one by one.

4. OPENFLOW CHANNEL FAILURE DETECTION

We can easily assume that an OpenFlow channel failure 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 for processing keep-alive messages with multiple controllers is impossible unlike other messages, because each switch should check availability of each controller.

It is not desirable that many keep-alive messages are exchanged when the rate of other messages are low. Controllers are often designed to insert flow entries into switches proactively so that controllers and switches do not need to communicate with each other frequently. Moreover, according to the results of a recent study by Shalimov et al.[12], 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 like CPython. If a controller need to handle many keep-alive messages, such controllers cannot be used due to many keep-alive messages although other messages will be exchanged not so frequently.

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 the switches can forward packets only with existing configurations.

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 a switch receives 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 a channel state, active when one or more channels are available, checking when executing a keep-alive procedure, or inactive when all channels are unavailable. When the channel state is active or inactive, a switch sends an echo request message over every channel 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 an echo reply message via one or more channels, the switch sets the channel state to active, and regards that previous messages have 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 may lose 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 an echo reply timeout value. 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.

4.2 Detection at Controllers

A limitation in controllers is that a message initiated by a controller is sent over one channel, not multiple channels, at the same time. This means that the controller should try to exchange request/reply messages over one channel, and when the timeout has expired, the controller should retry to send 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 may update Flow Tables to reroute traffic. If controllers share what messages arrive at which controllers, the controllers can use such data 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. 4.3 for the sequence ID). As with switches, each controller holds a channel state, active, checking, or inactive, and 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 the message including the sequence ID and the DPID of the switch to other controllers, and sets the Latest Sequence ID to the sequence ID in the message.

When a controller receives a message from another controller and a sequence ID in the received message is more than the Latest Sequence ID, the controller sets a state of a channel 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 checking, the controller does not start the timer. When the controller receives the same message from the channel to the switch, 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 same message from the switch, the controller regards the channel state is changed 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 that its channel is active. If a channel from a controller to a switch is inactive but another controller has an active channel to the switch, the controller knows that at least one channel from another controller is active. 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 may elect a backup controller in advance, and another controller may run a leader election algorithm among controllers that have active channels.

4.3 Sequence Message ID 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 time and a port in a switch becomes down and up, some controllers receive a port down message after another controller receives 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[10] considers this problem only in the case of OpenFlow channel role transition, but the same problem occurs when reordering asynchronous messages in the multiple controller environments.

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 port status changed. A controller should ignore messages whose sequence ID is less than the ID of messages that have previously processed.

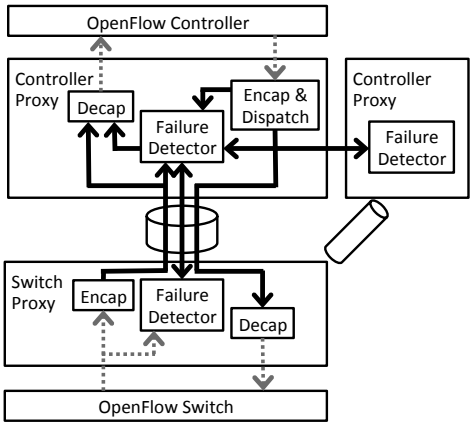

Figure 2: Overview of Prototype OpenFlow Channel Proxy. The gray dotted lines are flow of OpenFlow messages, and the black solid lines are flow of messages in our own format.

5. PROTOTYPE IMPLEMENTATION AS 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 we allow the proxy to intercept the messages.

Figure 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. 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. 2) runs near or on a controller. It accepts a connection 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 headers of our format consists of a sequence ID as described in Sec. 4.3, a datapath ID of a switch, and a message type such as echo request/reply used in Sec. 4.1 for failure detection, notification for arrival of a message, and request for sending a message to a switch in Sec. 4.2.

