Taproot: Resilient Diversity Routing with Bounded Latency

Eman Ramadan, Hesham Mekky, Cheng Jin, Braulio Dumba, Zhi-Li Zhang
{eman,hesham,cheng,braulio,zhzhang}@cs.umn.edu
Department of Computer Science & Engineering, University of Minnesota – Twin Cities, USA

ABSTRACT

As we increasingly depend on networked services, ensuring resiliency of networks against network failures and providing bounded latency to applications become imperative. Adding ample redundancy in the network substrate alone is not sufficient; resilient routing mechanisms that can effectively take advantage of such topological diversity also play a critical role. In this paper, we present *Taproot*, a *resilient diversity routing algorithm* that ensures *bounded latency* for packet delivery under failures by leveraging a *preorder* routing structure with precomputed routing rules. Leveraging the *centralized* control plane and programmable *match-action* rules in the data plane, we describe how Taproot can be realized in SDN networks. We implement Taproot in OVS and conduct extensive simulations and experiments to demonstrate its superior performance over existing solutions. Our results show that by tuning the latency allowance upon failure, Taproot reduces/eliminates the number of disconnected src-dst pairs even under 10 link failures. Finally, as a use case, we illustrate the impact of control channel failures on SDN data plane/application performance, and employ Taproot to provide a "hardened" SDN control network with bounded latency against failures. Our results show that Taproot immediately detects the failure and re-routes the control messages to a different path avoiding failed links/nodes. Hence, the control channel is maintained without interruption, or involvement from the controller, and the throughput was not affected.

CCS CONCEPTS

• **Networks** → **Routing protocols**; **Network protocol design**; **Data path algorithms**; *In-network processing*.

KEYWORDS

Resilient Routing, Link Failures, Latency-Complete Preorder Graphs (PrOG), Data Path Algorithm, SDN, OpenFlow

ACM Reference Format:
Eman Ramadan, Hesham Mekky, Cheng Jin, Braulio Dumba, Zhi-Li Zhang. 2021. Taproot: Resilient Diversity Routing with Bounded Latency. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '21), September 20–21, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3482898.3483364

SOSR '21, September 20–21, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-9084-2/21/09…$15.00
https://doi.org/10.1145/3482898.3483364

1 INTRODUCTION

With our growing dependence on various kinds of cloud-based Internet services on a daily basis, network resiliency has become increasingly critical. One way to enhance network resiliency is to introduce redundancy, e.g., by adding more nodes and links for a more diverse network topology. However, with more network components being added, the probability of network failures also grows with the number of nodes and links. It is reported in [14] that multiple failures occur on a daily basis in many of today's large data center networks, whereas it has been long known that link failures occur frequently in carrier networks [16, 26, 33]. With the "softwarization" and virtualization of networks and introduction of software switches and virtualized network functions, the probability of network element failures would likely rise further, due to server overloads or software bugs.

Besides network resiliency, latency has become another key requirement for many (interactive) Internet services such as e-commerce, video streaming, video conferencing, cloud gaming which have gained rapid popularity. The importance of latency will further grow with the rise of Internet-of-Thing (IoT) applications and cyber-physical systems.

As epitomized by Google's design philosophy that "failures are the norm" [10], it is imperative that networks be designed with *built-in resiliency*. As alluded to earlier, enhancing redundancy in the network fabric, e.g., by adding more switches and links, is not sufficient. Routing is central to any data network as it determines how packets will be forwarded. Hence better routing schemes that can effectively take advantage of topological diversity to overcome network failures while also providing latency guarantees (apart from meeting the network bandwidth requirements) are called for. We will use the term *resilient diversity routing* to refer to a routing scheme that can effectively take advantage of the network topological diversity to proactively prepare for failures, and dynamically route around failed links/nodes without relying on *converged route recomputation*; and such a resilient diversity routing is said to have *bounded latency* if it can further guarantee packet delivery from its source to its destination within a desired latency bound.

Classical distance vector and link state routing protocols such as RIP, OSPF and IS-IS are *reactive* in that upon detecting network failures, they resort to and rely on route recomputation to identify a new path to forward packets. The route recomputation can take 100 milliseconds (ms) or longer to converge [8, 12, 16, 26], with no guarantees on routing latency. MPLS/IP fast rerouting schemes (e.g., [19, 28, 31, 34]) often proactively provision a fixed number of *static* backup routes to prepare for potential link failures. Unfortunately, these schemes often can only protect against one or two link/node failures and cannot fully exploit the rich path diversity

in the underlying network to ensure resiliency. While more sophisticated routing mechanisms that further exploit path diversity, e.g., based on *acyclic directed graphs* (or DAGs), have been proposed in the literature, none can ensure network (routing) resiliency under multiple link/node failures while also providing bounded latency (see §2 for further discussion).

With a logically centralized control and a programmable data plane, SDN enables more flexible and sophisticated mechanisms (e.g., flow-based, non-shortest path routing) for fine-grained route control and traffic engineering. However, SDN also introduces new challenges in terms of resilient routing. Failures in the SDN data plane can potentially cause greater damage, due to lack of autonomy in SDN switches for adaptive route recomputation on their own [20, 23]. This problem is further exacerbated when the failures affect the SDN *control network*, through which the SDN controller communicates with SDN switches. For example, currently OpenFlow switches are often connected to an SDN controller via either an "in-band" channel using the same data plane network fabric, or an "out-of-band" channel with a dedicated (physical) control network. In either case, resilient diversity routing (ideally with bounded latency) is needed to "harden" the control network to ensure reliable communications between switches and SDN controller(s) for SDN control operations. Slow recoveries from network failures are particularly problematic for SDNs, where loss of connectivity between the SDN controller and data plane switches (and among multiple SDN controller instances) can create cascading failures, bringing down the entire network.

As eloquently argued in [23], moving the responsibility of ensuring connectivity to the data plane by endowing switches with *local rerouting* capabilities is advantageous over (*reactive*) routing schemes that rely on the control plane – which operates at a much slower time scale – for route recomputation. Thus, the DDC scheme is proposed in [23] which dynamically reconstructs *directed acyclic graphs* (DAGs) under failures, using link reversal operations [13]. Unfortunately, DDC may incurs $O(n^2)$ link reversal operations in the worst case before the forwarding state converges, where n is the number of nodes in a network. For large n, this convergence time can be significant.

In this paper, we are therefore interested in tackling the following fundamental problem: Considering a fixed source and destination node pair (s, d) and a set of (intermediate) nodes and links which form a subgraph $G_{s\rightarrow d}$ of the underlying network topology G. For example, the links and nodes in such a subgraph $G_{s\rightarrow d}$ are chosen so that they meet the minimum (link) bandwidth requirement and that the end-to-end latency along any path (with no more than a certain hop count) from s to d falls within a desired latency bound. We pose the following question: *Does there exist a routing structure, $\mathcal{R}$, on $G_{s\rightarrow d}$ – namely, a set of preconfigured/precomputed routing rules (e.g., in the forms of SDN match-action rules) – that can fully take advantage of the path diversity in the given subgraph $G_{s\rightarrow d}$, and ensure connectivity between s and d under arbitrary failures as long as the network is not partitioned?*

In this paper, we answer this question affirmatively. Going a step further, we demonstrate that it is possible to achieve a *stronger* objective that we set out to attain: we say that a routing structure (or a routing algorithm), $\mathcal{R}$, achieves *resilient diversity routing with bounded latency* if the following holds: Under any arbitrary failure Φ that does not partition a network, if there exists a path from a source s to a destination d in $G_{s\rightarrow d}$ that meets a given latency bound τ, then there must exists a (*precomputed*) route produced by $\mathcal{R}$ (or equivalently, forwarding state in the form of match-action rules pre-installed by $\mathcal{R}$ at the switches) that is *not affected* by the failure Φ. In other words, switches can dynamically adapt to failure and select the appropriately (pre-installed) rules to forward packets from s to d that meet the latency bound after the failure *without resorting to route recomputation* (either by the SDN controller or the switches themselves). This routing structure is based on the notion of *routing via preorder* first proposed in [32]. Here we are the first to establish why a lesser routing structure, e.g., a collection of (*fixed*) paths or a direct acyclic graph (DAG) with $O(n^k)$ size, is in general not sufficient to attain the desired *full* resilient diversity routing property. We also develop a complete theory (with performance analysis) for resilient diversity routing with bounded latency.

