Computer Communications 36 (2013) 656–665

Contents lists available at SciVerse ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

OpenFlow: Meeting carrier-grade recovery requirements ☆

Sachin Sharma *, Dimitri Staessens, Didier Colle, Mario Pickavet, Piet Demeester

Gaston Crommenlaan 8 bus 201, 9050 Ghent, Belgium

ARTICLE INFO

Article history:
Available online 24 September 2012

Keywords:
Carrier-grade
OpenFlow
Protection
Restoration

ABSTRACT

OpenFlow, initially launched as a technology-enabling network and application experimentation in a campus network, has a disruptive potential in designing a flexible network, fostering innovation, reducing complexity and delivering the right economics. This paper focuses on fault tolerance of OpenFlow to deploy it in carrier-grade networks. The carrier-grade network has a strict requirement that the network should recover from the failure within a 50 ms interval. We apply two well-known recovery mechanisms to OpenFlow networks: restoration and protection, and run extensive emulation experiments. In OpenFlow, the controlling software is moved to one or more hardware modules (controllers) which can control many switches. For fast failure recovery, the controller must notify all the affected switches about the recovery action within ms interval. This leads to a significant load on the controller. We show that OpenFlow may not be able to achieve failure recovery within a 50 ms interval in this situation. We add the recovery action in the switches themselves so that the switches can do recovery without contacting the controller. We show that this approach can achieve recovery within 50 ms in a large-scale network serving many flows.

1. Introduction

There is almost no practical way to experiment with new protocols in sufficiently realistic settings (e.g. at the scale carrying real traffic) to gain the confidence needed for their widespread deployment. In this context, OpenFlow [1] has caught the attention of many researchers and router vendors. It is developed in a clean-slate future internet program by Stanford University, which aims to offer a programmable network to test new protocols in current Internet platforms. If operators want to program the behavior of networking devices such as routers or switches, they require direct programming of the forwarding hardware. The core idea of OpenFlow is to provide direct programming of a router or switch to monitor and modify the way in which the individual packets are handled by the device. It is based on the fact that most modern routers/switches contain a proprietary FIB (Forwarding Information Base) which is implemented in the forwarding hardware using TCAMs (Ternary Content Addressable Memory). OpenFlow provides the concept of FlowTable that is an abstraction of the FIB. In addition to this, it provides a protocol to program the FIB via "adding/deleting/modifying" entries in the FlowTable. This is achieved by one or more separate devices (so-called controllers) that communicate with the OpenFlow switches via the OpenFlow protocol. The switch/router that exposes its FlowTable through the OpenFlow protocol is called an OpenFlow switch/router.

The standard switch/router consists of tightly interlinked elements that handle forwarding of packets (data plane) as well as controlling of forwarding (control plane). The control plane of these switches/routers implements almost all the standard protocols. However, only few of the protocols are required and effectively used. This makes switches/routers complex, expensive, and difficult to extend with new functions. OpenFlow addresses these issues by separating the control and forwarding plane. An OpenFlow network has a centralized programming model, where one or more controllers manage the underlying switches. This design is highly flexible, since it is the controller that can decide what actions (forward or drop) have to be taken for the different packets and at the same time forwarding can be done in hardware. Furthermore, the new functions can also be deployed very easily by just changing the software of one or more controllers.

An entry in the FlowTable consists of (1) a "packet header" that defines the flow, (2) "statistics" which keep track of matching packets per flow, and (3) "actions" which define how the packet should be processed. When a packet arrives at the OpenFlow switch, it is compared against the Flow Entries in the FlowTable. If a match is found, the actions of that entry are performed. If no match is found, the packet (a part thereof) is forwarded to the

☆ All the authors are working with the research group of Internet-Based Communication Networks and Services (IBCN), Department of Information Technology (INTEC), Ghent University – IBBT.

* Corresponding author. Address: Ghent University – IBBT, Department of Information Technology (INTEC), Gaston Crommenlaan 8 bus 201, 9050 Ghent, Belgium. Tel.: +32 9 331 49 77; fax: +32 9 331 48 99.

E-mail addresses: Sachin.Sharma@intec.ugent.be (S. Sharma), Dimitri.Staessens@intec.ugent.be (D. Staessens), Didier.Colle@intec.ugent.be (D. Colle), Mario. Pickavet@intec.ugent.be (M. Pickavet), Piet.Demeester@intec.ugent.be (P. Demeester).

0140-3664/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.comcom.2012.09.011

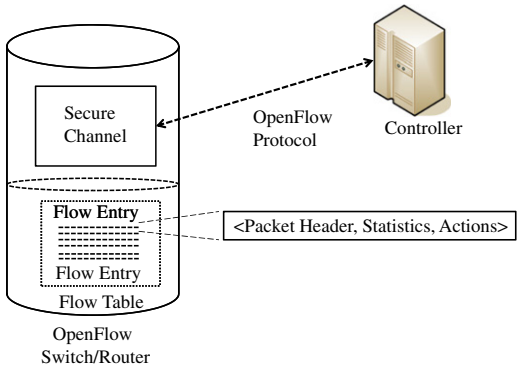

Fig. 1. OpenFlow principle.

controller over the secure channel (shown in Fig. 1). Thereafter, the controller can determine how the packet can be handled. It may return the packet to the switch indicating the forwarding port, or it may add the valid Flow Entries in the switches.

Large-scale experimental testbeds are available for researchers in the US through the GENI (Global Environment for Network Innovations) [2], in Japan through the JGN2plus research network [3], and in Europe through the OFELIA (OpenFlow in Europe: Linking Infrastructure and Applications) [4] projects. Several algorithms resulting from experiments on those testbeds have been deployed in many networks supporting various applications. Currently, industrial players such as Deutsche Telekom, Google, Microsoft, Verizon, and Yahoo have shown substantial interest towards OpenFlow and have formed ONF (Open Networking Foundation) [5] to standardize the OpenFlow protocol.

At present, one of the European projects named SPARC (SPlit ARchitecture Carrier-Grade Networks) [6] examines how carrier-grade networks can benefit from OpenFlow. Carrier-grade networks have a high capacity to support hundreds of thousands of customers and assume extremely high availability. Carrier-grade supports services such as cellular-telephone conversations, credit-card transactions that assume the availability of a reliable network. The requirement for the availability in commercial telecom networks is typically 99.999% [7]. The disruption of communication can suspend critical operations. In fact, there is a service level agreement between the business customer and a service provider to deliver a high-quality service. If a network operator is unable to meet the agreement, the service providers have to compensate for the loss of service. A failure recovery requirement in a carrier-grade network is sub-50 ms [8], which implies that the failure should be detected and the traffic should be recovered within a 50 ms interval.

