Class–based Traffic Recovery with Load Balancing in Software–Defined Networks

Davide Adami, Stefano Giordano, Michele Pagano, Nicola Santinelli
Department of Information Engineering, University of Pisa
Address: Via G. Caruso 16, 56122 Pisa, Italy
Emails: (d.adami, s.giordano, m.pagano)@iet.unipi.it, n.santinelli@studenti.unipi.it

***Abstract*—In the last years, Software-Defined Networking (SDN) has emerged as a flexible paradigm driving innovation in future network architectures. Unfortunately, resilience, a key requirement in IP networks, is not an inherent feature of the SDN architecture. Indeed, the automatic reconfigurability of traffic paths in case of links or nodes failures is no more available. This paper deals with the design and validation of an SDN control application for class-based traffic recovery with load balancing. The features of the OpenFlow protocol have been exploited to timely detect link failures as well as to estimate links utilization. The performance of different strategies for class-based traffic recovery has been evaluated in an emulated environment, based on the widely used Mininet tool, highlighting the effectiveness of the proposed solution in enterprise networks.**

Keywords*—*SDN; OpenFlow; traffic recovery; protection; restoration; load balancing.

I. INTRODUCTION

Software-Defined Networking (SDN), a recent approach to programmable networks decoupling control and data planes, is a promising way to develop and deploy flexible and innovative network services with CapEx and OpEX reduction [1][2][3][4].

The most deployed SDN protocol, OpenFlow [5][6], enables the communication between the data plane and a remote *controller*, where both control plane and applications reside. Each OpenFlow-compliant switch has a *flow-table* containing a set of *rules* (*i.e.*, flow-table entries) suitable for flows handling, where a flow is intended as a sequence of packets sharing some header fields value. Each packet received by the switch is compared against the flow-table. If a matching entry is found, the related action (*e.g.*, forward out a specified port or drop) is performed and some counters (*e.g.*, packets/bytes Tx/Rx) are updated. Otherwise, the packet is sent to the controller, which establishes how to handle it and updates the rules in the switches flow-table.

Resilience is a well-satisfied requirement in IP/MPLS networks [7], but it is an open issue for SDNs. Indeed, the automatic reconfigurability of traffic paths in case of links or nodes failures is no more available, since switches flow-table entries are now defined and updated only by the controller. By default, upon failure occurrences the controller does not perform any action.

Our paper aims at overcoming this problem by introducing a new SDN control application, developed on top of POX [9]. Moreover, to take into account the different requirements of traffic flows in terms of recovery time, our controller supports *Restoration* as well as *Protection*.

Restoration requires the computation of the recovery (*i.e.*, back-up) path (RP) only after a failure has been detected within the working (*i.e.*, primary) path (WP) and signaled to the controller through the OpenFlow `PortStatus` message. On the contrary, *Protection* involves the computation and set-up of the OpenFlow rules for both WP and RP at the same time. In our architecture two different *Protection* strategies are available: “dedicated protection” and “on-the-fly protection”. In the first case both WP and RP are simultaneously activated and user traffic is replicated on them (1+1 protection), without any service outage in case of failures. Instead, in the second case the RP is activated by the controller through the removal of the forwarding rules for the WP only after a failure is detected.

Resilience in SDNs has been already addressed in the literature without taking into account the dynamics of network traffic. For instance, in [8], Sharma et al. proposed a fast restoration mechanism for OpenFlow networks, in which all the forwarding rules are statically defined by the network administrator. On the contrary, we consider variable costs of the links depending on the current utilization, estimated in real-time by the controller exploiting OpenFlow counters. Traffic paths are computed accordingly, leading to flows distribution over the entire network.

In order to evaluate the behaviour of our application, we tested it in the Mininet [12] emulation environment. It is worth highlighting that the developed software modules may be ported in a real network without changes thanks to the Mininet features.

The rest of the paper is organized as follows. Section II highlights the main features of our control application, describing its components and their interaction with the POX platform. Then, Section III reports the emulation results for both the load balancing and traffic recovery functionalities. Finally, Section IV concludes the paper with some final remarks.