In particular, we present *Taproot*, a resilient diversity routing algorithm with bounded latency, based on the novel notion of *latency-complete* preorder graphs (*PrOGs*). Given a network represented as a graph, we show how latency-complete PrOGs can be constructed in $O(n^3)$ for all source-destination pairs, and establish the correctness of the construction algorithms. We demonstrate how Taproot can be realized in SDN, where pre-computed latency-complete PrOGs are pre-installed in SDN switches as a set of *match-action* rules. We also detail the SDN data operations in terms of the forwarding state maintained by each switch, the failure (and recovery) handling mechanisms via the *activation* and *deactivation* processes, the (*local*) information exchange process by neighboring switches to update the forwarding state in response to changes in the network, and packet forwarding mechanisms where switches select *eligible* outgoing links for packet forwarding based on the latency information carried in packet headers, and update the latency information as packets traverse them. We also use a simple example to illustrate how the "count-to-infinity" problem suffered by a *purely* distributed routing algorithm such as Bellman-Ford is completely eliminated in Taproot, which also relies on information exchange among neighboring nodes for local forwarding state updates. This showcases the power of *joint* centralized (static) route computation and distributed (dynamic) state adaptation and selection enabled by the SDN paradigm (*endowed with local state update capabilities*), which otherwise cannot be achieved via a *purely centralized* or a *purely distributed* routing paradigm. We remark that under Taproot, packets may dynamically adapt and traverse different paths in response to the changing network states; however, as all these paths are part of a *pre-computed/installed* latency-complete PrOG, the SDN control plane still *retains the full visibility* to the data plane.

We have implemented Taproot using the Open vSwitch (OVS) platform (with slight modifications), and conducted extensive experiments to evaluate its correctness and performance. Through simulation results, we illustrate that Taproot is capable to optimally exploit the inherent topological diversity, with superior performance over existing fast rerouting schemes based on pre-configured (static) backup paths. We also compare Taproot with DDC [23] via experiments, and demonstrate that Taproot can adapt to arbitrary failures rapidly, with minimal performance degradation, whereas DDC suffers noticeable delay with considerable performance impact.

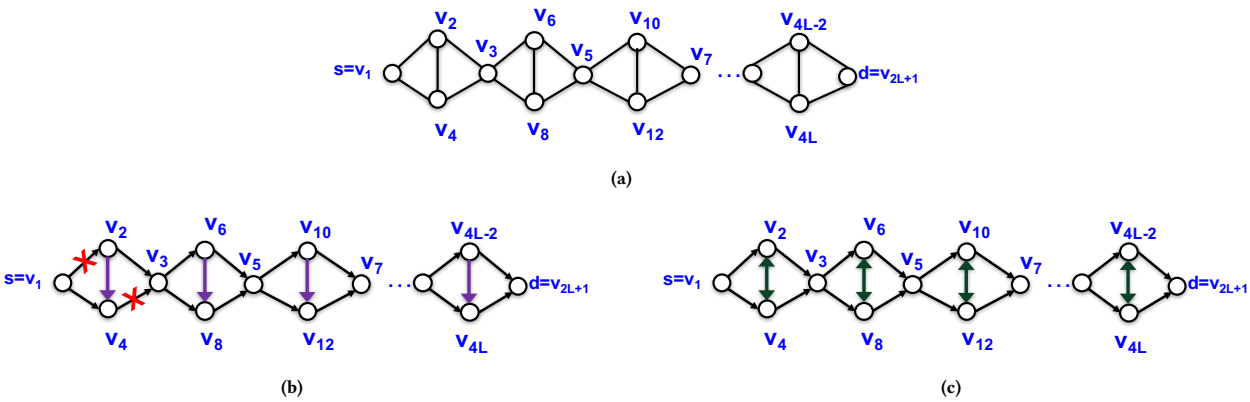


Figure 1: Toy Example (a) network topology; (b) a DAG is not *optimal resilient*, where there exist failure scenarios (e.g., failures of both (v_1, v_2) and (v_4, v_3)) which render d not reachable from s using the DAG; (c) a preorder graph containing maximal diversity.

Finally, we study the effects of the SDN control network failures on both the data plane and application performance in a testbed setting an SDN data plane and control network emulation. This use case illustrates the efficacy of Taproot in "hardening" the network fabric to be resilient against failures, while existing mechanisms based on "native" layer-2 or layer-3 routing mechanisms take longer time to converge affecting throughput.

2 RELATED WORK

Most existing (resilient) routing schemes such as ECMP or other forms of multi-path routing [24, 25, 37], IP Fast Rerouting (see, e.g., [12, 19, 22, 28, 29, 31, 34, 38] and the references therein), or MPLS-based link or path protection routing (e.g., [6, 31] and the references therein) are path-based. In the following, we use a simple example to illustrate that apart from enumerating all possible paths, path-based routing is not optimally resilient.

Consider the simple topology shown in Fig. 1a which has a total of $n = 3L + 1$ nodes. We see that there are in fact $O(2^L)$ paths from $s = v_1$ to $d = v_{2L+1}$ of length $2L$ (and paths of length $2L + 1$, and so forth.) It is not too hard to see that using any two *pre-specified* paths for routing (say, the upper path $P_1 := s, v_2, v_3, v_6, \ldots, d$ as a primary path and the lower path $P_2 := s, v_4, v_3, v_8, \ldots, d$ as a backup path) is not resilient against arbitrary failures which do not partition the network. In fact, apart from pre-installing all paths (of length $2L$ and $2L + 1$), using any K (pre-specified) paths is not optimally resilient in that it fully takes advantage of the available topological diversity: it is possible to pick one edge from each path and fail them (while not partitioning the network) which would render all paths invalid. This problem comes from the fundamental limitations of paths: paths are both *rigid* (as a sequence of links) and *fragile* (failing any link on the path renders the whole path invalid). Unfortunately, enumerating and installing all paths are generally infeasible, as their number may be in the order of $O(n!)$ for an n-node (relatively dense) graph.

Would pre-specifying a (static) DAG suffice? In Fig. 1b we provide an example of a DAG (with a directed vertical edge $v_2 \downarrow v_4$) and a two-link failure which renders this DAG invalid. A failure of this type would also render a DAG with any k directed vertical edges invalid. To be optimally resilient against arbitrary failures by fully taking advantage of the available topological diversity, we need a collection of $O(2^L)$ DAGs[1]. Short of pre-installing all (static) paths or DAGs, one has to *dynamically reconstruct* either a new path[2] or a new DAG in order to be resilient against arbitrary failures. For the simple topology shown in Fig. 1a, it is easy to see how such a new path or DAG may be constructed quickly. For a general topology, reconstructing a new *feasible* path or DAG without knowledge of the global topology may not be trivial.

In [13, 23], dynamic link reversal is used to reconstruct a new feasible DAG; unfortunately this may take up to $O(n^2)$ steps. The routing scheme proposed in [7] dynamically"reconstructs" a path (or rather a walk) by utilizing "match-action" rules in SDN switches and a tag ("dynamic packet state") carried in the packet header to implement a graph search algorithm. However, packets may take up to $O(n)$ steps in the worst case to reach the destination, as they "walk around" (e.g., via depth first search) the network to search for an available path. The paths traversed by packets are not guaranteed to be within a given latency bound either.

It is worth noting here that our notion of resilient diversity routing is closely related to the notion of *ideal forwarding connectivity* defined in [11] with a slight subtle difference (and a *stronger condition*): our notion restricts to the *precomputed*– thus *static* – routes (or equivalently, the forwarding states or rules) produced by a routing algorithm $\mathcal{R}$ *before* failures, where the ideal forwarding connectivity allows for *dynamic* forwarding choices made "on-the-fly" *after* the failure. Hence those routing schemes [7, 21, 23, 27] which *dynamically* reconstruct forwarding states or routes are considered to achieve *ideal forwarding connectivity*, but they do *not* attain the resilient diversity routing property.

[1]Each DAG contains the L vertical edges (v_{4L-2}, v_{4L}) oriented either upward or downward, $l = 1, \ldots, L$; there are a total of 2^L such DAGs. Given an arbitrary failure Φ, if there exists a path from s to d, then it is contained in one of these DAGs.

[2]This is the approach taken by [21, 27] where a new path is computed "on the fly" based on the failed link information carried in the packets. This approach avoids creating transient forwarding loops, but still incurs significant delay and overhead due to the dynamic shortest path recomputation.