Failure recovery in OpenFlow requires modification and addition of the Flow Entries in the OpenFlow switches. This paper studies fault tolerance of OpenFlow networks in large-scale carrier-grade networks. We implement two well-known mechanisms of failure recovery i.e. restoration and protection in OpenFlow networks. In the case of restoration, alternative paths are not established until a failure occurs. In the case of protection, the alternative paths are reserved before the failure occurs in the network. The controller in restoration must notify all the affected switches about a recovery action immediately. This leads to a significant load on the controller that delays the recovery action within the switches. We show that OpenFlow because of its centralized architecture may not be able to achieve failure recovery within a 50 ms interval in this situation. For protection, we add the recovery action in the switches themselves so that the switches can do recovery without contacting the controller. We perform an extensive emulation of our recovery mechanisms in different network scenarios and test OpenFlow with increased number of flows. We show that protection can achieve recovery within 50 ms in a large-scale network serving many flows.

The rest of the paper is organized as follows: Section 2 presents network resiliency, Section 3 describes the emulation environment, Section 4 presents results, Section 5 describes additional considerations, and finally Section 6 concludes.

2. Network resiliency

Network resilience is the ability to provide and maintain an acceptable level of service in presence of failures. We first describe general mechanisms that are used in carrier-grade networks to recover from the failure. Then, we describe the integration of those mechanisms in OpenFlow in order to fulfill the network availability requirements of carrier-grade networks.

The recovery mechanisms [10,11] that are used in carrier-grade networks are divided into two categories, i.e. restoration and protection. In the case of restoration, recovery paths can be either pre-planned or dynamically allocated, but resources required by the recovery paths are not allocated until a failure occurs. Thus, when the failure occurs, additional signaling is required to establish the restoration path. However, in the case of protection, the recovery paths are always pre-planned and reserved before the failure occurs. Hence, when the failure occurs, no additional signaling is needed to establish the protected path, traffic can immediately be redirected. In segment protection, an end-to-end working path is divided into segments, each of which is protected by a unique backup segment. However, in path protection, an end-to-end working path is protected by a unique backup path. There are different types of protection schemes available for carrier grade networks. These are 1 + 1, 1:1, 1:N and M:N protection [11].

Resilience is achieved in carrier-grade networks by first designing a network topology with the failures in mind in order to provide the alternate paths. The next step is adding the ability to detect the failures and react to them using a proper recovery mechanism. Loss of Signal (LOS) and Bidirectional Forwarding Detection (BFD) [9] are widely used to detect the failures in carrier-grade networks. LOS can detect the failure in one particular port of the forwarding device, whereas BFD detects the failures in the path between any two forwarding devices. BFD is a simple Hello protocol that in many aspects is similar to the detection components of many routing protocols like OSPF (Open Shortest Path First). A pair of systems (end-to-end devices) transmits BFD packets periodically between each other, and if a system stops receiving the BFD packets, the path between the neighboring systems is assumed to have failed.

2.1. Resilience for OpenFlow network

The failure can be detected in OpenFlow by a Loss of Signal. It causes an OpenFlow port to change to the "down" state from the "up" state. The OpenFlow port is the port bounded to the OpenFlow switch to transmit and receive packets. This mechanism detects only the link-local failures and may be used in restoration. However, for path protection, end-to-end failure detection in any path in the forwarding switches is required.

Recovery in OpenFlow can be done in essentially two different ways. One approach is to support the recovery mechanisms of a specific implemented protocol like MPLS (Multi Protocol Label Switching Protocol) [12,13] into OpenFlow. The other approach is to build resilience, supporting recovery of arbitrary flows, regardless of the type of traffic they carry. We explore this option for OpenFlow networks.

Fast restoration in OpenFlow can be implemented in the controller. It requires an immediate action of the controller after a

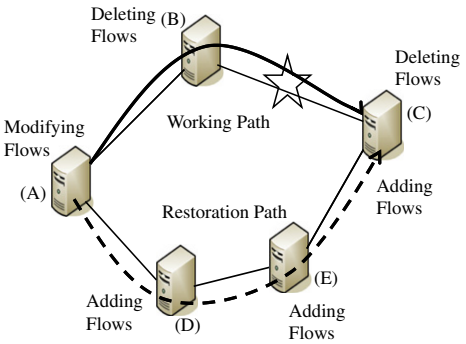

Fig. 2. Restoration mechanism.

notification of a change in a link status. Failure recovery can be performed by removing affected Flow Entries and installing new entries in the affected switches as fast as possible following the failure notification [14,15]. The restoration mechanism can be seen in Fig. 2 which consists of the OpenFlow switches A, B, C, D and E. Assuming the controller knows the network topology, we can calculate a path from a source node to a destination node. In Fig. 2, the controller first installs the path $< ABC >$ by adding the Flow Entries in the switches A, B and C. Once the controller receives the failure-notification message of the link BC, it calculates the new path, $< ADEC >$. For the OpenFlow switch A, as the flow in the Flow Entry for the working path $< ABC >$ and restoration path $< ADEC >$ is identical but the action is different (i.e. to forward to the switch B or D), the controller modifies the Flow Entry at A. In addition, for the restoration path $< ADEC >$, there are no Flow Entries installed in the switches D, E, and C related to this flow, the controller must add these entries in the respective switches. The Flow Entry in C for the working path $< ABC >$ and the restoration path $< ADEC >$ is different since the incoming port is assumed to be a part of the matching header in the Flow Entry. Once all the affected flows are updated/installed in all the switches, the flow is recovered. After the immediate action of restoration, the controller can clean up the other switches by deleting the Flow Entries at B and C related to the older path $< ABC >$.

In the time between the failure detection and the completion of restoration, the data packets may be lost. In order to further reduce the packet loss resulting from delay in the restoration action, we can turn to protection. Protection removes the need of the OpenFlow switches to contact the controller for modification and addition of the Flow Entries to establish the alternative path. This can be accomplished by pre-computing the protected path and establishing it together with the working path. In 1 + 1 protection, the traffic is duplicated at both the protected and working path, and in 1:1 protection, the traffic is transmitted to the protected path upon the failure at the working path. Protection allows fast recovery but requires a larger FlowTable.

To implement a protection scheme, we applied the Group Table concept specified for OpenFlow in its version 1.1 [16]. Unlike the FlowTable, a GroupTable consists of Group Entries that in turn contain a number of actions. To execute any specific entry in the GroupTable, the Flow Entry forwards packets to a Group Entry having a specific group ID. Each Group Entry consists of the group ID (must be unique), a group type and a number of action buckets (shown in Fig. 3A). An action bucket consists of an alive status (e.g. watch port and watch group in OpenFlow version 1.1 [16]) and a set of actions that are to be executed based on the value of the associated alive status. OpenFlow introduces the fast-failover group type [16] in order to perform fast failover without needing to involve the controller. This group type is important for our protection mechanism. Any group entry of this type consists of two or more action buckets with a well-defined order. A bucket is considered alive if its associated alive status is within a specific range (i.e. watch port or watch group is not equal to 0xffffffffL). The first action bucket describes what to do with the packet under normal conditions. If this action bucket has been declared as unavailable that is due to change in status of a bucket (i.e. 0xffffffffL), the packet is treated according to the next available bucket. The status of the bucket can be changed by the monitored port going into the "down" state or through other mechanisms such as BFD.

In our 1:1 path protection mechanism, the above Group Table concept of OpenFlow is used without any modification. We used BFD to detect the failures. Once BFD declares the failure in the working path, the action bucket associated with this path in the GroupTable is made unavailable by changing the value of the alive status.