II. SYSTEM ARCHITECTURE

The application proposed in this paper has been developed leveraging on POX [9], a platform for the rapid development and prototyping of network control software using Python. As shown in Figure 1, the aforementioned application consists of four main components, briefly described in this Section.

978-1-4799-7470-2/14/$31.00 ©2014 IEEE

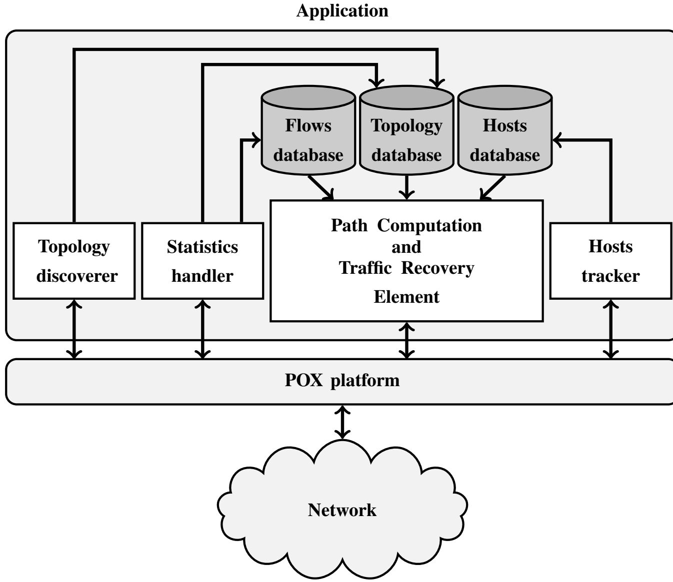


Figure 1. System architecture

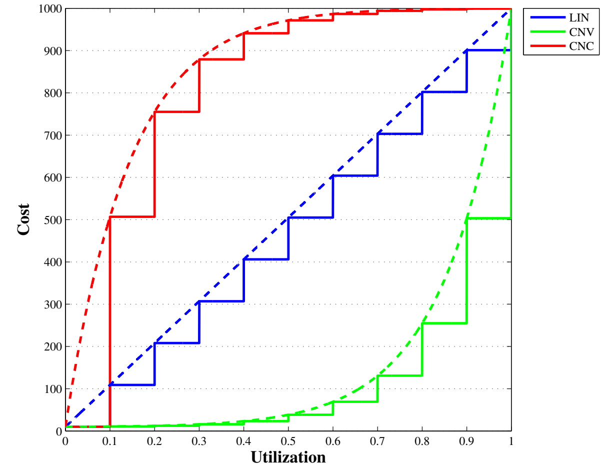


Figure 2. Metrics $(X = 3, Y = 1)$

A. Topology discoverer

Topology discoverer allows to infer the network topology. The underlying infrastructure is represented as a directed graph G, stored inside the *Topology database*. A switch in the network corresponds to a node in the graph, while a link between two switches is mapped into one or two arcs between the associated nodes, depending on whether the link is simplex or full-duplex, respectively. The considered component is based on the `openflow.discovery` module [10], provided within the Betta branch of the POX repository. To accomplish its task, `openflow.discovery` sends specially-crafted Link Layer Discovery Protocol (LLDP) messages out of each OpenFlow switch.

B. Hosts tracker

By means of ARP traffic analisys, *Hosts tracker* keeps updated information about hosts location within the network. The collected data are stored in a soft-state manner inside the *Hosts database*, containing mappings between each host (*i.e.*, IP and MAC addresses) and its connected switch (*i.e.*, datapath ID and port number).

C. Statistics handler

Statistics handler gathers OpenFlow counters (*i.e.*, statistics) related to physical ports belonging to each switch of the network. The processing of such an information allows to keep the *Topology database* constantly updated. In particular, it is possible to assign a cost to each arc of G according to the utilization level experienced by its corresponding link.

For the sake of simplicity, let us consider a unidirectional link with nominal bandwidth B bps between switches S_i and S_j. Switch S_j keeps a counter for the number of bytes received until time t, that is $N(t)$. *Statistics handler* retrieves the values of this counter by polling the considered switch every T_s seconds. The difference between the current value (*i.e.*, $N[k]$) and the previous one (*i.e.*, $N[k-1]$) provides the number of bytes received on port b during the k-th polling interval, from which both the average port load $\hat{R}[k]$ and the link utilization level $\hat{u}[k]$ can be easily derived.