3 BACKGROUND AND NETWORK MODEL

3.1 Topological Diversity, Resilient Routing and Bounded Latency

To formalize the fundamental question we posed in the introduction, we first introduce some basic notations. We represent a network topology as a graph $G = (V, E)$. G is assumed to be a connected graph. In addition, for each link $l = (i, j) \in E$, we use $\lambda(l) = \lambda(i, j)$ to denote the *latency* of link l. The matrix $\Lambda = [\lambda(i, j)]$ represents the latency matrix[3], where $\lambda(i, j) = 0$ if $(i, j) \notin E$. For any source and destination pair, s and d, let $P_{s\to d} := v_1(= s), v_2, \ldots, v_L(= d)$ be a path from s to d, where $(v_i, v_{i+1}) \in E$, $i = 1, \ldots, L-1$. (We will drop the subscript $_{s\to d}$ when the context is clear.) The latency of path P is then defined as $\lambda(P) = \sum_{i=1}^{L-1} \lambda(i, i+1)$.

Given a source-destination pair (s, d) and a subgraph $G_{s\to d}$ containing a set of intermediate nodes and links from s to d, a routing structure or algorithm $\mathcal{R}$ outputs a (directed) subgraph $\mathcal{R}_{s\to d}$, containing (directed) paths from s to d in $G_{s\to d}$, thus how packets from s are forwarded to d along the links in $G_{s\to d}$.

Now consider a network failure Φ which brings down a subset of links in E, denoted by E_Φ, and knocks off a subset of nodes in V, denoted by V_Φ[4]. The resulting graph that survives Φ is $\tilde{G} = (\tilde{V}, \tilde{E})$, where $\tilde{V} := V \setminus V_\Phi$ and $\tilde{E} := E \setminus E_\Phi$. We say s and d are *partitioned* if *no* paths from s to d exist in $\tilde{G}$ (i.e., the failure Φ partitions $\tilde{G}$ into two or more connected components.) However, if s is *not* partitioned from d, but all paths produced by the routing algorithm $\mathcal{R}$ *before* the failure Φ are *no longer valid* after the failure (i.e., no path in $\mathcal{R}_{s\to d}$ is contained in $\tilde{G}$), we say $\mathcal{R}$ is not resilient against the failure Φ. Hence, in order to restore connectivity between s and d, $\mathcal{R}$ must resort to route recomputation after Φ. Furthermore, given a latency requirement τ, we say a path $P_{s\to d}$ is *latency compliant* (w.r.t. τ) if $\lambda(P) \le \tau$. In the following we define *resilient diversity routing with bounded latency*:

Resilient Diversity Routing with Bounded Latency. Given a source-destination pair (s, d) and a subgraph $G_{s\to d}$, we say a routing structure (algorithm) $\mathcal{R}$ achieves *resilient diversity routing with bounded latency* (w.r.t. $G_{s\to d}$ and a set of latency bounds $\{\tau_{s\to d}\}$), if under any arbitrary failure Φ, there exists a *latency-compliant* path in $\tilde{G}_{s\to d}$ w.r.t. $\tau_{s\to d}$ (i.e., s can still reach d via a path P in $\tilde{G}_{s\to d}$ such that $\lambda(P) \le \tau_{s\to d}$), there must also exist a (precomputed) *latency-compliant* route from s to d (w.r.t. $\tau_{s\to d}$) in $\mathcal{R}_{s\to d}$ that is *not* affected by the failure Φ.

Given the above definitions, we see that a resilient diversity routing structure with bounded latency is capable of fully taking advantage of the topological diversity and redundancy provided by the network while also ensuring bounded latency. Attaining such property seems to be a tall order, given that it was not even clear whether optimal resilient routing (without latency bound) could even be achieved. Apart from resilient routing schemes designed for specific topologies (e.g., Fat-Tree or Leaf-Spine data center networks [4, 5, 15, 30, 35]) that have been "customer-designed" for data center networks, so far mostly negative results are known, see, e.g., [11] where it is shown that resilient routing based on interface-specific-forwarding (ISF) proposed in [22, 28, 29] is not resilient against arbitrary k link failures when $k > 1$.

[3]We can easily also accommodate network bandwidth requirements into our formulation. However, for clarity of presentation, we have ignored them – we may simply assume that only links that meet the minimal link bandwidth are selected for routing, and thus are eligible for candidate route precomputation.

[4]V_Φ can be empty. In this case, only link failures occur. We consider link failures because any node failure is equivalent to the failure of all its adjacent links.

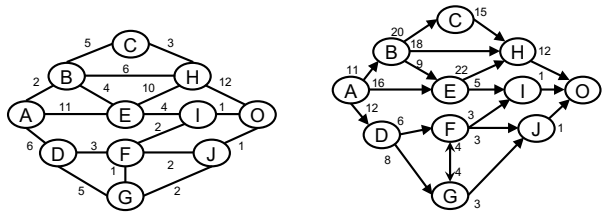
3.2 Preorders & Latency-Complete PrOGs

Routing via preorder proposed in [32] is the first resilient routing scheme that employs a new routing structure other than paths or DAGs. In the previous section we have demonstrated for the first time that DAGs are not sufficient. In the following we introduce the novel notion of *latency-complete* preorder graph (PrOG) which subsumes the τ-complete PrOGs defined in [32]. As we will show in §4, instead of a sequence of τ-complete PrOGs, a *single* latency-complete PrOG (with appropriately annotated latency information) suffices to attain resilient diversity routing *with or without bounded latency*.

Any directed graph forces an order among its nodes, we will now define this order and label the nodes accordingly. Mathematically, a *preorder* $\lesssim$ on a node set $U \subset V$ is a binary relation that is *reflective* and *transitive*. For any $u, v \in U$, if $v \lesssim u$, we say v is a predecessor of u, and u a successor of v; also, v is a *child* of u, and u a *parent* of v, if $v \lesssim u$ but $u \not\lesssim v$ (we denote this relation by $v \lesssim u$), and $\nexists\, w \in U$ such that $v \lesssim w \lesssim u$. If $v \lesssim u$ and if $u \lesssim v$, they are *siblings*. The corresponding (directed) graph to the preorder $(U, \lesssim)$, where there is a *(uni-)directed* edge $v \to u$ if u is a parent of v, and a *(bi-)directed edge* $v \leftrightarrow u$ if u and v are siblings, is referred to as a *preorder graph* or *PrOG*. (Note that if $\lesssim$ is *antisymmetric*, it yields a *partial order*, denoted by $(U, \prec)$, and results in a DAG.) Given $S \subseteq U$, $w \in U$ is called an *upper bound* of S, if $\forall\, v \in S$, $v \lesssim w$, and it is a *least upper bound* of S, if for any upper bound w' of S, $w \lesssim w'$. The (greatest) lower bounds of S can be similarly defined.

Given a network represented as a graph $G = (V, E)$, we say a preorder $\lesssim$ defined on G is a *(routing) preorder* from s to d – and the induced preorder graph (an *oriented* subgraph of G) a *routing PrOG* – if the following conditions hold: i) the preorder $\lesssim$ is *compatible* with G in that u is a parent or sibling of v *if and only if* $(u, v) \in E$; ii) s is the *unique* greatest lower bound of U, and d is the *unique* least upper bound of U; and iii) for any $u \in U$, $s \lesssim u \lesssim d$. Fig. 2b shows an example of a routing preorder from s to d. Intuitively, the conditions ii) and iii) above imply that there is a (directed) path (i.e., a chain of uni-or bi-directed edges) from s to d and any node u in the PrOG induced by $\lesssim$ is on a (directed) path from s to d. We will use $G_{s\to d}$ as a (generic) notation to denote the *induced* (routing) PrOG defined by a routing preorder from s to d, and drop the adjective "routing" for conciseness. As bi-directed edges are allowed, a *PrOG* is in general not a DAG. Fig. 1c shows a PrOG from $s = v_1$ to $d = v_{2L+1}$ for the toy network example in Fig. 1a.

