Received July 25, 2019, accepted August 15, 2019, date of publication August 26, 2019, date of current version September 12, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2937528

Amaru: Plug&Play Resilient In-Band Control for SDN

DIEGO LOPEZ-PAJARES[1], **(Member, IEEE), JOAQUIN ALVAREZ-HORCAJO**[1], **(Member, IEEE), ELISA ROJAS**[1], **A. S. M. ASADUJJAMAN**[2], **(Member, IEEE), AND ISAIAS MARTINEZ-YELMO**[1]

[1]Departamento de Automatica, University of Alcala, 28805 Alcala de Henares, Spain
[2]Network Planning Department, Banglalink Digital Communications Ltd., Dhaka 1212, Bangladesh

Corresponding author: Elisa Rojas (elisa.rojas@uah.es)

This work was supported in part by the Comunidad de Madrid through Project TAPIR-CM under Grant S2018/TCS-4496, and in part by the University of Alcala through Project CCG2018-EXP/076 and the Formacion del Profesorado Universitario (FPU) Program.

ABSTRACT Software-Defined Networking (SDN) is a pillar of next-generation networks. Implementing SDN requires the establishment of a decoupled control communication, which might be installed either as an out-of-band or in-band network. While the benefits of in-band control networks seem apparent, no standard protocol exists and most of setups are based on ad-hoc solutions. This article defines Amaru, a protocol that provides plug&play resilient in-band control for SDN with low-complexity and high scalability. Amaru follows an exploration mechanism to find all possible paths between the controller and any node of the network, which drastically reduces convergence time and exchanged messages, while increasing robustness. Routing is based on masked MAC addresses, which also simplifies routing tables, minimizing the number of entries to one per path, independently of the network size. We evaluated Amaru with three different implementations and diverse types of networks and failures, and obtained excellent results, providing almost on-the-fly rerouting and low recovery time.

INDEX TERMS SDN, OpenFlow, in-band control, resilient networks, path exploration.

I. INTRODUCTION

Control communication is pivotal for the thriving Software-Defined Networking (SDN) paradigm, which –by definition– cracks the network architecture into control (logical) and data (physical) planes [1]. The main purpose of the control communication is to connect both planes and it may be deployed either as an out-of-band or an in-band network [2], or even as a hybrid solution [3]. Out-of-band control is arranged as a dedicated network, isolated from the data plane links and devices, thus requiring additional resources; while in-band leverages the already existing deployment to perform the communication, hence sharing the resources with the data plane. Finally, hybrid solutions share the advantages and disadvantages of both modes, as control communication might either be routed through dedicated or –the already existing– data links.

The justification to implement each approach resides in diverse aspects [2], which we could summarize as follows:

1) As the network grows in size, in-band control is more scalable and cheaper. For instance, a dedicated out-of-band control network for carrier networks is almost inconceivable due to its high cost.
2) In-band networks are less prone to congestion due to the high-capacity nature of the data plane links.
3) In-band networks are more resilient to link failures as they might provide several routes without extra cost, while the out-of-band approach is restrained to a single route per deployed out-of-band network.
4) Bootstrapping is almost immediate for out-of-band approaches, but requires additional configuration for in-band communications.
5) Out-of-band networks are usually less vulnerable to security issues.

The three first statements illustrate clear advantages of in-band control in Software-Defined Networking (SDN). In fact, some SDN deployments are only possible using in-band control [2]. However, differently from out-of-band networks, no standard or official protocol exists to set up in-band networks for SDN. As a consequence, current in-band control is usually based on inefficient manual configurations, which hinders the main benefits of the in-band approach, as stated in the fourth point. Therefore, there is an urge for a (1) resilient and (2) automatic in-band protocol for SDN.

The associate editor coordinating the review of this manuscript and approving it for publication was Zehua Guo.

In particular, the former requirement, resiliency, is better for in-band than for out-of-band, but we should carefully analyze how the alternative routes are provided. Basically, the two main fault recovery approaches are *protection* and *restoration* [4], which pre-install and calculate the alternative routes on-the-fly when failures occur, respectively. According to literature, paths should be recovered within 50 ms to avoid further issues [5], and recent studies prove that only the protection mechanism accomplishes this requirement. However, pre-installing routes might not be scalable, or even possible, at all times.

The latter requirement, fast –and automatic– bootstrapping, is also a challenge for in-band control in SDN [6], [7], as most of the current approaches either bootstrap the network manually, leverage traditional routing protocols, or even rely on an additional out-of-band control network to configure it, which increases complexity and bootstrapping time in comparison with simple out-of-band networks.

In this article, we define, implement and evaluate Amaru, a protocol to deploy in-band control for SDN that provides three main features: (1) fast automatic bootstrapping and (2) multiple alternative routes to boost resiliency based on the protection scheme, while (3) reducing the number of table entries to prevent any scalability issues. To the best of our knowledge, Amaru is the only existing protocol that provides these three features. As such, Amaru tries to accomplish the above-mentioned challenges for in-band control.

The article is structured as follows: In Section II, we review the current state of the art regarding in-band control deployments and protocols for SDN. In Section III we define the Amaru protocol and its core features. Afterwards, in Section IV, we describe the different implementations developed to prove the behavior of Amaru. Finally, we evaluate and discuss the protocol in Section V, and provide the main conclusions in Section VI.

II. RELATED WORK

Beheshti and Zhang [8] formulate one of the first works to examine the **resiliency of in-band control** in SDN. Specifically, the authors define a protection metric and evaluate the algorithms to provide the best protection scheme. However, they do not question how to pre-install the alternative routes, its scalability or automatic deployment of the in-band network. The protection and restoration approaches are initially compared for in-band OpenFlow networks by Sharma *et al.* [4], [9], where the conclusion is that only protection grants a recovery under the required 50 ms [5]. In order to minimize the overall recovery time, failures should be detected as fast as possible, independently of how alternative routes are provided later on. In the literature, we can also find **fast failure detection** algorithms, which are mainly software-based solutions based on monitoring [10], [11].

Apart from detection time, recovery will also depend on the protection or restoration scheme in use. In the case of **protection**, rerouting is almost immediate, but then minimizing the number of table entries is crucial. Feng *et al.* [12] provide an algorithm for this purpose. Hu *et al.* [13] investigate protection with multiple controllers and propose a scheme based on local rerouting and reverse forwarding. Huang *et al.* [14] also investigate an optimizer to provide robust end-to-end global rerouting while minimizing resource utilization, but exclusively for single-link failures. While the previous ones are merely studies, Goltsmann *et al.* [15] plan to provide backup routes via direct calculation of the network nodes, which is faster but it might overload them (as nodes need to be aware of the whole topology, to perform breadth-first search, to install their own routes, etc.). However, this latter approach has only been designed for single-link failures as well. Chan *et al.* [16] designed a disjoint path planning to achieve fast reroute after failure. However, it requires backup controllers and failure detection depends on the monitoring cycle. González *et al.* [17] provide a resilient OpenFlow channel through MPTCP, but it requires an out-of-band channel for the initial setup, hence exclusively using in-band communication is unfeasible. Also leveraging MPTCP for in-band control, Raza and Lee [18] design a heuristic algorithm to compute disjoint and short-lengthed paths, but they leave as future work the associated protocol definition (that is, the procedure of constructing the in-band control connections).

Otherwise, if **restoration** is implemented, i.e. no alternative route is pre-installed, creation of new paths should be fast enough. RASCAR [19] intelligently calculates the best routes considering that shortest-path strategies do not always minimize recovery time of control paths. A similar approach is followed by MORPH [20], which reassigns controllers when the main one is affected by a security attack.

Only a few approaches implement resilient in-band control via **automatic bootstrapping**. ResilientFlow [6] is one of the first works to mention this challenge and it partially addresses it. It is a distributed control channel module that lets switches maintain their own control communication channels. Nevertheless, it requires a specific module installed plus an initial Open Shortest Path First (OSPF) [21] configuration, and also periodic exchange of network topology maps, increasing the signaling overhead. FASIC [22] provides a protection scheme based on OVSDB and automatic setup; but it requires continuous monitoring and downtime after failure claims to be around 5 seconds, which is smaller than using standard OVS configuration, but much higher than the 50 ms requirement. Medieval [23], [24] also claims to be plug&play, robust (via the creation of two spanning trees per controller) and self-stabilizing; however, it requires the pre-installation of diverse rules in the switches. Renaissance [25], [26] is a distributed in-band control system that creates one single tree per controller to obtain k-fault-resiliency. Nonetheless, if only one controller is deployed, only one path is generated and, as a link-state approach, the authors focus the analysis on its self-stabilizing properties and they do not evaluate recovery after multiple failures. Furthermore, bootstrapping requires several seconds and depends on the network update interval. The in-band control approach by Asadujjaman *et al.* [27] tackles multiple failures and implements automatic bootstrapping.

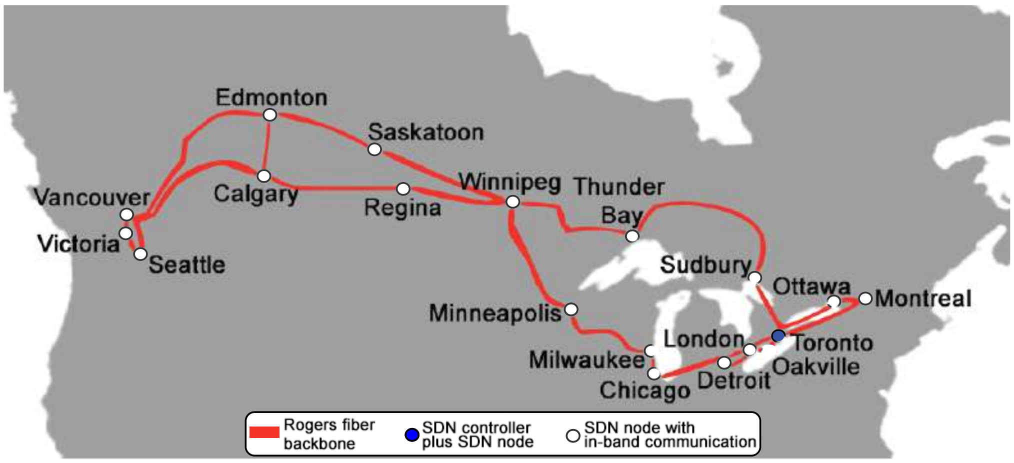


FIGURE 1. The Rogers topology as an example of carrier network for in-band control in SDN.

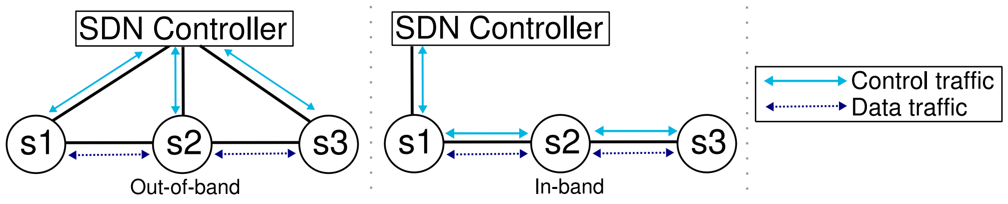


FIGURE 2. Example of out-of-band and in-band control in SDN.

