A DAG-Based Forwarding Paradigm for Large Scale Software Defined Networks

Stefano Avallone and Usman Ashraf

***Abstract*—The Software Defined Network (SDN) paradigm represents a major breakthrough in the networking field, due to its unprecedented capabilities in terms of flexibility and programmability. SDNs have been successfully deployed in data centers and small to medium enterprises. However, adopting the SDN paradigm in the context of wide area networks is more challenging, due to a number of factors including the higher probability that node and link failures occur and the unavailability of a dedicated control channel. In this paper, we present a DAG (Directed Acyclic Graph) based forwarding paradigm addressing the challenges that arise when the SDN concept is applied to large scale networks. Specifically, the proposed paradigm aims to limit the number of entries required on the SDN switches, to provide a fast local restoration of single node/link failures without the intervention of the SDN controller and to prevent the possibility of having inconsistent forwarding tables during updates. The proposed paradigm does not require any extension to the OpenFlow protocol and we show how it can be implemented by only using standard features. The DAG-based forwarding paradigm requires to compute a DAG between every pair of ingress-egress switches and to design an index-based hashing scheme to balance the load across the paths in the DAG while avoiding TCP reordering issues. In this paper, we present heuristic algorithms providing a solution to such problems and report the results of a simulation study conducted to assess the performance of the proposed forwarding paradigm.**

***Index Terms*—Software defined networks, traffic engineering, fast local restoration.**

I. Introduction

SOFTWARE defined networking has revolutionized the networking paradigm and paved the way for centralized decision making in networks. In sharp contrast to the decades old mechanisms of distributed routing, the software defined networking paradigm assumes a centralized approach in which the central controller assumes all the responsibilities of routing, synchronization, failure detection and recovery, resilience and even security. The nodes become merely forwarding devices with all the intelligence residing at the central controller. The separation of the control and data planes has opened up new possibilities of leveraging the power of computer networking but at the same time introduced new challenges. SDN networks offer complete control of the network and unprecedented traffic engineering possibilities. OpenFlow [1], which is the de facto standard for software defined networking, offers abstraction which hides the network complexity and allows operators to model the network as a single big switch without worrying about low-level details, resulting in simpler network management.

SDNs are typically employed in medium to large scale networks and in terms of scalability, perhaps the most applicable instance of SDNs manifests itself in the form of Software Defined Wide Area Networks (SD-WANs). While SDNs have evolved significantly during the recent years, Wide-Area Networks (WANs) pose bigger challenges to the design of software defined networks as compared to Data Center Networks (DCNs) which are the traditional deployment destination for software defined networking. The primary difference is that in DCNs, faults such as link and device failures are less frequent, are localized and easier to handle due to the underlying structure of DCNs. In DCNs, control networks are often deployed in parallel with the communication networks with dedicated links ensuring a significantly lesser probability of the data/control plane disconnect. Moreover, faults are localized and recovery is easier since DCNs are typically managed by a single entity. In contrast, faults in wide area SDNs are more frequent and often result in a disconnect between the control and data planes. The problem is exacerbated due to the fact that faults may span across several organization entities, making the detection and coordinated recovery more complicated. The coordination within a DCN is easier because of smaller communication delays, but becomes difficult in WANs with variable delays and packet losses resulting in difficulties in converging to a consistent network configuration. With the imminent exponential increase in the number of users and applications, ensuring scalability of SDN solutions is the cornerstone of research in this domain and there are several issues that must be addressed in order to ensure seamless scalability of Software Defined Wide Area Networks.

The first major research problem is scalability in terms of the required number of rule entries in the network. In order to support a broad range of applications, SDNs typically employ rules which are based on flows. The centralized controller applies sophisticated mathematical models based on its omni-knowledge about the network, and installs appropriate rules (which are significantly different from the

Manuscript received August 2, 2018; revised December 23, 2018 and July 8, 2019; accepted November 25, 2019. Date of publication November 28, 2019; date of current version March 11, 2020. The associate editor coordinating the review of this article and approving it for publication was H. De Meer. *(Corresponding author: Stefano Avallone.)*

S. Avallone is with the Department of Computer Engineering, DIETI, University of Napoli Federico II, 80125 Naples, Italy (e-mail: stavallo@unina.it).

U. Ashraf is with the Department of Communications and Networking, CCSIT, King Faisal University, Al Hofuf 31980, Saudi Arabia (e-mail: uashraf@kfu.edu.sa).

Digital Object Identifier 10.1109/TNSM.2019.2956678

traditional destination-based rules) at corresponding switches. In SD-WANs, thousands or even millions of these rules may need to be installed at different switches. In order to ensure fast lookup and efficient forwarding performance, these complicated rules are typically stored in the Ternary Content Addressable Memory (TCAM), which permits parallel reading of rules. The TCAM memory is expensive and power hungry, and therefore, there is limited space [2] in the rule tables of switches. Thus, in order to ensure scalability of the SDN paradigm to large scale WANs, efficient use of the TCAM switch table space is a major concern. Second, millions of flows entering and leaving the system and with dynamically varying traffic load can lead to uneven traffic distribution, congestion and suboptimal performance. Fine grained load balancing (as opposed to coarse grained) can provide substantial benefits by distributing the load evenly across the network. Third, network failures involving devices and communication links are quite common in large scale networks, and designing a resilient solution which is robust against link or switch failure without requiring intervention of the controller and without modifying flow rules is an important design consideration. Fourth, updates to rule tables of a single switch for load balancing (traffic engineering) or enforcing the optimization model should not result in a cascading of flow updates across the network, i.e., the consistency of the flow tables of the other switches should not be affected and loops should not be introduced due to changes to the flow table of a switch. Finally, due to the structure of the SDN paradigm, the centralized controller can become a bottleneck resulting in significant latency as every incoming flow must be sent to the controller which then computes the optimal path and enforces the configuration across the network. Therefore, mechanisms must be put in place to alleviate the flow latency problem.

With the above goals in mind, this paper presents a holistic solution which addresses several fundamental research challenges to scalability in Wide Area Software Defined Networks. The proposed solution contributes to the state-of-the-art by proposing a novel forwarding paradigm for software defined networks which leverages the concepts of 'resilient' directed acyclic graphs, index-based hashing, and MPLS (Multi-Protocol Label Switching) labeling to ensure a scalable solution which offers efficient utilization of flow tables by enforcing a fixed number of flow entries independent of the number of flows in the network, provides fine grained load balancing by dynamically spreading out traffic between ingress-egress pairs over multiple concurrent paths (over DAG), provides instant routes for incoming flows without having to wait for path computation by the controller, and is robust against node and link failures. Efficient, polynomial time algorithms are proposed to compute the DAG, discover paths and to compute flow allocation tables. The proposed approach is validated by carrying out simulation based performance evaluation over real topologies with a related solution.

The remainder of this paper is organized as follows. An overview of the related work is given in Section II. The main concepts of the proposed DAG-based forwarding paradigm are illustrated in Section III, along with a description of how it can be implemented by means of the OpenFlow protocol. Section IV presents the proposed solution algorithms. We present a performance evaluation of our forwarding paradigm in Section V and conclude the paper in Section VI.

II. Related Work

In this section, we will review the state-of-the-art in scalable software defined networks, identifying the major directions of research and will also highlight limitations of the existing solutions. Research addressing scalability issues in SDNs extends to several areas [3] including optimal placement of multiple controllers, minimizing the number of forwarding rules through flow aggregation, consistent SDN switch updates, traffic engineering and efficient failure recovery among others. Below we review the state-of-the-art in each of these areas and discuss their individual impact on the problem at hand, i.e., scalability of SDNs.

The basic structure of the SDN paradigm implies that the centralized controller is a bottleneck and is therefore one of the biggest limitations in the scalability of the network. In order to address these issues, a number of distributed controllers [4], [5] have recently been proposed with the main objective of distributing the controller instances across the network in order to achieve load balancing and fault tolerance. Both the controllers ONOS [4] and ONIX [5] attempt to provide a uniform global view and abstract the details of the different instances of controllers in the network. Load distribution through instance replication is a classic solution to scalability problems and while these distributed controllers do provide substantial performance improvement, they can only be considered as the first step towards a truly scalable solution as several new research challenges came to light. The next big problem was how to optimally place the controllers in the network. It is evident that placing multiple controllers in the network reduces latency and improves load distribution and fault tolerance, but selecting the optimal positions is non-trivial. The problem of optimal controller placement problem was subsequently explored in software defined networks and several solutions were proposed [6]–[8], which typically minimize the delay between the switches and the controllers. In [9], source routing is exploited as an alternative approach to reduce the latency in the communication process, given that the controller has to communicate with the ingress switch only. In this paper, we adopt a different approach to address the above issues. The forwarding paradigm we propose is such that configuration updates requiring a communication between the controller and the switches are rather infrequent. As an example, recovering from a link/switch failure is performed locally without involving the controller.

The next research challenge in scalability was to limit the number of rules installed at individual switches in order to optimize the rule table space at different switches in the network so that fewer rules need to be updated/installed by the controllers. The Ternary Content Addressable Memory (TCAM), which allows reading rules in parallel, is both expensive and power hungry, and therefore rules need to be

minimized across the network as typical SDN switches support a maximum of up to 1M rule entries [2]. Research in this direction involved compression of rules using wildcard rules [10]–[12], compression of access rules [13] and compressing forwarding rules [14], [15] including approaches that focused on clever routing to aggregate forwarding rules [16]. Some solutions split the rules and distribute rules over the network with the objectives of minimizing the number of rule installations [17], or minimize energy consumption [18]. Some approaches split the table across multiple locations using specialized data structures [19], maximizing the usage of default route segments [20] and handling mice and elephant flows separately [12]. Other solutions in the area [21] advocate minimizing the rule installations through mathematical optimization by reducing the flow reconfigurations. Another solution called *JumpFlow* [22] aims to reduce flow table usage by exploiting the available VLAN identifier (VID) in the packet header to carry routing information. Our forwarding paradigm, instead, can scale to support any number of flows because the number of required flow rules is independent of the number of flows traversing the switch.