We now introduce the key notion of *latency-complete* PrOG from s to d. We say a PrOG $G_{s\to d}$ from s to d is *latency-complete* with respect to any $\tau > 0$ if there exists a (*simple*) path P from s to d in G with latency $\lambda(P) \le \tau$, P is contained in $G_{s\to d}$ (as a *directed* path). For any $u \in G_{s\to d}$ and an *outgoing* link $u \to v$ (for a bi-directed edge $u \leftrightarrow v$, the outgoing direction $u \to v$ is considered an outgoing link of u), we augment it with a *(minimum) latency*



(a) Weighted Topology (b) Latency-Complete PrOG

Figure 2: Example of Latency-Complete PrOG, src = A, dst = O, with min latency of each node to O

state $\tau^d(u \to v)$, where $\tau^d(u \to v) := \min_{P_{u\to d} \in G_{s\to d}} \lambda(P)$, and $P_{u\to d}$ denotes a path (segment) from u to d in $G_{s\to d}$. We see that $\tau^d(u \to v)$ represents the minimum latency that can be attained by any path from u to d in G *via the outgoing link* $u \to v$. We refer to the resulting PrOG augmented with the minimal latency states as the *augmented* latency-complete PrOG; thereafter all PrOGs in the paper will be assumed to be augmented latency-complete PrOGs. For conciseness, we drop the adjective "augmented" (and sometimes, "latency-complete") unless emphasis is needed. The PrOG in Fig. 1c is a latency-complete PrOG from $s = v_1$ to $d = v_{2L+1}$ for the toy network example in Fig. 1a. For the network topology in Fig. 2a (where numbers above the edges are link latency), an *augmented* latency-complete PrOG from $s = A$ to $d = O$ is shown in Fig. 2b, where the number above an outgoing link is the minimum latency state $\tau^d(u \to v)$. The following theorem regarding the resiliency and diversity of latency-complete PrOGs holds trivially by its definition.

Theorem 1:Resilient Diversity of Latency-Complete PrOG. Given any arbitrary failure Φ that does not partition s and d, for any $\tau > 0$, if there exists a path $\tilde{P}$ from s to d in the surviving graph $\tilde{G}$ such that $\lambda(\tilde{P}) \le \tau$, then $\tilde{P} \in G_{s\to d}$. Hence, if a (precomputed) latency-complete PrOG $G_{s\to d}$ is employed for routing from s to d, it attains *resilient diversity routing with bounded latency.*

4 TAPROOT: RESILIENT DIVERSITY ROUTING WITH BOUNDED LATENCY

In this section, we present Taproot – a resilient diversity routing with bounded latency. We first provide an overview of Taproot. We then describe the control plane (i.e., PrOG construction) operations – in particular, algorithms for constructing (augmented) latency-complete PrOGs– as well as the data plane operations such as the state maintained by the switches, deactivation/activation processes, and eligible outgoing link selection for packet forwarding.

4.1 Overview of Taproot

Based on the notion of latency-complete PrOG, Taproot leverages the *centralized* control plane and programmable *match-action* rules in the data plane to achieve resilient diversity routing with bounded latency. Taproot operations are divided into control plane operations and data plane operations. Given a network topology G, the centralized control plane first constructs an augmented latency-complete PrOG, $G_{s\to d}$, for each source and destination (s, d). We present the construction algorithms in §4.2; the total construction time for all source-destination pairs is $O(n^3)$. The augmented latency-complete $G_{s\to d}$ is mapped to a set of *match-action* rules that are pre-installed in switches. In particular, the minimum latency state associated with each outgoing link will be installed in the SDN data plane, as part of the forwarding state that will be maintained and *updated* by the switches as the network state changes, e.g., as some links/nodes fail or recover. Switches also maintain the status of outgoing links (e.g., up or down) and the rules associated with them. As an outgoing link goes down or up, this may affect the minimum latency state. If this state changes, a switch will update it and inform its upstream neighbors. As a switch becomes a "sink" (see the definition below), it will invoke a *deactivation* process; likewise, when a link comes back up, an *activation* process may be invoked. Upon receiving a packet, a switch will select an *eligible* outgoing link based on the latency information carried in the packet header and the minimum latency state associated with the outgoing links. The latency information carried in the packet header will be updated as a packet traverses each switch. The data plane operations are described in §5.

As an illustration, we will use an example to describe the basic operations of Taproot. Consider the network shown in Fig. 2a where the link weight indicates latency. For the source-destination pair, $s = A$ and $d = O$, the augmented latency-complete PrOG $G_{s\to d}$ is shown in Fig. 2b, which is pre-installed in the data plane for resilient routing of packets from s to d. Now consider a flow f from $s = A$ to $d = O$ with a latency requirement $\tau_f = 12$ under normal operations and a relaxed latency requirement $\tilde{\tau}_f = 14 (\ge \tau_f = 12)$ under failures. These two latency-constraints are equivalent to selecting a so-called τ_f-complete sub-PrOG of $G_{s\to d}$ as a *primary* PrOG, and a $\tilde{\tau}_f$-complete sub-PrOG of $G_{s\to d}$ as a *backup* PrOG according to the definitions mentioned in [32], as shown in Fig. 3a and Fig. 3b respectively. A τ-complete PrOG is a PrOG that contains all paths from s to d with latency $\le \tau$. Hence, we can consider that only the outgoing links in Fig. 3a are used for forwarding packets of flow f under normal operations (i.e., without failures) to meet the latency bound τ_f, but all links in Fig. 3b can be used for forwarding packets of flow f under failures. Both are subsets of the augmented latency-complete PrOG $G_{s\to d}$ shown in Fig. 2b.

Using Fig. 3b, suppose the link $I \to O$ goes down. This failure renders I a *sink* node (i.e., it has no outgoing link). In this case, node I simply *deactivates* the incoming links $E \to I$ and $F \to I$ by notifying E and F not to forward packets from $s = A$ to $d = O$ to it. However, this does not affect the reachability from F to O; F simply uses the other outgoing link $F \to J$ to forward packets from A to O. On the other hand, the deactivation message from I to E causes node E to become a sink node also. This would trigger E to deactivate its incoming link, $B \to E$, which in turn triggers B to deactivate the link $A \to B$. Any existing packets destined to $d = O$ that are buffered at these immediate nodes {I, E, B} will simply be rerouted back to A *if* they have not exceeded their relaxed latency deadline $\tilde{\tau}_f$, so that A can re-route them to O using its current active outgoing link $A \to D$. Otherwise, these existing packets will be dropped. As a result of this failure and subsequent (*local*) actions at the affected nodes, the original PrOG dynamically shrinks by shedding the deactivated links, and the packets from $s = A$ are now

routed solely using the remaining unaffected portion of the original PrOG, namely a sub-PrOG on the node set $\{A, D, F, G, J, O\}$. If later link $F \rightarrow J$ also fails, in this case node F will activate the back-up link (a bi-directed edge) and forward packets of flow f along $F \rightarrow G$, so that the new path becomes $A \rightarrow D \rightarrow F \rightarrow G \rightarrow J \rightarrow O$ with a total latency of 13, which still satisfies the relaxed latency deadline upon failure $\tilde{\tau}_f = 14$. Link recovery events will be handled via an activation process which restore the failed outgoing link with the associated rules.

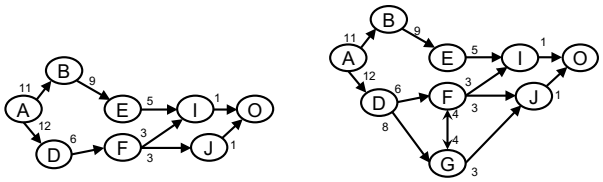
4.2 Control Plane Operations

A latency-complete PrOG can be constructed in a centralized way in two phases: i) First, we perform a breadth-first search on the network, starting with the destination d. For each node, we record its and its neighbors' minimum latency towards the destination; ii) Second, we start with the source s and prune any branch whose latency exceeds τ. The centralized algorithm, helps prevent loops and avoid the count-to-infinity problem during failures, as will be shown. Now we explain the algorithm in details, along with complexity analysis and proof of correctness.

Phase I: Latency Calculation In this phase, we compute the minimum latency for each node to reach the destination d. Each vertex $u \in V$ maintains L_u^d which represents its min latency to d. Initially, $L_u^d = \lambda(u, d)$ if $(u, d) \in E$; otherwise, $L_u^d = \infty$. Node u also maintains the min latency of its neighbors to reach the destination d without using node u as their nexthop represented by $L_{v,\bar{u}}^d$ (similar to split-horizon advertisements.) We use a min-priority queue to store vertices sorted by their L_u^d values in an ascending order. When vertex u is visited, for each neighbor v, u calculates its min latency to the destination d *without* using v as a nexthop and updates $L_{u,\bar{v}}^d$ of this neighbor v. Upon the update of $L_{u,\bar{v}}^d$, the value of L_v^d is updated if a lower latency is found as following: $L_v^d = min\{L_v^d, L_{u,\bar{v}}^d + \lambda(v, u)\}$. If a lower value is found, this means that node v can reach destination d via node u as its nexthop. (That is why node v has to be excluded while node u calculates $L_{u,\bar{v}}^d$.) In this case, node v is pushed to the queue if it was not queued before[5]. Using the topology shown in Fig. 2a, the source node $s = A$, and the destination node $d = O$. Let us consider node E, its min latency to the destination $L_E^O = 5$ (using I as a nexthop.) Now, when E calculates its min latency for node I, $L_{E,\bar{I}}^d = 22$ (using H as a nexthop) and its min latency through I has to be excluded. Thus, in case $I \rightarrow O$ fails, I can forward its traffic to E as it has another route to the destination through node H.