1:1 path protection can be seen in Fig. 3B. When a packet arrives at the OpenFlow switch (A), the controller installs two disjoint paths: one in $< ABC >$ and the other in $< ADEC >$. The OpenFlow switch (A) is the node that actually needs to take the switching action on the failure condition i.e. to send the packet to B during the normal condition and to send the packet to D during the failure condition. For this particular flow of the packet, the Group Table concept can be applied at the OpenFlow switch (A). The Group Entry in this switch may contain two action buckets: one for output port B and the other for output port D. Thus, one entry is added in the FlowTable of switch A, which points the packet to the above Group Entry in the GroupTable. For the other switches, B and C for the working path, and D, E and C for the protection path, only al Flow Entry is added. Thus in our case, the switch in C contains two Flow Entries: one for the working path $< ABC >$ and the other for the protection path $< ADEC >$. Once a

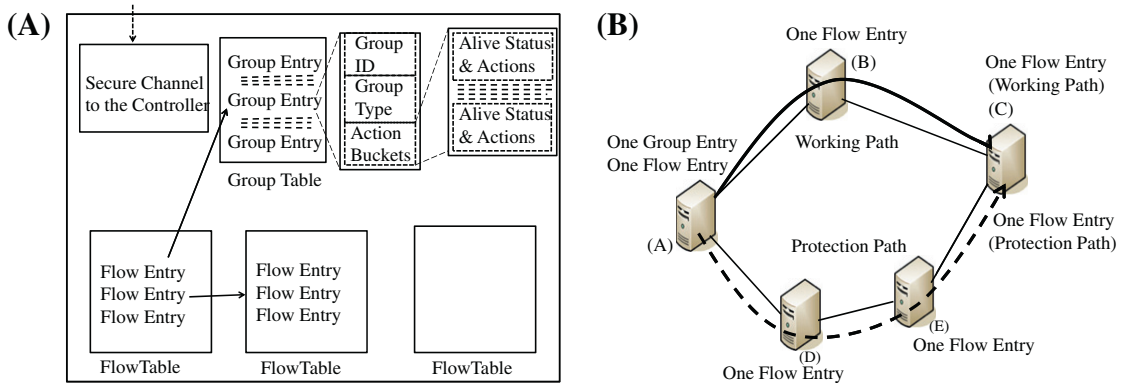

Fig. 3. (A) Group table concept (B) protection mechanism.

failure is detected by BFD in the working path, the OpenFlow switch (A) can change the alive status of the first bucket in the Group Entry. Thus action related to the next bucket, whose output port is D, can be taken. As the Flow Entries in D, E and C related to $< ADEC >$ path are already present, there is no need to install these in the respective switches upon failure.

3. Emulation environment

In [14], we have shown that restoration meets the 50 ms recovery requirement when there are two flows in a small network. However, as carrier-grade networks are typically large, serving many flows, we emulated restoration and protection on increasing number of flows in a large-scale European network. The emulated European topologies, testbed and methodology are described in this section.

3.1. Emulation testbed and topologies

Our virtual-wall emulation testbed is a generic test environment (based on emulab [17]) for advanced network, distributive software and service evaluation. It consists of 100 physical nodes (dual processor, dual core servers and up to six 1 Gb/s interfaces per node) interconnected by a non-blocking 1.5 Tb/s Force10 Ethernet switch (shown in Fig. 4A). It uses the concept of Virtual LAN (VLAN) to build arbitrary topologies. The nodes in our testbed may be used for network emulation (bandwidth, delay, packet loss) of a real network environment.

We emulated an extensive failure recovery experiment using the topologies that were developed within the COST 266 action project. In this project, a basic reference topology (BT topology in Fig. 4B) and variations of the BT topology (e.g. Fig. 5A and 5B), suited for a pan-European network, were designed. The variations of the BT topology, Core Topology (CT), Large Topology (LT), Ring Topology (RT) and Triangular Topology (TT), were obtained by varying the total number of nodes and the degree of meshedness. The CT and LT differ with respect to the number of nodes. The BT consists of 28 nodes and the CT consists of 16 nodes. The other derived topologies contained the same number of nodes as the BT, but the difference lies in the degree of meshedness. The topology called Ring topology (RT), Fig. 5B, is much sparser than the BT. The details of these topologies can also be found in [19]. We used the BT, CT and RT topologies for our experiment.

In our experiment, each node of the COST 266 topologies acted as an OpenFlow switch. In our emulation, we connected a server to each OpenFlow switch (not shown in Fig. 4B and 5). We built these topologies in our virtual-wall testbed. A virtual-wall node (shown in Fig. 4A) acted as an OpenFlow switch as well a server in our emulation. In our emulation, one CPU core of a virtual-wall node has been assigned to an OpenFlow switch and another CPU core has been assigned to a server. Each of the OpenFlow switches has also been provided a dedicated interface to a switched Ethernet LAN, which establishes a connection to the single controller i.e. out-of-band connection. Out-of-band means that there is a separate channel for the control and data plane i.e. the failure in the data plane does not affect communication between the switches and the controller.

3.2. Emulation methodology

There are many extensions for the OpenFlow protocol. These extensions have been released publicly in the form of OpenFlow

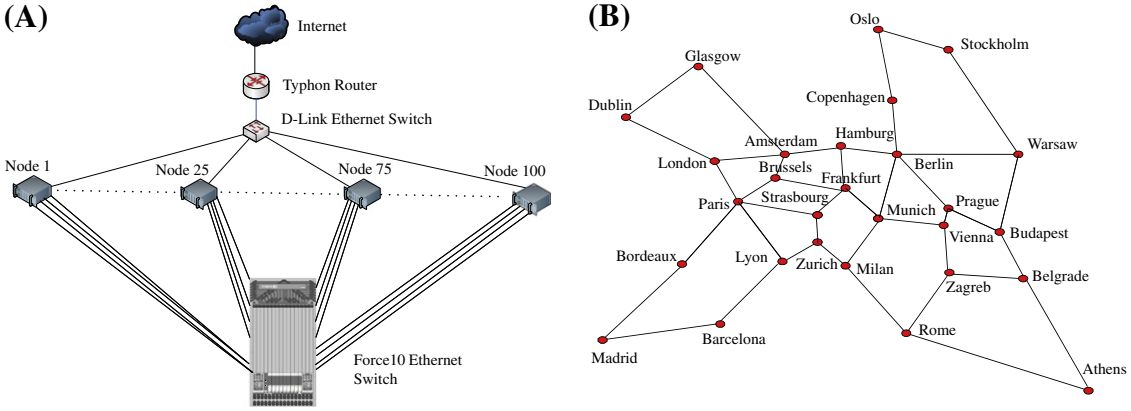

Fig. 4. (A) Virtual wall testbed (B) BT topology.