When a Switch Proxy receives an OpenFlow message from a switch, the Encap module assigns a sequence ID

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

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 in our own format to Controller Proxies if necessary. The Failure Detector closes the channel to the switch if the channel state is 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 the Switch Proxies. The other is to share messages received from Switch Proxies with other Controller Proxies. The Failure Detector module monitors connections to Switch Proxies as described in Sec. 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 employ 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.

6. EVALUATION

We have measured how much failure detection delay becomes short, and the overhead on the throughput and latency caused by our proposed mechanism.

The evaluation environment is in Fig. 3. One Switch Proxy and two Controller Proxies (Main and Sub) are connected via a Packet Filter. The OpenFlow Switch is connected to the Switch Proxy, and the Controller Proxy Main establishes a channel to the OpenFlow Controller. The Controller, the Switch Proxy, and the Controller Proxies have Intel Core 2 Duo T7400 CPU. The Controller has 4GB RAM, the Switch Proxy has 1GB RAM, and each Controller Proxy have 512MB RAM. The Packet Filter has Intel Atom C2358 CPU, and 4GB RAM. The above PCs other than the Controller run Ubuntu 14.04.2, and the Controller runs Ubuntu 12.04. The switch has Intel Xeon X5255 CPU, 8GB RAM, and runs CentOS 6.2. All links are directly connected by 1Gbps, and the latency between them is less than one msec. To simulate channel failures between the Switch Proxy and the Controller Proxies, we set filter rules to the Packet Filter to discard packets from the Switch Proxy to the Controller Proxies if necessary.

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 since we have installed a packet filter rule on the Packet Filter. An interval of sending an echo request and a timeout value of waiting an echo reply are summarized in Tab. 1. Table 1 also include default values of the intervals and the timeout values in some controllers and switches[3]. A Controller Proxy sets a timeout for one second until receiving the same message from a Switch Proxy when the Controller Proxy receives a message from other Controller Proxies.

The procedure of the measurement is as follows. A measurement server collects and monitors logs from proxies, and controls the switch and the packet filter rules when necessary.

1. The Switch Proxy intercepts an echo reply message, and notifies to the management server.
2. One second later, the management server installs filter rules on the Packet Filter to discard packets from the Switch Proxy to the Controller Proxies.
3. The Packet Filter notifies the completion of the installation to the management server.
4. The management server asks the switch to send a port status message.
5. The proxies detect channel failures between them, and notify the failures to the management server. When the controller or the switch closes a channel, the proxies notify the termination of the channel to the management server.

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

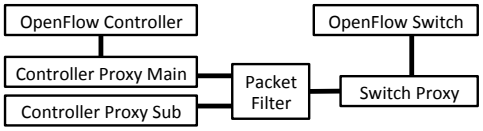

Figure 3: Connections for Evaluation

Table 1: 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

[2]http://zeromq.org/

[3]We cannnot find the keep-alive procedure in the Ryu source code. We cannot confirm that Ryu transmits echo requests periodically by running Ryu.

Table 2: 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

Table 3: 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

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 Proxy Main and Sub.

The results are summarized in Tab. 2. Without our proposed mechanism, delay includes the duration until next echo request is sent in addition to timeout values. We can see that the delay is reduced to timeout values with our proposed mechanism.

6.2 Overhead on Throughput and Latency

We have evaluated overhead on controller message processing performance and latency caused by our proposed mechanism. We run cbench[15] that fakes one switch on the switch, and the cbench controller in Trema at the controller. An RTT between two Controller Proxies is about 0.2 msec.

Table 3 shows average throughput and latency for 10 seconds. Failure Detection Disabled means we disabled our proposed mechanism. No Channel Failure means both connections between Switch Proxy and Controller Proxy Main and Sub are available. In Failure cases, a connection to a controller proxy noted is unavailable.

In terms of latency, the case "Failure: Controller Proxy Main" takes 1 msec longer than other cases. This is because the messages are transferred via Switch Proxy - Controller Proxy Sub - Controller Proxy Main. In other cases, the messages are transferred only via Switch Proxy and Controller Proxy Main.