However, it does not provide protection (understood as pre-installed backup routes) and controller-to-switch communication is source-routed, which increases packet overhead. Bentstuen and Flathagen [7] exclusively focus on the bootstrapping problem, by outlining several steps to pursue automatic setup.

Finally, we would like to highlight some other relevant works related with the topic. For example, Hark *et al.* [28] propose bilateral communication among SDN controllers via in-band control implemented with VLANs plus One-Hot-Coding to reduce the number of table entries. However, it does not address in-band communication per se, automatic bootstrapping or resiliency aspects. eTDP [29] presents a protocol to discovery the topology and initial network deployment, but it does not analyze runtime in-band control or the creation of multiple paths for resiliency. Gather [30] presents a method to reduce the number of table entries for in-band routing via the use of DNS-like grouping. Their authors evaluate algorithms to group addresses. Görkemli *et al.* [31] evaluate the performance of dynamic control planes and introduce the concepts of ''control flow table'' and ''control plane manager''. Nevertheless, it lacks a deeper analysis on automatic bootstrapping and recovery after failure.

Consequently, the main contribution of Amaru in relation to the current state of the art is to provide automatic bootstrapping, enhanced resiliency, while preserving scalability and low-complexity of in-band networks. So far, none of the current approaches accomplish all of these requirements.

III. AMARU

The core principle of Amaru is to strengthen the resiliency of in-band SDN communication, and –additionally– provide it via a fast and automatic bootstrapping. To achieve this goal, it explores all possible paths between any network device and the ones directly connected to the SDN controller, which guarantee backup routes after link or node failure. At the same time, Amaru should not affect network performance and, for this reason, it tries to reduce the signaling overhead.

As previously mentioned, one particular application of Amaru are carrier networks, where nodes are located very far ones from the others. Figure 1 depicts this use case, where only Toronto is connected to the SDN controller and the rest are controlled via in-band communication, as deploying out-of-band control will basically mean duplicating the already existing network and its associated maintenance cost. The architectural differences between out-of-band and in-band control are briefly summarized in Fig. 2. In the in-band approach, resources are saved (some links are not deployed) at the cost of leveraging the same data links both for data and control traffic.

We describe Amaru in detail in the next sections. Firstly, we explain the hierarchical labels leveraged in Amaru and the distributed mechanism to assign them, in Sections III-A and III-B, respectively. Afterwards, we describe the forwarding logic, in Section III-C, and the procedure to follow in case of network reconfiguration, in Section III-D. Then, we describe why Amaru boosts the

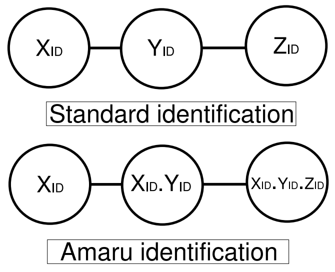


FIGURE 3. Hierarchical labeling in Amaru.

resiliency and scalability of in-band SDN communication in Sections III-E and III-F. Finally, we clarify how Amaru can be applied to multiple root nodes in Section III-G.

A. HIERARCHICAL LABELING

The foundations of Amaru are based on GA3 [32], which creates multiple paths for data center network based on hierarchical labels, where labels are unique per node. In Amaru, the first node or *root* node obtains an initial label, and then the label is increased in one field per hop, by simply adding the identity of the hop traversed. For instance, let us assume that the first node X is labeled X_{ID}. If X is directly connected to Y, then the latter node obtains the composed label $X_{ID}.Y_{ID}$. If Y is connected to a third node Z, the label $X_{ID}.Y_{ID}.Z_{ID}$ will be assigned to Z, and so on. The key advantage of this type of labeling is that the node IDs indirectly represent the connectivity among nodes. For instance, $X_{ID}.Y_{ID}.Z_{ID}$ should have a direct link to $X_{ID}.Y_{ID}$, because they just differ in the final digit of their label (i.e. Z_{ID}). Accordingly, the added value of these labels is that they have a hierarchical nature and they enclose the path traversed from the *root* node, as depicted in Fig. 3.

In Amaru, these labels are calculated to mask the real MAC addresses of each network device (addresses that are usually *meaningless* for routing as they only convey manufacturing information), without modifying the standard Ethernet frame, i.e. using the same amount of bytes in the header. It is important to notice that the MAC addresses of the SDN switches are usually assigned by the SDN controller in software. Therefore, it is easy to mask them with new meaningful MAC addresses in SDN that would not be feasible in case of legacy switches. In fact, with the capability of SDN to dynamically assign MAC addresses to any switch interface, the assumption that MAC addresses should be non-meaningful globally unique numbers is no longer valid. In Amaru we evolve the notion of MAC addresses to capitalize on the aforementioned observation.

To distinguish these masked MAC from standard addresses, they are identified as Amaru MACs (AMACs), where the U/L (unique/locally administered) bit in the address field is set to 1. Afterwards, these AMACs contain up to 46 bits (48 of the original MAC minus the broadcast/multicast and the U/L bits) to represent the labels, which could be translated into 6 levels of around 8 bits each, for example. The number of levels represent the maximum length of each path (e.g. 6 levels means the longest path will have 6 hops towards the SDN controller). Accordingly, with the same frame size, one AMAC conveys the information towards the *root* node, which is the switch directly connected to the SDN controller in our case.

B. ROOT DISCOVERY AND LABELING PROCEDURE

As the main purpose of Amaru is to provide augmented resiliency for SDN in-band control, the Amaru protocol aims to find as many paths as possible from any SDN node towards the *root* node, that is, the node directly connected to the SDN controller. These paths will be generated via the assignment of multiple AMACs by exploration probes. In this way, alternative paths can be instantiated on-the-fly if any link, or node, fails, just by switching the AMAC in use.

To simplify the explanation, we will describe the behavior of Amaru in single-controller networks. The procedure starts when a node discovers a direct connection to an SDN controller. This node will claim itself as *Amaru Root (ARoot)* and send an *Amaru Frame (AFrame)* through all of its ports. This AFrame will explore the whole network and the result of this process is the assignment of at least one hierarchical label, or *AMAC*, per node traversed. This exploration procedure is implemented in a distributed manner (not as a centralized algorithm) and follows the principles of Breadth-First Search (BFS) [33], which explores one path between a couple of nodes (s, t) in a graph. Amaru extends BFS to discover multiple paths in a single search.

The procedure is implemented as follows: The first assigned AMAC represents the ARoot and contains only one level. The next level in the hierarchy (level-n AMAC) is identified as a *child* and it obtains the *father's* AMAC plus an additional ID. Hence, the more fields (i.e. levels) in the AMAC, the further the node is from the ARoot. For instance, Fig. 4 portrays an example of Amaru applied to a SDN network consisting of nine switches (s_i, where $i \geq 1$ and $i \leq 9$). As $s1$ is the ARoot, its AMAC is 1 and it propagates the next-level AMACs to its *children*, namely 1.7 towards $s2$ and 1.2 towards $s4$. Afterwards, $s2$ and $s4$ will propagate new AMACs to their neighbors as well, i.e. they will broadcast next-level AMACs through all their ports except the input port. As it is a distributed process, the sending switch decides the value of the next level (ID) of the hierarchy, which could be any, with the only condition that it should be unique and consistent (always the same) per child. Each AMAC obtained is saved by the switches and it is associated with the port receiving it.

To guarantee loop prevention, a node only forwards an AFrame if the AMAC contained in it is not a child of their own. Hence, the distribution of AMACs continues while switches receive new AMACs and ends when nodes receive AMACs deriving from them and, therefore, discard them. This is the key difference of Amaru in comparison with BFS, as BFS discovers a single path between nodes and stops the procedure when a node is visited for the second and later times, while Amaru discovers multiple paths because it

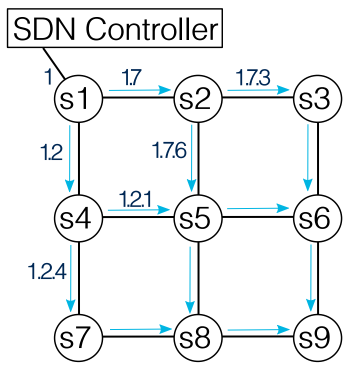


FIGURE 4. Initial propagation of AMACs from the ARoot.

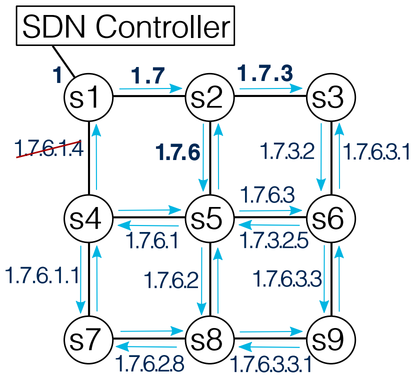


FIGURE 5. Propagation of label `1.7` in Amaru.

allows traversing the same node more than once, as far as the label being propagated is not a child of their own.

To illustrate how the Amaru procedure operates, Fig. 5 depicts the specific propagation of label 1.7 from $s1$ towards $s2$ in Amaru. Afterwards, $s2$ repeats the mechanism and propagates two child AMACs with values 1.7.3 and 1.7.6 to $s3$ and $s5$, respectively. This process continues until, eventually, a switch receives an AMAC that is a child of their own. For example, $s4$ sends 1.7.6.1.4 to $s1$, which is a child of 1 and, thus, it is discarded by $s1$. The same would happen in case that $s2$ would receive any AMAC starting with 1.7, for instance.

To understand how the assignment would look like after the propagation finishes, Fig. 6 represents a simplified version of the previous network, reducing it to just four nodes. For instance, $s4$ has two AMACs, 1.2 and 1.7.6.1, and they exactly represent the two possible paths towards the ARoot ($s4 - s1$ and $s4 - s5 - s2 - s1$), as well as their associated costs, one and three hops, respectively.

Finally, Fig. 7 exemplifies what would happen if two additional SDN switches are attached to the previous network. The direct consequence is that new paths towards the ARoot appear, namely 1.2.1.3.1.9 for $s2$, 1.7.3.2.8.1 for $s4$ and 1.7.3.2.8 for $s5$. For instance, now $s4$ can also reach $s1$ traversing $s4 - s5 - s6 - s3 - s2 - s1$ and that is the reason why $s4$ obtains a third AMAC from $s5$ with value 1.7.3.2.8.1, which represents that exact new route. Note that these AMAC values are just examples, not fixed by the topology.

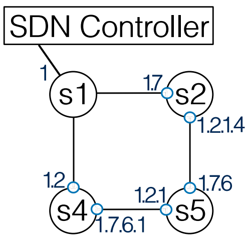


FIGURE 6. Assignment of AMACs in a four-node network.