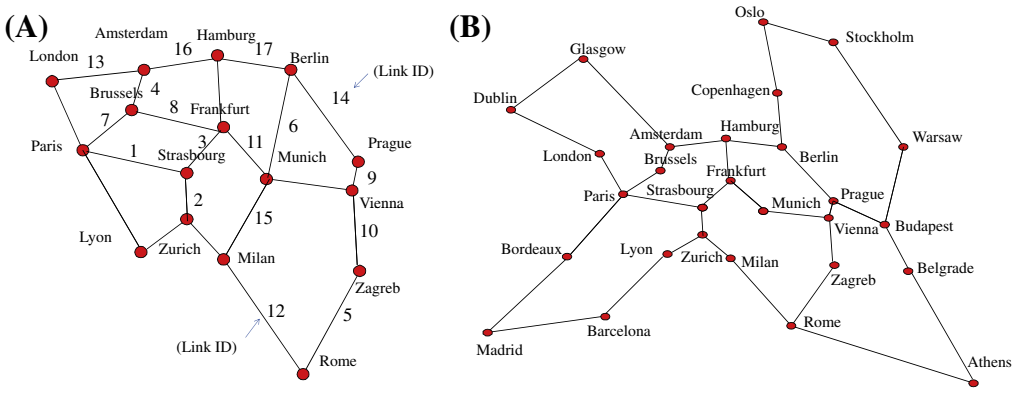

Fig. 5. (A) CT topology (B) RT topology.

versions. In April 2012, the OpenFlow version 1.3 has been released by ONF. In addition, many OpenFlow controllers are also available for controlling OpenFlow networks. These are NOX, Beacon, Onix, Helios and Maestro. We implemented restoration and protection in the NOX controller and the OpenFlow version 1.1 (developed by Ericsson [18]) and used these for our emulation.

To evaluate the recovery time, each server generated packets to all other servers using the Linux kernel module PKTGEN. Each server sent packets to all the other servers at the constant interval of 6 ms. The size of the PKTGEN packets was 64 bytes. We manually configured the routing table in each server to transmit the packets to the OpenFlow network.

For failure detection in protection, each working path was monitored by adding an additional BFD flow in the OpenFlow switches. BFD transmits a failure notification message when its session breaks. To receive the failure notification message, a virtual link (the link between veth1 and veth2 in Fig. 6) has been created between the OpenFlow switch and BFD (two different processes in each OpenFlow switch). Furthermore, an alias of the OpenFlow port (eth1:1) has been created for BFD to receive the packets from the OpenFlow port (eth1). The BFD failure detection time in our emulation was 40 to 44 ms. For restoration, we did not establish a BFD session. The OpenFlow switches detected the failure when the LOS was declared by the port as a "port down" event.

We derive an analytical model for the recovery time in the next section. The first experiment was performed to measure parameters of the analytical model. We used these parameters to verify our model. The second experiment was carried out on the CT topology. In this experiment, each server sent two different flows to all other servers. This experiment was performed with 480 flows in the network. The aim of this experiment was to find the recovery time experimentally and compare with the mathematical results that are calculated via our analytical model. The third experiment was similar to the second experiment but was performed with all the different topologies. The aim of this experiment was to compare the recovery time in different network scenarios. The fourth experiment was performed using the node failure instead of a link failure. The aim of this experiment was to test OpenFlow with a node failure condition. The fifth experiment was performed by increasing the number of flows in the CT topology. We increased the number of flows up to 24000 in this experiment. The aim of this experiment was to do a scalability experiment (i.e. how the number of flows affects the recovery time) in an OpenFlow network.

We now describe the second experiment in detail. In this experiment, we break a link at 0 s and find the recovery time. We describe the link break by failing the London–Amsterdam link in the CT topology (Fig. 7). Fig. 7 shows the traffic that was captured using the tcpdump utility in the NOX controller. At the beginning of the experiment, there were 16 spikes (from −106 to −92 s) after each one-second interval. These spikes were due to the traffic from the OpenFlow switches to learn the path to the destination. The one-second between the spikes occured because we have started sending the pktgen traffic after waiting one-second between each server. The one-second interval was used to avoid overloading the NOX controller as the switches can try to establish too many flows in a short time span. The spikes in Fig. 7B are higher than the spikes in Fig. 7A because protection establishes an alternative path together with the working path. In protection, each OpenFlow switch also established the BFD sessions (the spikes from −78 to −64 s in Fig. 7B) for each different path. There were small spikes periodically in the controller traffic. These were the echo messages that were sent to check aliveness of the controller links. The height and number of these spikes are different in Fig. 7A and 7B. This was due to the minor time difference between the start of both experiments. At the 0 s, we have failed the link London–Amsterdam by disabling the Ethernet interface at London. When the OpenFlow switch (London) detected this failure, the notification message was sent to the controller. Since the controller in restoration starts recovery upon a failure notification, there is a large spike at around 0 s in Fig. 7A. These were the flow-mod messages and the acknowledgements of these messages sent to reestablish the new path. In protection, the backup flows were installed before the failure occurs, the controller does not take any action upon failure. Therefore, we do not see any spike at 0 s in Fig. 7B.

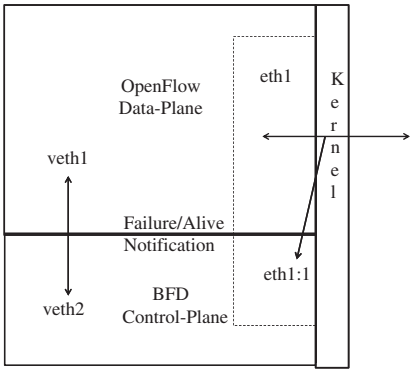

Fig. 6. Integration of BFD in OpenFlow.

Fig. 7 shows that in normal operation, the control network load is generally orders of magnitude lower. Implementing a high speed control network only for restoration (shown by critical time in Fig. 7A) will probably not make sense. Implementing protection mechanisms in the switches will be more cost-efficient, it may slightly increase the bandwidth requirement at flow setup time due to extra protection information to be sent to the switch, but highly decrease the bandwidth requirements during the failures by allowing the switch to perform the protection switching without the controller interference.

4. Results

This section is structured according to the number of different experiments performed for our emulation.

4.1. Analytical model and measurement of its parameters

In this section, we derive an analytical model to calculate the failure recovery time in an OpenFlow network. Our model is based on the failure recovery model described in [10,20]. We extended this model for our implemented restoration scheme.

Fig. 8 shows the recovery time in restoration i.e. when the failure is detected (T_{FD}), the failure notification message is sent, the affected flow is searched (T_{LU}), the new path is calculated (T_{CALC}) and then the flow-mod messages (T_{FM}) are sent, then again the next affected flow is searched (T_{LU}) and so on. When the switches receive the flow-mod message, the FlowTable is also updated with the new Flow Entry (T_{UPDATE}). We calculate the mathematical expression for the restoration time (T_R) for N affected flows (f) and it is shown in Eq. 1. T_{PROP} in Eq. 1 is the propagation time. $T_{LU,f}, T_{CALC,f}$ and $T_{FM,f}$ are the T_{LU}, T_{CALC} and T_{FM} for the affected flow f.

$$T_R = T_{FD} + T_{PROP} + \sum_{f=1}^{N}(T_{LU,f} + T_{CALC,f} + T_{FM,f}) + T_{PROP} + T_{UPDATE} \quad (1)$$

In our implemented mechanism, the affected port is searched in the flow path linearly for the link failure. If it is found, the flow is declared to be affected. The lookup time for the number of flows is shown in Fig. 9A.