In order to assign a cost $c_{ij}[k]$ to the arc (i, j), we have to define a wide-sense increasing function $f : [0, 1] \rightarrow [c_{\text{min}}, c_{\text{max}}]$, where $0 \leq c_{\text{min}} < c_{\text{max}}$, $c_{\text{max}} = 10^X$ and $c_{\text{min}} = 10^Y$. Among all the suitable metrics meeting such a requirement, we focused on three expressions, depicted as dashed curves in Figure 2:

$$f_{\text{LIN}}(u) = 10^Y + \left(10^X - 10^Y\right) u, \quad (1)$$

$$f_{\text{CNV}}(u) = 10^Y + 10^{uX} - 10^{uY}, \quad (2)$$

$$f_{\text{CNC}}(u) = 10^X - 10^{(1-u)X} + 10^{(1-u)Y}. \quad (3)$$

The *linear* (LIN) function (1) corresponds to a cost assignment policy proportional to the utilization. (2) is characterized by a *convex* (CNV) shape: significant costs are assigned only to the links experiencing high utilization levels. On the contrary, the *concave* (CNC) shape of (3) assigns high costs also for low utilizations. The user is free to choose the metric which best suites her purposes. In order to avoid route fluttering in case of small changes in the link utilization, we adopted a quantization step of 0.1, leading to the solid curves shown in Figure 2.

In case of full-duplex links, as usually occurs in real networks as well as in our emulation scenarios, the previous procedure must be applied for each direction.

In addition to the previous tasks, *Statistics handler* gathers OpenFlow counters related to the flows managed by each switch of the network. Therefore, the component is able to build and maintain the *Flows database*, containing mappings between flows description (*i.e.*, the match structure appearing in the flow-tables) and flows average rate, obtained through an estimation procedure similar to the one described for the traffic load.

D. Path Computation and Traffic Recovery Element

Path Computation and Traffic Recovery Element (PCTRE) computes the flow paths, installs the corresponding OpenFlow entries and performs class-based traffic recovery in case of failure. In our architecture three classes, characterized by an

increasing level of protection, are available: *Bronze*, *Silver* and *Gold*.

Our SDN control application assigns to *Bronze* flows a reactive traffic recovery strategy (*i.e.*, *Restoration*): the RP associated to such flows is computed, and eventually activated, only after the WP failure. On the contrary, *Silver* flows benefit from a proactive technique (*i.e.*, *Protection*), meaning that WP and RP are computed simultaneously. Nevertheless, the flow-table entries associated to the RP have a lower priority with respect to the ones of the WP. Therefore, for the former to be activated, the latter have to be removed. *Gold* flows benefit from a proactive traffic recovery strategy, as well. However, WP and RP have the same priority level. Thus, packets belonging to such flows are forwarded on two *link-disjoint* paths at the same time.

Knowing the graph representation for the network under test, the Dijkstra's algorithm execution leads to a set of updated shortest-path trees, each rooted at a different node of G (*i.e.*, a different switch of the network). Such an execution is triggered by any modification, concerning topology or costs, to the *Topology database*. A new flow is detected when the OpenFlow asynchronous message `PacketIn`, generated by the switch connected to the source, is received. Referring to the *Hosts database*, the component "translates" a couple of hosts (*i.e.*, flow source and destination) in a couple of switches (*i.e.*, path source and destination). Accordingly, it retrieves the relevant WP among the available ones and, in case of *Bronze* flows, installs the corresponding rules inside the switches flow-table. As previously mentioned, *Silver* and *Gold* flows require the computation of the RP, being link-disjoint from the WP. For this purpose, PCTRE creates a custom topology pruning the arcs of the WP from the real network. If an additional execution of the Dijkstra's algorithm allows to find the new path, both the WP and RP are installed through the appropriate flow-table entries with the correct priority values; otherwise an error message is sent to the network administrator.