In the context of software defined networks, traffic engineering is considered to be a key application and has been investigated in a number of papers [23]. In [24], authors address the maximization of network utilization when SDN is incrementally introduced into an existing network. However, the proposed approach relies on the usage of OSPF-TE to disseminate the knowledge of the available bandwidth on all the network links. The maximization of network utilization while taking the limited size of forwarding tables into account is addressed in [25]. An algorithm to compute multiple paths per ingress-egress pair is presented, though the problem how to split flows across the computed multiple paths is not addressed. Segment routing has been recently introduced [26], [27] as another technique enabling traffic engineering that can benefit from the centralized architecture of SDN. In [28], authors describe an experimental implementation of the segment routing paradigm with an enhanced version of an OpenFlow controller. Both offline and online traffic engineering algorithms based on the segment routing paradigm are proposed in [29]. While these approaches provide interesting benefits, one of their limitations is that the forwarding path must be encapsulated in a stack of MPLS labels and this creates network overhead by consuming bandwidth. Research to address this problem was subsequently carried out and in [30], authors proposed computation of routes that ensure minimum label stack depth. However, their solution is limited to node labels.

Some approaches focus on improving the traffic splitting mechanisms [31]–[33] employed by the traffic engineering approaches in SDNs to address the poor scalability problem of existing SDN-based traffic-splitting solutions as they generate excessive rules for rule-tables on switches. The basic problem is that existing multipath solutions for OpenFlow networks [34], [35] assume hash-based approaches like Equal-Cost Multi-Path (ECMP), which may lead to sub-optimal results. In [31], authors exploit bit-masking operations to partition a traffic demand between the available paths according to precomputed splitting ratios. However, their solution requires extensions to the current protocol, i.e., the ability to express matching field in a more flexible and generic way and to select matching operations. Similarly, in [32] authors attempt to achieve accurate traffic splitting by leveraging wildcard rules to split the traffic within the constrained rule-table size, and use incremental updates to minimize traffic overhead for each update. In [33] authors propose that for non-uniform flow size distribution, a dynamic load distribution scheme based on the collected load sharing statistics can find the most accurate traffic splits with minimal route changes. Differently from our work, these proposals focus on how to split the incoming traffic in predefined proportions locally at a single switch. Thus, they do not address the problems of balancing the load across the entire network and providing mechanisms for fast local restoration.

Failure recovery is another major thread of research which is crucial to the scalability of SDNs and there are several solutions in this area [36], [37], [38]. In fact, in view of the importance of error recovery in SDNs, recent versions of OpenFlow support for the *FastFailover* mechanism which are basically conditional rules in which alternate multihop paths can be used in the case of link failures. However, the controller program must anticipate every possible failure and proactively compute all alternate paths. Moreover, it does not allow programmers to fine-tune the optimization mechanism of switches. In [36], authors proposed SPIDER, which offers a periodic link-probing based detection mechanism, and fast re-routing of traffic even for far off failures, irrespective of controller availability. In [37], authors leverage the well known technique of local detouring by using flow grouping and aggregation methods for rapid and lightweight failure handling in OpenFlow networks. In [38], authors devise a mechanism to find alternate paths to handle high priority packets with minimal delay when a link failure occurs in the network. They also propose a technique to evenly distribute the traffic over all the paths, by finding a set of shortest paths to the destination.

Summarizing, due to the enormous size and exponential number of dynamic network flows, there are several important issues that must be addressed in order to realize truly scalable wide area software-defined networks. As discussed above, several solutions address various aspects of the scalability problem, but our proposed approach contributes to state-of-the-art by proposing a comprehensive approach which encompasses all of these diverse issues and challenges together, in order to achieve a complete solution which is practical and easily applicable to large scale SDNs.

III. DAG-Based Forwarding Paradigm: Concepts

The proposed DAG-based forwarding paradigm is illustrated in this section. The two main concepts it builds on are the usage of a Directed Acyclic Graph (DAG) for each ingress-egress pair, which determines how the flows between that ingress-egress pair are routed, and the usage of a resilient index-based hashing scheme to drive the forwarding of every

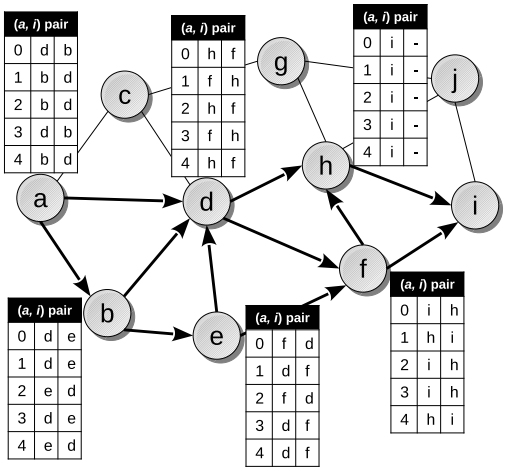


Fig. 1. Example illustrating a DAG between two nodes (a and i, links in bold) and the flow allocation tables used to forward packets of flows between those nodes.

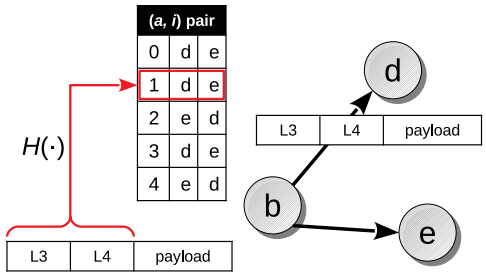


Fig. 2. Example illustrating the index-based hashing scheme ($M = 5$): the XOR folding of the selected header fields hashes to 1; then, the flow allocation table indicates that the packet must be forwarded to neighbor node d. If the link to neighbor node d is down, the packet must be forwarded to neighbor node e.

single flow. The following subsections present such two concepts, provide details about a possible implementation by means of the OpenFlow protocol and discuss the benefits of the proposed paradigm.

A. Requirements for Feasible DAGs

The main idea of the proposed DAG-based forwarding paradigm is to compute a DAG for every ingress-egress pair in the network. The purpose of such a DAG is to identify all and only links that can be used to forward packets of the flows between the given ingress-egress pair. This concept is illustrated in Fig. 1, where the links belonging to the DAG associated with the ingress-egress pair (a, i) are shown in bold. The proposed DAG-based forwarding paradigm is such that the packets of a given flow are forwarded along some path in the DAG between the ingress and egress nodes. By definition, all the paths in a DAG are cycle-free. However, given that every path in the DAG can be potentially taken by the packets of some flow, it is important to set additional requirements for feasible DAGs. We consider a DAG to be feasible if:

- The length of every path between the ingress and egress nodes in the DAG does not exceed the length of the shortest path between the ingress and egress nodes in the DAG multiplied by a given factor α (>1). The rationale is to avoid that packets take extremely long paths that consume resources and increase latency.
- Every node in the DAG has at least two neighbors, so that to guarantee protection against single node/link failures. Indeed, if a node/link becomes unavailable, the upstream node can redirect packets to another neighbor to have them delivered to the egress node.

Clearly, if longer paths are allowed, more resources are consumed, but more paths are available, which increases the chances of distributing the load across the DAG. Given that the computation of the DAGs is independent of the traffic load and does not need to be performed again in case of link/node failures, a proper value for α can be determined offline, as shown in Section V. The approach we propose in this paper to compute a DAG for every ingress-egress pair is presented in Section IV-A.

B. Resilient Index-Based Hashing Scheme

The availability of multiple paths in a DAG between a given ingress-egress pair can be exploited for traffic engineering purposes. Indeed, nodes can route flows on distinct paths in the DAG in order to best balance the traffic load on the network links. However, a proper forwarding strategy needs to be defined to fully exploit such possibility. Firstly, it is necessary to ensure that all the packets of a given flow follow the same path, in order not to incur TCP reordering issues. Secondly, the forwarding strategy should allow to easily tune the load distribution. Thirdly, in case of link/switch failures, packets must be rapidly redirected onto a good alternative path. To meet such requirements, we propose a *resilient* variant of the index-based hashing scheme, which is extremely flexible and exhibits a good performance in terms of load balancing [39]. The proposed scheme is illustrated in Fig. 2. Each node splits the incoming traffic into M bins by using a hash function with XOR folding of selected fields of the packet headers. As an example, this hash function may be expressed as:

$$H(\cdot) = (D_1 \oplus D_2 \oplus D_3 \oplus D_4 \oplus S_1 \oplus S_2 \oplus S_3 \oplus S_4) \; mod \; M \tag{1}$$

where S_i and D_i are the i-th octets of the source and destination IP addresses, respectively. The value obtained by calculating the selected hash function on a packet is used as an index in a *flow allocation table*. Each entry in the table provides both the active and the backup outgoing link for the packet. The active outgoing link is used when it is alive; when the active outgoing link is down, the backup link is used.

We highlight that each node keeps a distinct flow allocation table for each ingress-egress pair whose associated DAG includes the node. Also, we observe that the requirements mentioned above are met. Indeed, the XOR folding of (properly selected) packet headers' fields for all the packets of a flow hashes to the same value; hence, all the packets of a flow follow the same path and TCP reordering issues are avoided. Also, the assignment of bins to outgoing links in the flow allocation tables can be computed so as to enforce a given load

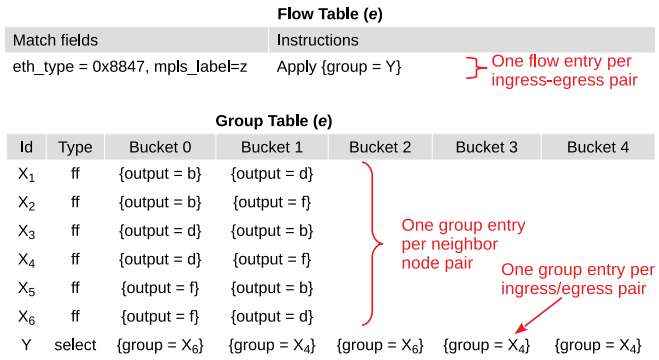


Fig. 3. Example of flow table and group table configuration.

distribution across the network. Finally, fast recovery from node/link failures is ensured by the availability of a backup outgoing link in the entries of the flow allocation tables; the requirements imposed on feasible DAGs ensure that packets will still be able to reach the egress node by traversing a path that is cycle-free and limited in length.