Analysis: For each destination d, each node in the network is visited once; to update the min latency of each neighbor, the algorithm iterates over the other neighbors. This yields a time complexity of $O(n^2)$ per destination. The total complexity of this phase is $O(n^3)$ for *all* source-destination pairs. This phase is only done once. For different values of τ, we just repeat phase II as below.

[5]In case of ties while choosing the next vertex (i.e., multiple nodes can have the same min latency to the destination), the node with the largest number of neighbors is visited first, so that it has a higher probability of having neighbors with min latency values which it can use to calculate its min latency and report to its other neighbors. If more than one vertex have the same number of neighbors, the node id is used as a tiebreaker.



(a) Primary PrOG τ_f = 12 **(b) Backup PrOG $\tilde{\tau_f}$ = 14**

Figure 3: Example of Primary and Backup PrOGs for Network in Figure 2

Phase II: Specifying Eligible Nexthops After calculating the minimum latency of each node and its neighbors, each node needs to specify a set of eligible nexthops based on the value of τ. First, we calculate the minimum latency from each node u to the source node s denoted by L_u^s, a neighbor v of u is considered an eligible nexthop if it satisfies the following condition: $L_{v,\bar{u}}^d \leq \tau - L_u^s - \lambda(u, v)$, which represents the available latency that must be satisfied at node v to reach the destination within τ. The set of eligible outgoing links of node u includes its neighbors with the min latency satisfying this condition. In this phase, we need to visit each vertex only once to calculate its eligible outgoing links set. For example, if $\tau = 12$, then E is not an eligible outgoing link for A, as its minimum latency to reach the destination is 16. We use such algorithm to specify the eligible nexthops, because we are not merely selecting those nexthops of u which are on the shortest paths from u to d (i.e., have lower latency than u), but also those neighbors of u which may have higher latency but still can be used to forward packets within the latency requirement τ, which can be used in case of failures of the shortest path links. The eligible outgoing links of each node form the *τ-complete*. That is why in Phase I, at each node u we not only need to calculate L_u^d, but also $L_{u,\bar{v}}^d$ for each neighbor v. This allows us to construct a PrOG that contains all possible (simple) paths meeting the latency requirement τ without creating cycles or loops. In other words, the construction ensures a *preorder* among the nodes involved. Correctness proofs can be found in §5.3.

Analysis: For each source s and latency requirement τ, the complexity of this phase is $O(n^2)$. It takes $O(n^2)$ to calculate the minimum latency of each node to the source s using Dijkstra's algorithm. Then, each node is visited once to specify its eligible nexthops by iterating once over its neighbors.

5 DATA PLANE: PACKET FORWARDING AND FAILURE HANDLING

In the data plane, each switch needs to maintain forwarding states to determine whether a link and a neighbor switch are eligible to use or not given a specific τ. Switches handle failure (and recovery) via *deactivation* (and *activation*) processes, and select *eligible* outgoing links for packet forwarding based on the latency information carried in packet headers, and update the latency information as packets traverse them. The (*local*) information is exchanged among neighboring switches to update the forwarding state in response to changes in the network.

5.1 Switch States and Packet Forwarding Under Normal Operations

For each pair of source s and destination d, a switch i needs to maintain the following information for each outgoing port j: a destination-based *latency* L_j^d (the minimum latency from j to destination d), and a *state* {Active, Inactive}. Each switch also maintains its minimum latency to reach the destination L_i^d based on the information of its neighbors.

Under normal operations, packets are required to be forwarded to the destination d within τ_f latency. We use two extra header fields in packets to represent its latency requirements, a `latency` $\mathcal{T}$ field (similar to the standard TTL field) and a `latency_offset` $\mathcal{T}'$ field. At source s, the `latency` field is set to $\mathcal{T} = \tau_f$ and the `latency_offset` field is set to $\mathcal{T}' = \tilde{\tau}_f - \tau_f$. When a node i receives a packet, an *active* outgoing port j is guaranteed to forward the packet to the destination within τ_f using simple path if it satisfies the following two conditions: i) $\mathcal{T} \geq L_j^d + \lambda(i,j)$: node j's latency to reach the destination L_j^d and the cost $\lambda(i,j)$ to reach node j form node i is less than the value of $\mathcal{T}$, and ii) $L_j^d < L_i^d$: the latency L_j^d of node j to reach the destination d is smaller than the latency L_i^d of node i, in order to progress towards the destination at each step. L_j^d stored at node i is the minimum latency to reach the destination without using node i as j's nexthop as mentioned in §4.2. Any time a packet is forwarded along a link $i \rightarrow j$, the `latency` field is decremented by $\mathcal{T} = \mathcal{T} - \lambda(i,j)$; the packet is dropped when this value reaches zero. Next, we illustrate how failures are handled by using the `latency_offset` field.

5.2 Handling Failures and Recoveries: Deactivation and Activation Processes

Failures can destroy part of the PrOG and may render some nodes to become "dead ends" or *sinks* with no valid outgoing links. Resilient routing with $\tilde{\tau}_f$-latency is achieved under arbitrary link/node failures by dynamically *deactivating* the directed edges that are affected by failures and *activating* certain backup directed edges.

Handling link/node failures: When a link failure happens, the packet is dropped or forwarded back to (any) predecessor (parent) node, say x, if the current node, say y, is a sink node. y also notifies x to no longer forward packets destined to the affected destination to it. If x also has no more successor nodes, it further notifies its parent(s) not to use it to forward packets to the destination. This process is repeated *recursively* till reaching a non-sink node or the source node s. To achieve this, a 1-bit deactivation/activation tag (*link_state* = {1 Active, 0 Inactive}) is added in the packet header to exchange the state of the link between neighbors for a list of affected source-destination pairs, which are included in the packet header. Upon receiving such tag, a node updates the `state` of its local port.

Using the PrOG in Fig. 2b as an example, suppose the link $D \rightarrow G$ fails. There still exists a path from D to O (i.e., $D \rightarrow F$). D simply routes all traffic from $s = A$ to $d = O$ to F. Now consider another scenario when link $I \rightarrow O$ fails, this renders I to become a *sink* (i.e., it has no outgoing link). In this case, I invokes the *deactivation* process which prepares a *deactivation tag* for this source-destination pair to notify its two predecessors E and F. Any packets destined to $d = O$ that are buffered at I will be rerouted back to E or F, if they have not exceeded their latency deadline. When E receives the *deactivation tag*, it invokes the *deactivation* process which changes the *state* of the outgoing port I to *Inactive*, and continues forwarding packets using its current active outgoing ports (in this case H.) Similarly, F will forward its traffic to O through J.

Handling link/node recovery: When a failed link/node recovers from failure, an *activation* process is initiated. If a node is no longer a sink after recovery, an *activation tag* is generated to notify its predecessor nodes about the affected source-destination pairs. This process is done *recursively* to re-activate the relevant portion of the PrOG. We emphasize that a key advantage of Taproot is that we are guaranteed to route flows using paths satisfying their given latency. When failed links/nodes are all recovered, Taproot will activate the same PrOG that was originally constructed for resilient routing. For example, when the link $I \rightarrow O$ recovers from failure, I will send an *activation tag* to notify its predecessors E and F. When E and F receive this *activation tag*, they reverse the actions taken during *deactivation*.

Packet forwarding under failures and recovery: With link failures, the latency associated with an outgoing port may change. The node needs to recalculate its minimum latency to reach the destination after failures, and inform its neighbors of its new minimum latency. This process is done *recursively* to update the minimum latency of each node to reach the destination after failure. The process stops when the minimum latency of a node is not affected by the propagated information from its neighbor nodes. During this process, the `latency_offset` is added to the remaining latency in the packet which is then re-routed immediately through the path that satisfies the new latency constraints. If no such path exists, the packets will be dropped. Once the network is recovered, the minimum latency gets updated on the affected nodes, and the updates are propagated to neighboring nodes as before. Eventually, each flow will take the path(s) satisfying its latency to reach the destination.

