TRANSACTIONS ON EMERGING TELECOMMUNICATIONS TECHNOLOGIES
Trans. Emerging Tel. Tech. 201X; **00**:1–14
DOI: 10.1002/ett

RESEARCH ARTICLE

New cooperative mechanisms for Software Defined Networks based on Hybrid Switches

Joaquin Alvarez-Horcajo[1], Isaias Martinez-Yelmo[1]*, Elisa Rojas[2], Juan A. Carral-Pelayo[1], Diego Lopez-Pajares[1]

[1]Dept. Automática, Universidad de Alcalá, 28801 Alcalá de Henares (Madrid) Spain
[2]Telcaria Ideas S.L., 28911 Leganés (Madrid) Spain

ABSTRACT

Software Defined Networking (SDN) aims to provide simplified network design, operation and management using a decoupled control plane. However, its centralized control and global network knowledge present scalability and reliability issues, which makes SDN deployment very challenging. In this paper, we propose and evaluate a hybrid switch with partial delegation of basic bridging and new cooperative mechanisms between controller and switches. This delegation offloads the SDN controllers while maintaining the capability to install forwarding rules on the switches. In this way, we take full advantage of hybrid switches in addition to using them as backwards compatible equipment which interoperates with traditional switches. We validate this proposal by implementing a hybrid OpenFlow switch on an open source software switch as a proof of concept. Scalability and path setup delay are improved with respect to traditional centralised SDN solutions, due to the reduction in controller load, and in turn due to the reduced traffic between switches and controller. Our cooperative mechanisms focus on recovering failures, obtaining the best performance of all approaches on higher loads, and providing a good trade-off between controller based and traditional distributed approaches.

*Correspondence
Dept. Automática, Universidad de Alcalá, 28801 Alcalá de Henares (Madrid) Spain
E-mail: isaias.martinezy@uah.es

1. INTRODUCTION

The Software-Defined Networking (SDN) architecture supports flow-level control with fine granularity and offers many advantages, such as avoiding switch by switch configuration, allowing fast deployment of new protocols without the need to design or deploy new switching equipment in the production network, as well as coexistence among production and experimental networks, and network programmability. Furthermore, its usage has been extended to high performance networks such as data center networks (DevoFlow[1], Hedera[2] and PAST[3]). However, proposals based strictly on SDN impose significant overheads [1]. The need to have complete control of all flows creates bottlenecks at the controller. For instance, these bottlenecks can have an impact on flow setup times [3]. As a solution, Kandoo [4] proposes to distribute the load between multiple controllers to solve the bottlenecks at the controller side. Other proposals distribute part of the network state among the switches as in DevoFlow [1], OpenState [5], or Difane [6]. Another limitation is that the number of flows per switch can be very high: hundreds of thousands of entries at switch tables may be needed for a large network [7]. These table sizes are impossible to implement in current TCAMs, which are needed for wildcard prefix matching. Thus, the use of SRAMs, which do not have the restrictions of TCAMS, has been proposed to improve network scalability as in PAST [3]. Alternatively, Tiny Packet Programs [8] or SmartPacket [9] incorporate state information in the packets. Thus, ongoing research is going into reducing the tasks performed on the SDN Control plane by moving these tasks onto the data plane [10, 11].

In order to improve the SDN architecture, we propose a cooperative scheme in which the SDN controller(s) and the switches work together, sharing the flow processing and management load in traffic forwarding operations and also during path recovery operations after network failures. With this scheme, not every flow in the network (such as broadcast, ARP, DHCP, etc.) requires exclusive controller intervention, and forwarding performance is improved by directly switching packets on the network devices without controller operation whenever possible. Path recovery is improved by the cooperation of hybrid

switches with the SDN controller when setting up new paths after network failures. Our design has similarities with other approaches like DevoFlow as it keeps flow handling in the data plane whenever possible, which prevents the overheads previously mentioned. Thus, the traffic that must be managed by a controller is greatly reduced and bottlenecks for the installation of all flows on OpenFlow tables are avoided. These improvements are obtained by taking advantage of hybrid switches, which can perform certain tasks autonomously using traditional distributed switching protocols on the data plane.

The main contributions of this paper are: first, the proposal of an SDN backwards-compatible switch architecture enabling autonomous forwarding on hybrid switches; second, the definition of a cooperative flow processing mode; and third, a software switch implementation of the proposed design along with an evaluation of the proposed solution that validates the expected improvements in performance and scalability.

The rest of the paper is structured as follows. Section 2 reviews the related work on SDN and switching. Section 3 explains the new proposed switch architecture. Section 4 provides a detailed explanation of the software switch implementation. Section 5 presents the experiments conducted to evaluate the proposal and the results obtained. Section 6 discusses the results and backwards compatibility issues related to the bridging delegation. Finally, Section 7 summarizes the conclusions of this work.

2. RELATED WORK

Despite the success of OpenFlow as the most popular southbound interface protocol for SDN, basic aspects of SDN architectures, like distribution of control in the network, are still subjects of intense discussion. We analyse the related work, grouping the proposals into three categories: first, the approaches dealing with the scalability of SDN; second, those that deal with resilience; and finally, those proposing partially delegated control.

Regarding **scalability of SDN**, the decoupling between control plane and data plane in SDN constitutes a possible bottleneck for scalability [12]. Moreover, the additional computational and network resource consumption may even lead to unrecoverable network outages [13] and also get a performance penalty [1, 14] as previously mentioned in the introduction. These performance penalties have been traditionally compared to more traditional bridging/switching solutions. For example, the work in [15] compares SPB 8021.aq and TRILL with OpenFlow NOX Routing for shortest path bridging and shows the inability of OpenFlow to perform local and sub-second reaction unless some kind of path protection is pre-installed. The work in [16] shows the high control message overhead of OpenFlow networks and its complexity compared with traditional distributed protocols such as OSPF and STP. Additionally, the work in [17] emphasises the complexity needed to assure the knowledge at controllers of the current active topology and the recovery mechanism after link failure, which requires path recomputation and usually long convergence times with respect to more traditional approaches.

Resilience of SDN to failures and convergence time after a failure have always been a key concern in network performance [18]. The work in [19] based on SDN/OpenFlow shows that leveraging the Link Layer Discovery Protocol (LLDP) [20] to monitor links and detect faults does not scale since it overloads the controller. Thus, it proposes a local mechanism on the switch side. Work on [21] analyses reactive path restoration and proactive path protection and shows that recovery time is unacceptably long in SDN. It proposes to improve the recovery time based on link redundancy using the Bidirectional Forwarding Detection (BFD) protocol [22] from IETF.

Regarding **delegation of control**, several proposals in the literature deal with bottlenecks in controllers by decentralizing the SDN control plane and a hardware and software upgrade both at the switches and at the controller as in [12]. [23] examines the concept of global and local knowledge to determine whether an SDN application should reside in the control or in the data plane, [24] proposes an API for programming the generation of packets in SDN switches (it presents the example of an ARP handler in the data plane). Other approaches propose to exploit parallel execution on the controllers [25], the synchronization of network state among multiple controller instances [26], or a distributed [27] or hierarchical [4] platform of SDN controllers. DevoFlow [1] reduces the control plane communication by handling most microflows (a large number) directly in the data plane, and only invoking the control plane (controller) for the setup of larger flows (a small number). OpenState [5] affirms that some level of control logic in the switches might be beneficial to offload logically centralized controllers. However, the traffic between switches and controllers still remains high in all of them.

Finally, there are other proposals based on more disruptive approaches. Difane [6] distributes the forwarding rules among different switches and forwards packets to the appropriate authorised switch to limit the state required at every switch. NEOD [28] embeds part of the management plane directly in the network devices, so that unusual network activities are monitored and captured earlier and traffic decreases, thus improving the reliability and scalability of the controllers during disaster events. Current research efforts are focused on moving the load on the SDN Control plane to the data plane [10, 11] in order to delegate most of the actions to the infrastructure equipment and leverage controllers.