The approach we propose in this paper to compute the flow allocation tables for all the nodes is presented in Section IV-B.

C. Implementation With OpenFlow

To show that the DAG-based forwarding paradigm is practically feasible and to evaluate how it scales, we describe how it can be implemented by means of the OpenFlow protocol (starting from version 1.3). An OpenFlow switch consists of one or more *flow tables* and a *group table*. A flow entry in a flow table consists of *match fields*, counters and a set of instructions. Match fields specify values against which the various packet header fields or the packet ingress port are matched. Counters track the amount of packets and bytes that matched a given flow entry. A number of instructions are defined to redirect a packet to another flow table, add metadata to the packet or perform a set of actions such as set a specified packet header field, redirect the packet to a group entry in the group table or set the output port for the packet. A group entry in the group table is associated with a group identifier and contains a group type, counters and *action buckets*. A group entry may consists of zero or more action buckets, each of which specifies a set of actions to be executed on the packets. Supported group types are *indirect* (the actions of the unique bucket are executed), *all* (the actions of all the buckets are executed), *select* (only the actions of the bucket chosen by a selection algorithm are executed) and *fast failover* (the actions of the first bucket that is associated with a live port are executed).

In current Autonomous Systems (ASes), border routers have to determine the egress node for each packet they receive. In IP-based networks, this is normally performed by using inter-domain rules provided by BGP (Border Gateway Protocol). To correctly take forwarding decisions, internal nodes, too, need to be aware of the egress node selected for each received packet. Multiple techniques can be used to this purpose, including redistributing BGP routes into the IGP (Interior Gateway Protocol) or configuring an MPLS (Multi-Protocol Label Switching) tunnel between the ingress and egress nodes. Similarly, in an SD-WAN, border switches need to determine the egress switch for each packet they forward and the information about the selected egress switch needs to be conveyed to internal switches in order for them to appropriately forward packets. Indeed, our proposed DAG-based forwarding paradigm requires every internal node receiving a packet to determine which ingress-egress pair it belongs to, in order to consult the corresponding flow allocation table and find the outgoing link for the packet. To this purpose, we propose that the ingress node for a packet determines the egress node (as discussed hereinafter) and adds an MPLS label to the packet identifying the ingress-egress pair it belongs to. Every intermediate node can therefore infer the ingress-egress pair each packet belongs to from the MPLS label carried by the packet. In this regard, we note that our paradigm does not require that intermediate nodes perform the usual label swapping mechanism, because the label value identifies the ingress-egress pair and hence it must be kept unchanged until the packet reaches the egress node. Concerning the ability of the ingress node to determine the egress node for a packet, various alternative schemes are possible. A solution involving SDN nodes only consists of having the ingress node send the first packet of a new flow to the controller, which (by using inter-domain rules provided by BGP) determines the flow entry that can be used by the ingress node to match the packets of the flow and add the label identifying the selected ingress-egress pair to them. Another solution, involving legacy border routers and therefore potentially leading to an easier transition to SDN, consists of configuring legacy border routers to add an MPLS label to the packets they forward to SDN nodes, which thus derive the ingress-egress pair the packet belongs to from the MPLS label carried by the packet. Finally, another alternative is provided by OpenFlow-hybrid switches, i.e., switches that support both traditional and OpenFlow operations. For instance, the popular Broadcom chipset [40] has a table that can match on destination IP prefix and set a metadata tag that can be matched in the subsequent TCAM. Thus, the former table can be used to store BGP routes and incoming packets can be tagged based on the matching BGP route.

Figure 3 illustrates how the DAG-based forwarding paradigm can be implemented by using the network and the DAG of Fig. 1 as an example. The case of an intermediate node is considered in the figure. The configuration of an ingress node (not shown) only differs in the flow entries of the flow table, which depend on the implemented solution (as discussed above). Intermediate nodes receive packets carrying an MPLS label that identifies the ingress-egress pair the packet belongs to. In the example of Fig. 3, node e of the network shown in Fig. 1 is considered. A single flow entry per ingress-egress pair is needed in the flow table. Indeed, all the packets belonging to the same ingress-egress pair have to be processed in the same manner. The ingress-egress pair a received packet belongs to can be derived from the MPLS label carried by the packet. Hence, each flow entry matches MPLS packets (*eth_type=0x8847*) with a specified label and redirects them to the group entry that refers to the ingress-egress pair corresponding to the label (in the example, the group entry identifier

is *Y*). Such group entry contains as many buckets as the number of bins in the flow allocation table corresponding to the ingress-egress pair the group entry refers to. This group entry is of type *select* and the selection algorithm used to determine the bucket whose actions are to be executed simply calculates the selected hash function (e.g., the one defined by eq. (1)).

The *i*-th bucket contains a single action to redirect the packet to another group entry that contains two buckets, one for the active outgoing link and one for the backup outgoing link specified in the *i*-th entry of the flow allocation table. In such a way, packets are forwarded according to the bin they are assigned to by the hash function. The latter group entry is of type fast failover to achieve fast restoration in case of failures: if the outgoing link specified in the first bucket is down, the one specified in the second bucket is used. This technique is based on the *liveness monitoring* feature of the OpenFlow switches, i.e., the ability to monitor the status (live or not) of any of their ports. Then, a bucket in a group entry is considered live if the port watched is live. In the example, bucket 0 redirects packets to group entry X_6, which sends them to neighbor f (if the link to f is live) or neighbor d (if the link to f is down), as specified in the first row of node e's flow allocation table (Fig. 1).

As shown by Fig. 3, let *N* be the number of neighbors of an intermediate node and *D* the number of ingress-egress pairs whose associated DAG includes that node, all the flow allocation tables of the node can be configured by means of $\Theta(D)$ flow entries in the flow table and $\Theta(N^2 + D)$ group entries in the group table. Therefore, the number of flow entries and group entries is independent of the number of flows crossing a switch. Finally, we note that the configuration of the egress node (not shown) is rather simple: a flow entry per ingress-egress pair is installed on the flow table to strip the MPLS header and deliver the packet to the intended destination.

D. Features of the DAG-Based Forwarding Paradigm

The DAG-based forwarding paradigm has been designed with the aim of achieving a number of properties. In this section, we list such properties and discuss how the proposed forwarding paradigm achieves them.

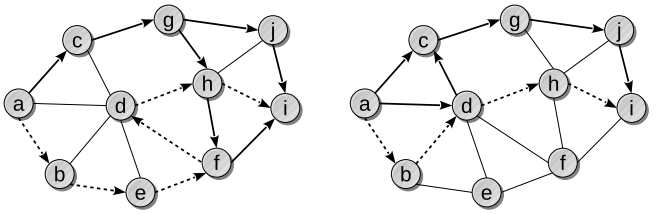
Scalability in terms of number of required entries: The flow table of every SDN switch only contains one flow entry for each ingress-egress pair whose associated DAG includes the switch. Thus, the number of flow entries, which impacts the time required to find a matching rule, is upper bounded by the number of ingress-egress pairs in the network and is independent of the number of flows within the network. In terms of memory occupation, we have to account for an additional group entry per ingress-egress pair, plus one group entry per neighbor node pair. Still, the amount of memory required to store all the required entries is independent of the number of flows within the network. Also, group entries do not need to be stored on a TCAM module because they are not matched against packet headers.

Fine-grained tuning of load distribution: The proposed forwarding paradigm allows to distribute the flows between an ingress node and an egress node across any of the paths available in the associated DAG. Such a possibility can be exploited to traffic engineer the distribution of the load by properly computing the paths taken by the flows. To this end, the assignment of bins to outgoing links in the flow allocation tables can be adjusted to achieve the desired goal. Clearly, the larger the number of bins in the flow allocation tables, the higher the granularity of adjustment.

No delay due to online path computation: The OpenFlow protocol provides that switches can forward packets that did not match any rule to the controller, which can then provide a rule for forwarding these packets. Such a feature may be exploited to perform an online computation of an ad hoc path every time a new incoming flow arrives. However, forwarding the first packet of a flow is significantly delayed due to the time the controller takes to compute a path and enforce the corresponding configuration. With our forwarding paradigm, all paths are already pre-computed, so this source of delay is eliminated.

Fast local restoration of single node/link failures: An important property of a forwarding paradigm for large scale SDNs is the ability to recover from single node/link failures without requiring the intervention of the controller. This property is important because the communication between switches and controller, which usually does not exploit dedicated out-of-band channels in the context of software defined WANs, may be interrupted due to the node/link failure. In such cases, if switches depended upon the controller to recover from the failure, no action would be taken to restore the network connectivity. In any case, having a node/link failure handled locally by switches eliminates the time required for the controller to compute an alternative configuration and enforce it. Our DAG-based forwarding paradigm possesses this property. Indeed, as described earlier, as soon as a switch port is detected to be not alive, the pre-configured backup outgoing link is used thanks to the fast failover mechanism. The requirements for feasible DAGs ensure that the new path followed by the packets is limited in length. Thus, the intervention of the controller is not required to recover from a single node/link failure.