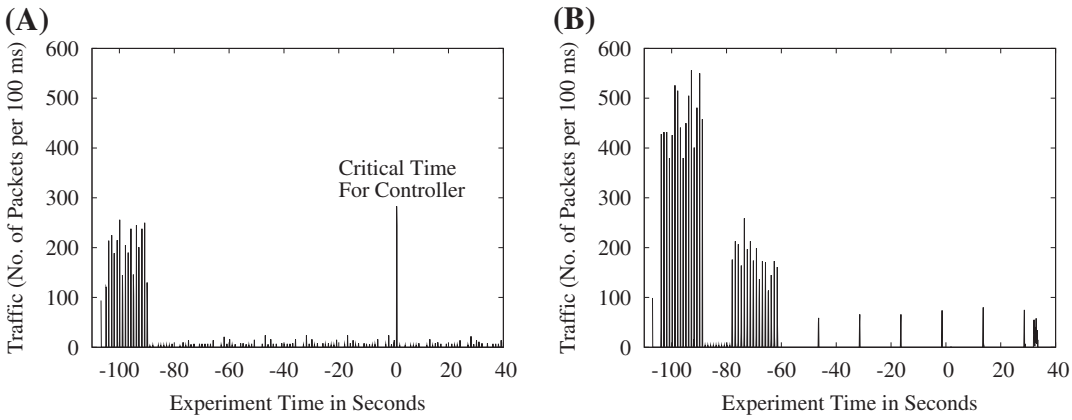

Fig. 7. (A) Restoration (NOX intensity) (B) protection (NOX intensity).

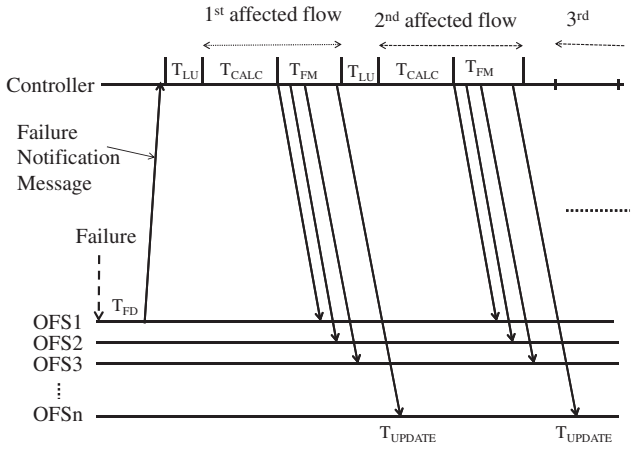

Fig. 8. Analytical model for restoration.

If the flow is declared as affected, a new shortest path is calculated in our algorithm. Fig. 9B shows the path calculation time in the RT, BT and CT topology. The paths calculated in our experiment are the shortest paths obtained via the Dijkstra algorithm. The time complexity of the implemented Dijkstra algorithm is $O(n^2)$ where n is the number of nodes. This time can be decreased to $E + n \times log(n)$ (E is the number of edges) by using a Fibonacci heap for storing the topology graph. The path calculation times are measured for all topologies used in the experiment and is shown in Fig. 9B.

We measured the flow-mod transmission capacity of the controller over a 1000 Mbps link. Fig. 10A shows the transmission capacity of the controller link when the flow-mod messages were transmitted over 1,2,4 and 12 TCP connections. We observed 38%, 100%, 100%, and 100% of the CPU utilization for 1,2,4 and 12 TCP connections respectively. The flow-mod transmission capacity decreased when the controller started transmitting flow-mod messages on all the 12 TCP connections. This is because of context switching between the different TCP connections. Fig. 10A shows the maximum flow-mod transmission capacity as 15000 to 40000 packets per second.

The FlowTable in the OpenFlow 1.1 software, implemented by Ericsson, is a linear table. If a match is found, the entry is modified in the table. Otherwise, a new entry is added at the end. The time for modification (minimum, average and maximum) and addition of an entry with respect to the number of Flow Entries in the table is shown in Fig. 10B.

We calculated each time (flow lookup, path calculation, flow-mod transmission and flow-entry addition) separately to determine the recovery time (Eq. 1) which we compare with the experimental results in the next section.

Now we describe the time complexity of our protection mechanism. The recovery time in the protection mechanism depends on the time the switch takes to modify the alive-status of the Group Entries. The switch takes O (1) time (approximately 5.8 microsecond) to modify the alive-status of one Group Entry. For n Group Entries, this results in an $O(n)$ time. For our experiment, the number of Group Entries in a switch is equal to the number of different paths that are established by it. Thus, the number of Group Entries (per switch) for the CT, BT, and RT topology is 15, 27 and 27 respectively.

4.2. Emulation results

We now show the results of the experiment in which the link London–Amsterdam was failed in the CT topology (second experiment). Fig. 11 shows the traffic at the link London–Amsterdam (from −0.2 to 0.3 s), which was captured at Amsterdam. As the port at London was disabled at 0 s, London stopped transmitting the packets at this link. However, as the port at Amsterdam was not disabled, Amsterdam continued transmitting traffic on the same link until the controller establishes the new fault-free paths. There is no traffic in Fig. 11A after 0.240 s because all the traffic has been switched to some other fault-free path. However, the total traffic becomes equal to the BFD traffic after about 0.05 s in Fig. 11B. This is because only BFD traffic remained on this link. It was not switched to some other path.

Fig. 12 shows the traffic on the link London-Paris. Traffic London (Server) in Fig. 12 is the traffic from and towards London (server). The traffic from −0.2 to 0.4 s is shown in Fig. 12. After the failure recovery at 0 s, this was only the link connecting London, so all the traffic from and towards London (server) has to pass through this link. At the time of the link failure (at 0 s), Fig. 12 shows a drop in the total PKTGEN traffic on this link. The dropped traffic was the traffic that was coming from the link London–Amsterdam to the link London-Paris. Restoration/protection reroutes the affected traffic. Fig. 12A shows that there is a drop in the total traffic for approximately 0.190 s, followed by a step-wise increase in traffic until 0.240 s when all flows in our experiment were restored. Fig. 12B shows this time as approximately 0.040 s, followed by a stepwise increase until 0.050 s for protection. Fig. 12B shows a small decrease in the BFD traffic after failure because the BFD traffic from the link London–Amsterdam was completely lost after the link failure.