Our proposal is also giving back responsibility to the data plane. However, it goes beyond pure management functions to include delegation of basic bridging functions

Trans. Emerging Tel. Tech. 201X; **00**:1–14 © 201X John Wiley & Sons, Ltd.
DOI: 10.1002/ett
Prepared using ettauth.cls

in order to increase performance and enhance scalability of the overall network. We achieve this objective by using hybrid OpenFlow and legacy switches. This idea is not novel since it has been exploited before by network manufacturers to allow OpenFlow backwards compatible network equipment such as Open-Flow-Hybrid [29] or Brocade hybrid switch and its different operation modes [30, 31]. These approaches allow the coexistence of distributed forwarding (legacy protocols) with controller based forwarding by relying on out-of-band traffic management. Additionally, our proposal seamlessly combines both mechanisms, alternatively enabling one of them at the flow level (OpenFlow or classical distributed forwarding) and also defines a cooperative mode to show how hybrid switches may take advantage of the knowledge from controllers and how both hybrid switches and controller can work together in a cooperative way, one of the main contributions of this paper, to improve the resilience and scalability of networks.

3. HYBRID SWITCH DESIGN

This section explains how the proposed hybrid switch works internally. First it briefly describes how legacy OpenFlow switches work and later it presents how our proposed hybrid switch works, which has certain similarities with the Brocade approach [30, 31]. We detail how the flows can be treated differently according to the proposed hybrid switch approach. Finally, we examine how a new cooperative processing mode is possible, which supports cooperation between hybrid switches and controllers.

3.1. SDN Reference Architecture

Figure 1 illustrates the SDN architecture specified by the Open Networking Foundation (ONF). This architecture considers the following layers according to [32, 33, 34]: (i) The *Application Layer* contains the specific business applications to be deployed on the network. It may include only an abstracted view of the network, as well as the services provided by the SDN platform and it communicates with the network infrastructure through the Controller Layer using the SDN NorthBound Interface (NBI), which is not standardised. (ii) The *Control Layer* is the central and orchestrating point of a capable SDN network. Its tasks are performed by an SDN controller which is usually capable of using the requirements of applications to setup forwarding rules on the network elements. Controllers are critical components, so their failure and overload must be avoided [35]. (iii) The *Infrastructure Layer* is the composition of all network nodes. Its main function is forwarding packets and it communicates with the Control Layer via the SouthBound Interface (SBI). The most common SBI protocol is the standardised OpenFlow, but other protocols can be also

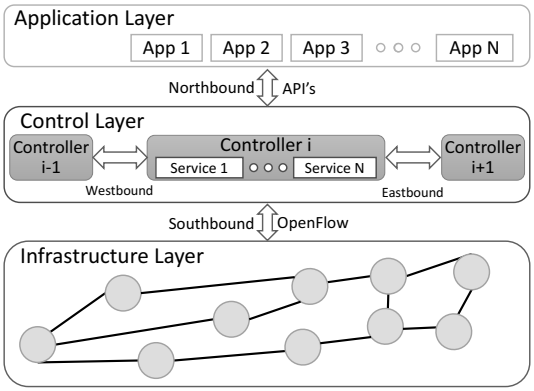


Figure 1. SDN Architecture

used, such as NETCONF [36]. In the rest of the paper we will suppose the use of OpenFlow for simplicity.

3.2. Autonomous bridging

The SDN architecture promotes full separation of the control and data planes. Packet forwarding is performed by the switch according to specific rules computed by the controller and installed at the switches via the SBI protocol, i.e. OpenFlow. Instead, we propose to keep the autonomous bridging capacities on hybrid switches as the default forwarding mechanism. That is, unless specific rules were previously installed by the controller, the forwarding of a packet with no matching OpenFlow rules is performed by the hybrid switch on its own. With this proposal, we try to capitalize on the best of both worlds, using the simple yet efficient forwarding mechanisms of classical bridges while allowing the desired degree of flow control from the controller. The controller may install the necessary rules to trap the desired and important flows to be handled apart from the rest (e.g. *elephant* flows in a data center, mission critical flows or even all of them if the network administrator so desires), but if no rule applies to a given packet the switch is left to its own to forward it according to a classical distributed bridging protocol.

Figure 2 presents a flowchart in which our proposal is integrated into an OpenFlow 1.3 switch. The upper part shows the behaviour of a pure OpenFlow switch, but when no rule matches the received packet it is sent to the autonomous switch (the lower part of the figure) to be forwarded instead of being dropped. The autonomous switch performs two main tasks, the Path Recovery mechanism, which is triggered due to network failures and the Distributed Bridging Engine implementing the desired autonomous learning and forwarding mechanisms. In this way, we greatly reduce the number of flows to be handled by the controller, the control plane messages overhead (see section 6.1) and the amount of state (rules) installed at the switches. Furthermore, the controller may regain full control of forwarding at any time by simply installing a

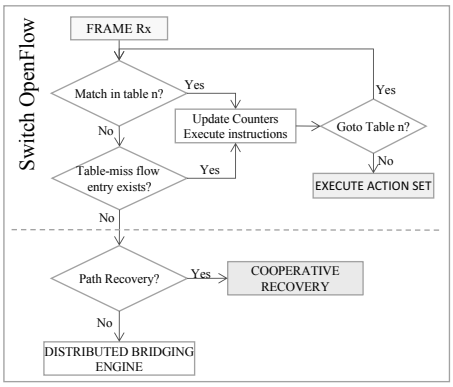


Figure 2. Frame processing on semi-autonomous OpenFlow switch

rule to forward any unmatched packet since controller rules have precedence over the local autonomous switching.

The Distributed Bridging Engine should provide an efficient use of the underlying network to ease the controller operation (computing resources for specific flows). We propose to use ARP-Path [37], a specific mechanism for layer 2 forwarding that provides shortest paths and loop prevention, but other protocols could also be used. We select ARP-Path since it allows to maintain inexpensive and simple hybrid OpenFlow switches as the SDN community proposes. Costs would be increased if more complex and computing demanding solutions (i.e. SPB or TRILL) were used as Distributed Bridging Engine. Furthermore, the design using more complex autonomous switches and a SDN controller will make it difficult to setup, which is undesirable.

3.3. Flow processing modes

The processing of the flows depends on the rules installed by the controllers. We distinguish three flow processing modes according to the degree of control delegated and the level of cooperation involved among hybrid switches and controller(s):

- *Autonomous Processing:* This is the preferred mode for improving the scalability of controllers. In this mode, the hybrid switch is in charge of establishing the path for the flows itself using the Distributed Bridging Engine if no OpenFlow rule matches. That is, only special flows, previously identified by the controller by installing a specific rule, are handled by the controller.

- *Controller Processing:* In this mode, the path selection is defined by the controller via OpenFlow rules. Flows must match these rules in order to be processed according to their specification. Controllers install a rule to forward non-matching packets to themselves. This mode leads to a regular SDN network according to the specification of OpenFlow 1.3.

- *Cooperative Processing:* Assuming that controllers have knowledge of the operation of the Distributed Bridging Engine local to switches, they could help switches with some specific tasks due to their global knowledge of the underlying network, for instance to compute restoration paths or to help in recovery mechanisms. In fact, any distributed process may benefit from this approach.

The cooperative flow processing mode is the most novel proposal of this section. In order to properly explain this mode, we provide a detailed example in section 4 to illustrate how we can take advantage of this mode and get benefits from its use.