The traffic recovery procedure is triggered by the reception of the OpenFlow asynchronous message `PortStatus`. From the *Flows database*, PCTRE identifies all the flows forwarded on the broken link and the class they belong to.

No action is required for *Gold* flows since traffic is already replicated on the RP. Flows, belonging to the *Silver* and *Bronze* classes, are sorted according to their bandwidth (extracted from the *Flows Database*) in decreasing order to minimize losses during the recovery phase. For *Silver* flows the OpenFlow rules associated to the WPs must be removed. Instead, *Bronze* flows require a longer recovery procedure. At first, PCTRE creates a new topology, pruning the broken link, and then runs the Dijkstra's algorithm, searching for the RP. If such a path is found, PCTRE installs the needed rules inside the switches flow-table, otherwise, the flow is not recovered.

III. Emulation Results

This Section describes the tests carried out to evaluate the effectiveness of our control application, tested within the Mininet emulation environment [11][12]. Network traffic has been generated thanks to the Iperf tool [13]. As far as the *Statistics handler* configuration, we considered a sampling time $T_s = 5$ seconds and the following values for the cost parameters: $X = 3$ ($c_{\max} = 1000$) and $Y = 1$ ($c_{\min} = 10$). Without loss of generality, we suppose that a flow is uniquely identified by the 5-tuple

$$< src_ip, dst_ip, proto_ip, src_port, dst_port > .$$

However, scalability can be achieved through aggregation, for instance considering the traffic flows generated by subnetworks.

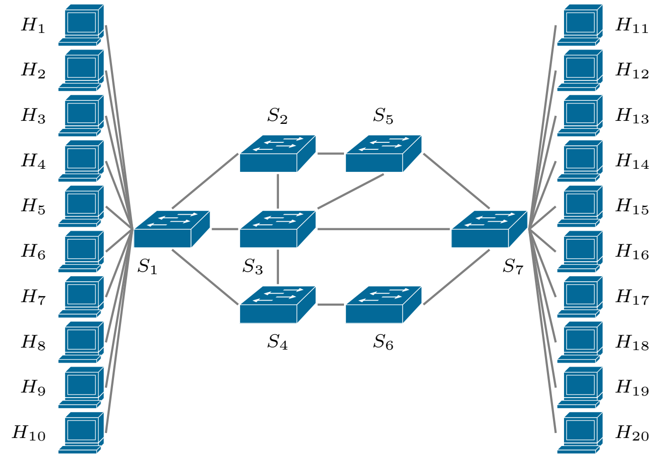


Figure 3. Emulated topology, load balancing tests

A. Load balancing

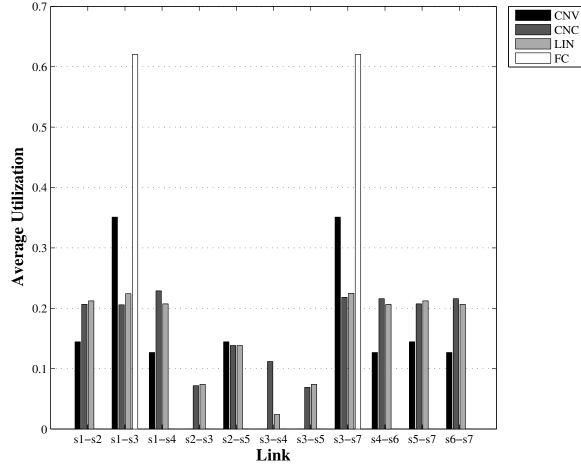
The following tests compare our load balancing strategies (LIN, CNV and CNC) with a baseline Fixed Cost (FC) assignment policy. For the last one, we chose unitary link costs and disabled the *Statistics handler* module (the part related to port counters, at least).

The reference topology is shown in Figure 3, where all the links are full-duplex with bandwidth $B = 10$ Mbps. Each host acts both as Iperf client and server. A client (traffic source) connected to switch S_1 (S_7) randomly chooses its server among the hosts connected to S_7 (S_1). Network traffic is generated according to the well-known ON/OFF source model, proposed in [14]. More precisely, train lengths follow a Pareto distribution with parameters $\alpha = 1.5$ (*i.e.*, shape) and $k = 20$ seconds (*i.e.*, location), corresponding to a mean value $T_{\text{ON}} = 60$ seconds. Intertrain distances follow a Pareto distribution as well, with $\alpha = 1.5$, $k = 10$ seconds and, thus, a mean value $T_{\text{OFF}} = 30$ seconds.