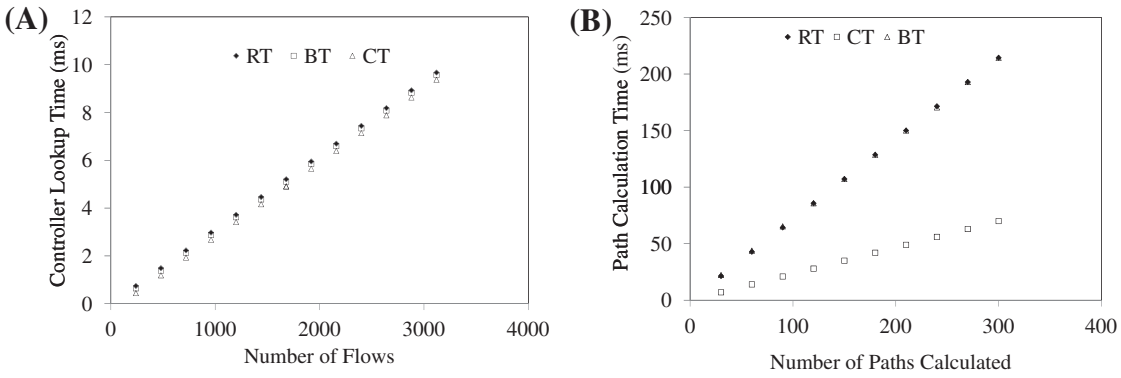

Fig. 9. (A) Flow lookup time (affected/unaffected) (B) controller path calculation time.

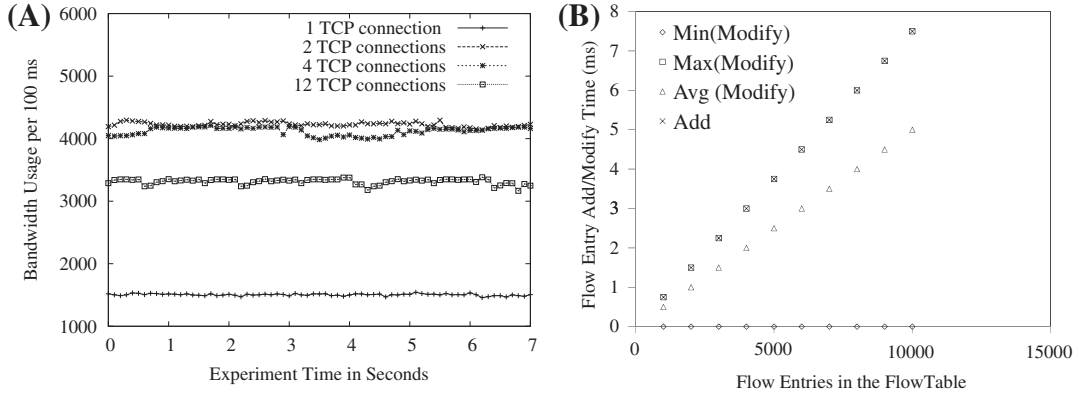

Fig. 10. (A) Controller flow-mod transmission capacity (B) flow entry modification/addition time.

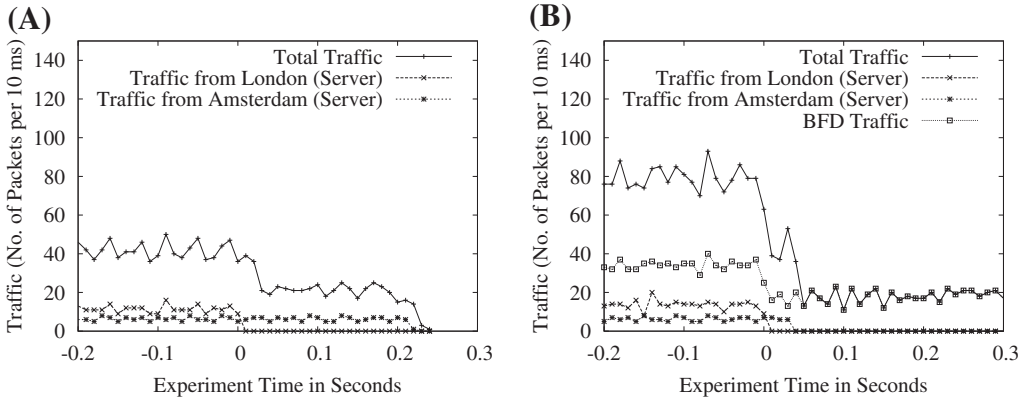

Fig. 11. (A) Restoration (traffic on the affected link) (B) protection (traffic on the affected link).

In our restoration mechanism, London detected the failure at about 0.187 s. However, restoration took approximately 0.054 s to restore all the flows (0.187 to 0.240 s). Fig. 13 shows a detail of 0.046 s interval (0.187 to 0.232 s) in which the NOX controller searched the affected flow, calculated the new path and sent the flow-mod traffic to the switches.

We did the link failure experiment for all the indicated links of Fig. 5A. The results are depicted in Fig. 14A. The x-axis shows the ID of the broken links and the number of affected flows. The y-axis shows the minimum restoration/protection time (the time it took to recover the first flow), the maximum restoration/protection time (the time it took to recover all the flows) and the average restoration/protection time (the expected time for any flow to be recovered after a failure). The recovery time is calculated as the number of packets dropped multiplied by the packet send interval. The links are ordered from left to right according to the number of flows that were affected by the link failure (Fig. 14A).

The Fig. 14A shows the difference in the time when the first flow was restored after each failure. The reason behind this is that the switches detected the failure at different time points. In our Linux node, it is difficult to measure this failure detection time in ms as there is some random time between the moment we break the link and the moment the Linux system has actually disabled the link (both receive and transmit disabled). The first flow was re-

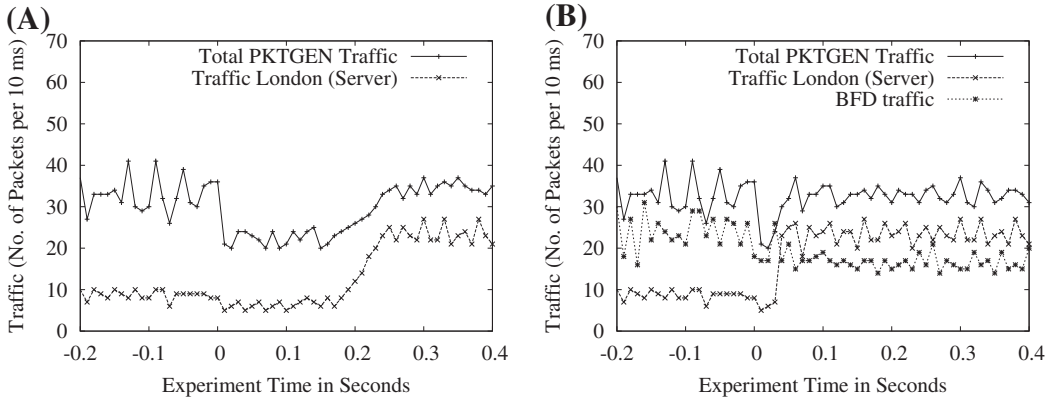

Fig. 12. (A) Restoration (traffic on restored link) (B) protection (traffic on protected path).