4. HYBRID SWITCH IMPLEMENTATION AND PATH RECOVERY COOPERATIVE PROCESSING

We decided to use an OpenFlow software switch for quick prototyping of our proposal. Nowadays, there are different software switches with OpenFlow support: the reference OpenFlow switch 1.0 implementation [38], Open vSwitch [39], ofsoftwitch13 from CPqD [40, 41], Indigo virtual switch [42], DPDK [43] or Linc [44]. All of them implement the OpenFlow control interface and delegate all the control to the controller, without any local switching. We have selected ofsoftwitch13 to implement our proposal since we consider it the most suitable choice for easy prototyping, as it is based in Linux user space. After explaining how our hybrid switch is implemented using ofsoftwitch13, we define three different recovery mechanisms which can be used with our proposed hybrid switch architecture.

4.1. Reference OpenFlow Software Switch 1.3

The ofsoftswitch13 switch from CPqD [40, 41] is implemented on a Linux-based system running in user-space and provides OpenFlow 1.3 [45] support. It is designed to support rapid and inexpensive experimentation on OpenFlow-based networks and easy prototyping of new features such as this work. It supports the necessary OpenFlow functions and data structures: *Meter Tables* for QoS support, *Group Tables* for enhanced forwarding mechanisms and *Flow Tables* to properly process the matched flows [38]. It also supports both physical and/or virtual ports. Finally, it translates OpenFlow data packets to an internal format *Oflib* and it uses the *NetBee* Link library to decode the incoming packets from the network interfaces.

4.2. Distributed Bridging Engine: ARP-Path

Based on our experience of layer 2 routing path exploration protocols, we have chosen ARP-Path [37] from The All-Path family as the Distributed Bridging Engine. ARP-Path

Trans. Emerging Tel. Tech. 201X; **00**:1–14 © 201X John Wiley & Sons, Ltd.
DOI: 10.1002/ett
Prepared using ettauth.cls

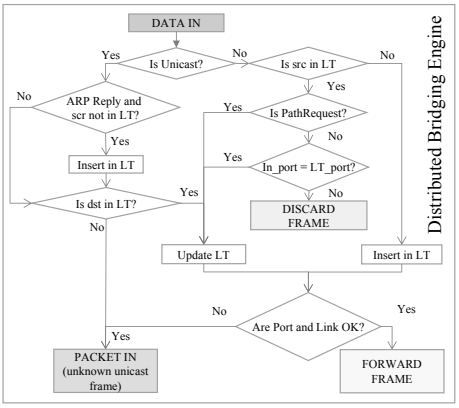


Figure 3. Distributed Bridging Engine mechanism.

is a lightweight protocol with a built-in loop prevention mechanism and native path diversity. It uses a backward learning mechanism inherited from transparent bridges. The four basic differences between ARP-Path bridges and standard transparent bridges are as follows. First, ARP-Path bridges flood frames over all network links instead of over a spanning tree topology (ARP-Path bridges do not need to block links to prevent loops). Second, the address learning mechanism at ports is modified so that once a MAC source address is learnt, this address cannot be associated with a different port for a very short *lock* time. Third, source address locking and learning occur only with broadcast frames (like the standard ARP Request packet), not with unicast frames (with the exception of ARP Reply packets). Fourth, frames with an unknown unicast destination address trigger the ARP-Path path recovery mechanism instead of the usual flooding of unknown frames of standard transparent bridges. Figure 3 shows a flowchart illustrating the basic path establishment mechanism. A detailed explanation of how paths are established can be found on [37] and [46].

4.3. Software Switch extension

We have extended the ofsoftswitch13 reference implementation to incorporate the ARP-Path protocol behaviour. More specifically, we have included three new data structures:

- *Hello packets:* ARP-Path capable switches periodically send *Hello* packets each three seconds to identify neighbour ARP-Path switches (when a *Hello* is received). Also, if a *Hello* packet is not received on a given port it is marked as unused or used by regular hosts.
- *Neighbour Table (NT):* Neighbour ARP-Path switches discovered by the *Hello* mechanism are stored in this table including the incoming port.
- *Learning and Blocking Table (LT):* This table stores the MAC address to output port association with the corresponding cache timer (after expiration an entry can be safely removed), which is set up to ten seconds. It is used by the switch to implement the basic learning and frame forwarding mechanisms according to the ARP-Path protocol rules.

These three data structures are used by the extended pipeline processing logic, which implements ARP-Path according to Figure 3, if and only if a received packet does not match any OpenFlow rule (Figure 2).

4.4. Path Recovery Mechanisms

We select the recovery mechanisms to illustrate the benefits and feasibility of cooperative mechanisms with controller and hybrid switches since they are an important mechanism, related to the basic functionality of packet forwarding and the operation performance on data networks. There are several proposals in relation to the selected Distributed Bridging Engine based on ARP-Path for path recovery in case of link or switch failure [47, 46, 37]. We propose to study three different recovery mechanisms by using our hybrid switches called: Distributed Path Recovery, Controller Based Path Recovery and Controller Assisted (Cooperative) Path Recovery. The last proposal shows the full advantages obtained by controller and hybrids switches cooperating together.

4.4.1. Distributed Path Recovery

The Distributed Path Recovery is performed by the hybrid switches alone. This mechanism is illustrated in Figure 4 where hosts *s* and *d* communicate via *s1-s3-s5* (shortest path) until the link *s1-s3* goes down. When a failure occurs (see Figure 4.a), a *Path Recovery Request* packet is flooded from the switch (s1) that does not have a matching entry at learning table due to the network failure. This operation is shown on Figure 4.b and its purpose is to reach the destination switch in order to trigger the path repair from the destination. Once the destination switch is reached, a *Path Recovery Reply* packet is flooded from the destination edge bridge and it reaches every hybrid switch in the network creating a kind of sink tree rooted at destination edge switch (and eventually, arriving to *s1*). Therefore, every hybrid switch in the network receiving the packet gets a viable path to reach the destination host (see Figure 4.d). The destination MAC address is only learned from the first port getting the *Path Recovery Reply* packet. Later flooded packets are discarded according to the same locking mechanism used in ARP-Path.

4.4.2. Controller Based Path Recovery

The Controller Based Path Recovery uses the standard approach in SDN networks to deal with network failures. When a network failure occurs the controller computes an alternative path and sets up the necessary rules on switches to forward the packets according to the new computed path. This mechanism is illustrated in Figure 5 where hosts

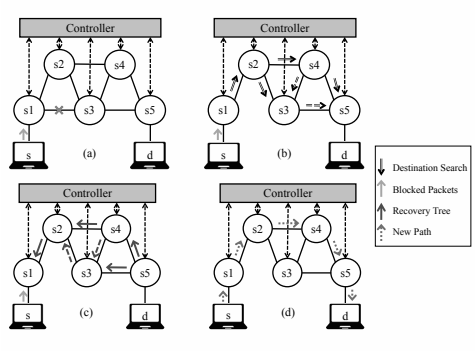


Figure 4. Distributed Path Recovery.

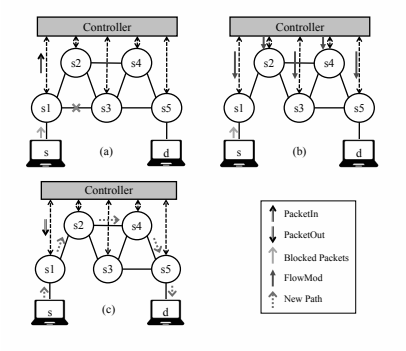


Figure 5. Controller Based Path Recovery.

s and *d* communicate via *s1-s3-s5* (shortest path) until the link *s1-s3* goes down.