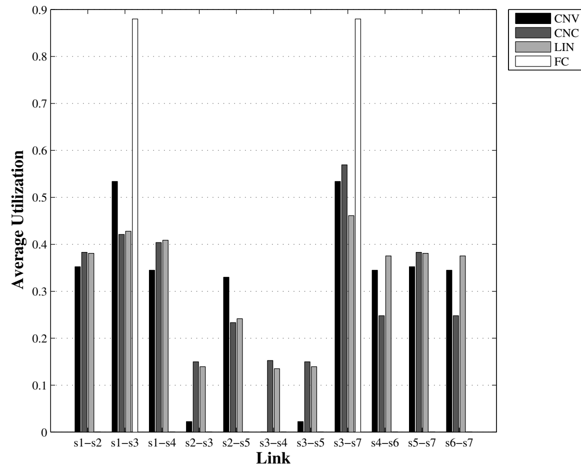
In the following subsections, we compare the average link utilization, loss rate and throughput for the different path computation strategies (the baseline FC and our policies LIN, CNV and CNC, introduced in Section II-D) taking separately into account UDP and TCP flows.

1) UDP flows: In the emulation, the rate R of each ON/OFF source is uniformly distributed in the range $[r_{\text{inf}}, r_{\text{sup}}]$. In Figure 4a and Figure 4b we present the performance comparison for low and high network utilization respectively. For the sake of brevity, only one direction is reported, since similar results are obtained for the opposite one.

In more detail, Figure 4a, corresponding to $R \in \mathcal{U}[0.064, 2]$ Mbps, shows that the FC strategy leads to forward

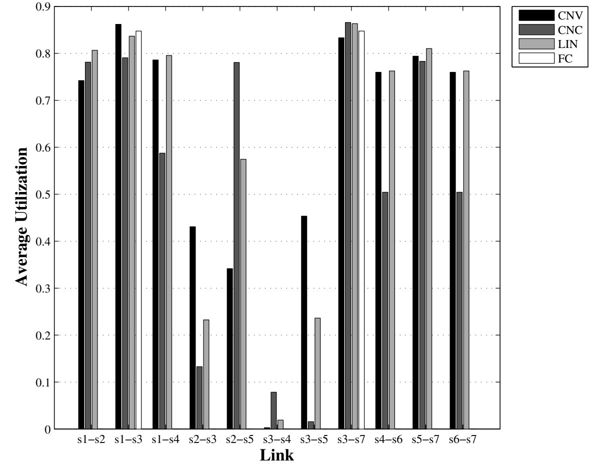


(a) $r_{\text{inf}} = 0.064$ Mbps, $r_{\text{sup}} = 2$ Mbps, $S_i \rightarrow S_j$ direction

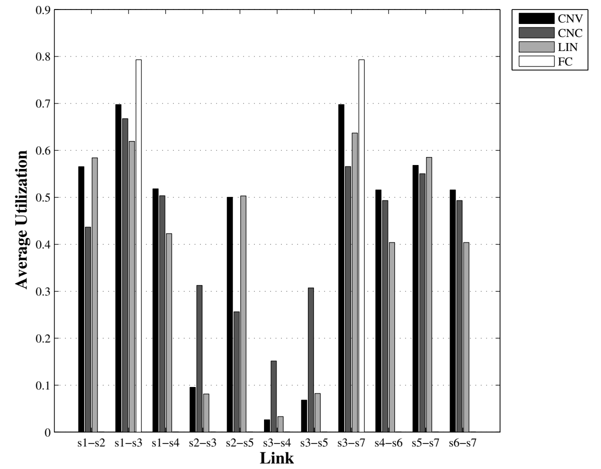


(b) $r_{\text{inf}} = 0.8$ Mbps, $r_{\text{sup}} = 3.2$ Mbps, $S_i \rightarrow S_j$ direction