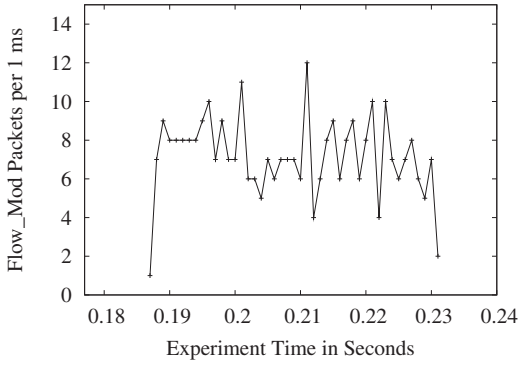

Fig. 13. NOX Flow-mod traffic in restoration.

stored in 2 to 3 ms. So the failure detection time in our experiment was approximately equal to the time when the first flow was restored. In Fig. 14A, the maximum restoration time after failure detection is shown by the link ID 17, which is equal to 60 ms. Fig. 14A shows the dependence on the number of flows that have been restored. Fig. 14A also shows that all the flows were protected between 42 to 48 ms in protection. This includes the failure detection time of BFD which was approximately 40 to 44 ms. Fig. 14B shows the difference between the analytical and experimental calculation of the restoration and protection time. In the restoration calculation, the failure detection time is assumed the time when the first flow restored in our experiment. In the protection calculation, it is assumed 44 ms. As the average path length in the CT topology is 3, we assumed that the controller has transmitted 4 flow-mod message per affected flow in restoration. Fig. 14B shows the difference of ± 9 ms for restoration and the difference of -2 to + 4 for protection. This difference includes the error in the recovery time calculation that is generated by the packet send interval (6 ms) in our experimental results.

Now we show the results of the link failure and the node failure experiment on all different topologies (the third and fourth experiment). In the link failure experiment, a link was broken, and the number of affected flows and the recovery time were calculated. In the node failure experiment, the failure was given by running "poweroff" command on the CT OpenFlow switches that are present in all the topologies, and the number of affected flows and the recovery time were calculated. In this experiment, the controller receives a FIN message because an OpenFlow switch disconnects from the controller. The controller then starts the same restoration activity as performed in the link failure experiment. The results of both experiments are depicted in Fig. 15. The x-axis shows the number of affected flows. The y-axis shows the recovery time after the first flow was restored in the experiment. (Link) and (Node) in Fig. 15 are referred to the link and node failure experiment respectively. The number of servers in the CT, BT and RT are 16, 28 and 28 respectively. As each server transmitted packets to all the other servers present in the topology, we observed different number of flows affected by failing the same link or the same node in different topologies. Fig. 15 shows that the restoration time depends on the number of affected flows in all the topologies. The results also show that the CT topology has less restoration time than the BT or RT topology because the path calculation time in the CT is less than the BT or RT topology.

To test scalability, we did the experiment on the CT topology, where the number of flows from each server was increased by a factor n (1 to 100) and the link London–Amsterdam was failed during the experiment (the fifth experiment). The factor n means each server transmitted n different flows to all other servers. In the factor 1, there were 240 flows in the network and 33 were affected by the failure. In the factor 100, there were 24000 flows in the network and 3300 were affected by the failure. The experimental and analytical results are shown in Fig. 16. We found a linear increment in the restoration time. We observed approximately 2.5 s restoration time when we increased the number of flows by the factor 100. However, in protection, we did not observe dependence on the increased number of flows. This is because in protection, we established 15 Group Entries (per switch) for all the flows in the CT topology, and modification of the affected entries has taken less than a 1 ms time ($O(n)$).

We evaluated restoration and protection in mesh topologies. Carrier-grade networks often feature a ring topology for the aggregation segments. For resiliency on rings, typically protection is used, as each switch has only two directions. If a connection is broken, traffic is sent along the other direction on the ring. A standardized protection solution for packet networks is Resilient Packet Ring (RPR, IEEE 802.17). RPR has two modes of protection, wrapping and steering. In wrapping, when a failure is detected, traffic going towards and from the failure direction is wrapped (looped) back to go in the opposite direction on the other ring (subject to the protection hierarchy). In steering, traffic is redirected in the source node to the opposite ring. Wrapping and steering can be easily implemented in OpenFlow by the GroupTable concept where the first action bucket in a Group Entry contains the action for the working path and the second action bucket contains the action for the protection path. Performance-wise, this will perform similarly (or even better, since only 2 action buckets are needed) to the protection on a mesh, so 50 ms protection in Openflow can be met on a ring.

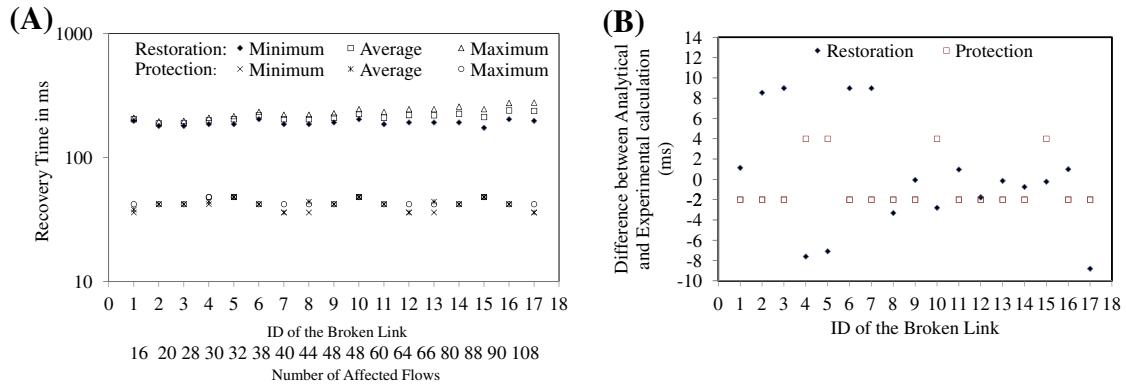

Fig. 14. (A) Recovery time (experimental) (B) recovery time difference.

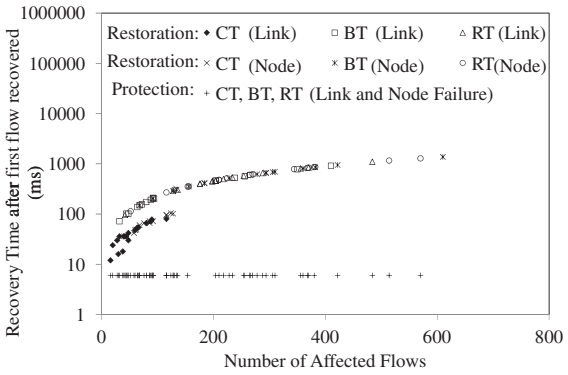

Fig. 15. Link and node failure experiment on the CT, BT and RT topologies.

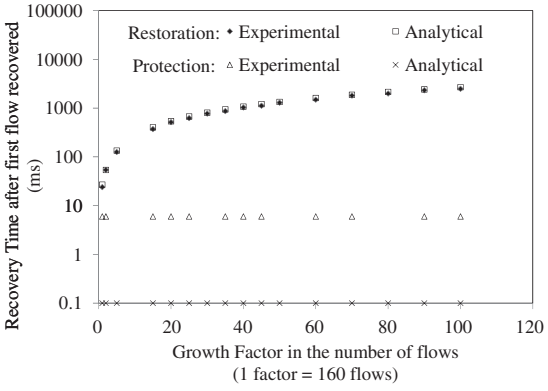

Fig. 16. Scalability experiment.