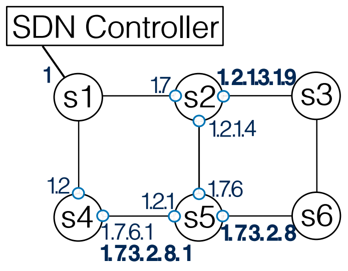


FIGURE 7. Extended assignment of AMACs in a six-node network.

The values obtained by $s3$ and $s6$ are omitted for simplicity, but we can assume that one of the AMACs assigned at $s6$ would 1.7.3.2, for instance.

C. FORWARDING LOGIC

As already introduced in the previous section, each AMAC represents one possible route towards the ARoot, i.e. towards the SDN controller. As each field of an AMAC is associated with a switch port, AMACs indirectly convey the list of switch ports (i.e. next hops) to be traversed towards destination. Therefore, when an SDN switch wants to send traffic to the controller, it just needs to check the available AMACs, pick one of them and use it as its own MAC instead of the manufacturer's. Afterwards, the forwarding logic follows the principles of source routing [34], that is, the traffic is sent through the port whose associated AMAC is related with the destination AMAC of the packets.

For example, in the case of Fig. 7, if $s5$ chose 1.7.6 to forward the traffic, the first step would be sending the traffic towards $s2$ indicating 1 as destination and 1.7.6 as source of the frames. Subsequently, $s2$ receives this traffic and sends it through the port associated to 1.7. Eventually these frames arrive at $s1$, which simply forwards it to the SDN controller.

In the opposite direction, i.e. controller-to-switch communication, the only difference is that the SDN controller needs to be aware of –at least– one of the available AMACs of a node to send the traffic. These can be easily resolved via the ARP/NDP procedure previous to any communication: instead of providing the real MAC address, SDN switches will mask it with any of the available AMACs, which will be used afterwards by the controller for future communications. For instance, $s5$ could provide 1.2.1 in Fig. 7 and, accordingly, all frames from the SDN controller would follow the path $s1 - s4 - s5$. As a consequence, paths are not mandatorily bidirectional.

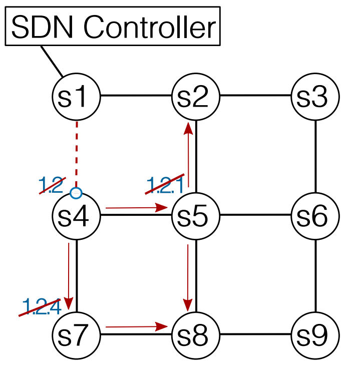


FIGURE 8. **Reconfiguration and deletion of AMACs after link failure.**

It is important to note that source routing does not introduce any type of packet overhead in Amaru, as the frame size remains exactly the same by simply substituting the real MAC by one of the associated AMACs, independently of the hops to be traversed. Thus, the only difference is table look-up, which is now performed checking if the prefix of the AMAC is contained in the node, instead of matching a whole fixed MAC.

D. NETWORK RECONFIGURATION

Diverse events might affect address assignment after startup, such as links failing or new nodes joining the network. Amaru performs local reconfiguration based on up/down event follow-up. The simplest mechanism requires that nodes periodically exchange *Hello* messages with their neighbors, but other procedures might also be appropriate, like triggering the renewal of addresses in Amaru when receiving a `PORT_STATUS` message if OpenFlow is the specific SDN control protocol. Thanks to this event registration, Amaru nodes are aware of which AMACs should be created, deleted, or even just temporarily disabled.

1) LINK FAILURE RECONFIGURATION EVENT

As AMACs are assigned to specific ports of Amaru nodes, when a link goes down, the involved nodes just need to disable the associated AMACs of that port. Accordingly, these nodes might decide whether to disable the whole tree or wait for the link to recover. To dismantle the tree of the affected AMACs, the procedure requires sending a AFrame exclusively towards the leaf nodes. A flag set in the AFrame distinguishes the deletion from the assignment procedures.

For example, Fig. 8 illustrates the procedure after link $s1 - s4$ fails, where the affected AMAC is 1.2 in $s4$. This node could remove that AMAC from the list of possible paths and propagate its deletion towards all neighbors. Specifically, the figure shows how 1.2.1 would be deleted at $s5$ and 1.2.4 at $s7$. The procedure would continue until all nodes update their tables.

Once the link is up again, the affected node $s4$ will repeat the usual assignment process by sending an AFrame through the port just recovered, which will eventually reach all the previously affected nodes, hence recovering the initial state.

2) NODE FAILURE RECONFIGURATION EVENT

A node event produces the same effect as a link event, but repeated n times, where n is the number of links attached to the node.

E. MULTIPLE PATHS AND ENHANCED RESILIENCY

By definition, Amaru maximizes in-band network resiliency as it potentially explores all possible paths in the topology. According to literature, this number easily ranges in millions of possibilities in networks comprised of twelve or more nodes [35]. Such a huge amount of possible routes is not needed in any networking scenario, thus we decided to restrict the learning of paths (i.e. of AMACs) based on the following parameters:

- ***N***: indicates the maximum number of AMACs that a node is allowed to learn. For example, if $N = 3$, only three AMACs, and hence paths, will be saved in the node.
- ***L***: provides the number of fields of the prefix of an AMAC that should differ from the already learned to be accepted. For instance, if $L = 2$ and the node already has an AMAC with value 1.2.3.4 and a new one arrives with value 1.2.5.6, the node will not save it as it shares the first two digits with an already existing one; but if the value is 1.1.5.6 instead, it will be learned.

To understand how these parameters work, we should consider that each node will be receiving one AMAC at a time, based on latency from the root. Therefore, AMACs are assigned in this order and, for example, $N = 3$ would mean that only the three first AMACs received would be saved, discarding the rest and any other associated AFrame. Some other parameters could be defined as they are orthogonal to the procedure of Amaru. In the end, the main objective is to reduce the number of AMACs, and the number of updates in case of link failure, while providing enough alternative paths for enhanced resiliency.

In Amaru, any node might use any of these multiple paths towards the root at zero cost, as reconfiguration of the AMAC address assignment is straightforward and based on local information. In switch-to-controller communications, if an AMAC is suddenly not available due to link failure, nodes can use any of the alternative ones with no need of additional messages or configuration. In the case of controller-to-switch communications, the ARoot will only have one AMAC per destination node and the update is not immediate. Nevertheless, they can obtain the new AMAC after receiving any message from the destination node. For example, in OpenFlow-based networks, link failures are notified to the controller via a `PORT_STATUS` message, which could serve this purpose. Additionally, signaling is considerably reduced as AFrames are only sent when specific events occur, such as network initialization or link failure.

F. SCALABILITY AND SHORTEST-PATH METRIC

As an exploration protocol, Amaru reduces convergence time, number of exchanged messages and table entries for in-band control routing. Regarding convergence time, it is drastically reduced as it directly depends on the time that the probe frame requires to traverse the network, without additional messages for convergence, as the exploration is performed at once. Accordingly, the number of exchanged messages decreases as well. Furthermore, Amaru paths are stable at all times, even if the procedure has not finished. In the case of table entries, one AMAC (i.e. one table entry) represents one path to the root node. For example, if a node requires three routes (which is basically one main and two alternative ones in case of failure), it will simply install three table entries, independently of the network size.

Finally, Amaru can potentially work with two different metrics to perform shortest-path routing from the root: number of hops and latency. Although, by definition, an AMAC represents the number of hops (e.g. an AMAC with four levels traverses three hops until reaching the root), due to the nature of the exploration AFrame probe, nodes receive AMACs in a specific order. Thus, the first assigned AMACs represent the fastest paths from the root towards the node receiving it [36]. Thus, nodes in Amaru could save the fastest AMACs as an alternative metric to hop count, because fewer hops not necessarily represent minimum-latency paths. The election of one metric or another is up to the network manager, or could be even based on artificial intelligence algorithms. For instance, the path from the root to the node could be the fastest one, but not necessarily in the opposite direction, so this node could choose an alternative (non-bidirectional) path towards the root based on minimum hops instead. For simplicity, the remainder of the article is strictly based on the minimum-hop metric.

G. SCENARIOS WITH MULTIPLE ROOT NODES

Diverse ARoots can coexist in Amaru with no additional logic. If each root node chooses a unique label (e.g. three ARoots with AMACs 1, 5 and 7), their children will be clearly distinguished and will cause no conflict. Even if a label collision between ARoots occurs, the Amaru procedure can progress as usual, although potencial overlaps of AMACs might appear at some point in the network. In that case, only the fastest arriving AFrame will be forwarded, hence reducing the number of alternative path discovered, but maintaining robustness.

IV. IMPLEMENTATION

To evaluate the functionality and performance of Amaru, we have developed three implementations, increasing in complexity and closeness to reality. The first one is a Python-based simulator, which helped us to prove the assignment procedure and refine the protocol. With this initial implementation, we proved that Amaru potentially explores all possible paths between the root and any other node. After validating Amaru with the Python-based simulator, a second implementation with the OMNeT++ simulator confirmed the behavior of Amaru with packets. Finally, we also developed Amaru in an SDN software switch to examine its feasibility with real traffic and scenarios. In the three implementations, the value of the next AMAC is created by concatenating the output port to the father's AMAC. In this way, we ensure the identification is unique per child.

Researchers particularly interested in these implementations are encouraged to check the source code, available in GitHub [37].

A. PYTHON-BASED SIMULATOR

As a first approximation of the Amaru protocol, an ad-hoc simulator based on Python was developed to verify the functionality and correctness of Amaru. This simulator only models transmission and reception events (which represent a packet exchange), so latencies are not considered and, hence, paths are exclusively based on the hop count metric. As input, it takes diverse topologies generated by the Brite topology generator [38] and it evaluates each of them by placing the SDN controller on every possible node. For every node in each topology, it obtains the number of exchanged packets and learned AMACs according to the Amaru protocol.

This simulator is based on events that represent a packet exchange. The initial event starts on the root node, i.e. where the controller is attached. This event creates as many subsequent events as neighbors of the node, which are equivalent to a packet transmission and handled applying the logic of Amaru. First, the receiving node checks if it can accept new AMACs (the available space is limited by the N parameter) and, if so, it checks if the AMAC is valid according to the L parameter. If any of the two checks obtains a negative result, the node rejects the proposed AMAC address. Otherwise, the node stores the new AMAC and propagates its children to all its neighbors.

B. OMNET++ SIMULATOR

A simulation model for Amaru was also developed in the OMNeT++ 5.3 framework. The nodes defined in the simulator implement an SDN switch and maintain an array with a list of AMAC address per port. In comparison with the previous simulator, the purpose of this implementation was three-fold: (1) to introduce latencies in the links and measure how the assignment is affected, (2) to assess the communication of the SDN switches with the controller, and (3) to evaluate the resiliency of Amaru upon one or more link failures.

In this simulator, upon startup, the SDN controller sends an initial message to the node connected to it, containing a level-1 AMAC. This node assumes the ARoot role and floods an AFrame through all its ports (except the incoming port) with level-2 AMACs, where the second level matches with the output port, as stated in the description of Amaru. Afterwards, the AFrame is only broadcast if the AMAC address is loop free and the learning condition results positive according to the previously described parameters, N and L.