Consistency of forwarding tables during configuration updates: The traffic load is time-varying and adjustments to the load distribution may be needed to keep the network operating efficiently. However, if adjusting the load distribution involves modifying the configuration on multiple nodes, the risk of having inconsistent configurations which temporarily lead to routing loops may arise. Our forwarding paradigm is exempt from this problem. Indeed, changes in flow allocation tables, that can be done, e.g., to adjust the load distribution across the network, may only alter which of the paths available in the DAG are taken by the flows routed along that DAG. However, DAGs are cycle free by definition and, due to the requirements we impose for feasible DAGs, do not contain paths that are longer than the shortest path more than a configurable percentage. Therefore, it is guaranteed that packets are forwarded along loop-free and adequately short paths even while the configuration of the flow allocation tables is updated. As discussed in the next



(a) Adding a loop-less path (dashed) to a DAG including loop-less paths only (bold) introduces a loop (d-h-f-d) in the DAG

(b) Adding a path of length 4 (dashed) to a DAG including paths of length at most 5 (bold) introduces a path of length 6 (a-b-d-c-g-j-i)

Fig. 4. Example illustrating the need of checking the subgraph after the inclusion of a new path.

section, changes in the DAG structure are extremely infrequent, thus the disruption that such operation may cause can be neglected.

Support for traffic differentiation: The proposed forwarding paradigm enables to easily differentiate traffic flows by properly modifying the assignment of flows to bins. To this end, the available bins can be partitioned into different sets, each associated with a distinct traffic class. Distinct hash functions, one per traffic class, can then be defined. The algorithm used to select a bucket inside a group entry can be extended in such a way to compute the hash function corresponding to the traffic class a packet belongs to (the traffic class can be derived, e.g., from the Type of Service fields of the IP header). Thus, packets belonging to different traffic classes are mapped onto distinct sets of bins and each set of bins can guarantee a different treatment (e.g., by forwarding packets along shorter and less loaded paths or by enqueuing packets into priority queues on the outgoing port).

IV. DAG-Based Forwarding Paradigm: Solutions

We present our approach to find a DAG for every ingress-egress pair and compute the flow allocation tables for all the nodes. The idea is that flow allocation tables can be adjusted to adapt to variations in the traffic load, while DAGs should not need to be modified. In the following, we assume that the Software Defined Network is modeled as a directed graph $G = (V, E)$, where V is a set of vertices each representing an SDN switch and E is a set of edges each representing a link between two SDN switches. Given two nodes $u, v \in V$, the capacity of link $u \rightarrow v \in E$ is denoted by $c(u \rightarrow v)$.

A. Computing Feasible DAGs

As mentioned earlier, DAGs do not need to be computed often, because a change in the traffic load should be addressed by adjusting the flow allocation tables. Consequently, the approach we propose here to compute feasible DAGs does not take into account any traffic load, but it only depends on the topological properties of the network. We first present an overview of the proposed approach and then discuss the details of the proposed algorithm, named RDAG (Resilient Directed Acyclic Graph), by illustrating its pseudo-code.

RDAG is an iterative algorithm that keeps a directed subgraph of G and eventually returns it as the sought feasible DAG for a given ingress-egress pair (s, d). The loop invariant is that the directed subgraph is loop free and the length of no path between s and d exceeds the length of the shortest path between s and d multiplied by a given factor $\alpha(> 1)$. The directed subgraph is initialized to the shortest path between s and d in G. In every iteration, a node of the subgraph is explored to ensure that it has at least two neighbors in the directed graph. If such a condition is not met, the directed subgraph is augmented by adding an alternative path from the node to the egress node d that does not include the current next hop of the node. The algorithm ends when all the nodes of the directed subgraph have been explored.

We observe that adding a loop-less path to the directed subgraph does not ensure that all the possible paths in the directed subgraph are loop-less (Fig. 4a). Likewise, adding a path of length less than the maximum allowed one does not ensure that all the possible paths in the subgraph have a length less than the maximum allowed one (Fig. 4b). Therefore, when augmenting the directed subgraph with a new path, it is necessary to check that the inclusion of the new path does not introduce loops or paths of length exceeding the maximum allowed one. Fortunately, this check can be performed by means of a Depth-First-Search (DFS) visit. Indeed, a directed graph is acyclic if and only if the DFS visit returns no back edges [41]. Also, a DFS visit allows to sort the nodes of a DAG in a topological ordering, which is such that if a link $u \rightarrow v$ exists in the DAG, then u precedes v in the ordering. Thus, the ingress (egress) node is the first (last) node in this ordering. The topological ordering allows to easily determine the maximum distance in the DAG between every node and the egress node. Indeed, the maximum distance of a node is one plus the maximum among the neighbors' maximum distance to the egress node. Therefore, the maximum distance to the egress node can be computed for all the nodes by iterating over all of them, starting from the penultimate node in the topological ordering. Thus, a DFS visit on a directed subgraph D checks whether D is a DAG and, in that case, returns a sorted list of nodes in a topological ordering and the length of the longest path between each node in D and the last node in the topological ordering (the egress node).

We now describe the RDAG algorithm in more details (Algorithm 1). The directed subgraph D is initialized to the shortest path in G between the ingress and egress nodes. A DFS visit on D is performed to determine the initial ordering of the nodes. For all the nodes, the attribute *done* is initialized to false. This attribute is set to true once the node has been explored and is used to ensure that every node is explored just once. Then, nodes in D are explored one by one in a reverse topological order, starting from the penultimate node (in the pseudo-code, we use the PREVIOUS(x, l) function to retrieve the predecessor of element x in the list l). In every iteration, if the explored node has been already marked as done or it has already more than one next hop in D, it is marked as done and its predecessor in the topological ordering is explored next (lines 31–32). Otherwise, the explored node u has a single neighbor (v) in D (line 10) and we attempt to find an alternative path to the egress node. For this purpose, we consider a copy (G_{pruned}) of the input graph G and

Algorithm 1 Pseudo-Code of the RDAG Algorithm

```
RDAG (G = (V, E), s, d, α)
 1  D = ∅
 2  SP = SHORTEST_PATH (G, s, d)
 3  PATH_ADD (SP, D)
 4  DFS (D, s)
 5  for each u ∈ V
 6      u. done = FALSE
 7  u = PREVIOUS (D. sorted . last, D. sorted)
 8  while u <> NIL
 9      if u. done == FALSE AND | Adj (D, u)| = 1
10          v = Adj (D, u). first
11          G_Pruned = G
12          if v == d
13              REMOVE_EDGE (G_Pruned, u → d)
14          else REMOVE_VERTEX (G_Pruned, v)
15          found = FALSE
16          L^su = MAX_DIST_FROM_SOURCE (D, u)
17          L^ud = 0
18          while ! found AND L^su + L^ud ⩽ α · length [ SP ]
                AND KSP_HAS_NEXT(G_Pruned, u, d)
19              P = KSP_NEXT (G_Pruned, u, d)
20              D_Augm = D
21              PATH_ADD (P, D_Augm)
22              DFS (D_Augm, s)
23              if IS_ACYCLIC(D_Augm) AND
                   MAX_DIST_TO_DEST(D_Augm, s)
                   ⩽ α · length [ SP ]
24                  found = TRUE
25              L^ud = P. length
26          if found
27              D = D_Augm
28              u. done = TRUE
29              u = PREVIOUS (D. sorted . last, D. sorted)
30              continue
31      u. done = TRUE
32      u = PREVIOUS (u, D. sorted)
```

Algorithm 2 MINMAXUTIL Program

variables

$$f_l^{sd} \in [0, F^{sd}]\ \forall l \in DAG^{sd}, \forall (s,d) \in P$$
$$U \in \mathbb{R}^+$$

minimize U

subject to

1) $$U \geqslant \sum_{(s,d)\,|\,l \in DAG^{sd}} \frac{f_l^{sd}}{c(l)} \qquad \forall l \in E$$

2) $$\sum_{v \in \mathcal{N}^{sd}(u)} f_{u \to v}^{sd} - \sum_{v \in \mathcal{N}^{sd}(u)} f_{v \to u}^{sd} = \begin{cases} F^{sd} & \text{if } u = s \\ -F^{sd} & \text{if } u = d \\ 0 & \text{else} \end{cases} \qquad \forall u \in V, \forall (s,d) \in P$$

prune the link $u \to v$, in case v is the egress node, or the vertex v otherwise. Then, we look for a path between u and the egress node d in the pruned graph. A k-shortest loop-less path algorithm [42] provides, one-by-one and in increasing order of length, the shortest paths between u and d in the pruned graph. The path P returned by the k-shortest path algorithm is tentatively added to a copy (D_{Augm}) of the subgraph D. A DFS visit on D_{Augm} determines whether D_{Augm} is acyclic, finds the length of the longest path between s and d and topologically sorts the nodes. If D_{Augm} is acyclic and the length of the longest path is less than the maximum allowed path length, the path P is actually added to D, node u is marked as done and the exploration of the nodes restarts from the penultimate node in the new topological ordering (lines 26–30). Otherwise, a new path returned by the k-shortest path algorithm is considered.

To avoid that a number of shortest paths between u and d in the pruned graph are uselessly considered, we compute the length L^{su} of the longest path between the ingress node s and u in D (line 16). Given that we already have a topological ordering of the nodes in D, computing such a value only requires to relax all the edges of D (whose weights must be set to -1) [41]. Thus, as soon as the k-shortest path algorithm returns a path with a length L^{ud} such that $L^{su} + L^{ud}$ exceeds the maximum allowed path length, we can stop processing the shortest paths between u and d. Indeed, since shortest paths are returned in increasing order of length, none of the following shortest paths can be added to the subgraph without violating the constraint on the maximum path length. In such a case, node u is marked as done (despite it only has one neighbor) and the predecessor of u in the topological ordering is explored. The algorithm ends when the ingress node s is marked as done and the directed subgraph D is returned.

The complexity of RDAG is dominated by the inner while loop (lines 18–25). In the worst case (since the nodes in D are a subset of those in G), a DFS visit on D requires $O(|V| + |E|)$, while obtaining the next path from the k-shortest path algorithm requires $O(|V|(|E| + |V| \log |V|))$. The outer while loop is repeated at most $|V|$ times (once for each node in D), hence the complexity of RDAG is $O(|V|^2(|E| + |V| \log |V|))$.