In terms of throughput, we measured 30 percent decrease for "No Channel Failure" case from "Failure Detection Disabled" case, and 5 to 10 percent decrease from Failure Channel cases. In "No Channel Failure" case, Controller Proxy Main receives 1.5 times more messages than in "Failure Detection Disabled" case, including the messages from Switch Proxy, Controller, and Controller Proxy Sub. Overhead on processing our own failure detection mechanism can be measured by comparing "Failure Detection Disabled" case and Failure cases, and the overhead seems 10 to 15 percent.

7. DISCUSSION

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

We cannot see any overhead on latency when a Switch Proxy and a Controller Proxy connected to a controller directly establish a connection. When a connection between Switch Proxy and Controller Proxy Main is failed, the latency is increased because messages are transferred via Controller Proxy Sub. This route adds two kinds of delay to the overall latency, a message encode/decode delay in Controller Proxy Sub, and latency between two Controller Proxies. If the latency between the Switch Proxy and the Controller Proxies is large and the processing time in the controller is long, this overhead on latency will be negligible.

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

One of problems to shorten intervals of executing a keep-alive procedure is the number of messages that controllers process. If switches and controllers try to shorten failure detection delay to the timeout values without our proposed mechanism, the switches and the controllers should always exchange keep-alive messages. If the switches generate messages other than ones for the keep-alive mechanism at a low rate, our proposed mechanism works well without significantly increasing the number of messages that the controllers process, because our proposed mechanism generates additional messages only when a message is sent or received. Our proposed mechanism is not suitable to networks where the switches and the controllers generate message at high rate. In such cases, the switches and the controllers sometimes generate more messages than the cases of shortening the intervals of the keep-alive messages. We will need to address the problem in a different approach, such as adaptively sending keep-alive messages according to intervals of messages arrived.

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 flooding the messages to other controllers only when a controller has not received the messages from other controllers, like an idea of Trickle algorithm[8].

We do not note anything about the design of controllers. Some controllers has a function to elect a master controller for each switch, such as ONOS[1], 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, each channel state, or which channel controllers use for sending request messages, are provided by controller platforms. The controllers may run their coordination algorithm to choose such parameters.

8. CONCLUSION

We have proposed a mechanism to detect OpenFlow channel failures quickly with fewer messages, compared to a simple 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 inactive 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 timeout values and overhead on latency is negligible. The overhead on throughput cannot be negligible, but this is due to the increased number of messages at controllers.

Future works includes minimizing overhead on throughput by restricting messages shared among controllers, detecting failures of all channels to a switch at controllers, etc.

9. REFERENCES

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

[2] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller fault-tolerance in software-defined networking. SOSR '15, pages 4:1–4:12, Jun. 2015.

[3] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD). RFC 5880 (Proposed Standard), Jun. 2010.

[4] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs, and P. Skoldstrom. Scalable fault management for openflow. IEEE ICC 2012, pages 6606–6610, Jun. 2012.

[5] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. OSDI '10, pages 1–6, Oct. 2010.

[6] K. Kuroki, M. Fukushima, and M. Hayashi. Redundancy Method for Highly Available OpenFlow Controller. *International Journal on Advances in Internet Technology*, 7(1):114–123, Jun. 2014.

[7] K. Kuroki, N. Matsumoto, and M. Hayashi. Scalable OpenFlow Controller Redundancy Tackling Local and Global Recoveries. In *The Fifth International Conference on Advances in Future Internet*, pages 61–66, Aug. 2013.

[8] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. NSDI '04, pages 15–28, Mar. 2004.

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

[10] Open Networking Foundation. OpenFlow Switch Specification Version 1.5.1. Mar. 2015.

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

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

[13] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester. Fast failure recovery for in-band OpenFlow networks. DRCN 2013, pages 52–59, Mar. 2013.

[14] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester. OpenFlow: Meeting carrier-grade recovery requirements. *Computer Communications*, 36(6):656 – 665, 2013.

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

[16] A. Tootoonchian and Y. Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. INM/WREN 2010, pages 1–6, 2010.