Path Recovery controller-side functions: We have implemented an SDN application using the Ryu SDN framework [48]. The Controller Based Path Recovery supports the following functions:

- *Host Table (HT) maintenance:* HT contains the hosts directly connected to every switch. The controller updates this table when it receives a packet from an edge switch containing the MAC address and port of a host connected to it.
- *Path Recovery Computation:* If the controller receives a *PacketIn* from a switch with a *Path Recovery Request* for a given destination (see Figure 5.a), it gets the source and destination MAC addresses from the received Request and computes an alternative path according to the new topology obtained from LLDP. We have implemented an Equal Cost Multipath (ECMP) algorithm based on shortest paths (Dijkstra) to obtain the desired new path. Once the new path is computed, the controller installs the necessary OpenFlow rules on hybrid switches to setup the desired path (see Figure 5.b). Additionally, a PacketOut is used to send the original unmatched packet due to the network failure.

Path Recovery switch-side functions: In addition to the controller functions, switches also perform some new tasks to complete the Path Recovery.

- *Attached Host notification:* The hybrid switch informs the controller of the MAC address and port where a host is attached so that the controller keeps an updated *HT*.
- *Path Recovery Request:* If a unicast frame is received by a switch with no matching entry in its LT Table, it forwards the packet to the controller (PacketIn) (see Figure 5.a) to send a *Path Recovery Request*.

Finally, frames are sent using the newly acquired path from the controller computation (Figure 5.c).

Although the path restoration resides exclusively in the controller, there is some cooperation between hybrid switches and controller(s). Hybrid switches proactively notify the controller(s) of their attached hosts, since controllers need this information to recompute the paths on a *Path Recovery Request*. This approach also shows the coexistence of local and OpenFlow rules in hybrid switches and how the SDN architecture can be used, if desired, to handle only selected flows (e.g. flows affected by failures) instead of dealing with all existing network flows.

4.4.3. Controller Assisted Path Recovery

With this Path Recovery mechanism, we have devised and implemented an alternative cooperative mechanism to the previous Controller Based Path Recovery. Hybrid switches and controllers work together to setup a new alternative path under failure conditions. With this Path Recovery mechanism, controllers assist hybrid switches with the recovery procedure, but the autonomous operation of hybrid switches is used to establish the new paths. This mechanism is illustrated in Figure 6 where again hosts *s* and *d* communicate via *s1-s3-s5* (shortest path) until the link *s1-s3* goes down.

Path Recovery controller-side functions: We have implemented an SDN application using the Ryu SDN framework [48]. The Controller Assisted Path Recovery supports the following functions:

- *Host Table (HT) maintenance:* HT contains the hosts directly connected to every switch. The controller updates this table when it receives a packet from an edge switch containing the MAC address and port of a host connected to it.
- *Path Recovery Reply generation:* If the controller receives a *PacketIn* from a switch requiring a Path Recovery for a given destination (see Figure 6.a), it gets the MAC destination address from the packet and sends a *PacketOut* encapsulating the *Path Recovery Reply* packet to the destination edge switch (obtained from the HT), to trigger the recovery (see Figure 6.b).

Trans. Emerging Tel. Tech. 201X; **00**:1–14 © 201X John Wiley & Sons, Ltd.
DOI: 10.1002/ett
Prepared using ettauth.cls

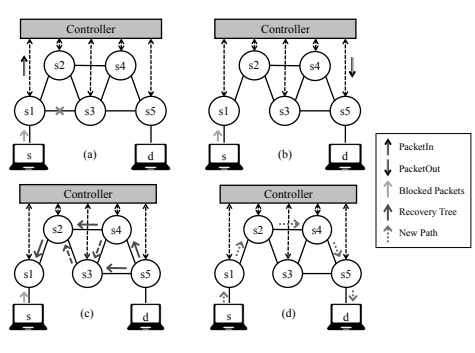


Figure 6. Controller Assisted Path Recovery.

Path Recovery switch-side functions: In addition to the controller functions, switches must also perform some new tasks to complete the desired Path Recovery.

– *Attached Host notification:* The hybrid switch informs the controller of the MAC address and port where a host is attached so that the controller keeps an updated *HT*.

– *Path Recovery Request:* If a unicast frame is received by a switch with no matching entry in its LT Table, it forwards the packet to the controller (PacketIn) (see Figure 6.a) to send a *Path Recovery Request*.

– *Path Recovery Reply flooding:* After receiving the PacketOut with an encapsulated *Path Recovery Reply* packet, an edge switch floods the multicast *Path Recovery Reply* packet (see Figure 6.c). The *Path Recovery Reply* packet is flooded from the destination edge bridge in the same way as it is done on the Distributed Path Recovery. Thus, the *Path Recovery Reply* packet reaches every hybrid switch in the network creating a kind of sink tree rooted at destination edge switch (and eventually, arriving to *s1*). Therefore, every hybrid switch in the network receiving the packet gets a viable path to reach the destination host.

Finally, frames are sent using the newly acquired path from the Path Recovery Reply packet flooding (Figure 6.d).

This cooperative recovery process avoids unnecessary computation on the controller for establishing an alternative path based on the traditional SDN centralized approach. It also avoids any unnecessary broadcast explorations like the Distributed Path Recovery mechanism to locate the destination edge switch, and leverages the information stored by the controller since path computation is unnecessary with this approach. Therefore, this last proposal makes an effective trade-off among performance, computation cost and bandwidth consumption.

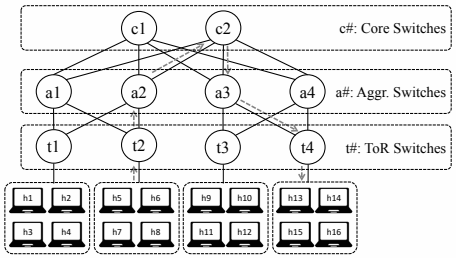


Figure 7. VL2 topology: Dc=Da=4 and four hosts per ToR switch

5. EVALUATION

We evaluated our hybrid switch proposal in different scenarios to analyze the amount of controller messages needed, CPU load, network throughput, delay, and path diversity achieved. We also verified the seamless compatibility of the proposed autonomous bridging with other SDN applications such as firewalls and the Controller Assisted Path Recovery mechanism previously explained in section 4.4. This evaluation is a very important work, since to the best of our knowledge, it is the first detailed work on measuring the performance of hybrid switches. Furthermore, we focus on the proposed *Controller Assisted Path Recovery* mechanism since it fully exploits the cooperation among hybrid switches and controller(s).

5.1. Testbed

We emulated several network scenarios with Mininet [49] on an Ubuntu 14.04.3 Linux machine with an i7-2600 3.40GHz CPU, 24GB RAM and SSD disk. A VL2 topology, as shown in Figure 7,was used for most of the experiments. Due to the performance limitations of the ofsoftswitch13 used for the implementation, interswitch link speeds are limited to 100 Mbps and host-switch link speed are limited to 10 Mbps. Switches are connected through an out-of-band control network to the Ryu controller. We designed the experiments to run at full CPU capacity in order to maximize the available switching capacity. The testbed supports up to 120-140 Mbps of overall throughput before saturation. Experiments were repeated at least ten times, computing the 95% confidence level until obtaining confidence intervals smaller than 10%, which validates the averaged obtained results.