B. Computing the Flow Allocation Tables

Our approach to compute the flow allocation tables for every node consists of two steps. First, the estimated traffic matrix is optimally routed across the computed DAGs. Then, flow allocation tables are determined to route flows accordingly.

1) Solving the Optimal Routing Problem: An estimate of the traffic matrix, i.e., the set of demands between all the ingress-egress pairs, can be easily obtained. Indeed, by exploiting the counters associated with the group entry related to an ingress-egress pair (Fig. 3), it is possible for the controller to contact the ingress node for a given ingress-egress pair and retrieve the rate of the traffic demand between the ingress and egress nodes. Given the traffic matrix, the Linear Programs (LPs) illustrated in Algorithms 2 and 3 optimally solve the problem of routing the traffic demands across the given DAGs. Periodically, the controller can retrieve an updated traffic matrix and decide whether to compute a new solution for the routing problem. In the following, we denote by P the set of ingress-egress pairs, by DAG^{sd} the set of links included in the DAG computed for the ingress-egress pair (s, d), by f_l^{sd} the amount of flow belonging to the demand (s, d) that is routed on link l, and by $\mathcal{N}^{sd}(u) = \{v \in V \,|\, u \to v \in DAG^{sd}\}$ the set of neighbors of node u in DAG^{sd}.

The MINMAXUTIL Linear Program (Algorithm 2) routes the traffic demands of all ingress-egress pairs (each across the associated DAG) in such a way to minimize the maximum utilization among all the network links. To this purpose, we

Algorithm 3 MAXMINUTIL Program

variables

$$f_l^{sd} \in [0, F^{sd}] \ \forall l \in DAG^{sd}, \forall (s,d) \in P$$
$$U^{sd} \in \mathbb{R}^+ \quad \forall (s,d) \in P$$

maximize $\sum_{(s,d)\in P} U^{sd}$

subject to

1) $\frac{f_l^{sd}}{c(l)} \geqslant U^{sd}$

$\forall l \in DAG^{sd}, \forall (s,d) \in P$

2) $\sum_{(s,d)\,|\,l\in DAG^{sd}} \frac{f_l^{sd}}{c(l)} \leqslant \mu$

$\forall l \in E$

3) $\sum_{v\in\mathcal{N}^{sd}(u)} f_{u\to v}^{sd} - \sum_{v\in\mathcal{N}^{sd}(u)} f_{v\to u}^{sd} = \begin{cases} F^{sd} & \text{if } u = s \\ -F^{sd} & \text{if } u = d \\ 0 & \text{else} \end{cases}$

$\forall u \in V, \forall (s,d) \in P$

Algorithm 4 Pseudo-Code of the Algorithm to Compute Flow Allocation Tables

```
COMPUTEFAT (N^sd(u), L, T, R, replace)
1  for each n ∈ N^sd(u)
2      load[n] ← 0
3  for i ← 1 to R.size
4      if R[i] == replace
5          for each n ∈ N^sd(u)
6              load[n] += L[i]
7              dist[n] ← ||T − load||_2
8              load[n] -= L[i]
9          R[i] ← arg min_n dist[n]
10     load[R[i]] += L[i]
```

introduce variable U, which is constrained to be greater than the utilization of every network link (constraint 1). The goal of minimizing the maximum link utilization is thus achieved by minimizing U. Constraint 2 represents the usual flow conservation constraint, which is enforced separately for each ingress-egress pair. The optimal value μ of the objective function, i.e., the minimum value of the maximum utilization among all the links, is passed as input to the MAXMINUTIL Linear Program (Algorithm 3). The goal of this LP is to balance the load of each traffic demand across the corresponding DAG. This is done by introducing a variable U^{sd} for each ingress-egress pair (s, d), which is constrained to be less than the amount of flow belonging to the demand (s, d) that is routed on each link of the corresponding DAG (constraint 1), and by maximizing the sum over all the variables U^{sd}. Constraint 2 ensures that the maximum link utilization does not exceed the minimum value computed by MINMAXUTIL, while constraint 3 represents the flow conservation constraint. Solving MAXMINUTIL returns the amount of flow belonging to each traffic demand that needs to be routed on each link in order to achieve an optimal distribution of the traffic load. Next, we show how such traffic distribution can be enforced by properly computing the flow allocation tables.

2) Deriving the Flow Allocation Tables: Given the solution returned by MAXMINUTIL, it is possible to determine the proportions in which a node u included in the DAG associated with the ingress-egress pair (s, d) has to share the incoming flow belonging to the demand (s, d) among its neighbors:

$$\lambda_{u\to v}^{sd} = \frac{f_{u\to v}^{sd}}{\sum_{w\in V\,|\,w\to u\in DAG^{sd}} f_{w\to u}^{sd}}$$

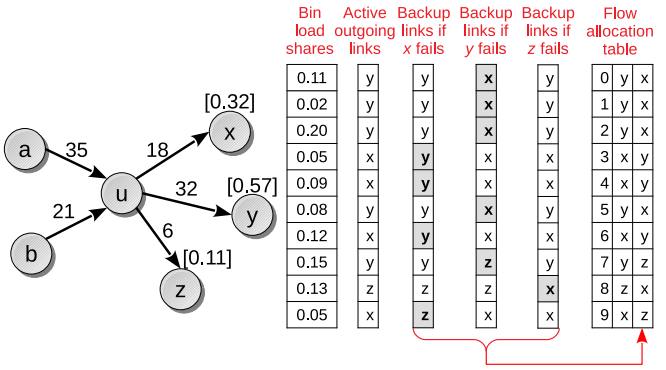


Fig. 5. Example illustrating how to compute flow allocation tables starting from the solution of the LPs. This example refers to a single ingress-egress pair.

An example is shown in Fig. 5, where the proportions in which node u must share the incoming traffic of a given demand among its neighbors are shown in square brackets next to the neighbors. Such proportions are computed by dividing the flow routed on the links to the neighbors by the incoming traffic (56), as per the solution returned by MAXMINUTIL. The flow allocation table on a node (related to a given ingress-egress pair) must be determined such that the incoming traffic is split according to the proportions defined above. To this end, we leverage the availability of counters maintained for each group bucket, besides those maintained for each group entry, to determine the actual load share for each bin of a flow allocation table. The load share of a bin is the ratio of the amount of traffic hashed to the bin to the total amount of incoming traffic that belongs to the ingress-egress pair associated with the given flow allocation table. Then, neighbors must be assigned to bins in such a way that, for every neighbor, the sum of the load shares over all the bins assigned to the neighbor approximates the proportion computed for the neighbor as shown above. As an example, in the flow allocation table of node u (Fig. 5), the sum of the load shares over all the bins assigned to nodes x, y and z must be, respectively, 0.32, 0.57 and 0.11. The algorithm proposed to select outgoing links in such a way to ensure a given set of proportions is shown in Algorithm 4. Given the set $\mathcal{N}^{sd}(u)$ of neighbors in the DAG associated with the ingress-egress pair (s, d), the vector L of the load shares of all the bins in the flow allocation table of node u related to the ingress-egress pair (s, d) and the vector T of the target proportions, the algorithm fills in the elements of the vector R one at a time, starting from the first one. The i-th element of R is set to the neighbor that minimizes the Euclidean distance between the vector T of the target proportions and the vector including, for every neighbor, the sum of the load shares of the bins assigned so far to that neighbor. When the algorithm ends, vector R provides the set of active outgoing links for the flow allocation table.

The algorithm shown in Algorithm 4 is also used to compute the backup outgoing links for a flow allocation table. To this purpose, the algorithm only sets the elements of the vector R that equal the input parameter *replace*. When the algorithm is run to compute the active outgoing links, all the elements of vector R are initialized with an invalid value and *replace* equals such invalid value, so that all the elements of R are set. The algorithm is then run as many times as the number of neighbors, in order to determine the set of backup links to use in case each neighbor becomes unreachable. When the algorithm is run to compute the backup links in case the link to x fails (Fig. 5), vector R is initialized to the set of active outgoing links, *replace* is set to x and the vector T of target proportions is adjusted to consider that the flow routed on the link to x must be shared among the remaining neighbors. The algorithm will therefore only modify the elements of vector R equal to x in such a way to ensure the given target proportions. The set of backup outgoing links is then obtained by taking the neighbors that replace the unreachable neighbor, for every unreachable neighbor, as shown in Fig. 5.

C. Discussion on the Operations of the DAG-Based Forwarding Paradigm

In this section, we review the different components of the proposed DAG-based forwarding paradigm and discuss aspects like how frequently they need to be executed and how their execution can be distributed among multiple controllers.

Computation of DAGs: For each ingress-egress pair, the RDAG algorithm can be executed to determine the associated DAG. The complexity of a single execution of RDAG is polynomial in the number of vertices and edges of the network graph. Given the need of information about the whole topology, the controller is best positioned to execute RDAG. Configuring switches to enforce a new DAG for a given ingress-egress pair can be done without disrupting network connectivity by updating switches in the reverse topological ordering. In this way, it cannot happen that a switch forwards a packet belonging to an ingress-egress pair to a switch that has not been configured yet to forward packets belonging to that ingress-egress pair. Fortunately, however, enforcing a new set of DAGs is not expected to be done on a daily or even weekly basis. Indeed, DAGs are computed based only on topological properties of the network, hence neither changes in the traffic matrix nor link/node failures require to compute a new set of DAGs. Finally, we observe that the load of computing DAGs and performing the corresponding configurations for all the ingress-egress pairs can be shared among a set of distributed controllers, each of which taking care of a distinct sets of ingress-egress pairs and hence operating independently from the others.