Communication between the controller and switches is performed by following the next hop based on the AMACs contained in exchanged messages and according to the stored information in the AMAC address list. A node considers itself

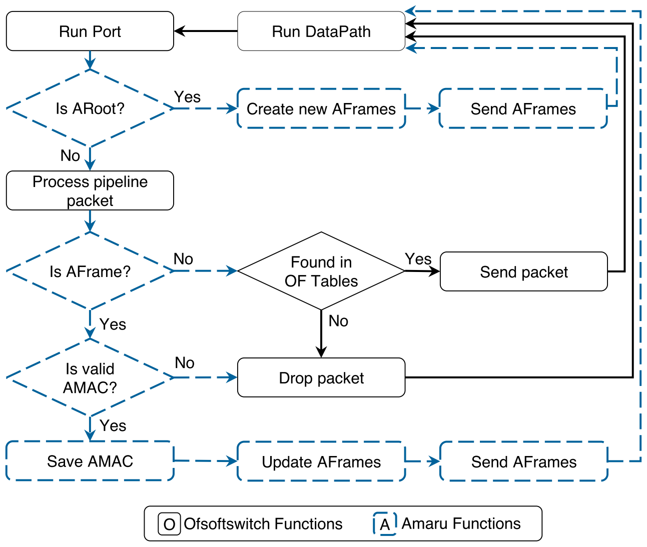


FIGURE 9. Flowchart of the Ofsoftswitch13 implementation.

as the destination of a control packet if the next hop matches one of the AMAC addresses in its list.

In addition, the simulator provides the *Link Failure Notification (LFN)* message to manage network dynamic events, as explained in section III-D. When a LFN message is received, the AMAC dismantle procedure is initiated through the flood of an AFrame with a field that indicates that the attached AMAC should be discarded instead of learned.

C. OFSOFTSWITCH13 SOFTWARE SWITCH IMPLEMENTATION

Finally, we developed the address assignment of Amaru in the Ofsoftswitch13 software switch [39]. We selected this software switch as it is easy to prototype and implement protocols on it (in comparison to Open vSwitch [40], for instance), while still performing reasonably well for real deployments [41]. In this case, the most significant part of the implementation was testing real-traffic scenarios and designing the AFrame.

Regarding the implementation of the switch, we modified the work-flow of the software switch to support the Amaru protocol (see Fig. 9, which distinguishes the original functions of Ofsoftswitch13 from the ones added for Amaru). The modification starts when the switch detects that it is directly connected to the SDN controller and it identifies itself as ARoot (see "*Is ARoot?*"). Then, it starts the AMAC address learning procedure. Afterwards, each received frame is checked (see "*Is Amaru Frame?*"). If the switch detects an AFrame, it must verify the correctness of the incoming AMAC according to the parameters and logic aforementioned, and propagate it if so or discard it otherwise. The process continues until all AMACs are propagated.

Finally, the control frame size of Amaru is illustrated in Fig. 10, in comparison with the ones of the Rapid Spanning Tree Protocol (RSTP) [42] and the OSPF protocol [21] (two of the best-known protocols for bridging and routing, respectively), as a reference. Amaru requires the smallest control frame of them all. Once the paths are created, in-band control is performed via standard Ethernet packets and it is important to highlight that Amaru does not introduce any overhead in them. As a result, the amount of bits available to represent labels is limited to 46 bits, as stated in Section III-A. The implementation of Amaru will use one byte per level, hence representing up to 6 levels of hierarchy (as most nodes should be within a few hops to the root to avoid high latencies), which are enough to provide good results, as illustrated in Section V. In any case, the network manager could configure the amount of bits to use, as this will not affect the principles of Amaru.

V. EVALUATION AND DISCUSSION

A. TESTBED

Our hardware infrastructure consists of 7 computers powered by Intel(R) Core(TM) i7 processors with 24 GB of RAM, all of which are interconnected via a GbE Netgear GS116 switch, for emulation and simulation purposes. To validate both the Python and OMNeT++ simulation models, we have also evaluated in the same conditions the Amaru software switch implementation based on *Ofsoftswitch13*. This latter evaluation was performed using the *Mininet* [43] emulation platform, which makes it possible to assess our protocol using a real Linux TCP/IP stack.

B. EXPERIMENTAL SETUP

The evaluation studies the three core features of Amaru: automatic bootstrapping, scalability and resiliency. For this reason, three main tests were scheduled to evaluate them: convergence time, packet consumption and throughput during failure events, respectively. Additionally, an initial test to validate the tools being leveraged in the tests was also performed. Table 1 summarizes all of these tests, which we further describe below.

First, we validate the developed tools for the three feature tests, which basically consisted in checking that the implementation of Amaru in three different platforms (from more abstract to more realistic: Python simulator, OMNeT++ simulator and Ofsoftswitch13 switch) were consistent. The reason to use these three platforms is because the Python simulator helped us in prototyping Amaru, while the OMNeT++ simulator had other similar protocols implemented to be compared to, and finally Ofsoftswitch13 because it is a software switch that could be used for in-band communication in real SDN deployments.

For the evaluation of the core features of Amaru, the best option would be to compare Amaru with the current state of the art. However, no current work accomplishes the same features than Amaru and their source codes were not available in all cases. For these reasons, we picked two of the closest related works: ResilientFlow [6] and Asadujjaman *et al.* [27] for comparison. In particular, firstly, we compare Amaru with RSTP and OSPF for automatic bootstrapping and scalability, because ResilientFlow is based

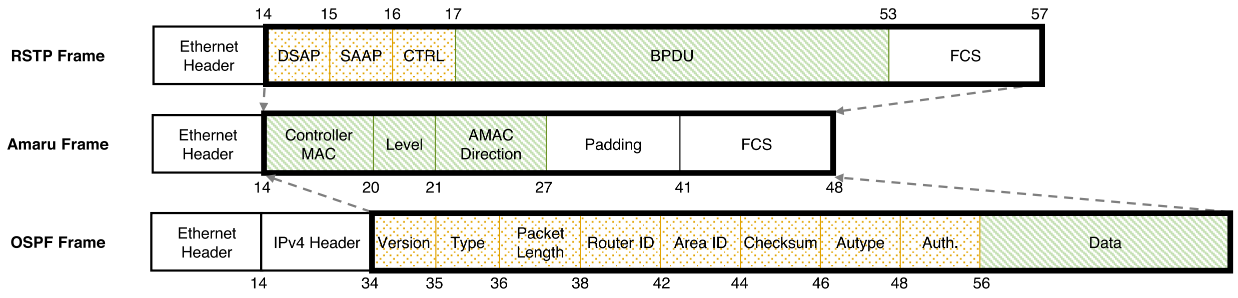


FIGURE 10. Comparison of Amaru, RSTP and OSPF control frames.

TABLE 1. Summary of the evaluation performed for Amaru.

Proof	Protocols	Platforms	Objective	Measure unit	Reason
Tools validation	Amaru	Python OMNeT++ ofsoftswitch13	To validate the Amaru protocol developed in various platforms/tools	Number of packets	To properly model the protocol and avoid design errors
Convergence time	Amaru RSTP	OMNeT++	To measure the time that each protocol needs to obtain the paths	Time	To prove Amaru is faster than standard alternatives
Packet consumption	Amaru RSTP OSPF	OMNeT++	To measure the number of packets that each protocol needs to obtain the paths	Number of packets	To prove Amaru optimizes network resource consumption
Recovery	Amaru Asadujjaman *et al*	OMNeT++ ofsoftswitch13	To measure the time that each protocol spends in restoring from a path fail	Time	To compare Amaru with similar SDN in-band protocols

on OSPF and also because they are standard protocols that could be leveraged for in-band control in SDN. Secondly, we evaluate Amaru against Asadujjaman *et al.* for resiliency. The three comparison tests were performed in OMNeT++ as these protocols were implemented using this simulator. Additionally, we also used the SDN switch Ofsoftswitch13 for the recovery test to prove that Amaru is also feasible in real scenarios.

As for the network scenarios used in the tests, we deployed diverse topologies to evaluate Amaru; specifically three types, from a symmetric graph to a real network deployment. Firstly, a synthetic mesh-like topology comprised of 10 nodes, as depicted in Fig. 11, served as initial proof-of-concept for Amaru, as it provides multiple paths that should be learned by the protocol.

Secondly, we employed the Barabasi and Bonabeau [44] and Waxman [45] topologies from the Brite topology generator [38]. Although these types of networks are also synthetic, they claim to illustrate some of the most representative topologies of the Internet. The idea behind the use of these topologies was to evaluate the packet consumption and convergence time of Amaru. Thus, the experiments were conducted configuring different average number of links per node (3, 5, 7) and different network sizes, from 10 to 60 nodes using 10-node steps. Node placement follows a heavy-tailed model, which emulates the world population distributions and the network size is increased incrementally, adding new nodes to the already existing ones and connecting them, which is more realistic than populating the nodes randomly in the scenario. The link bandwidth is constant and the Waxman parameters to the default values $\alpha = 0.15$ and $\beta = 0.2$.

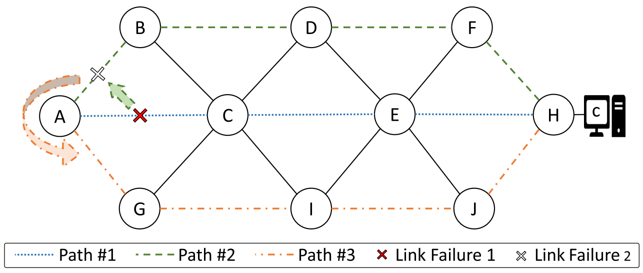


FIGURE 11. Synthetic topology.

Furthermore, the minimum number of runs per experiment was set up to 10 to calculate standard deviations. Table 2 presents a summary of these deployments. No background traffic was included in these experiments as the idea was to test how Amaru performed in a network deployment from scratch.

Finally, we also tested a real network topology adapted from Rogers fiber backbone network [46], as shown in Fig. 1 and Fig. 16. This topology encompasses 19 nodes connected by 24 links spanning approximately 4,920 km, with a total fiber length of approximately 11,183 km [27].

C. RESULTS

In this section, we analyze the results of the four types of tests previously described.

1) DEVELOPED TOOLS VALIDATION

The first step is to compare the different implementations detailed in section IV. The main objective is to ensure the expected behavior of Amaru and to validate the

TABLE 2. Experimental setup for the Brite topologies.