5.2. Controller Offloading

First, we evaluated the load reduction obtained at the controller by delegation of bridging. On the VL2 topology depicted in Figure 7, several UDP flows were set as follows using *IPerf* [50]: host h1 sends traffic to every other host, host h2 sends to every other host but h1, host h3 sends to every other host but h1 and h2 and so on, for a total of $n*(n-1)/2=8*15=120$ flows. The flows were instantiated in order, starting from host h1 to h16 with a short time

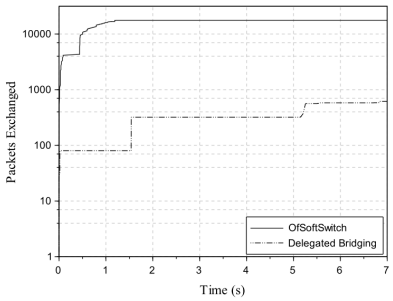


Figure 8. Cumulative controller-switch packets exchanged

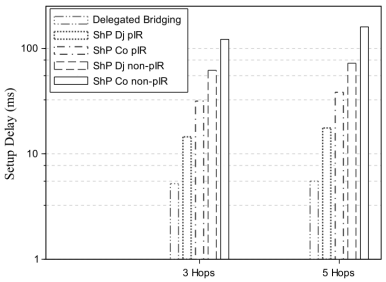


Figure 10. Average flow setup time

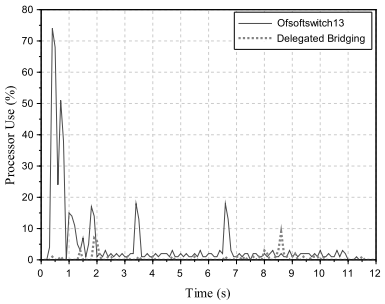


Figure 9. Ryu controller CPU load

period between each flow since we are only interested in the path setup overhead (messages to/from controller) and CPU load. OpenFlow rules are not pre-installed, thus tables at switches are empty at the beginning of every run.

Figure 8 shows the total accumulated number of packets processed by the controller across time. For non-autonomous bridging (standard OpenFlow) we can see a peak when network starts operation: the number of packets is high because the controller must discover the network topology and configure all the switches. We see how autonomous bridging exchanges a number of packets one order of magnitude lower, even during network initialization.

Figure 9 compares CPU load at the controller for the same experiment. Again, autonomous bridging clearly outperforms the centralized OpenFlow approach showing negligible use of the CPU even during network initialization. This improvement is because flows are established autonomously by the switches and the controller does not need to perform any calculation to establish new paths.

5.3. Path Setup Delay

We designed this experiment to compare the path setup times obtained using autonomous bridging and different optimal shortest path schemes (disjoint versus congruent paths, with or without pre-installed rules).

We set up 12 UDP flows on the topology of Figure 7, running from h1 to h5, h9 and h13, from h6 to h2, h10 and h14, from h11 to h3, h8 and h15 and from h16 to h4, h8 and h12. Hence, there are two flows between every pair of ToR switches (one in each direction) and every host is involved in one flow. Flows start in sequence and run at 3 Mbps each until the end of the experiment.

Figure 10 illustrates the average flow path setup time obtained after twenty runs for every routing scheme. Flows are separated into two groups according to the number of hops needed to reach from source ToR switch to destination, either 3 or 5 hops. We see how autonomous bridging performs better on average compared to all the controller-based approaches, irrespective of whether the rules are pre-installed (pIR) or not (non-pIR), and whether the paths selected are congruent (Co, paths coincide in both directions) or not (Dj, disjoint). Among the controller-based approaches, pre-installation of rules reduces path setup delays, as expected. Routing via congruent paths always exhibits greater delays because it needs more rules to be installed than for non-congruent paths. It is worth noting that path setup for autonomous bridging (ARP-Path) is faster than routing with pre-installed rules. This is because the standard ARP packets are used (besides their standard address resolution function) to establish the paths (with negligible additional delay), and because processing time at the Distributed Bridging Engine is smaller (comparisons are simpler) than processing all necessary OpenFlow pre-installed rules.

Figures 11(a) and 11(b) show the CDFs of path setup delay for the above mentioned tests. Autonomous bridging outperforms all controller based approaches, even with pre-installed rules. It is worth noting that pre-installed rules exhibit lower variance in path setup delays. When rules are not pre-installed when using controller based approaches, the path setup delay depends not only on the transmission time between switches and controllers, but also depends on the processing time at the controller. This dependency increases setup times and makes it quite unpredictable, producing a high variance. Autonomous bridging removes unpredictable delays in path setup for new flows when controller intervention is not strictly needed.

Trans. Emerging Tel. Tech. 201X; **00**:1–14 © 201X John Wiley & Sons, Ltd.
DOI: 10.1002/ett
Prepared using ettauth.cls

(a) Non pre-installed rules (b) Pre-installed rules

Figure 11. Delay setup time

	r	Path		r	Path
h6-h10	1	t2-a2-c2-a3-t3	h6-h14	1	t2-a2-c2-a3-t4
	2	t2-a2-c1-a4-t3		2	t2-a2-c1-a4-t4
	3	**t2-a2-c1-a3-t3**		3	t2-a2-c2-a4-t4
	4	**t2-a2-c1-a3-t3**		**4**	**t2-a2-c1-a3-t4**
	5	t2-a1-c2-a3-t3		**5**	**t2-a2-c1-a3-t4**
h11-h3	1	t3-a3-c1-a2-t1	h11-h7	**1**	**t3-a3-c1-a1-t2**
	2	t3-a4-c1-a1-t1		2	t3-a4-c1-a1-t2
	3	t3-a3-c2-a1-t1		3	t3-a3-c1-a2-t2
	4	t3-a3-c1-a1-t1		**4**	**t3-a3-c1-a1-t2**
	5	t3-a3-c2-a2-t1		5	t3-a3-c2-a2-t2
h16-h4	1	t4-a4-c2-a1-t1	h16-h8	1	t4-a3-c1-a2-t2
	2	t4-a4-c2-a2-t1		**2**	**t4-a4-c1-a1-t2**
	3	t4-a3-c1-a2-t1		3	t4-a4-c1-a1-t2
	4	t4-a3-c2-a1-t1		4	t4-a4-c1-a1-t2
	5	t4-a4-c1-a1-t1		**5**	**t4-a4-c1-a1-t2**

Table I. Example of multipath diversity

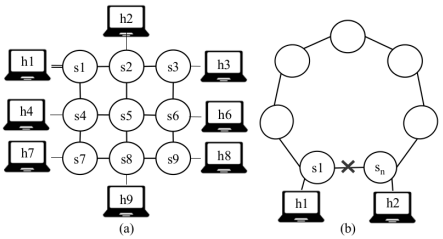


Figure 12. 3x3 mesh and *n*-ring topologies

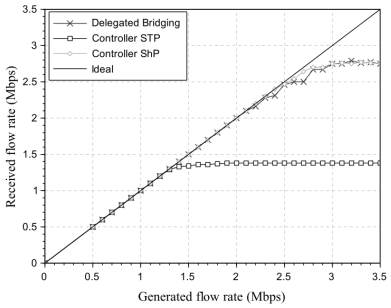


Figure 13. Overall throughput

5.4. Path Diversity

We study now how the ARP-Path based Distributed Bridging Engine exploits the intrinsic path diversity of certain topologies like VL2 to distribute the load. We use the same experiment setup as in the previous section. Table I presents the paths obtained in 5 repetitions of the experiment. We choose to depict only 6 representative flows to reduce table size. We see how the paths taken by the flows may vary or not, between consecutive repetitions, depending on which is the fastest one in each iteration. For example, the flow from h6 to h14 follows the same path (t2-a2-c1-a3-t4) in repetitions 4 and 5, which is different from the paths selected on repetitions 1, 2 and 3. This is because ARP-Path finds the minimum latency path at the moment of flow start and network conditions vary between repetitions. This produces a beneficial load balancing effect already known in this protocol [37].

5.5. Hybrid Switch Performance

Due to the computing limitation of our testbed, we use the topology shown in Figure 12.(a) to run the performance tests. It consists of a 3x3 square mesh of switches connected by 10 Mbps links, with a single host attached to every switch but the central one (s5). Host h1 is attached to s1 via a 100 Mbps link so that possible bottlenecks are restricted to interswitch links.

In the experiment, hosts h2 to h9 start a single UDP flow (of equal rate) each, to host h1 (a total of 8 flows arriving to h1). Then, we measured the amount of traffic received at h1 from every source under three different routing schemes: autonomous bridging; shortest path (ShP) with optimal paths precomputed and installed at the switches by a Ryu controller; and a suboptimal spanning tree (STP) (i.e. when the spanning tree does not coincide with the used paths, then paths are not shortest paths) also precomputed and

installed by the controller. We repeated the experiment increasing the flow rate (starting from 0.5 Mbps) in steps of 100 Kbps to test network throughput.