5. Additional considerations

5.1. Memory size requirement in protection

Our results show that protection is better than restoration because the former reduces the time required for fault recovery and avoids the sudden increase of traffic load in the controller at the time of failure detection. However, protection needs to maintain additional information of alternative paths together with the working path. Thus, the memory requirement in protection is more than restoration.

In restoration, the controller replaces the Flow Entry of the working path in the ingress OpenFlow switch (e.g. switch A in Fig. 2), which implies that one Flow Entry per flow is required at any time in this switch. However, the ingress OpenFlow switch for protection (e.g. switch A in Fig. 3B) installs an additional Group Entry for failure recovery. Thus, the additional memory requirement of the ingress OpenFlow switch for protection is the size of GroupTable i.e. the number of Group Entries in the GroupTable.

If d is the degree of the ingress OpenFlow switch, then the maximum number of output actions for the working paths can be d and the maximum number output actions for the protection paths can be $d-1$. Thus, the additional memory requirement of the ingress OpenFlow switch for protection (i.e. the maximum size of GroupTable) is $d \times (d-1)$.

As the TCAM memory is expensive, it is the size requirement of this memory which is important for the OpenFlow switches. The size requirement of this memory depends on implementation of the OpenFlow switch. The OpenFlow switch implemented in HP procurve 5400 zl series [21] manages FlowTables in hardware and in software. The FlowTable in software has the full set of Flow-Entries, and the FlowTable in hardware has the subset of Flow Entries. The FlowTable in hardware is managed using a TCAM that translates a Flow Entry into a TCAM entry. When the packet does not match to any TCAM entry, the packet is forwarded to software. If the matching entry is found in software, it is installed in the TCAM and the packet is forwarded.

Thus, the Flow Entries related to the protection path can be installed in software, and the Flow Entries related to the working path can be installed in hardware. Once a failure occurs in the working path, the Flow entries related to the protection path can be moved to the TCAM. Furthermore, as the GroupTable requires a 32 bit match on the Group ID, the GroupTable can be present in the static or dynamic RAM. Thus, for this type of OpenFlow switch implementation, the protection path does not require the additional Flow Entries to be installed in the TCAM. Therefore, the TCAM memory requirement of protection can be equal to restoration.

5.2. Reliability of the control plane

In this paper, we considered the failure in data-plane side i.e. recovery from failure when a data traffic path fails. However, because Openflow is a split architecture (relying on the controller to take action when a new flow is introduced in the network), reliability of the control plane is also an important issue. The controller should also be resilient against targeted attacks. There are multiple options for control plane resiliency. One can provide two controllers, each on a separate control network and when a connection to one controller is lost, the switch can switch over to the backup network. This is a very expensive solution. Another option is to try to restore the connection to the controller by routing the control traffic over the data network. When a switch loses

connection to the Openflow controller, it can send its control traffic to a neighboring switch, which will require the controller to detect such messages and establish Flow Entries for routing the control traffic through this neighbour switch. This through-the-data-plane solution is an intermediate step towards full in-band control. An effective scheme for carrier grade networks may be to implement out-of-band control in the failure-free scenario, switching to in-band control for switches which lose the controller connection after a failure. In-band control is supported in the Openflow specification. There could be a situation where the controller itself crashes. In this situation, we can have two controllers so that when the one controller crashes then OpenFlow switches can rely on a backup controller. In future work, we will consider those situations in OpenFlow networks.

6. Conclusion

In this paper, we have presented restoration and path protection for OpenFlow to deploy it in a carrier grade network. We ran extensive experiments on pan-European network topologies and tested OpenFlow in a real environment via our virtual-wall testbed facility. We showed that the low-cost devices in OpenFlow can restore traffic, but its dependency on the centralized controller means that it will be hard to achieve 50 ms restoration in a large-scale carrier grade network. We used the group table concept (recently proposed for OpenFlow) to implement protection. In this paper, we proposed the first implementation of protection based on the group table concept. Finally, we showed that OpenFlow can achieve the carrier-grade requirement of a 50 ms interval if protection is implemented in these networks to recover from the failure.

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7) under Grant agreement n^o 258457 (SPARC) and n^o 258365 (OFELIA).

References

[1] N. McKeown, T. Andershnan, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in campus networks, ACM Computer Communication Review 38 (2) (2008) 69–74. New York, USA.
[2] GENI. Avilable from: <http://www.geni.net>.
[3] JGN2plus research and development test-bed network. Avilable from: <http://www.jgn.nict.go.jp/>.
[4] OFELIA. Avilable from: <http://www.fp7-ofelia.eu/>.
[5] ONF. Avilable from: <https://www.opennetworking.org/>.
[6] SPARC. Avilable from: <http://www.fp7-sparc.eu/>.
[7] D. Collins, Carrier-Grade Voice Over IP, McGrawa Hill, 2000.
[8] B. Jenkins, D. Brungard, M. Betts, N. Sprecher, S. Ueno, MPLS-TP Requirements, RFC 5654, IETF, 2009.
[9] D. Katz, D. Ward, Bidirectional Forwarding Detection, RFC-5880, IETF, 2010.
[10] J.P. Vasseur, M. Pickavet, P. Demeester, Network Recovery: Protection and Restoration of Optical, SONET-SDH, IP and MPLS, Morgan Kaufmann, 2004.
[11] E. Mannie, D. Papadimitriou, Recovery (Protection and Restoration) Terminology for Generalized Multi-Protocol Label Switching (GMPLS), RFC 4427, IETF, 2006.
[12] A.R. Sharafat, S. Das, G. Parulkar, N. McKeown, MPLS-TE and MPLS VPNs with Openflow, ACM Computer Communication Review 41 (4) (2011) 452–453. New York, USA.
[13] D. Jocha, A. Kern, A. Takacs, P. Skoldstrom, MPLS-Openflow Based Access/Aggregation Network, GENI Engineering Conference, Puerto Rico, US, 2011.
[14] S. Sharma, D. Staessens, D. Colle, M. Pickavet, P. Demeester, Enabling Fast Failure Recovery in OpenFlow Networks, Design of Reliable Communication Networks (2011) 164–171. Krakow, Poland.
[15] D. Staessens, S. Sharma, D. Colle, M. Pickavet, P. Demeester, Software defined networking: meeting carrier grade requirements, Local & Metropolitan Area Networks (2011) 1–6. North Carolina, USA.
[16] OpenFlow Switch Specification: Version 1.1.0 (Wire Protocol 0x02): <http://www.openflow.org/>, 2011.
[17] Emulab Network Emulation. Available from: <http://www.emulab.net011>.
[18] Ericsson OpenFlow and NOX Controller Software. <https://github.com/TrafficLab>.
[19] S.D. Maesschalck, D. Colle, I. Lievens, M. Pickavet, P. Demeester, C. Mauz, M. Jaeger, R. Inkret, B. Mikac, J. Derkacz, Pan-European optical transport networks: an availability-based comparison, Photonic Network Communications 5 (3) (2003) 203–225.
[20] V. Sharma, F. Hellstrand, Framework for Multi-Protocol Label Switching (MPLS)-based Recovery, RFC 3469, IETF, 2003.
[21] OpenFlow switch HP procurve 5400 zl series. <http://www.openflow.org/wp/switch-hp/>.