Analysis and Comparisons: In the worst case, for each link failure, the deactivation/activation processes take $O(n)$ steps, which is the length of the longest simple path from s to d passing by every node in the network to propagate the deactivation/activation. Thus, they are guaranteed to terminate because it is limited by the length of the longest simple path included in the PrOG. On the contrary, link reversal techniques [23] can take $O(n^2)$ in the worst case for DAG recomputation. For techniques using graph search algorithms [7], they can take more than $O(n)$ as the packet may be sent to the same node multiple times after being sent to a certain child node, to be forwarded to the next child. Both techniques do not take into account latency constraints while trying to forward packets to the destination. Furthermore, we only need to reroute a small portion of packets forwarded to the deactivated portion of the PrOG, if they have not exceeded their latency deadlines; otherwise they will be dropped. New packets can be sent using other paths embedded in the PrOG satisfying the relaxed latency constraint. However, in [23], until the convergence process ends, packets bounce around in the network and loops may be formed. Also, in [7], packets may have multiple returns to sources, while "walking around" searching for a path to the destination, according

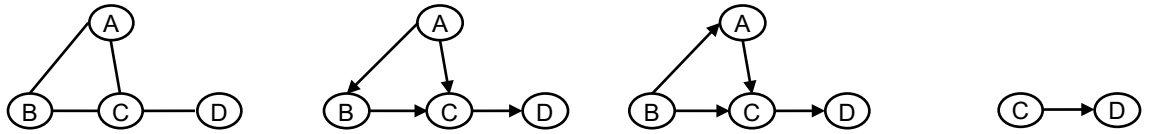


Figure 4: Count-To-Infinity

to the order specified by the graph search algorithm. Moreover, in Taproot multiple link failures can be handled simultaneously following the same deactivation/activation procedure, as there is no global exchange of information or routing table recalculation, only the affected portions of the PrOG are deactivated/activated, and local minimum latency recalculation and exchange may take place. All the (eligible) routes are already pre-installed, and each switch *dynamically* selects eligible routes for packet forwarding locally and independently.

5.3 Proofs of Correctness

We formally establish the correctness of Taproot through a series of theorems.

Theorem 2. Given a weighted graph $G = (V, E)$, a flow f from a source node s to a destination node d, with a latency requirement τ. The constructed τ-complete PrOG in Phase II contains all possible paths between s and d satisfying the required latency.

Proof. For a path $P := v_1(= s), v_2, \ldots, v_K(= d)$ to be included in the constructed PrOG, the latency of a path P must be no larger than τ (i.e., $\lambda(P) = \sum_{i=1}^{K-1} \lambda(v_i, v_{i+1}) \leq \tau$). To include this path, node v_i must add node v_{i+1} to its set of eligible nexthops. Let the minimum latency to reach the source node s from node v_i be $L^s_{v_i}$, and the minimum latency to reach the destination node d from node v_{i+1} without using node v_i as its nexthop be $L^d_{v_{i+1},\bar{v}_i}$. If $L^s_{v_i} + L^d_{v_{i+1},\bar{v}_i} + \lambda(v_i, v_{i+1}) \leq \tau$, then $L^d_{v_{i+1},\bar{v}_i} \leq \tau - L^s_{v_i} - \lambda(v_i, v_{i+1})$, which is the condition used in Phase II to check whether node v_{i+1} is an eligible nexthop to node v_i. If this condition is satisfied, this path is included in the constructed PrOG.

Theorem 3. Packet forwarding using the τ-complete PrOG guarantees packet delivery within τ using simple paths.

Proof. Non-simple paths means a node is visited more than once, which means the latency of the selected nexthop v is larger than the minimum latency of the current node u which violates the forwarding procedure explained above, so this nexthop is not eligible to use.

Theorem 4. Upon a link failure, if there is still a path between the source node s and the destination node d that satisfies the latency constraint $\tilde{\tau}_f$ (the relaxed latency in case of failures) in the original graph G, it will still be included after deactivating parts of the PrOG leading to failures.

Proof. Since the constructed τ-complete PrOG contains all possible paths that satisfy the relaxed latency constraint $\tilde{\tau}_f$, then if there is a path between s and d in the original graph G, it is still included in the constructed τ-complete PrOG. Upon link failure, only links leading to a sink node are deactivated, which means these links

Flow Table

Match	Action
src=A, dst=O, in_port = D	Group: 1
src=A, dst=O, in_port = G	Group: 1
src=A, dst=O, in_port = *	- Invoke activation/deactivation - Group: 1

Group Table

Group Id	Group Type	Action Bucket	State
1	fast failover	watch_group: 2, Group: 2 watch_group: 3, Group: 3 output: in_port Invoke deactivation	Active Active Active
2	select	Output: I Output: J	Active Active
3	select	Output: G	Active

Figure 5: Match-Action Rules for Switch F

can't be used to reach the destination, so they can't be part of the path that still connects s and d after the failure.

Lastly, we remark that while Taproot also relies on information exchange among neighboring nodes for local forwarding state updates, it does *not* suffer from the "count-to-infinity" problem plaguing a *purely* distributed routing algorithm such as Bellman-Ford. We use the classic "textbook" example topology shown in Fig. 4(a) to illustrate this point: In Bellman-Ford distant vector routing algorithm with poisonous reverse, if the link $C - D$ goes down, nodes A, B and C will get into a "count-to-infinity" loop before they realize that they all lose connectivity to D. Under Taproot, the PrOGs used for packet forwarding from A, B and C to D are depicted in Figs. 4(b)-(d) respectively. When the link $C - D$ goes down, C would immediately realize that it has lost connectivity to D; furthermore, as the outgoing links $A \rightarrow C$ and $B \rightarrow C$ are deactivated, both A and B also realize that they have lost connectivity to D. This simple example illustrate the power of *joint* centralized (static) route computation and distributed (dynamic) state adaptation and selection enabled by the SDN paradigm (*endowed with local state update capabilities*) which otherwise can not be achieved via a *purely centralized* or a *purely distributed* routing paradigm.

6 TAPROOT IMPLEMENTATION

We have implemented Taproot with two components: a controller and a programmable data plane to support new features introduced by Taproot. Our Taproot controller first computes the Latency-Complete PrOG for each flow with the relaxed latency requirement $\tilde{\tau}_f$ in case of failures, which already includes the paths satisfying τ_f used for normal operations. Then, for each switch, it determines the

priority of each outgoing link based on its min latency to destination, and translates these into a set of *match-action* rules to be installed at the relevant switches. Fig. 5 shows how the routing preorder in Fig. 3 is coded in *match-action* data plane rules for switch F.

For data plane operations, we extend Open vSwitch, which supports group table and group chaining, with some customized operations. When a packet is received, it is matched in the flow table based on its `source`, `destination`, and `ingress port (in_port)`. The corresponding action is to direct the packet to the group table (`Group 1`). We use separate groups for the outgoing links satisfying τ_f $\{I, J\}$ (`Group 2`), and the outgoing links satisfying $\tilde{\tau}_f$ $\{G\}$ (`Group 3`). The group type of `Group 1` is "fast failover" to give priority to links satisfying τ_f, as it executes the actions associated with the first live bucket determined by the state of its corresponding port or group. However, not all outgoing links are eligible for forwarding a packet due to the latency constraints. Thus, two extra header fields are added to each packet (`latency, latency_offset`), and we maintain at each switch the (per-destination) latency for its outgoing ports. Upon link failures affecting both outgoing links $F \rightarrow I$ & $F \rightarrow J$, `Group 2` becomes *InActive*, and F forwards its traffic to O through G. When G also fails, `Group 3` also becomes *InActive*, and F becomes a *sink* node, so it invokes the *deactivation* procedure described in the previous section. Each switch generates an *activation* or *deactivation* tags that can be piggybacked in data packets in response to link/node failure and recovery events, or sent as separate packets if there are no data packets. If a switch receives a packet that does not match the listed default incoming ports, then it is received from an outgoing link and is either an `activation` or a `deactivation` tag (indicated by the additional header field `link_state`). The switch updates the corresponding link state and latency accordingly, and still tries to forward the packet if it has other active outgoing links (indicated by the last rule in the Flow Table). When a failed link is up or an `activation` message is received, its corresponding bucket becomes active again and can be used.

7 EXPLORING TOPOLOGICAL DIVERSITY

In this section, we compare Taproot with existing path-based protection routing schemes and study how effective they are in leveraging the topological diversity inherent in the network to maintain connectivity between source (src) and destination (dst) nodes under various failure scenarios.