Type of network topologies	Barabasi [44] & Waxman [45]
Average links per node (*degree*)	3, 5, 7
Number of nodes per topology	10, 20, 30, 40, 50, 60
Node placement	Heavy-tailed
Growth type	Incremental
Bandwidth channel distribution	Constant (equal for all links)
Waxman topology parameters	$\alpha = 0.15$ & $\beta = 0.2$
Minimum number of runs	10

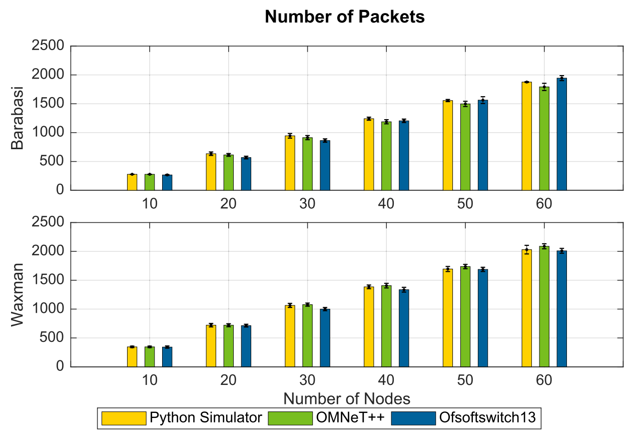


FIGURE 12. Comparison of the average number of packets exchanged for the three implementations of Amaru.

correctness of the models in the Python and OMNeT++ simulators, particularly in comparison with the behavior of the Ofsoftswitch13 implementation.

On the one hand, we should guarantee that all implementations learn a similar amount of AMACs, as defined by the parameters N and L. In fact, we proved it, although we found out that some of the values of these AMACs could differ due to the different modeling of the network delays in each platform. This result is expected since our Python tool only models the address propagation mechanism, but it does not consider network delays. OMNeT++ and Ofsoftswitch13 also differ for two reasons: (1) OMNeT++ cannot exactly model the delay of real platforms and (2) the order to flood packets in each node is not the same in both platforms due to their inner internal behavior.

On the other hand, although slight differences exist in the implementations, the exploration mechanism follows the same exact procedure and the expected number of packets should be the same. Therefore, apart from comparing the number of assigned AMACs, we also measured the average number of exchanged packets in all platforms. Figure 12 shows how this number is very close among all the three developed implementations (a maximum difference of around 5%), as expected.

As a result, we can conclude that our simulation models are good enough with respect to a real implementation of Amaru (the one with Ofsoftswitch13), and hence we can use these tools to perform more exhaustive experiments as follows.

2) CONVERGENCE TIME

The first core feature of Amaru to be examined is the setup convergence time. We can define the convergence time as the time that the Amaru protocol requires to obtain the number of desired paths according to the parameters N and L. This convergence time mainly depends on the size of the network and the average node degree. Although, control and data plane messages share network links, data plane traffic will not affect the convergence time of Amaru because no packet will be usually produced until the setup is finished.

To perform the evaluation, we compare Amaru against RSTP. The reason for this comparison is that RSTP is a simple protocol that provides a spanning tree from a root node, which could be also used to establish an in-band tree to achieve communication with a controller (though it does not provide multiple paths). RSTP exchanges periodic BPDU messages based on a configurable timer, which can be set up in OMNeT++ to a minimum of 1 second. This parameter will not affect the convergence time, as part of the protocol procedure is event-based after the first BPDU messages are received and, because of this, the convergence time is smaller than 1 second. Other routing protocols, such as OSPF, are not examined for this specific metric because they are not pure plug&play approaches, as they require an initial (and manual) configuration. Furthermore, the convergence time for OSPF was considerably higher than both Amaru and RSTP, even when the timers were set to their lowest values (without jeopardizing the routing stability).

Figure 13 presents best-case and worst-case results for Amaru in comparison with RSTP, using the OMNeT++ simulator for both protocols. The experiments were conducted in both Barabasi and Waxman topologies for a different number of nodes and different degrees of connectivity. As convergence time in Amaru also depends on the predefined parameters, we consider the best-case scenario to have $N = 2$ and $L = 3$ (in yellow), and the worst-case one with $N = 8$ and $L = 4$ (in green). The best case is $N = 2$ as it is one with the lowest amount of exchanged messages, and the worst case is $N = 8$, as we believe a redundancy of 8 paths is good enough for the resiliency of the network. The parameter L does not affect so much the convergence time, and we found out that a value around 3 and 4 was reasonable for path diversity. Furthermore, we include the results of RSTP (in blue).

For any set of results, the convergence time increases with the number of nodes in the network. On the contrary, an increase in the degree of the network nodes does not necessarily produce the same behavior. This fact can be initially counterintuitive, but the reason is that if we increase the degree of the network nodes, we are not changing the size of the topology, but only the number of existing connections. Thus, the number of existing paths also increases but it might not inexorably lead to a higher convergence time. Actually, it depends on how the nodes are connected.

We executed 10 runs with different topologies with the same characteristics to later calculate the confidence intervals, as shown in Fig. 13. Moreover, it is important to

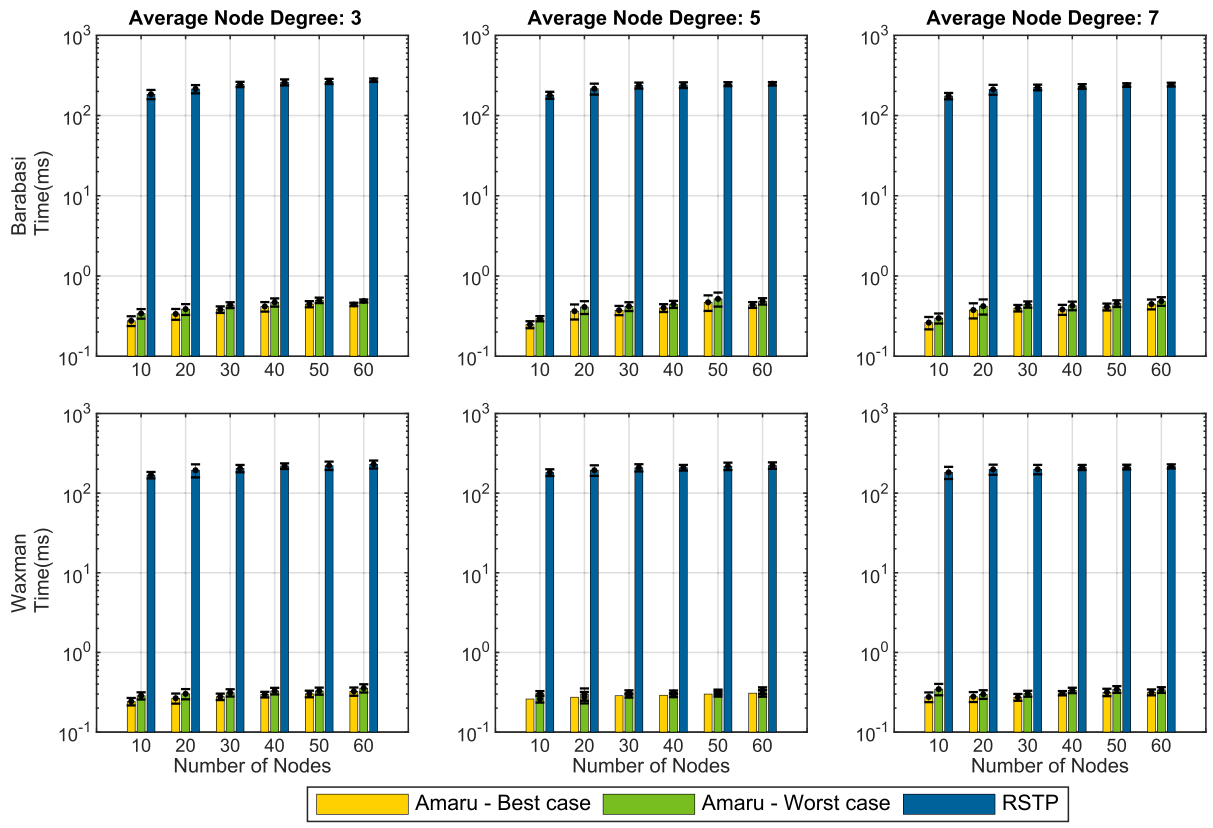


FIGURE 13. Comparison of convergence time in Amaru and RSTP.

highlight how Amaru clearly outperforms RSTP, as it converges 500 times faster than RSTP, in average. The small convergence time of Amaru is due to its exploration approach, which allows to exchange packets quickly without using over-dimensioned timers and/or repeated message exchanges among neighbors, both usually common in link-state protocols. Furthermore, Amaru provides N different paths to each node while RSTP only provides one.

3) PACKET CONSUMPTION

Another feature to consider in Amaru is its scalability. Thus, we analyzed the total number of packets required to complete a full exploration and address configuration procedure in Amaru. For this evaluation, we leveraged the OMNeT++ simulator and compared Amaru with the OMNeT++ implementations of RSTP and OSPF.

Figure 14 depicts the comparison of Amaru with respect to RSTP, as in the case of convergence time. Once again, experiments with Barabasi and Waxman topologies were performed with values of N equal to 2 and 8, and values of L equal to 3 and 4. Furthermore, different degree of nodes have been considered, namely 3 (yellow line), 5 (green-dashed line) and 7 (blue-dotted line). RSTP values are represented with markers to distinguish them from the Amaru ones. Amaru outperforms RSTP in most of the cases (in average, RSTP requires a number of packets 7,7% higher than Amaru) but, what is more important, Amaru always requires a smaller number of packets in larger topologies. Additionally, we observed there is no difference on using values of L equal to 3 or 4 in the case of Amaru. In general, as the size of the topology increases, the total number of exchanged packet consumption linearly increases, independently of the protocol.

In addition to the previous results, we also compared the required number of packets of Amaru with respect to the use of OSPF. The reason for this is that Amaru can obtain multiple paths per root node and RSTP not, while OSPF obtains the full knowledge of a topology in a distributed way to later calculate any desired k-shortest paths according to certain metric. In Fig. 15, we can observe how the number of packets in OSPF is one order of magnitude larger than Amaru (note the logarithmic scale), more specifically, OSPF requires around 30 times more packets than Amaru in average. The main reason is because OSPF requires recurring updates from neighbors every time that a new node is reached, while Amaru conveys all the information at once while traversing the network, so we could say that OSPF requires around 30 iterations to transfer the same amount of information than Amaru.