Figure 13 shows the traffic received by host h1 versus the traffic offered from each source. As expected, STP is the first scheme to reach saturation because it does not use all available links. In contrast, autonomous bridging achieves similar marks to ShP (since both route though shortest paths) and outperforms STP by a factor of 2 in terms of throughput. The advantage of autonomous bridging is that, in contrast to ShP, the controller does not need to take part in the path setup since paths are obtained on the fly by the switches when they are needed.

5.6. SDN/OpenFlow backward compatibility and Flow processing modes

We also performed some tests to verify compatibility of autonomous bridging with OpenFlow forwarding, more precisely the coexistence of the different flow processing modes (autonomous, controller-based and cooperative) explained in section 3.3.

5.6.1. Controller-based Path Selection

Controller based forwarding uses only OpenFlow rules. These rules can be pre-installed or not; if not, the path is set up on the fly by the controller. Path set-up delays for this approach were shown in section 5.3.

5.6.2. Distributed Bridging coexisting with Controller Based Packet Filtering

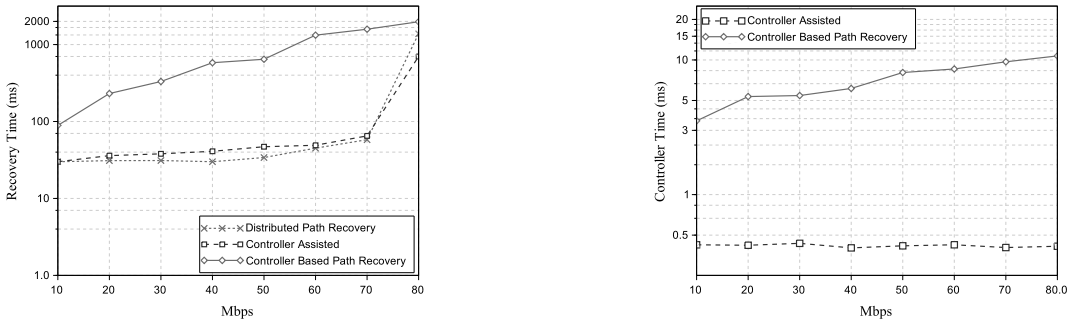
We set up paths via OpenFlow to configure a firewall application and combine it with autonomous bridging for the remaining flows. We used the topology shown in Figure 12.(a). The controller first introduces all necessary rules to filter the traffic on the network via *PacketIn* packets. These rules work as follows: host h1 can neither send nor receive IP packets to/from h3. Host h2 can neither send nor receive IP packets to/from h4. The switches populate their OpenFlow tables according to the instructions from the controller. When a packet that does not match any rule arrives, according to OpenFlow 1.3 it should be discarded. However, autonomous bridging is active and can perform autonomous flow processing and forward it according to the ARP-Path protocol. We establish two UDP flows, one between h1 and h2 and the other between h4 and h2, and observe how the flow between h1 and h2 is exchanged normally but only the ARP packets (request & reply) between h4 and h2 are forwarded because UDP traffic is filtered as specified by the rules installed on the switches. In this manner we verify the compatibility of autonomous bridging with a controller-based firewall-like application. However, some other services, such as an active QoS service requiring the modification or elimination of pre-existing flows setup locally by the autonomous bridging operation will not be feasible because the controller does not know about these flows. This seems to be an interesting additional feature in order to provide full visibility of state to the controller. Of course, it is up to the controller to gain full control if needed by simply installing a forward-me-all rule as with OpenFlow 1.3.

5.6.3. Path Recovery Performance

Finally, we evaluate the different Recovery mechanisms previously proposed on section 4.4. We use a 7-ring topology as shown in Figure 12.(b). This selected topology represents a worst-case scenario: a failure between two adjacent switches forces reconfiguration at all other switches on the ring. UDP traffic is set from host h1 to h2 and, once the path is established between the two hosts, the link between the switches is removed to simulate a link failure. When the failure is detected by the switch, each one of the proposed Recovery mechanisms is evaluated. Each one of the 7 switches in the ring topology uses 100 Mbps links in order to compare our results with those shown in [21]. Different traffic speeds between the broken links are evaluated from 10 Mbps to 80 Mbps. These speeds are set up with a multiflow group of 1 Mbps independent flows. Furthermore, we also established background traffic of 50 Mbps along the recovery path in the same direction as the established flows. The Recovery time was measured as the worst recovery time of all flows that needed to be recovered from the broken link. The results are shown in Figure 14(a).

Additionally, we can see on Figure 14(a) how the controller based recovery offers the worst performance for all traffic speeds. This is expected, because as the number of flows increases, the controller needs to perform the necessary computations per flow, and recovering all flows requires proportionally more time. In contrast, it can be observed how the Controller assisted Recovery and the Distributed Recovery always perform better than the Controller Based recovery. Only at the highest level of throughput do these two recovery mechanisms show a significant increase in the Recovery time, while still beating the Controller based Recovery in every case. This is because the cooperative recovery packets used by hybrid switches and regular data traffic experience queueing times, which are only relevant under high loads. Finally, it is important to note how the Controller Assisted Recovery performs better under high data loads and the Distributed Recovery performs better under smaller data loads. This behaviour is due to the processing time at the controller and the fact that the delay on the control network between controller and hybrid switches is not negligible. Controller assisted recovery, is only more effective when data traffic is high enough, making queuing delays bigger than the delay on the control network. Nevertheless, the performance of both methods is quite similar. Thus, we consider Controller Assisted Recovery to be the best option since it has the best behaviour under high loads and always requires less broadcast traffic than Distributed Recovery in every case. In addition to the previous results, we have also measured the controller processing time per flow for the Controller Assisted and Controller Based recoveries. The results in

Trans. Emerging Tel. Tech. 201X; **00**:1–14 © 201X John Wiley & Sons, Ltd.
DOI: 10.1002/ett
Prepared using ettauth.cls



(a) Recovery Time

(b) Controller Processing Time per flow

Figure 14. Recovery Performance

Figure 14(b) show a difference of about one order of magnitude between them. The processing time when using Controller Assisted Recovery is negligible when compared to the Controller Based Recovery. Furthermore, it is interesting to note how the processing per flow increases with the number of existing flows when using Controller Based Recovery. This effect is due to the fact that during the processing time of each flow, the Ryu controller must queue all remaining Path Recovery Requests for their later processing, which has a non-negligible effect.

Other approaches require a constant exchange of packets (i.e. BFD based systems [22]) but our proposal uses the same data stream to check the status of links without probe packets, which reduces the number of packets inserted in the links. Our Distributed Recovery and Controller Assisted Recovery provide competitive recovery times, even better than alternative methods when BFD interarrival packet time is greater than 5 ms [21].

Based on the obtained results, we can see the advantages of using hybrid switches to reduce the load on controllers and to reduce in certain aspects the dependency on them. Furthermore, the cooperation between hybrid switches and controllers can help to increase the performance of certain network operations such as Recovery procedures.

6. DISCUSSION

This section aims to discuss, based on our experience with our implementation, several aspects of networks based on hybrid switches related to performance, network state and alternative platforms for widely implementing our proposal.