Computation of an optimal solution to the routing problem: Given an estimate of the traffic matrix, the MINMAXUTIL and MAXMINUTIL Linear Programs need to be solved to determine how to optimally route the traffic matrix across the network. Given that it is required the knowledge of the DAGs computed for all the ingress-egress pairs and an estimate of the traffic matrix, this task is suited to being performed by a centralized controller. The controller needs to periodically check (by retrieving the counters on the ingress switches) whether the traffic matrix used the last time the routing problem was solved is still a reasonable estimate of the current traffic matrix. If not, the controller may need to compute another solution of the two LPs, to avoid a decrease in network performance. Thus, the frequency of this task depends on how variable the incoming traffic is. Computing a solution to the routing problem can also be distributed among a set of controllers in different ways. One possibility is that the controller detecting the need to compute a new solution to the routing problem informs the other controllers by sharing the new traffic matrix it estimated. All the controllers use the same input to the LPs and hence obtain the same solution. Afterwards, each controller computes the flow allocation tables (as described next) for a subset of the switches or for a subset of the ingress-egress pairs.

Computation of the flow allocation tables: The solution to the LPs provides, along with the current load shares of the bins, the input to the COMPUTEFAT algorithm that computes the flow allocation table on a given node for a given ingress-egress pair. Given the target proportions in which a node has to split the incoming traffic of an ingress-egress pair (as returned by the LPs), it is necessary to recompute the flow allocation table whenever the actual proportions in which the incoming traffic is split (as determined by the hash function and the assignment of bins to neighbors) substantially diverge from the target proportions. Computing flow allocation tables and configuring the corresponding group entries can be done by the unique controller or by the controller that is in charge of configuring the specific node or the specific ingress-egress pair. For this purpose, the controller has to periodically retrieve the counters associated to the buckets of the group entry corresponding to the specific ingress-egress pair in order to obtain the bin load shares and determine whether it is necessary to execute the COMPUTEFAT algorithm again and update the group entry accordingly.

V. PERFORMANCE EVALUATION

We present the results of a simulation study conducted with the ns-3 network simulator to evaluate the performance of the proposed DAG-based forwarding paradigm. We used ns-3.29 patched[1] to add support for both full-duplex CSMA (Carrier Sense Multiple Access) links [43] and OpenFlow 1.3 compliant devices [44]. We implemented a controller[2] in ns-3 that learns the network topology by elaborating the information received by the switches, computes a DAG for every ingress-egress pair, solves the Linear Programs described earlier and configures flow tables and group tables on all the switches as illustrated in the previous sections. Flow allocation tables are initialized by assuming a uniform distribution of the flows among the bins (i.e., the load share of every bin is $1/M$, where M is the number of bins) and recomputed after $5s$ by considering the actual load shares of the bins, as obtained by retrieving the counters associated with the buckets of the group entries.

[1] code available at https://github.com/stavallo/ns-3-dev-git/tree/ofswitch13.
[2] code available at https://github.com/stavallo/ofswitch13/tree/dag-fp.

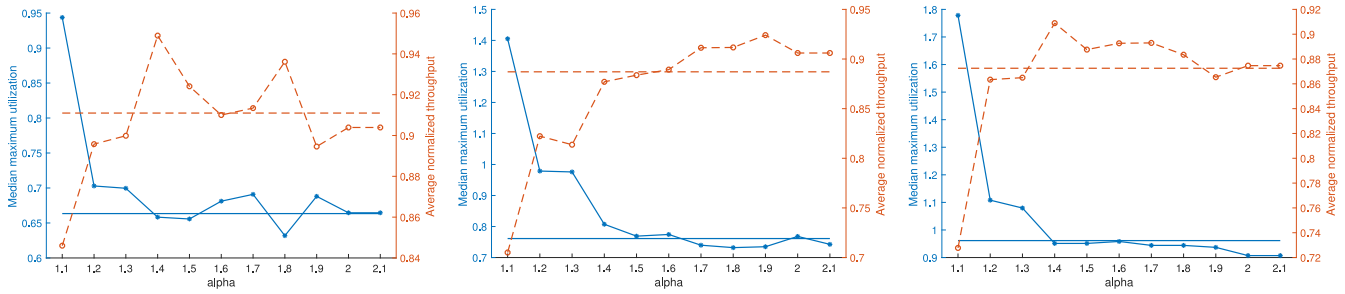


Fig. 6. Median of the maximum link utilization returned by the Linear Programs for every topology and every considered traffic matrix, along with the corresponding average throughput normalized to the sum of the traffic demands, obtained as α varies from 1.1 to 2.1.

Three topologies from the SNDlib library [45] have been used in the simulations: *germany50* (50 nodes and 88 links), *ta2* (65 nodes and 108 links) and *zib54* (54 nodes and 81 links). For each topology, 30 distinct traffic matrices were selected among those available in the library. Each traffic matrix involves from 100 to 250 ingress-egress pairs with different traffic demands. We assume that the controller has the complete knowledge of the traffic matrix and computes the routes for all the ingress-egress pairs accordingly. The actual amount of traffic between each ingress-egress pair is generated by means of a variable number of TCP flows whose average bit rates range from a tenth to a hundredth of the actual amount of traffic to generate. The traffic generator for every single TCP flow follows an on/off pattern, as implemented by the *OnOffApplication* ns-3 model. Each simulation run lasts for a simulated time of 45 seconds.

A. Tuning the DAG-Based Forwarding Paradigm Parameters

We first present an analysis aimed at evaluating the impact of various parameters on the performance of the proposed DAG-based forwarding paradigm. We first consider α, the parameter of the RDAG algorithm that limits the length of the paths included in each DAG. For each of the three considered topologies, we ran the RDAG algorithm to determine the set of DAGs with α ranging from 1.1 to 2.1. Then, for each topology and each value of α, we solved the MINMAXUTIL and MAXMINUTIL Linear Programs to determine the maximum link utilization corresponding to each of the 30 traffic matrices considered for each topology. The solid curves in Fig. 6 show the median of such maximum link utilization values (on the left y-axis), for each value of α. The solid horizontal lines indicate instead a value which is 5% higher than the minimum among such median values. As it can be expected, small α values lead to DAGs including few paths other than the shortest paths, thus causing a reduced ability of balancing the load across the network, which manifests itself through a high maximum link utilization. Increasing α leads to more paths being included in the DAGs, with a consequent improvement in the load balancing and decrease of the maximum link utilization. Starting from a certain α value, however, subsequent increases in the α value do not correspond to a better balancing of the load and the median of the maximum link utilization values seems to oscillate around a steady state value. Given that high α values might lead to long paths being selected for forwarding traffic, we suggest to set α to the minimum value among those corresponding to a median maximum link utilization that is no more than 5% higher than the minimum median value. Based on this approach, we set $\alpha = 1.4$ for the *germany50* and *zib54* topologies and $\alpha = 1.7$ for the *ta2* topology.

To validate the approach proposed to set α, we ran ns-3 simulations to evaluate the throughput achieved with different values of α. For this set of simulations, we set the number of bins to 10 and the number of TCP flows per ingress-egress pair to 20. The dashed curves in Fig. 6 show the overall throughput (on the right y-axis) achieved for each α value, normalized to the total amount of traffic to be generated according to the traffic matrix and averaged over the 30 considered traffic matrices. The dashed horizontal lines indicate instead a value that is 5% smaller than the maximum achieved throughput. It can be observed that the curves representing the median maximum link utilization and the normalized throughput have a rather specular behavior, which implies that it is likely to achieve a high throughput by selecting α with the proposed approach. In fact, the throughput achieved with the selected value of α is the maximum or very close to the maximum (it is above the horizontal dashed line) for all the topologies.

Having determined the value for the α parameter for each topology, we now focus on the impact of the number of bins. Based on the content of its header fields, each packet incoming to a node is hashed to one of the bins available in the flow allocation table associated with the ingress-egress pair the packet belongs to and is forwarded to the neighbor associated with the selected bin. For each topology and each number of bins considered (1, 5, 10, 15, 20, 25 and 30), we ran ns-3 simulations to measure the normalized throughput achieved with each of the 30 traffic matrices and with 5 distinct assignments of port numbers to TCP flows (so as to change the way flows are hashed to bins). Each data point in Fig. 7 represents the normalized throughput averaged over 150 simulations (30 distinct traffic matrices times 5 distinct assignments of port numbers) and is associated with the corresponding 95% confidence interval. Figure 7 shows 3 curves each corresponding to a different number of TCP flows per ingress-egress pair: 10, 20 and 30.

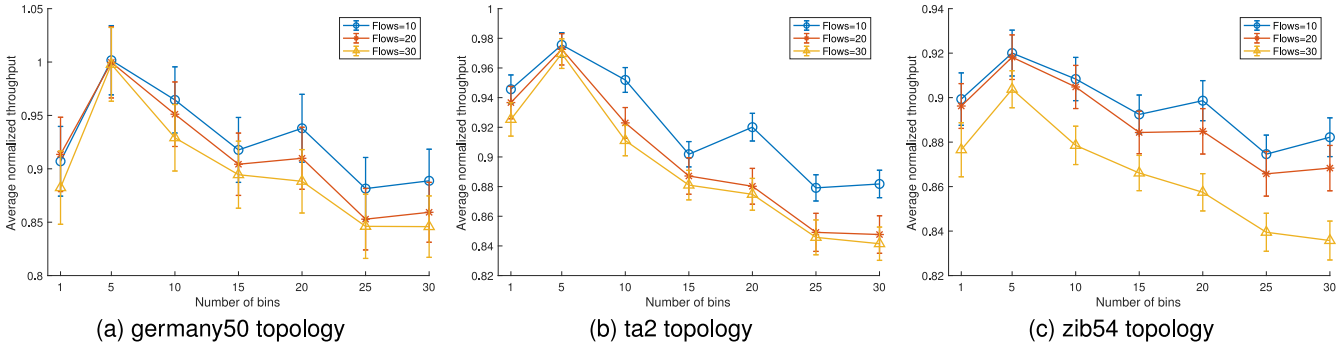


Fig. 7. Average and 95% confidence interval of the normalized throughput over the considered 30 traffic matrices and 5 distinct assignments of port numbers to TCP flows, for different number of bins.