From these results, we can conclude that Amaru offers good scalability since its packet consumption is smaller than other alternatives, such as RSTP and OSPF, while offering multiple low-latency paths. The main reason for this could

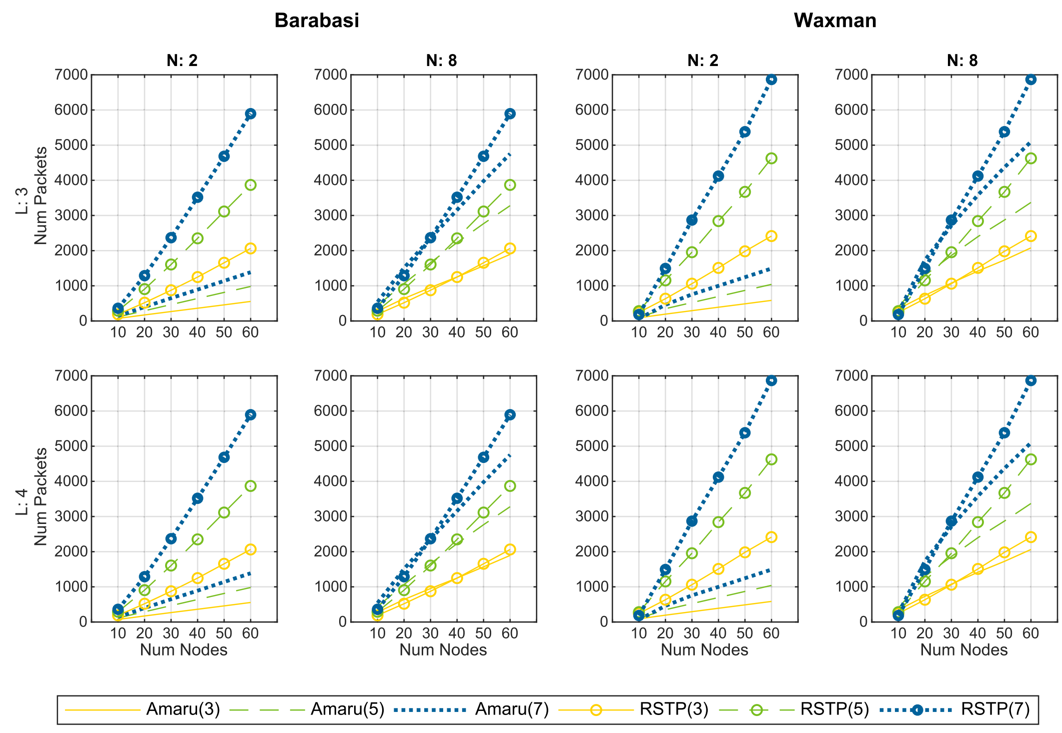


FIGURE 14. Comparison of average number of packets in Amaru and RSTP.

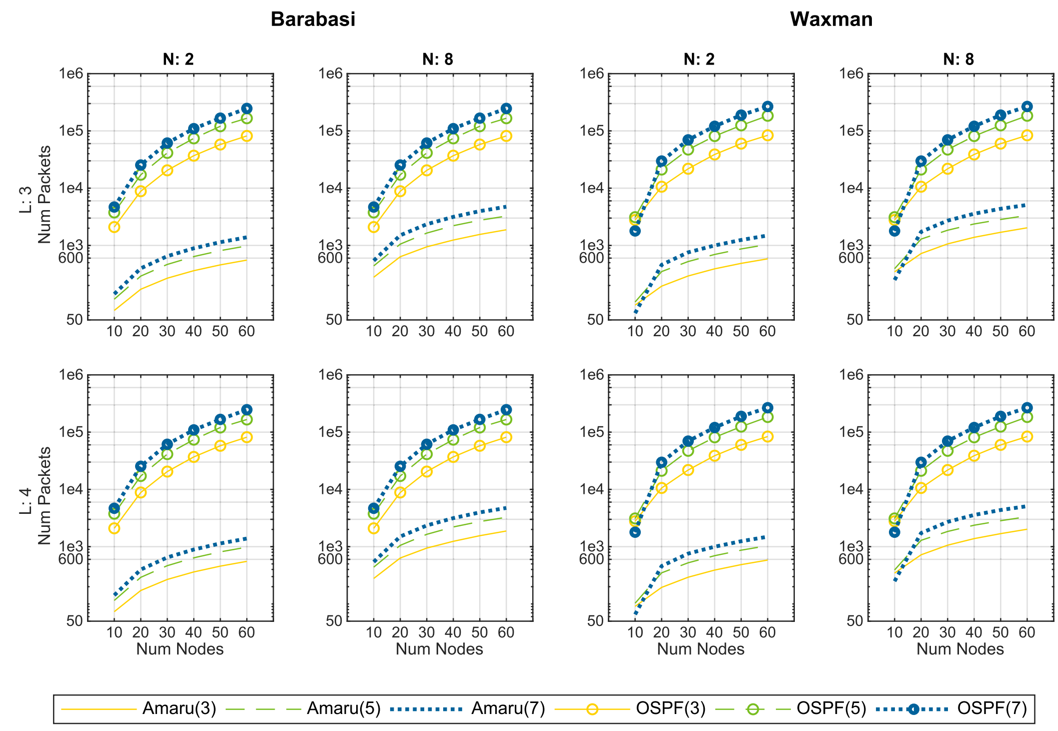


FIGURE 15. Comparison of average number of packets in Amaru and OSPF.

be the same as in the case of the convergence time: the exploration nature of Amaru, which allows carrying the information at once through the network without the need of multiple message exchanges among neighbors, as RSTP and OSPF require.

4) RECOVERY TIME

Finally, we performed different tests to evaluate the recovery time of Amaru, by studying it from two perspectives: switch-to-controller and controller-to-switch communications. We leverage the Ofsoftswitch13 implementation to measure the

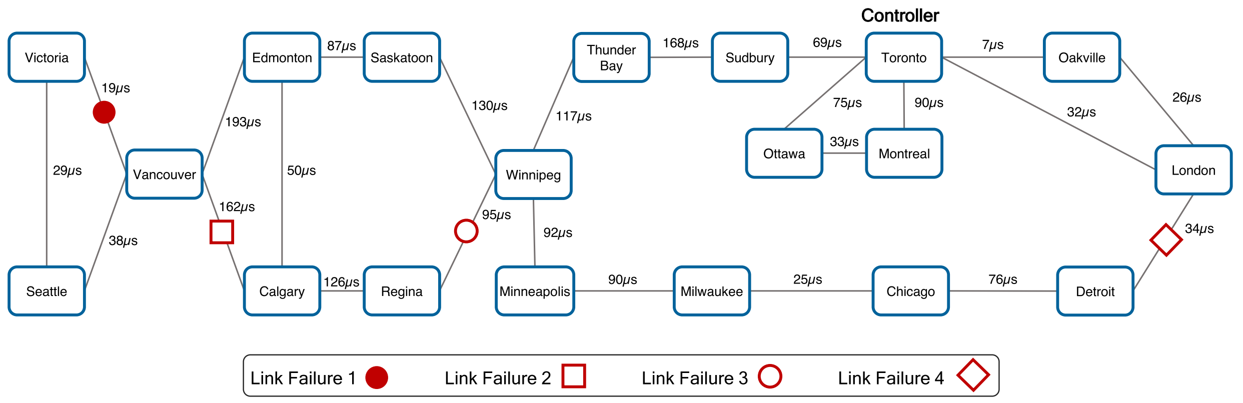
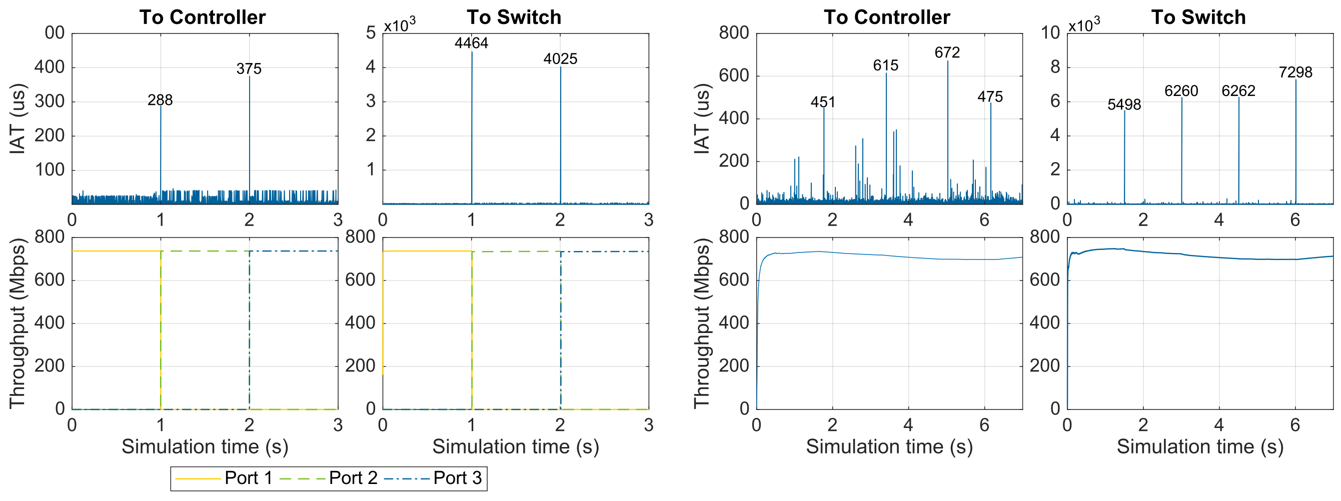


FIGURE 16. Rogers topology [46] and evaluated link failures.



(a) Recovery time on synthetic network (Fig. 11)

(b) Recovery time on Rogers network (Fig. 16)

FIGURE 17. Recovery times of Amaru in the Ofsoftswitch13 implementation.

throughput of in-band traffic, as well as the inter-arrival time (IAT) between received in-band packets. The IAT will show an increase in its value after a failure due to the recovery time required to overcome it. In this case, we leveraged two well-known topologies as link failures must be scheduled to later on measure the recovery time; the first topology is symmetric to measure how fast similar paths are selected, while the second one is a real scenario: the Rogers topology [46].

The first experiment uses the topology in Fig. 11, where the controller is connected to switch H and the link speed is 1 Gbps. The test consists of evaluating the throughput from A to H, where the ARoot is connected. Node A has saved three routes according to the Amaru procedure: (1) A-C-E-H, (2) A-B-D-F-H and (3) A-G-I-J-H, associated to ports 1, 2 and 3 of the node, respectively. The first path established by Amaru for in-band traffic is through port 1. The first link failure happens at time 1 s and the in-band traffic is rerouted through the path through port 2. Finally, a second failure occurs at time 2 s and the traffic is redirected once again, this time through port 3. The throughput is measured based on a signaling traffic of 750 Mbps, which is high enough to evaluate Inter-Arrival Times (IATs) with high granularity, but without exceeding the forwarding capabilities of the switch.