6.1. Offloading

The use of hybrid switches reduces the load on controllers and improves network performance, scalability and resilience. Eliminating the need for the controller to specify and control all flows irrespective of their importance dramatically reduces the processing overhead at controllers and its required communication bandwidth (section 5.2). It also shortens the queue size at controller reception buffers, decreasing response times of all flows specifically configured, e.g. at OpenFlow tables. Furthermore, it reduces the number of rules needed to be installed per switch, which in turn accelerates the installation of relevant rules in the network. Even though all necessary rules could be initially pre-installed, the network initialization time would increase, reducing its availability and growing complexity with increasing network sizes. In addition, reconfiguration after a network failure would require additional controller intervention to re-accommodate the flows. Thus, reducing the number of flows managed by the controller improves network availability and resilience. Moreover, cooperative recovery consistently maintains a good trade-off between processing time and broadcast recovery traffic even on high failure rates since the benefits due to cooperative operation do not directly depend on the number of recovered failures. On the contrary, the controller based approach will quickly degrade with the number of failures as the increasing load on the controller will produce greater recovery times. Regarding the distributed approach, every failure recovery will produce twice as much broadcast messages as the collaborative recovery.

6.2. Global Knowledge and Cooperative Processing

The use of our hybrid switch makes the controllers lose the global network knowledge typical of SDN. They do not have complete information about paths for all flows established in the network; although statistics remain available. An adequate balance between local and global knowledge is key to achieving scalability in decentralized SDN architectures because global knowledge is not needed in many cases [23]. For instance, section 5.6.2 shows how a controller based stateless firewall can be easily configured without losing its functionality.

Stateless firewall related rules can be installed and used as expected on any regular OpenFlow network, while other applications may need additional (or even global) information to operate properly. A common use case for global knowledge is QoS active management of premium flows. In this case, non premium flows should be rerouted or discarded when needed in order to accommodate premium flows to the desired Service Level Agreement. We see as future work, out of scope of this paper, an extension of OpenFlow that adds the capability to receive the generated state (e.g. established flows) from the Distributed Bridging Engine, to complete the knowledge of the state at hybrid switches. Also, a mechanism to handle possible conflicts between SDN applications/services and our hybrid switches should be devised to facilitate the integration with existing SDN deployments. Furthermore, the global knowledge on controllers would help to select the most appropriate mechanisms, cooperative or not (i.e. the recovery mechanisms revised in this paper), according the deployed network, and the requirements of the offered/implemented services. It would be even possible to change dynamically how the network operates depending on its conditions and to improve the provisioning of the running SDN applications/services. However, standardised APIs (i.e. [51]) are needed to coordinate controllers, applications/services and hybrid switches.

6.3. Other platforms for hybrid switches

If used, hybrid switches would be needed both in hardware and software implementations. Software switches are used at the network edge for reasons of proximity to VMs, flexibility and configurability, and hardware switches at the core to maximize overall performance. The restrictions (flow installation delay and rate, maximum number of flows, etc.) are different for pure software than for hardware switches. Our hybrid switch has been implemented using the quick prototyping ofsoftswitch13 [40, 41]. For higher performance, either hardware implementations of the OpenFlow switch would be needed, or high performance virtual switches like Open vSwitch. Some interesting possibilities are a hardware implementation on NetFPGA (extending the first OpenFlow switch implementation) [52] or using the promising P4 language [53, 54] for extending its use in any software/hardware P4 compatible platform.

7. CONCLUSIONS

We have proposed and evaluated a hybrid switch concept along with cooperative mechanisms between controller and switches. The switches can both autonomously take charge of basic bridging, and cooperate in path recovery with the controller, in order to reduce state, and improve scalability and resiliency. This delegation offloads the SDN controllers (section 5.2) but maintains the capability of the controller to install all necessary specific forwarding rules (section 3.3). Additionally, the simultaneous existence of forwarding decisions on the control plane (controllers) and also on the data plane (switches) makes it possible to handle flows not only based on either controller or switch decisions, but also on cooperative actions involving both controllers and switches if desired. This new approach opens new possibilities for cooperative interactions between the controller and the network. A proof of concept has been implemented on an open source software switch (section 4) to evaluate empirically the advantages of this partial configurable decentralisation of the control plane. ARP-Path was the selected bridging protocol (see section 4.2) due to its native loop prevention mechanism, its similar performance to shortest path algorithms, and due to not requiring specific control packets except in the case of path recovery mechanisms (section 5.5). Delegation of bridging functions not only implies the offloading of rules managed by controllers, it also improves path setup delay (section 5.3) due to the reduction of load on the controller, since the processing and queueing times related to controller communication decrease. Finally, the advantages of cooperative flow processing have been demonstrated, which can be used with the proposed hybrid switch. The Controller Assisted Recovery has demonstrated in section 5.6.3 its best performance under higher loads, and obtains a good trade-off between controller processing time and in-band broadcast traffic. Furthermore, under small loads, its performance is very close to the best results from other recovery strategies. Thus, we think that this proposal opens a new avenue for improving current SDN architectures and highlights how open issues still remain in this area based on switches and controllers cooperation (section 6.2).

8. ACKNOWLEDGEMENTS

This work was supported in part by grants from Comunidad de Madrid through Project TIGRE5-CM (S2013/ICE-2919) and from Ministerio de Economia y Competitividad through Programa de Garanta Juvenil (PEJ-2014-A-41867). Thanks to Javier Luzón who helped in the selection of OpenFlow switches and to Matt Hutton for the English language editing.

REFERENCES

1. Curtis AR, Mogul JC, Tourrilhes J, Yalagandula P, Sharma P, Banerjee S. DevoFlow: Scaling Flow Management for High-performance Networks. *SIGCOMM Comput. Commun. Rev.* Aug 2011; **41**(4):254–265.
2. Al-Fares M, Radhakrishnan S, Raghavan B, Huang N, Vahdat A. Hedera: Dynamic flow scheduling for data center networks. *Proceedings of the 7th USENIX Conference on Networked Systems Design and*