It can be observed from Fig. 7 that results are similar for the three topologies. The optimal number of bins turns out to be 5 and the throughput decreases as the number of bins increases beyond this optimal value. Considering that nodes have 2 or 3 neighbors in a DAG, 5 bins is likely the best compromise between the need of having at least as many bins as the number of neighbors (otherwise, no packet could be forwarded to some neighbors) and the need of limiting the number of bins (compared to the number of flows) in order to guarantee some level of flow aggregation that smooths the on/off pattern of the individual flows and produces aggregate flows with a more regular bit rate. Another observation is that the throughput slightly decreases as the number of flows per ingress-egress pair increases. This result can be likely explained because most of the network links are saturated and packets arriving when the transmission queues are full are dropped. Packet drops trigger the TCP congestion control slow start mechanism. The higher the number of flows entering the slow start phase simultaneously, the stronger its effect, i.e., the alternation of periods when TCP senders slow down (causing a throughput decrease) and periods with large bursts of traffic that induce a new congestion. In fact, we observed that the total number of packets dropped with 30 flows per ingress-egress pair is generally twice the total number of packets dropped with 10 flows per ingress-egress pair. Finally, we mention that the normalized throughput only slightly fluctuates as the assignment of flows to bins changes. In fact, for each traffic matrix and for each number of bins, the standard deviation (not shown) of the 5 values of normalized throughput achieved with distinct assignments of port numbers is less than 0.09. This result proves the effectiveness of running the COMPUTEFAT algorithm 5s after the beginning of a simulation to recompute the assignment of bins to neighbors based on the actual load shares of the bins.

B. Comparative Evaluation of the DAG-Based Forwarding Paradigm

The proposed DAG-based forwarding paradigm is compared to the approach presented in [25] (Algorithm 3), which shares with our proposed solution the goal of maximizing the amount of flow that can be routed across the network while limiting the forwarding table size and whose performance has been analyzed in-depth by the authors. For each ingress-egress pair, 10 distinct paths are computed by using the k-shortest path algorithm. Each path is assigned an amount of flow by solving a bicriteria approximation of the bounded path-degree max flow problem ([25, Algorithm 1]). In our simulations, we assign each of the flows between a given ingress-egress pair to one of the paths so as to meet the computed flow values for the paths. Another controller has been consequently implemented to configure the SDN switches according to the approach we compare our DAG-based forwarding paradigm to.

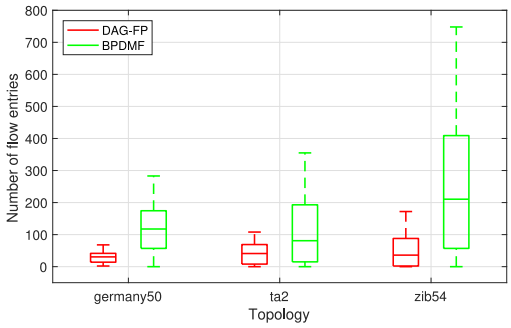


Fig. 8. Number of flow entries required by the algorithms under test.

We first compare our forwarding paradigm (DAG-FP) to the bounded path-degree max flow algorithm (BPDMF) in terms of the number of required flow entries. For each of the considered topologies, Fig. 8 summarizes the number of flow entries installed on each node in all the simulations we conducted (which are described next). On each box, the central mark indicates the median, and the bottom and top edges of the box indicate the 25th and 75th percentiles, respectively; the whiskers extend to the most extreme data points not considered outliers. It turns out that BPDMF requires many more flow entries than DAG-FP (the median for BPDMF is 2 to 6 times higher than the median for DAG-FP). Basically, the reason of such difference is that, for every ingress-egress pair, BPDMF configures a number of single paths, each of which requiring its own flow entry, while DAG-FP configures a DAG which allows flows to take any of the paths included in the DAG.

The first set of simulations aim to evaluate the performance of the algorithms under test when no link failure occurs. We set α as described in the previous subsection, the number of

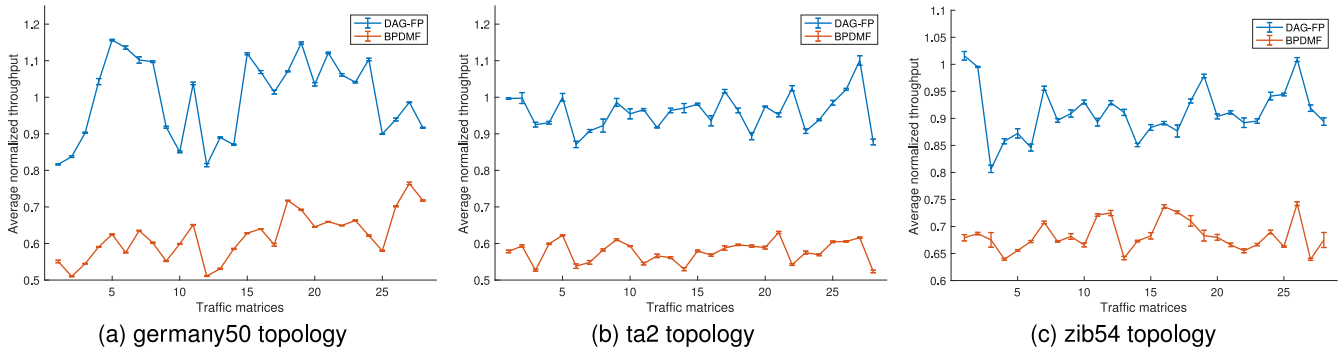


Fig. 9. Average and 95% confidence interval of the normalized throughput achieved for different traffic matrices.

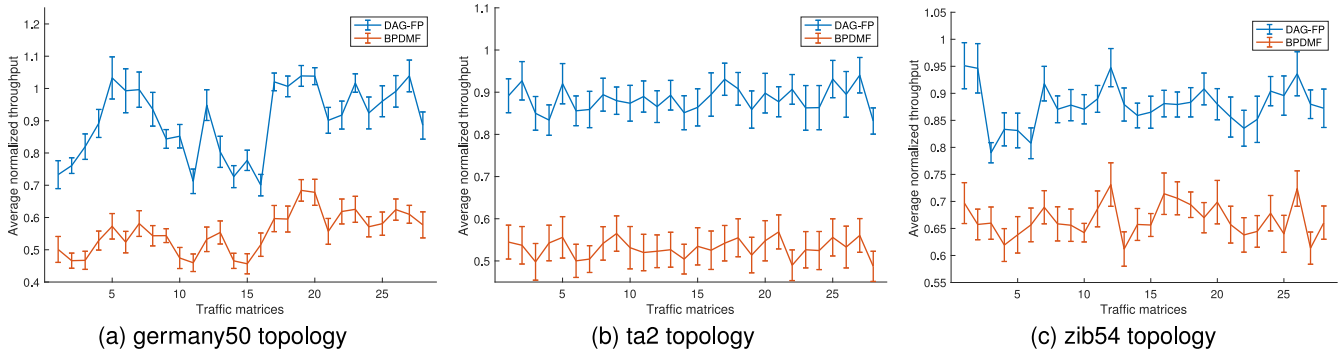


Fig. 10. Average and 95% confidence interval of the normalized throughput achieved after the link failure for different traffic matrices.

bins to 5 and the number of flows per ingress-egress pair to 20. For each topology and for each of the 30 traffic matrices considered, we ran 10 ns-3 simulations by using distinct seeds for the random variables involved in the generation of traffic (each TCP flow is associated with two random variables which determine the duration of the "on" and "off" periods). Figure 9 shows the average and the 95% confidence interval of the normalized throughput achieved by the two algorithms under test in the 10 runs, for each traffic matrix. It can be observed that the DAG-based forwarding paradigm achieves a significantly higher throughput than BPDMF for all the topologies and for all the traffic matrices.

In the second set of simulations, we simulate a link failure after 15 seconds from the beginning of the simulation, in order to evaluate the ability of the proposed paradigm to react to link failures. We stress that BPDMF does not provide any restoration mechanism. However, in order to present a fair comparison to our paradigm, when BPDMF is used, we simulate that the controller is *instantaneously* informed of the link failure, so that it can *instantaneously* instruct the involved ingress switches to redirect the flows routed on the paths affected by the link failure to other paths. Therefore, results in Fig. 10 are obtained by disregarding the time that is required for the node that detects the failure to notify the controller and for the controller to communicate the new configuration to the involved ingress nodes. When our forwarding paradigm is used, instead, nodes detecting a failure can autonomously take appropriate actions to react to the failure, without the need to communicate with the controller.

For every topology and traffic matrix, we conducted 10 different simulations where the failing link is selected randomly. For each simulation, we measured the average throughput over the interval following the failure. Fig. 10 shows the average and the 95% confidence interval of the normalized throughput achieved after the link failure in the 10 runs, for each traffic matrix. As indicated by the wider confidence intervals, distinct link failures have a different impact on the throughput achieved after the failures. Nonetheless, it can be observed that our DAG-based forwarding paradigm still outperforms BPDMF in all the considered cases.

VI. Conclusion

In this paper, we presented a forwarding paradigm for large scale Software Defined Networks based on the definition of a DAG for each ingress-egress pair and on the use of an index-based hashing scheme to balance the load across the paths in the DAG. The proposed paradigm addresses the main challenges currently faced by software defined area networks. Indeed, our proposed paradigm enables fast local restoration of link/node failures, prevents the risk of having inconsistent forwarding tables during configuration updates and eliminates the delay associated with online path computation. Furthermore, we proved through extensive simulation studies that the DAG-based forwarding paradigm achieves significantly higher throughput than an alternative solution while requiring much less flow entries, which is another extremely

desirable feature in the context of software defined wide area networks.