Figure 17 shows what happens during these failures in switch-to-controller (left side) and in controller-to-switch communication (right side). The upper figures illustrate the IAT from consecutive received in-band packets in microseconds. Both of them present an increase in the IAT exactly when failures take place. As expected, the IAT in the controller-to-switch traffic is bigger than the IAT in the switch-to-controller traffic because of the extra time required to notify the controller of the failure occurrence. The recovery time is below 400 μs in the switch-to-controller direction and below 5 ms in the controller-to switch direction, hence smaller than the 50 ms to guarantee a successful recovery [5]. The differences in time of each direction are caused by the fact that only switches save all path towards the controller, so that is why the switch-to-controller recovery is almost immediate (μs), but the controller-to-switch

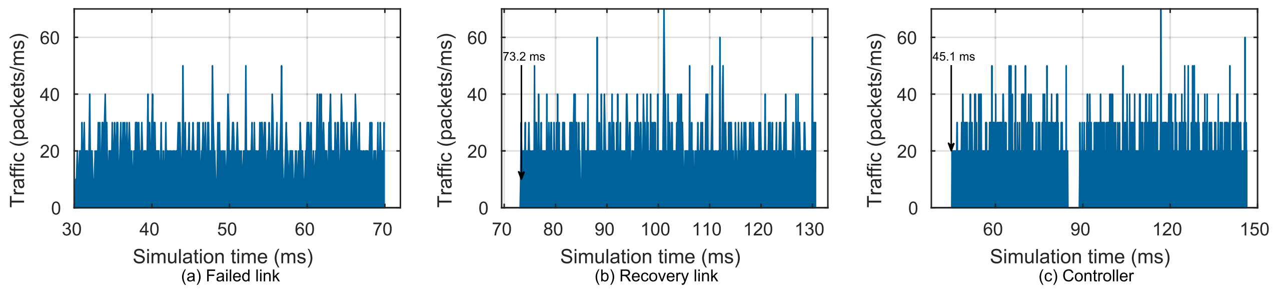


FIGURE 18. Recovery times in Amaru from three points of view, after a failure in Victoria-Vancouver link at time 70 ms.

direction requires the reception of traffic from the switch to be updated. Furthermore, the lower figures clearly depict how the throughput changes from one port to other port in switch A after a failure and how throughput is almost unaffected in the switch-to-controller direction and slightly affected (negligible in the graph) in the controller-to-switch direction.

The second experiment follows the same methodology and setup as the first one, but using the Rogers topology [46] instead (shown in Fig. 16). Additionally, the number of link failures is increased to four to check failures at different distances from the SDN controller. The in-band traffic is established between the Toronto and Victoria nodes. The first path goes from Victoria to Vancouver and that link then fails, redirecting the traffic through Seattle. Three more link failures are illustrated in the figure, all of them affecting the current route at each time. The obtained results are very similar to the previous ones, even considering that this time the topology is based on a real network deployment. Again, the switch-to-controller traffic has a smaller recovery time (always below 700 μs) than the controller-to-switch traffic, which always has a recovery time below 7.3 ms. This latter value is slightly bigger than the previous one (5 ms) because the network is bigger and there are more propagation latencies, but the value is still under the maximum 50 ms. Moreover, the throughput is again almost unaffected by the link failures.

It is important to note that the parameters L and N increase the scalability of Amaru, as it potentially explores all paths, but they also constrain its resiliency. Though improbable, a corner case would be if all learned paths used at least one of the failed links. In this case, none of the learned AMACs could work as alternative routes and the Amaru discovery mechanism should be relaunched, either locally or, in the worst case, globally. In these cases, the recovery time will be increased up to a time equal to the Amaru convergence time. The convergence time for the synthetic 10-node topology is around 125 μs and up to 1.9 ms in the Rogers topology, considering this traffic has priority above other types. Thus, we can conclude that, even in the worst case, the recovery time will continue presenting excellent small values, under the required 50 ms according to literature [5].

Finally, to have a comparison with the closest approach in the state of the art, we repeated the same simulation scenario than in the work from Asadujjaman *et al.* [27] to measure convergence time after link failure. The results are illustrated in Fig. 18 and show that, after the Victoria-Vancouver link failure, the recovery time is 3.2 ms (from the link failure at time 70 ms, to recovery at 73.2 ms), more than 10% smaller than the 3.8 ms obtained in the closest related work [27]. This is because Amaru already knows the end-to-end backup path when there is a failure, whereas the compared work has to go through hop by hop signaling upon failure.

D. DISCUSSION

As proved by the evaluation results, the advantages of Amaru are as follows:

1) **Fast automatic bootstrapping:** Amaru paths are populated automatically and its convergence time is smaller than standard alternatives leveraged for in-band control, such as RSTP and OSPF, as proved in Section V-C.2.
2) **Good scalability:** Amaru provides multiple paths towards the SDN controller, following a protection scheme. Moreover, it only requires a single entry per path, independently of the network size, which drastically reduces the forwarding table size (e.g. 10 different paths towards the controller are represented by just 10 entries). Additionally, the amount of exchanged messages to build these forwarding tables is smaller than RSTP (even considering than RSTP generates only one path) and much smaller than OSPF, as illustrated in Section V-C.3.
3) **Enhanced reliability:** Amaru implements a protection scheme (that is 1 primary path + $N - 1$ backup ones) that enhances the network reliability are rapidly available even after multiple failures occur. Recovery times are below the 50 ms and high throughput is almost unaffected, as shown in Section V-C.4.

On the other hand, there are some aspects in Amaru that require further analysis and could be consider its main drawbacks:

1) **New protocol:** As a new protocol, it requires some agent in the switches to support it for implementation.

However, we have developed it in Ofsoftswitch13 and proved that it is feasible in real scenarios.

2) **AMAC design:** Amaru masks real MAC addresses with AMACs, which have an available space of 46 bits, as previously explained. By default, this field is leveraged as 6 levels of around 8 bits, which represents paths of 6 hops with around $2^8 = 256$ neighbors per switch, but other configurations are also feasible (such as 12 levels of around 4 bits) and it is up to the network administrator to decide the most suitable setup.
3) **Selection of parameters:** Amaru discovers all paths among all nodes and the SDN controller, which is not desirable for real networks; thus, the parameters N and L should be defined prior to deployment. By default, Amaru is set to $N = 8$ and $L = 4$, which provides excellent performance for the evaluated topologies, but further scenarios should be analyzed.

VI. CONCLUSION

The Amaru protocol provides automatic bootstrapping, enhanced resiliency and good scalability for in-band control in SDN. To the best of our knowledge, only one work is close to accomplish these three features and Amaru outperforms it. As an exploration protocol, it follows a completely different approach to the current literature, which is mostly based on link-state knowledge. This exploration strategy drastically reduces the convergence time and the number of exchanged packets, while guaranteeing the network paths are stable at any time.

Amaru has been tested via simulation (based on Python and OMNeT++) and emulation (Ofsoftswitch13 and Mininet), and results are promising. Furthermore, Amaru potentially discovers all paths between any couple of network nodes, but for good scalability we currently limit the learning by setting two parameters (N and L), which is enough to portray an excellent behavior in carrier networks. Nevertheless, we aim to further refine this learning by analyzing their optimal values and implications (number of messages and table entries), and by applying other types of parameters and other techniques (such as machine learning), hence saving only the best routes from the whole set to guarantee resiliency at any failure event.

In order to deploy Amaru in real SDN scenarios, the only requirement is to have some SDN software switch or a similar agent installed in the switches, as proved by our proof-of-concept implementation with Ofsoftswitch13. The main advantage of Amaru is that currently no in-band control standard exists, so new SDN switches envisioning this feature could consider Amaru as part of their designs. Additionally, SDN switches based on P4 could also be programmed to support it.

Finally, as future work, we will analyze the implications of deploying Amaru in large-scale networks with high connectivity, as well as in mobile and dynamic networks, specifically with more than one operator involved. Additionally, the combination of in-band and out-of-band resiliency with multiple controllers could be also examined [47]. We will also study the inclusion of the protocol in the ONF community, as we trust Amaru might contribute as a relevant step forward in regard to the in-band control of SDN, where there is no defined standard yet. Additionally, some other envisioned research topics are leveraging Amaru for multipath routing, either as a distributed protocol or centralized mechanism (e.g. to calculate shortest paths in the SDN controller), and we will also analyze its applications in graph theory, in comparison with BFS and similar algorithms.

ACKNOWLEDGMENT

The authors thank David Carrascal for developing a topology parser to test the same scenarios with the different simulators of Amaru.

REFERENCES

[1] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[2] A. Jalili, H. Nazari, S. Namvarasl, and M. Keshtgari, "A comprehensive analysis on control plane deployment in SDN: In-band versus out-of-band solutions," in *Proc. 4th IEEE Int. Conf. Knowl.-Based Eng. Innov. (KBEI)*, Tehran, Iran, Dec. 2017, pp. 1025–1031.

[3] R. Amin, M. Reisslein, and N. Shah, "Hybrid SDN networks: A survey of existing approaches," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 4, pp. 3259–3306, 4th Quart., 2018.

[4] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Fast failure recovery for in-band OpenFlow networks," in *Proc. 9th Int. Conf. Design Reliable Commun. Netw. (DRCN)*, Mar. 2013, pp. 52–59.

[5] B. Niven-Jenkins, D. Brungard, M. Betts, N. Sprecher, and S. Ueno, *Requirements of an MPLS Transport Profile*, document RFC 5654, 2009, pp. 1–31. doi: 10.17487/RFC5654.

[6] T. Watanabe, T. Omizo, T. Akiyama, and K. Iida, "ResilientFlow: Deployments of distributed control channel maintenance modules to recover SDN from unexpected failures," in *Proc. 11th Int. Conf. Design Reliable Commun. Netw. (DRCN)*, Mar. 2015, pp. 211–218.

[7] O. I. Bentstuen and J. Flathagen, "On bootstrapping in-band control channels in software defined networks," in *Proc. IEEE Int. Conf. Commun. Workshops (ICC Workshops)*, May 2018, pp. 1–6.

[8] N. Beheshti and Y. Zhang, "Fast failover for control traffic in Software-defined Networks," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2012, pp. 2665–2670.

[9] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "In-band control, queuing, and failure recovery functionalities for OpenFlow," *IEEE Netw.*, vol. 30, no. 1, pp. 106–112, Jan. 2016.

[10] S. S. W. Lee, K.-Y. Li, K. Y. Chan, G.-H. Lai, and Y. C. Chung, "Software-based fast failure recovery for resilient OpenFlow networks," in *Proc. 7th Int. Workshop Reliable Netw. Design Model. (RNDM)*, Oct. 2015, pp. 194–200.

[11] D. Kotani and Y. Okabe, "Fast failure detection of OpenFlow channels," in *Proc. Asian Internet Eng. Conf. (AINTEC)*, 2015, pp. 32–39. [Online]. Available: http://doi.acm.org/10.1145/2837030.2837035

[12] S. Feng, Y. Wang, X. Zhong, J. Zong, X. Qiu, and S. Guo, "A ring-based single-link failure recovery approach in SDN data plane," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp. (NOMS)*, Apr. 2018, pp. 1–7.

[13] Y. Hu, W. Wendong, G. Xiangyang, C. H. Liu, X. Que, and S. Cheng, "Control traffic protection in software-defined networks," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2014, pp. 1878–1883.

[14] H. Huang, S. Guo, W. Liang, K. Li, B. Ye, and W. Zhuang, "Near-optimal routing protection for in-band software-defined heterogeneous networks," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 11, pp. 2918–2934, Nov. 2016.

[15] P. Goltsmann, M. Zitterbart, A. Hecker, and R. Bless, "Towards a resilient in-band SDN control channel," Univ. Tübingen, Tübingen, Germany, Tech. Rep., 2017. [Online]. Available: https://ub31.uni-tuebingen.de/xmlui/bitstream/handle/10900/78147/KuVS-FG-Netsoft17-7.pdf

[16] K. Chan, C. Chen, Y. Chen, Y. Tsai, S. S. W. Lee, and C. Wu, ''Fast failure recovery for in-band controlled multi-controller OpenFlow networks,'' in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Oct. 2018, pp. 396–401.