Implementation, NSDI'10, USENIX Association: Berkeley, CA, USA, 2010; 19–19.
3. Stephens B, Cox A, Felter W, Dixon C, Carter J. PAST: Scalable ethernet for data centers. *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, ACM, 2012; 49–60, doi:10.1145/2413176.2413183.
4. Hassas Yeganeh S, Ganjali Y. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, ACM: New York, NY, USA, 2012; 19–24.
5. Bianchi G, Bonola M, Capone A, Cascone C. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *SIGCOMM Comput. Commun. Rev.* Apr 2014; **44**(2):44–51.
6. Yu M, Rexford J, Freedman MJ, Wang J. Scalable flow-based networking with difane. *SIGCOMM Comput. Commun. Rev.* Aug 2010; **41**(4):–.
7. Qian C, Lam SS. A scalable and resilient layer-2 network with ethernet compatibility. *IEEE/ACM Transactions on Networking* Feb 2016; **24**(1):231–244.
8. Jeyakumar V, Alizadeh M, Kim C, Mazières D. Tiny Packet Programs for Low-latency Network Control and Monitoring. *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, ACM: New York, NY, USA, 2013; 8:1–8:7.
9. Moghaddam RF, Cheriet M. SmartPacket: Re-Distributing the Routing Intelligence among Network Components in SDNs. *CoRR* 2014; **abs/1412.0501**. URL http://arxiv.org/abs/1412.0501.
10. Zheng K, Wang L, Yang B, Sun Y, Uhlig S. Lazyctrl: A scalable hybrid network control plane design for cloud data centers. *IEEE Transactions on Parallel and Distributed Systems* 2016; **PP**(99):1–1.
11. Mekky H, Hao F, Mukherjee S, Zhang ZL, Lakshman T. Application-aware data plane processing in sdn. *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, ACM: New York, NY, USA, 2014; 13–18.
12. Bhandarkar S, Behera G, Khan KA. Scalability Issues in Software Defined Network (SDN): A Survey. *Advances in Computer Science and Information Technology (ACSIT)* 2015; **2**.
13. Guan X, Choi BY, Song S. Reliability and Scalability Issues in Software Defined Network Frameworks. *Research and Educational Experiment Workshop (GREE), 2013 Second GENI*, 2013; 102–103.
14. Gelberger A, Yemini N, Giladi R. Performance Analysis of Software-Defined Networking (SDN). *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on*, 2013; 389–393.
15. Soeurt J, Hoogendoorn I. Shortest path forwarding using OpenFlow. *Technical Report*, University of Amsterdam 2012. URL https://www.os3.nl/_media/2011-2012/courses/rp1/p25_report.pdf, bibtex: soeurt_shortest_2012.
16. Awobuluyi O. Periodic control update overheads in OpenFlow-based enterprise networks. *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*, 2014; 390–396. Bibtex: 6838691.
17. Hakiri A, Gokhale A, Berthou P, Schmidt DC, Gayraud T. Software-defined networking: Challenges and research opportunities for future internet. *Computer Networks* 2014; **75, Part A**:453 – 471.
18. Yeganeh S, Tootoonchian A, Ganjali Y. On scalability of software-defined networking. *Communications Magazine, IEEE* February 2013; **51**(2):136–141.
19. Kempf J, Bellagamba E, Kern A, Jocha D, Takacs A, Skoldstrom P. Scalable fault management for OpenFlow. *Communications (ICC), 2012 IEEE International Conference on*, 2012; 6606–6610.
20. 802.1AB: Link Layer Discovery Protocol (LLDP). http://standards.ieee.org/getieee802/download/802.1AB-2009.pdf.
21. van Adrichem NL, van Asten BJ, Kuipers FA. Fast Recovery in Software-Defined Networks. *Software Defined Networks (EWSDN), 2014 Third European Workshop on. IEEE*, 2014; 6.
22. Katz D, Ward D. *Bidirectional Forwarding Detection (BFD)*. No. 5880 in Request for Comments, IETF, 2010. URL http://www.ietf.org/rfc/rfc5880.txt, published: RFC 5880 (Proposed Standard).
23. Stevens M, Ng B, Streader D, Welch I. Global and local knowledge in SDN. *Telecommunication Networks and Applications Conference (ITNAC), 2015 International*, 2015; 237–243.
24. Bifulco R, Boite J, Bouet M, Schneider F. Improving SDN with InSPired Switches. *ACM Symposium on SDN Research (SOSR '16)*, 2016.
25. Cai Z. Maestro: Achieving Scalability and Coordination in Centralizaed Network Control Plane. PhD Thesis, Rice University, Houston, TX, USA 2012. AAI3521292.
26. Tootoonchian A, Ganjali Y. HyperFlow: A Distributed Control Plane for OpenFlow. *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, USENIX Association: Berkeley, CA, USA, 2010; 3–3.
27. Koponen T, Casado M, Gude N, Stribling J, Poutievski L, Zhu M, Ramanathan R, Iwata Y, Inoue H, Hama T, *et al.*. Onix: A Distributed Control Platform for Large-scale Production Networks. *Proceedings of the 9th USENIX Conference on*

Operating Systems Design and Implementation, OSDI'10, USENIX Association: Berkeley, CA, USA, 2010; 1–6.
28. Song S, Hong S, Guan X, Choi BY, Choi C. NEOD: Network Embedded On-line Disaster management framework for Software Defined Networking. *IM*, Turck FD, Diao Y, Hong CS, Medhi D, Sadre R (eds.), IEEE, 2013; 492–498.
29. Wackerly S. OpenFlow-hybrid mode. *Technical Report*, HP 2014. URL https://wiki.opendaylight.org/images/1/1d/ODL_Hybrid_Mode.pdf.
30. Hybrid OpenFlow, the Brocade Way â ipSpace.net by @ioshints 2012. URL http://blog.ipspace.net/2012/06/hybrid-openflow-brocade-way.html.
31. The Practical Path to SDN: Brocade OpenFlow Hybrid Port Mode 2013. URL http://community.brocade.com/t5/SDN-NFV/The-Practical-Path-to-SDN-Brocade-OpenFlow-Hybrid-Port-Mode/ba-p/417.
32. Software-defined networking (SDN) definition - open networking foundation. URL https://www.opennetworking.org/sdn-resources/sdn-definition.
33. Cabaj K, Wojciech J, Kukliński S, Radziszewski P, Truong Dinh K. SDN architecture impact on network security. *Position Papers of the 2014 Federated Conference on Computer Science and Information Systems*, vol. 3, 2014; 143–148, doi:DOI:10.15439/2014F473.
34. Sharma S, Staessens D, Colle D, Pickavet M, Demeester P. Openflow: Meeting carrier-grade recovery requirements. *Computer Communications* 2013; **36**(6):656 – 665. Reliable Networkbased Services.
35. Pashkov V, Shalimov A, Smeliansky R. Controller failover for SDN enterprise networks. *Science and Technology Conference (Modern Networking Technologies) (MoNeTeC), 2014 International*, 2014; 1–6.
36. Enns R, Bjorklund M, Bierman A, Schönwälder J. Network Configuration Protocol (NETCONF). RFC 6241 Oct 2015, doi:10.17487/rfc6241. URL https://rfc-editor.org/rfc/rfc6241.txt.
37. Rojas E, Ibanez G, Gimenez-Guzman JM, Carral JA, Garcia-Martinez A, Martinez-Yelmo I, Arco JM. All-path bridging: Path exploration protocols for data center and campus networks. *Computer Networks* 2015; **79**(0):120 – 132, doi:http://dx.doi.org/10.1016/j.comnet.2015.01.002.
38. McKeown N, Anderson T, Balakrishnan H, Parulkar G, Peterson L, Rexford J, Shenker S, Turner J. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* April 2008; **32**(2):69–74.
39. OvS: Open vSwitch. http://openvswitch.org/.
40. CPqD: OpenFlow 1.3 Software Switch. https://github.com/CPqD/ofsoftswitch13.
41. Leão Fernandes E, Esteve Rothenberg C. OpenFlow 1.3 software switch. *In Salão de Ferramentas XXXII Simpósio Brasileiro de Redes de Computadores - SBRC'2014*, 2014.
42. Indigo Virtual Switch. http://www.projectfloodlight.org/indigo-virtual-switch/.
43. DPDK: Data Plane Development Kit. http://dpdk.org/.
44. LINC: OpenFlow software switch. https://github.com/FlowForwarding/LINC-Switch.
45. OpenFlow Switch Specification v1.3.2. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.2.pdf.
46. Ibanez G, Carral J, Arco J, Rivera D, Montalvo A. ARP-Path: ARP-Based, Shortest Path Bridges. *Communications Letters, IEEE* July 2011; **15**(7):770–772, doi:10.1109/LCOMM.2011.060111.102264.
47. Ibanez G, Naous J, Rojas E, Rivera D, Carral JA, Arco JM. A simple, zero-configuration, low latency, bridging protocol. *Proceedings of Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, 2010.
48. Ryu SDN framework. URL http://osrg.github.com/ryu/.
49. Mininet: An instant virtual network on your laptop (or other PC) - mininet. URL http://mininet.org/.
50. iperf - the network bandwidth measurement tool. URL https://iperf.fr/.
51. Gutiérrez PAA, Rojas E, Schwabe A, Stritzke C, Doriguzzi-Corin R, Leckey A, Petralia G, Marsico A, Phemius K, Tamurejo S. Netide: All-in-one framework for next generation, composed sdn applications. *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, 2016; 355–356, doi:10.1109/NETSOFT.2016.7502408.
52. Naous J, Erickson D, Covington GA, Appenzeller G, McKeown N. Implementing an openflow switch on the netfpga platform. *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, ACM: New York, NY, USA, 2008; 1–9.
53. Bosshart P, Daly D, Gibb G, Izzard M, McKeown N, Rexford J, Schlesinger C, Talayco D, Vahdat A, Varghese G, *et al.*. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.* Jul 2014; **44**(3):87–95.
54. P4 Language Web Page. URL http://p4.org/.

DOI: 10.1002/ett
Prepared using ettauth.cls