REFERENCES

[1] N. McKeown *et al.*, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[2] *NoviSwitch*. Accessed: Dec. 17, 2018. [Online]. Available: https://noviflow.com/products/noviswitch/

[3] O. Michel and E. Keller, "SDN in wide-area networks: A survey," in *Proc. 4th Int. Conf. Softw. Defined Syst. (SDS)*, 2017, pp. 37–42.

[4] P. Berde *et al.*, "ONOS: Towards an open, distributed SDN OS," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw.*, 2014, pp. 1–6.

[5] T. Koponen *et al.*, "Onix: A distributed control platform for large-scale production networks," in *Proc. 9th USENIX Conf. Oper. Syst. Design Implement.*, vol. 10, 2010, pp. 1–6.

[6] A. Sallahi and M. St-Hilaire, "Optimal model for the controller placement problem in software defined networks," *IEEE Commun. Lett.*, vol. 19, no. 1, pp. 30–33, Jan. 2015.

[7] G. Wang, Y. Zhao, J. Huang, Q. Duan, and J. Li, "A k-means-based network partition algorithm for controller placement in software defined network," in *Proc. IEEE Int. Conf. Commun. (ICC)*, 2016, pp. 1–6.

[8] Y. Hu, T. Luo, N. C. Beaulieu, and C. Deng, "The energy-aware controller placement problem in software defined networks," *IEEE Commun. Lett.*, vol. 21, no. 4, pp. 741–744, Apr. 2017.

[9] M. Soliman, B. Nandy, I. Lambadaris, and P. Ashwood-Smith, "Exploring source routed forwarding in SDN-based WANs," in *Proc. IEEE Int. Conf. Commun. (ICC)*, 2014, pp. 3070–3075.

[10] A. R. Curtis *et al.*, "Devoflow: Scaling flow management for high-performance networks," in *Proc. SIGCOMM CCR*, 2011, pp. 254–265.

[11] A. Iyer, V. Mann, and N. Samineni, "Reducing switch state and controller involvement in openflow networks," in *Proc. IFIP Netw. Conf.*, Brooklyn, NY, USA, 2013, pp. 1–9.

[12] Y. Nakagawa, K. Hyoudou, C. Lee, S. Kobayashi, O. Shiraki, and T. Shimizu, "DomainFlow: Practical flow management method using multiple flow tables in commodity switches," in *Proc. CoNEXT*, 2013, pp. 399–404.

[13] S. Zhang, F. Ivancic, C. Lumezanu, Y. Yuan, A. Gupta, and S. Malik, "An adaptable rule placement for software-defined networks," in *Proc. IEEE/IFIP Int. Conf. Depend. Syst. Netw. (DSN)*, Atlanta, GA, USA, 2014, pp. 88–99.

[14] W. Braun and M. Menth, "Wildcard compression of inter-domain routing tables for openflow-based software-defined networking," in *Proc. 3rd Eur. Workshop Softw. Defined Netw. (EWSDN)*, 2014, pp. 25–30.

[15] M. Rifai *et al.*, "Too many SDN rules? Compress them with MINNIE," in *Proc. IEEE Glob. Commun. Conf. (GLOBECOM)*, Dec. 2015, pp. 1–7.

[16] H. Huang, P. Li, S. Guo, and B. Ye, "The joint optimization of rules allocation and traffic engineering in software defined network," in *Proc. IEEE 22nd Int. Symp. Qual. Service (IWQoS)*, 2014, pp. 141–146.

[17] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the 'one big switch' abstraction in software-defined networks," in *Proc. CoNEXT*, 2013, pp. 13–24.

[18] F. Giroire, J. Moulierac, and T. K. Phan, "Optimizing rule placement in software-defined networks for energy-aware routing," in *Proc. IEEE Glob. Commun. Conf. (GLOBECOM)*, Austin, TX, USA, 2014, pp. 2523–2529.

[19] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *Proc. IEEE INFOCOM*, 2013, pp. 545–549.

[20] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "OFFICER: A general optimization framework for OpenFlow rule allocation and endpoint policy enforcement," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2015, pp. 478–486.

[21] U. Ashraf, "Rule-minimization for traffic evolution in software-defined networks," *IEEE Commun. Lett.*, vol. 21, no. 4, pp. 793–796, Apr. 2017.

[22] Z. Guo *et al.*, "Jumpflow: Reducing flow table usage in software-defined networks," *Comput. Netw.*, vol. 92, pp. 300–315, Dec. 2015.

[23] A. Mendiola, J. Astorga, E. Jacob, and M. Higuero, "A survey on the contributions of software-defined networking to traffic engineering," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 2, pp. 918–953, 2nd Quart., 2017.

[24] S. Agarwal, M. Kodialam, and T. V. Lakshman, "Traffic engineering in software defined networks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2013, pp. 2211–2219.

[25] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "On the effect of forwarding table size on SDN network utilization," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2014, pp. 1734–1742.

[26] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir. (2015). *Segment Routing Architecture*. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-spring-segment-routing/

[27] A. Bashandy, C. Filsfils, S. Previdi, B. Decraene, S. Litkowski, and R. Shakir. (2018). *Segment Routing With MPLS Data Plane*. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-spring-segment-routing-mpls/

[28] A. Sgambelluri, F. Paolucci, A. Giorgetti, F. Cugini, and P. Castoldi, "Experimental demonstration of segment routing," *J. Lightw. Technol.*, vol. 34, no. 1, pp. 205–212, Jan. 1, 2016.

[29] R. Bhatia, F. Hao, M. Kodialam, and T. V. Lakshman, "Optimized Network Traffic Engineering using Segment Routing," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 657–665.

[30] A. Giorgetti, P. Castoldi, F. Cugini, J. Nijhof, F. Lazzeri, and G. Bruno, "Path encoding in segment routing," in *Proc. IEEE Glob. Commun. Conf. (GLOBECOM)*, 2015, pp. 1–6.

[31] D. Tuncer, M. Charalambides, S. Clayman, and G. Pavlou, "Flexible traffic splitting in OpenFlow networks," *IEEE Trans. Netw. Service Manag.*, vol. 13, no. 3, pp. 407–420, Sep. 2016.

[32] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford, "Efficient traffic splitting on commodity switches," in *Proc. 11th ACM Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2015, pp. 1–6.

[33] N. Wu, S.-H. Tseng, and A. Tang, "Accurate rate-aware flow-level traffic splitting," in *Proc. 56th Annu. Allerton Conf. Commun. Control Comput. (Allerton)*, May 2018, pp. 774–783.

[34] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *Proc. ACM SIGCOMM Conf.*, 2013, pp. 3–14.

[35] C.-Y. Hong *et al.*, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM Conf.*, 2013, pp. 15–26.

[36] C. Cascone, D. Sanvito, L. Pollini, A. Capone, and B. Sansó, "Fast failure detection and recovery in sdn with stateful data plane," *Int. J. Netw. Manag.*, vol. 27, no. 2, 2017, Art. no. e1957.

[37] P. Thorat, S. Jeon, and H. Choo, "Enhanced local detouring mechanisms for rapid and lightweight failure recovery in openflow networks," *Comput. Commun.*, vol. 108, pp. 78–93, Aug. 2017.

[38] V. Muthumanikandan and C. Valliyammai, "Link failure recovery using shortest path fast rerouting technique in sdn," *Wireless Pers. Commun.*, vol. 97, no. 2, pp. 2475–2495, 2017.

[39] Z. Cao, Z. Wang, and E. Zegura, "Performance of hashing-based schemes for Internet load balancing," in *Proc. IEEE INFOCOM Conf. Comput. Commun. 19th Annu. Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, vol. 1, Mar. 2000, pp. 332–341.

[40] Broadcom. *High-Capacity StrataXGS Trident II Ethernet Switch Series*. Accessed: Jul. 26 2018. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56850-series/

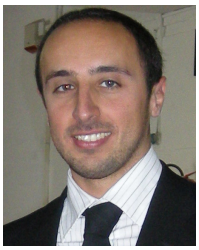
[41] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, U.K.: MIT Press, 2001.

[42] Y. Yen, "Finding the K shortest loopless paths in a network," *Manag. Sci.*, vol. 17, no. 11, pp. 712–716, Jul. 1971.

[43] L. J. Chaves. *Full Duplex Extension for CSMA in ns-3*. Accessed: Mar. 1, 2018. [Online]. Available: https://www.nsnam.org/bugzilla/show_bug.cgi?id=2354

[44] L. J. Chaves, I. C. Garcia, and E. R. M. Madeira, "OFSwitch13: Enhancing Ns-3 with OpenFlow 1.3 support," in *Proc. Workshop NS-3 (WNS3)*, 2016, pp. 33–40.

[45] *SNDlib*. Accessed: Mar. 9, 2018. [Online]. Available: http://sndlib.zib.de/home.action



Stefano Avallone received the M.Sc. and Ph.D. degrees from the University of Napoli Federico II in 2001 and 2005, respectively, where he is currently an Associate Professor with the Department of Computer Engineering. He was a Visiting Researcher with the Delft University of Technology from 2003 to 2004, and Georgia Institute of Technology in 2005. He is on the editorial board of *Ad Hoc Networks* (Elsevier) and the technical committee of *Computer Communications* (Elsevier). His research interests include wireless networks, 4G/5G networks, and software defined networks.

Usman Ashraf received the B.S. degree in computer science from FAST Lahore, in 2003, and the M.S. and Ph.D. degrees in computer networks from INSA Toulouse, France, in 2006 and 2010, respectively. He is currently with the College of Computer Science and IT, King Faisal University. He has several publications in prestigious international journals including IEEE COMMUNICATIONS LETTERS and the IEEE TRANSACTIONS ON MOBILE COMPUTING. He has more than seven years of teaching and research experience. In his last position, he chaired the Department of Computer Science, Air University Islamabad.