Figure 4. Average link utilizations: UDP flows



(a) $S_i \rightarrow S_j$ direction



(b) $S_i \leftarrow S_j$ direction

Figure 5. Average link utilizations: TCP flows

Table I. AVERAGE LOSS PERCENTAGES: UDP FLOWS

Transmission Rate (Mbps)	**Average Loss** (%)			
	CNV	**CNC**	**LIN**	**FC**
$R \in \mathcal{U}\,[0.064, 2]$	0.0894	0.0776	0.0830	0.2838
$R \in \mathcal{U}\,[0.8, 3.2]$	0.4912	0.7146	0.4124	28.5412

Table II. AVERAGE TCP THROUGHPUT

Average Throughput (Mbps)			
CNV	**CNC**	**LIN**	**FC**
2.3505	2.1778	2.0559	0.9016

the traffic only over 2 out of 11 links. On the contrary the load balancing path computation criteria try to reach a fair utilization among network resources, with the CNV policy using the lowest number of links.

Figure 4b shows the behavior for higher utilization, namely $R \in \mathcal{U}\,[0.8, 3.2]$ Mbps. While the previous considerations still apply to this case, it is worth noticing that CNV uses more links with respect to the previous scenario (*i.e.*, $S_2 - S_3$ and $S_3 - S_5$).

Table I reports the average loss percentages for the different policies. In case of low utilization the performance are quite similar with limited losses also in case of FC. On the contrary, for high utilization our variable cost strategies outperform FC, with slightly higher losses for CNC.

2) TCP flows: In this scenario we considered greedy sources, whose rate is tuned by TCP according to the congestion of the network. Figure 5 shows the average utilization for all the bidirectional links of the emulated network. Since FC always uses the shortest path between S_1 and S_7, the average throughput is strongly limited, while load balancing policies lead to an higher utilization of all the links and hence to an higher throughput (see Table II).

Once again, an FC assignment policy brings to poor load sharing among the infrastructure (2 links over the 11 available) with respect to a load balancing criterion.

B. Traffic Recovery tests

These tests aim at evaluating the performance of the traffic recovery strategies implemented within our control application. In more detail, we take into account two performance metrics:

Table III. OVERALL MTTD AND MTTR

	MTTD (ms)	**MTTR** (ms)
Protection (*Silver*)	23.7827 ± 0.5848	27.3668 ± 0.6403
Restoration	23.2668 ± 0.5895	36.8622 ± 0.7230

- *Time To Detect* (TTD) – the time interval between the failure occurrence and its detection by the controller, which includes the time needed by the switch to detect the port status change and the notification delay to the controller;
- *Time To Repair* (TTR) – the time interval between the WP failure and the RP activation, which is strictly related to the recovery strategy.

For their calculation, we exploited the `time` library [15] provided by Python. In particular, the method `time.time()` returns the time in seconds since "epoch" (*i.e.*, the beginning of times) as a floating point number. During each experiment, the following timestamps are taken into account:

- `tFail`, just before the link is torn down;
- `tDete`, just after the controller has received the `PortStatus` messages;
- `tProt`, just after the flow-table entries related to the WP have been removed (in case of *Silver* flow);
- `tRest`, just after an RP is computed and installed (in case of *Bronze* flow).

With the previous definition, we have:

$$\text{TTD} = \texttt{tDete} - \texttt{tFail}, \tag{4}$$

$$\text{TTR} = \begin{cases} \texttt{tProt} - \texttt{tFail} & \textit{Silver} \text{ flow}, \\ \texttt{tRest} - \texttt{tFail} & \textit{Bronze} \text{ flow}. \end{cases} \tag{5}$$

The emulated network topology is the same as in previous tests (see Figure 3), but in this case we considered only three CBR traffic flows (at 1 Mbps): $H_1 \rightarrow H_{11}$, $H_2 \rightarrow H_{12}$ and $H_3 \rightarrow H_{13}$. Due to variable cost assignment, these flows follow the paths $S_1 \rightarrow S_3 \rightarrow S_7$, $S_1 \rightarrow S_2 \rightarrow S_5 \rightarrow S_7$ and $S_1 \rightarrow S_4 \rightarrow S_6 \rightarrow S_7$, respectively.