7.1 Experimental Setup

For each src-dst pair in a given topology, we consider the following path protection and link protection techniques: the primary path (the shortest path (d) between this src-dst pair) is protected via i) a *backup path* which is calculated as the next shortest path after removing the primary path from the topology, or ii) a set of *backup links* – constructed as following: for each link, we find an alternative disjoint path to the nexthop after removing this link. In comparison, we employ several Latency-Complete PrOGs with different τ's, starting from $\tau = d$ (the shortest path latency) to $\tau = d+1, d+2, d+3$, and so forth. We use 4 AS topologies from RocketFuel [36] with varying sizes: AS1221 (83 nodes, 131 edges), to AS3257 (151 nodes, 288 edges) as examples.

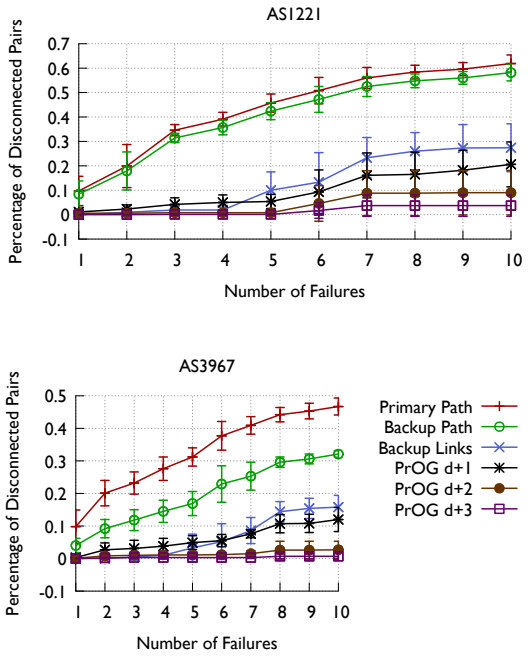


Figure 6: Percentage of Disconnected Pairs

7.2 Disconnection

We calculate the percentage of disconnected pairs (i.e., no path between src and dst) among all src-dst pairs in the topology to measure how each technique utilizes the topological diversity. For each routing scheme $\mathcal{R}$, a src-dst pair is considered *disconnected* with respect to $\mathcal{R}$ if the failed link(s) render all pre-computed (primary or backup) paths/rules *invalid*, i.e., packets from the src can no longer be routed to the dst even if a path still exists in the topology. The src-dst pair is considered disconnected if the failed edges: i) lie on the shortest path for primary path technique, ii) lie on both the shortest path and its alternative disjoint backup path, or iii) lie on any link of the shortest path and its backup disjoint path for backup links technique. For PrOG, the failed edges are removed, then we check if the src-dst pair is still connected. If the failed edge is bidirectional, we remove both directed edges.

From Fig. 6, we see that Latency-Complete PrOGs outperform existing path and link protection routing schemes under various failure scenarios; it is capable of effectively leveraging the topological diversity for resilient routing. We see that PrOGs with $\tau = d + 2$ are sufficient to reduce the number of disconnected src-dst pairs, and with $\tau = d + 3$, they can nearly avoid any disconnection even under 10 link failures. Moreover, not only Taproot utilizes the diversity in the network, but it also satisfies the latency constraints without the need to enumerate all paths satisfying this latency. Unlike the backup links technique which minimizes the number of disconnected pairs, but without any performance guarantee. However, our proposed solution prioritizes at each node the outgoing link with the smallest remaining cost to the destination, hence the latency experienced will increase incrementally based on the current state of failed link(s) and won't be arbitrary as the case of existing solutions.

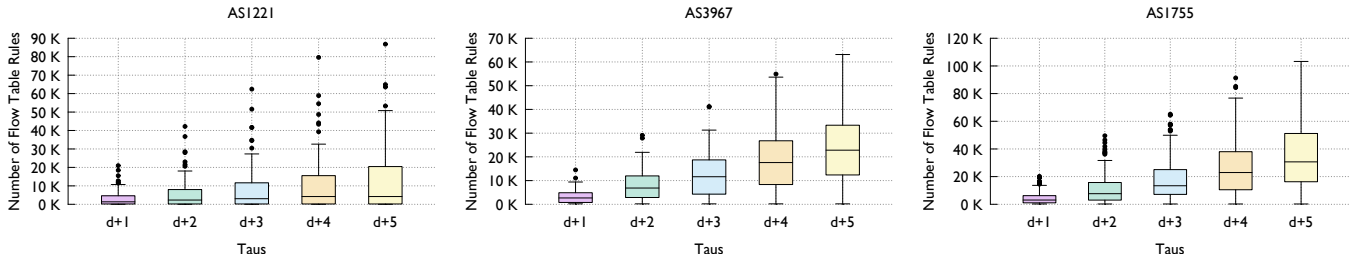


Figure 7: Flow Table Rules

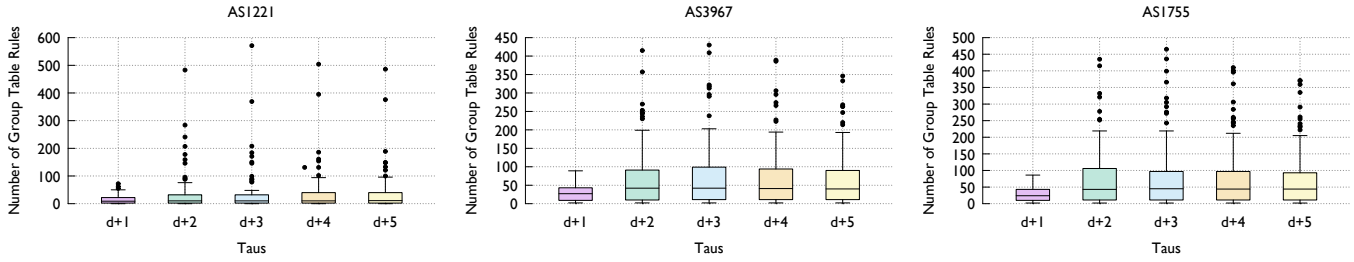


Figure 8: Group Table Rules

7.3 Maximum Number of Link Failures

We now consider the maximum number of link failures that Taproot can withstand before the network is partitioned. Using PrOGs with different values of τ, we calculate this metric by subtracting the length of the shortest path (i.e., src and dst are still connected) from total number of edges in each PrOG which gives us the maximum number of link failures without PrOG partitioning for each src-dst pair. Bidirectional edges are only counted once. Fig. 9 shows the box plot of the maximum number of link failures for each src-dst pair in AS3967 and AS3257. We remark that using PrOG with $\tau = d + 2$, Taproot can tolerate the failure of nearly 1/3 of the links in each topology. Even with $\tau = d$ (i.e., PrOG formed by the shortest paths only), Taproot can be resilient against multiple link failures as long as they do not partition the src and dst.

7.4 PrOG Overhead

The resiliency of Taproot becomes more robust with increasing the value of τ. This comes with the overhead of more rules to be installed in the switches. Fig. 7 and Fig. 8 show the box plots of the number of flow table and group table rules for each switch w.r.t. τ values. Using $\tau_F = d$ as initial latency and $\tilde{\tau}_f = d + 1, d + 2, ..$ in case of failures. Since at each switch the same outgoing links can be shared by multiple src-dst pairs, we aggregated the group table entries based on the unique sets of outgoing links. From these plots, we see that as the topology size increases, the number of rules increases. However, they are still below the limits specified in [1] (1M entries in wild card match flow tables and 10K entries for group table) in terms of TCAM entries. Part of the future work is to investigate different methods for efficiently representing and compacting the match-action flow table rules and state information in a way to minimize the space requirements.