[17] S. González, A. D. la Oliva, C. J. Bernardos, and L. M. Contreras, ''Towards a resilient OpenFlow channel through MPTCP,'' in *Proc. IEEE Int. Symp. Broadband Multimedia Syst. Broadcast. (BMSB)*, Jun. 2018, pp. 1–5.

[18] A. Raza and S. Lee, ''Gate switch selection for in-band controlling in software defined networking,'' *IEEE Access*, vol. 7, pp. 5671–5681, 2019.

[19] S. S. Savas, M. Tornatore, F. Dikbiyik, A. Yayimli, C. U. Martel, and B. Mukherjee, ''RASCAR: Recovery-aware switch-controller assignment and routing in SDN,'' *IEEE Trans. Netw. Service Manage.*, vol. 15, no. 4, pp. 1222–1234, Nov. 2018.

[20] E. Sakic, N. Đerić, and W. Kellerer, ''MORPH: An adaptive framework for efficient and Byzantine fault-tolerant SDN control plane,'' *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2158–2174, Oct. 2018.

[21] J. Moy, *OSPF Version 2*, document RFC 2328, Apr. 1998. [Online]. Available: https://rfc-editor.org/rfc/rfc2328.txt

[22] Y.-L. Su, I.-C. Wang, Y.-T. Hsu, and C. H.-P. Wen, ''FASIC: A fast-recovery, adaptively spanning in-band control plane in software-defined network,'' in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2017, pp. 1–6.

[23] L. Schiff, S. Schmid, and M. Canini, ''Medieval: Towards a self-stabilizing, plug & play, in-band SDN control network,'' in *Proc. ACM SIGCOMM Symp. SDN Res. (SOSR)*, 2015, pp. 1–2. [Online]. Available: http://eprints.cs.univie.ac.at/5745/

[24] L. Schiff, S. Schmid, and M. Canini, ''Ground control to major faults: Towards a fault tolerant and adaptive SDN control network,'' in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshop (DSN-W)*, Jun. 2016, pp. 90–96.

[25] M. Canini, I. Salem, L. Schiff, E. M. Schiller, and S. Schmid, ''A self-organizing distributed and in-band SDN control plane,'' in *Proc. 37th IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Atlanta, GA, USA, Jun. 2017, pp. 2656–2657.

[26] M. Canini, I. Salem, L. Schiff, E. M. Schiller, and S. Schmid, ''Renaissance: A self-stabilizing distributed SDN control plane,'' in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2018, pp. 233–243.

[27] A. Asadujjaman, E. Rojas, M. S. Alam, and S. Majumdar, ''Fast control channel recovery for resilient in-band OpenFlow networks,'' in *Proc. 4th IEEE Conf. Netw. Softw. Workshops (NetSoft)*, Jun. 2018, pp. 19–27.

[28] R. Hark, A. Rizk, N. Richerzhagen, B. Richerzhagen, and R. Steinmetz, ''Isolated in-band communication for distributed SDN controllers,'' in *Proc. IFIP Netw. Conf. (IFIP Netw.) Workshops*, Jun. 2017, pp. 1–2.

[29] L. Ochoa-Aday, C. Cervelló-Pastor, and A. Fernández-Fernández, ''ETDP: Enhanced topology discovery protocol for software-defined networks,'' *IEEE Access*, vol. 7, pp. 23471–23487, 2019.

[30] Y. Yan, J. Bi, Y. Zhou, and C. Zhang, ''Gather: A way to optimize the routing process of in-band control network,'' in *Proc. SIGCOMM Posters Demos*, 2017, pp. 12–14. [Online]. Available: http://doi.acm.org/10.1145/3123878.3131969

[31] B. Görkemli, S. Tatlıcıoğlu, A. M. Tekalp, S. Civanlar, and E. Lokman, ''Dynamic control plane for SDN at scale,'' *IEEE J. Sel. Areas Commun.*, vol. 36, no. 12, pp. 2688–2701, Sep. 2018.

[32] E. Rojas, J. Alvarez-Horcajo, I. Martinez-Yelmo, J. M. Arco, and J. A. Carral, ''GA3: Scalable, distributed address assignment for dynamic data center networks,'' *Ann. Telecommun.*, vol. 72, no. 11, pp. 693–702, Dec. 2017. doi: 10.1007/s12243-017-0569-4.

[33] E. Arjomandi and D. G. Corneil, ''Parallel computations in graph theory,'' in *Proc. 16th Annu. Symp. Found. Comput. Sci. (SFCS)*, Oct. 1975, pp. 13–18.

[34] M. C. Hamner and G. R. Samsen, ''Source routing bridge implementation (LANs),'' *IEEE Netw.*, vol. 2, no. 1, pp. 33–36, Jan. 1988.

[35] B. Roberts and D. P. Kroese, ''Estimating the number of s-t paths in a graph,'' *J. Graph Algorithms Appl.*, vol. 11, no. 1, pp. 195–214, 2007.

[36] E. Rojas, G. Ibáñez, J. M. Gimenez-Guzman, J. A. Carral, A. Garcia-Martinez, I. Martinez-Yelmo, and J. M. Arco, ''All-path bridging: Path exploration protocols for data center and campus networks,'' *Comput. Netw.*, vol. 79, pp. 120–132, Mar. 2015.

[37] *Open Source Implementation of Amaru*. Accessed: Aug. 20, 2019. [Online]. Available: https://github.com/gistnetserv-uah/Amaru/

[38] A. Medina, A. Lakhina, I. Matta, and J. Byers, ''BRITE: Universal topology generation from a user's perspective,'' Dept. Comput. Sci., Boston Univ., Boston, MA, USA, Tech. Rep., 2001. [Online]. Available: https://www.cs.bu.edu/brite/publications/usermanual.pdf

[39] *CPqD: OpenFlow 1.3 Software Switch*. Accessed: Aug. 20, 2019. [Online]. Available: https://github.com/CPqD/ofsoftswitch13

[40] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, ''The design and implementation of open vswitch,'' in *Proc. 12th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Oakland, CA, USA, 2015, pp. 117–130.

[41] E. L. Fernandes, E. Rojas, J. Alvarez-Horcajo, Z. L. Kis, D. Sanvito, N. Bonelli, C. Cascone, and C. E. Rothenberg, ''The road to BOFUSS: The basic OpenFlow user-space software switch,'' 2019, *arXiv:1901.06699*. [Online]. Available: https://arxiv.org/abs/1901.06699

[42] *IEEE Standard for Local and Metropolitan Area Networks—Common Specification. Part 3: Media Access Control (MAC) Bridges—Amendment 2: Rapid Reconfiguration*, IEEE Standard 802.1w-2001, 2001.

[43] Mininet. *Mininet: An Instant Virtual Network on Your Laptop (or Other PC)*. Accessed: Aug. 20, 2019. [Online]. Available: http://mininet.org/

[44] A.-L. Barabási and E. Bonabeau, ''Scale-free networks,'' *Sci. Amer.*, vol. 288, no. 5, pp. 60–69, 2003.

[45] B. M. Waxman, ''Routing of multipoint connections,'' *IEEE J. Sel. Areas Commun.*, vol. 6, no. 9, pp. 1617–1622, Dec. 1988.

[46] *Rogers Fibre Backbone*. Accessed: Aug. 20, 2019. [Online]. Available: https://www.rogers.com/enterprise/wholesale#network

[47] T. Hu, P. Yi, Z. Guo, J. Lan, and Y. Hu, ''Dynamic slave controller assignment for enhancing control plane robustness in software-defined networks,'' *Future Gener. Comput. Syst.*, vol. 95, pp. 681–693, Jun. 2019.

DIEGO LOPEZ-PAJARES (M'16) received the master's degree in telecommunications engineering, in 2016. He has been a Researcher with the GIST-NETSERV Research Group, University of Alcala, since 2015, focusing on topics related to delay-tolerant and SDN networks, specifically low-latency routing solutions and the multipath problem. These topics comprise the basis of his Ph.D. In addition, he actively participates in several GIST-NETSERV research projects, such as TIGRE5-CM, TAPIR-CM, EsPECIE, and SIMPSONS.

JOAQUIN ALVAREZ-HORCAJO (M'17) received the master's degree in telecommunications engineering from the University of Alcala, in 2017. After having worked at Telefonica as a Test Engineer for COFRE and RIMA networks, he was awarded a Grant for University Professor Training (FPU) from the University of Alcala, where he is currently a Researcher. His research interests include software defined networks (SDN), Internet protocols, and new generation protocols. At present, he is especially interested in topics related to advanced switches and SDN networks. He has participated in various competitive projects funded through the Community of Madrid Plan, such as TIGRE5-CM and TAPIR-CM.

ELISA ROJAS received the Ph.D. degree in information and communication technologies engineering from the University of Alcala, Spain, in 2013. As a Postdoctoral Researcher, she was with IMDEA Networks and, later on, as a CTO of Telcaria Ideas S.L., and an SME dedicated to both research and development of virtualized network services. She has participated in diverse projects funded by the EC, such as FP7-NetIDE or H2020-SUPERFLUIDITY. She is currently a Professor with the University of Alcala. Her current research interests include SDN, NFV, high performance and scalable Ethernet, the IoT, and data center networks. She was involved in three different areas in relation with SDN: as a Researcher in several EU projects, as a Designer and Developer in an SDN project for a Network Operator, and as a Professor.

A. S. M. ASADUJJAMAN (M'17) received the M.Sc. degree in information and communication technology (ICT) from the Bangladesh University of Engineering and Technology, in 2017. He is currently a Core Network Lead Engineer with Banglalink Digital Communications Ltd., Dhaka. His research interests include computer network reliability and performance. He is currently interested in software defined network (SDN) reliability and performance.

ISAIAS MARTINEZ-YELMO received the Ph.D. degree in telematics from the Carlos III University of Madrid, Spain, in 2010. After working as a Postdoctoral Assistant at the Carlos III University of Madrid, he became and remains a Teaching Assistant with the Automatics Department, University of Alcala, Spain. His research interests include peer-to-peer networks, content distribution networks, vehicular networks, NGN, and Internet protocols. Nowadays, he is especially interested in advanced switching architectures and software defined networks (SDN). He has participated in various competitive research projects funded by the Madrid Regional Government (Medianet and TIGRE5-CM), National projects (CIVTRAFF), and European projects (CONTENT and CARMEN). His research papers have been published in high impact JCR indexed research journals, such as *Communications Magazine*, *Computer Communications*, and *Computer Networks*. He was a Technical Program Committee Member of IEEE ICON from 2011 to 2013. In addition, he has been a Reviewer for high quality conferences (i.e., IEEE INFOCOM) and scientific journals IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY, *Computer Communications*, and *ACM Transactions on Multimedia Computing*.