To test the recovery performance for *Silver* and *Bronze* traffic classes (indeed, the $1 + 1$ protection used for *Gold* class avoids any traffic disruption), every link belonging to the aforementioned paths is torn down in separate simulation sets, each consisting of 100 runs. Table III shows the average values (over all links) of the Mean Time To Detect (MTTD) and the Mean Time To Repair (MTTR), together with their 90% confidence interval: the MTTD does not depend on the recovery strategy, while the MTTR assumes higher values for *Bronze* flows due to the time necessary for calculating the recovery path and installing the rules along the switches of the new path.

We highlight that the proposed application allows a flow to be recovered (on average) in less than 30 or 40 milliseconds, depending if the class it belongs to is whether *Silver* or *Bronze*, respectively. Therefore, the SONET/SDH requirement of 50 milliseconds is (on average) satisfied.

IV. CONCLUSION

This paper proposed a new SDN control application for class-based traffic recovery with load balancing, implemented and tested in the Mininet emulation environment. In more detail load balancing has been achieved by considering variable cost assignments policies, based on OpenFlow network statistics, in the routing algorithm. Moreover, specific recovery techniques have been implemented for different traffic classes, ranging from Restoration in case of *Bronze* traffic to $1 + 1$ protection for *Gold*.

A wide set of tests has been carried out to evaluate the performance of our control application. For the different cost functions considered in this work, load balancing leads to lower losses and higher bandwidths with respect to a fixed cost assignment policy. As far as class-based traffic recovery is concerned, recovery occurs (on average) in less than 30 (*Silver* class) or 40 milliseconds (*Bronze* class), meeting the SONET/SDH requirement of 50 milliseconds.

ACKNOWLEDGEMENTS

This work was partially supported by the Italian Ministry of University and Research funded FIRB project GreenNet.

REFERENCES

[1] ONF, "Software-Defined Networking: The New Norm for Networks", ONF White Paper, 2012.

[2] OPEN NETWORKING FOUNDATION, homepage, https://www.opennetworking.org/.

[3] A. LARA, A. KOLASANI, B. RAMAMURTHY, "Network Innovation using OpenFlow: A Survey", *IEEE Communications Surveys and Tutorials*, vol. 16, no. 1, pp. 493-512, 2013.

[4] SDNCENTRAL, What's Software-Defined Networking (SDN), http://www.sdncentral.com/what-the-definition-of-software-defined-networking-sdn/.

[5] N. MCKEOWN, T. ANDERSON, H. BALAKRISHNNAN, G. PARULKAR, L. PETERSON, J. REXFORD, S. SHENKER, J. TURNER, "OpenFlow: Enabling Innovation in Campus Networks", *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69-74, 2008.

[6] ONF, "OpenFlow Switch Specification", Version 1.0.0 (Wire Protocol 0x01), 2009.

[7] J. P. VASSEUR, M. PICKAVET, P. DEMEESTER, "Network Recovery: protection and restoration of optical, SONET-SDH, IP and MPLS", *Morgan Kaufmann*, 2004.

[8] S. SHARMA, D. STAESSENS, D. COLLE, M. PICKAVET, P. DEMEESTER, "Enabling Fast Failure Recovery in OpenFlow Networks", *8th International Workshop on the Design of Reliable Communication Networks (DRCN)*, pp. 164-171, 2011.

[9] POX, homepage, http://www.noxrepo.org/pox/about-pox/.

[10] POX, `openflow.discovery`, https://github.com/noxrepo/pox/blob/betta/pox/openflow/discovery.py.

[11] MININET, homepage, http://mininet.org/.

[12] B. LANTZ, B. HELLER, N. MCKEOWN, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks", *Hotnets-IX Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010.

[13] IPERF, homepage, http://iperf.fr/.

[14] W. WILLINGER, M. S. TAQQU, R. SHERMAN, D. V. WILSON, "Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level", *SIGCOMM '95 Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 100-113, 1995.

[15] PYTHON 2.7.8 DOCUMENTATION, `time` library, https://docs.python.org/2/library/time.html.