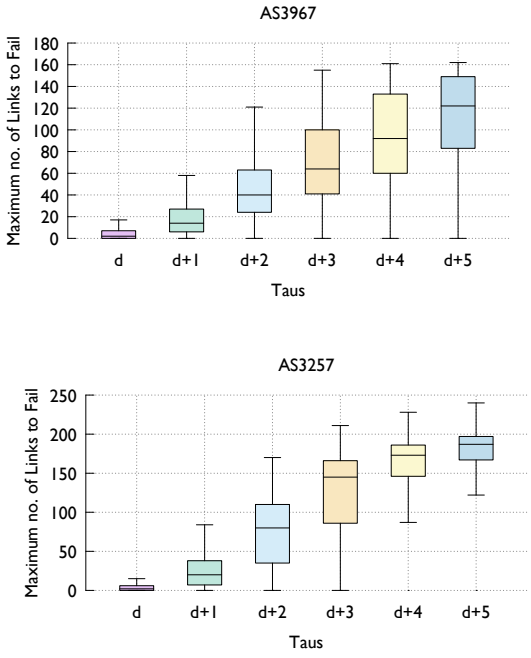


Figure 9: Max no. of Link Failures

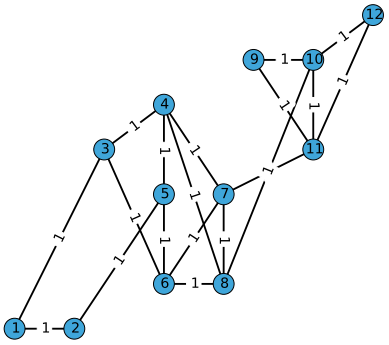


Figure 10: Google WAN topology

8 PERFORMANCE EVALUATION

In this section, we compare the performance of Taproot with DDC [23][6] using Mininet [2]. Although DDC is also robust against k arbitrary link/node failures as long as the network is not partitioned, it employs link reversals to reconstruct a new DAG when a node becomes sink (i.e., no outgoing link to destination). Moreover, it provides no latency guarantee either before or after failures. In contrast, using Latency-Complete PrOGs, Taproot has no convergence delay. Lastly, we consider a use case where we employ Taproot for the SDN control network.

8.1 Experimental Setup

For evaluation, we use Google's WAN topology [17] shown in Fig. 10 with a link capacity of 1 Gbps. A UDP traffic generator is used to send around $50K$ packets from $h1$ (attached to $s1$) to $h2$ (attached to $s12$), we use $\tau_f = 5$, and $\tilde{\tau}_f = 9$. The eligible outgoing links for each switch are prioritized according to the minimum latency to destination. We introduce 6 link failure sequentially: we fail one link first, then two, and so forth. After each link failure event, we measure the packet latency (in terms of no. of hops) for both Taproot and DDC, and convergence time and no. of link reversals for DDC.

8.2 Convergence Time and Latency

Table 1 shows the convergence time for the link reversal algorithm (DCC) to converge after link failures, which increases with the number of failures till reaching around 0.37 sec for 6 link failures involving around 182 link reversals. On the other side, there is no convergence in Taproot, since all paths satisfying the latency constraints are computed *a priori* and installed in the relevant switches. This is also reflected in the packet latency for each technique. In link reversal, packets experience longer latency till the convergence process is done, this is shown in Fig. 11. For example, in case of four link failures, packet latency reached around 17 hops, which is twice the length of the shortest path. In Taproot, the latency is always less than $\tilde{\tau}_f = 9$, regardless of the number of link failures, as the "surviving" rules result in packet forwarded along the next *available* (shortest) path(s).

8.3 Use Case: SDN Control Network

To show how Taproot quickly reacts to network failures, we simulate a data center network which consists of OpenFlow switches

[6]We obtained the code from the authors.

Table 1: Link Reversal Convergence

Links No.	Time (sec)	Link Reversals
1	0.03	5
2	0.03	7
3	0.1739	36
4	0.2486	80
5	0.3268	148
6	0.3741	182

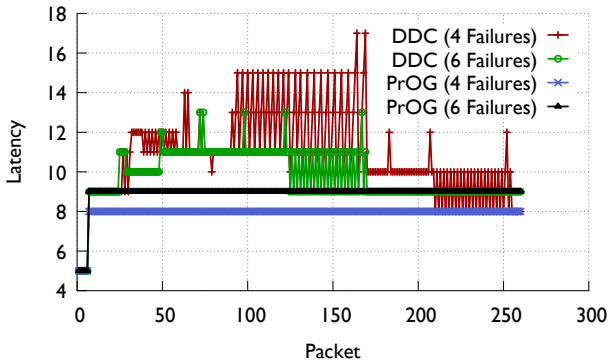


Figure 11: Latency

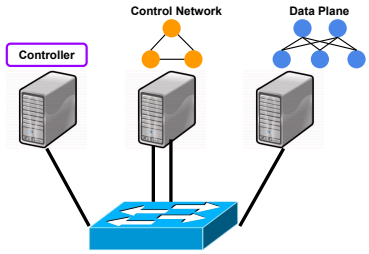


Figure 12: Real testbed setup for Taproot's use case of SDN control network.

and is controlled by a remote controller. A separate control network connects the controller and the data center network (i.e., data plane) to establish the "out-of-band" control channels. We set up the controller, the control network, and the data plane on three servers connected by a physical switch, as shown in Fig. 12. We launch multiple Open vSwitch (OVS) instances to build a leaf-spine data plane on the right server. Network namespaces are created as virtual hosts attached to the leaf switches. We use iPerf to generate flows between virtual hosts. The data plane is connected to a remote controller hosted on the left server through the control network running on the middle server. Three OVS instances are launched to form the control network, two of them attached to two physical network interfaces to carry the OpenFlow messages between the controller and the data plane. We demonstrate how fast Taproot reacts to failures in the control network in comparison to Spanning Tree Protocol (i.e., OVS instances are set in "standalone" (layer 2) mode with STP enabled) or OSPF (i.e., OVS instances run OSPF for layer-3 connectivity).

Consider one flow is sent between a pair of hosts in the data plane. The corresponding forwarding entries are periodically refreshed by the controller. After five seconds, a link in the control

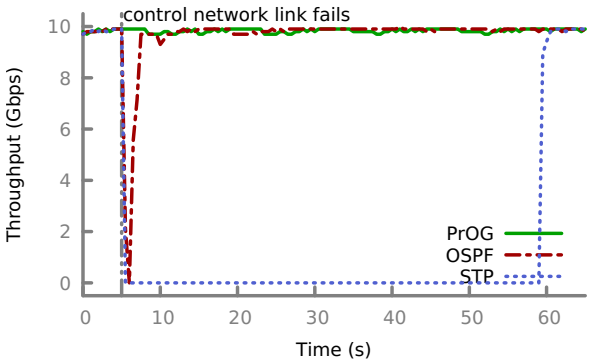


Figure 13: A flow's throughput in face of one control network link failure. Taproot outperforms STP and OSPF to recover from the control network failure.

network fails. Taproot immediately detects the failure and re-routes the control messages to a different path. Hence, the control channel is maintained without interruption, or involvement from the controller to change the routing rules. The data flow's throughput is not affected as shown in Fig. 13. If OSPF or STP is running in the control network instead, it takes a couple of seconds to tens of seconds for the control network to recover. During the convergence, the control channel between the controller and the data plane is broken, and due to the long convergence of the control network, the existing forwarding rules in the switches eventually expire and no forwarding entries can be added/re-installed. As a result, the throughput of the data flow drops to zero until the control channels get re-established and the forwarding rules re-installed.

9 DISCUSSION & FUTURE WORK

In this section, we elaborate on issues related to handling network topology changes and Taproot deployment.

• **Handling Topology Changes.** Operators may add new nodes to the network to increase its capacity. When a node X is added, it will calculate the minimum latency state of its outgoing links based on the information of its downstream neighbors and inform its upstream neighbors, which in turn would need to re-calculate their minimum latency state information and so on. This process continues till we reach the source node or a node whose minimum latency is not affected by this change. The other nodes in the network won't be affected. Hence, Taproot would run its control plane algorithm in the background, find the difference between the new Latency-Complete PrOG and the existing PrOG currently installed in the switches, and then only update the tables of the affected nodes. Similarly, the same procedure is followed if operators decide to remove a problematic device or a faulty node. The overhead as a result of this process depends on how far this added/removed node is from the source node which specifies how many upstream nodes are affected and require a new state update.

• **Installing OpenFlow Rules.** The issue of specifying the order of nodes to install OpenFlow rules is inherent in OpenFlow itself (see [18] for more details), and is orthogonal to our proposed solution. However, the same methods required to install OpenFlow tables for nodes along a single path would still apply to our proposed Taproot routing algorithm, but instead of having one node at each hop count from the source/destination node, we would have multiple nodes at the same level which can be handled in parallel at the same time.

• **Deployment Using Existing Hardware.** As presented above, Taproot requires some modifications to current OpenFlow specifications. It is however possible to implement Taproot *approximately* in existing hardware OpenFlow switches supporting Group Tables. For example, we can install a local controller at each switch to perform the local operations such as latency calculation and exchange, activation and deactivation processes where the activation/deactivation tags can be implemented using special VLAN tags/MPLS labels. Rules can be pre-installed in switches such that the outgoing links are prioritized according to their minimum latency to the destination. Hence, packets are always forwarded using those available outgoing links with the minimum latency, and they don't need to carry `latency` and `latency_offset` fields. This addresses the problem of the need to change the packet header and add bits to represent the `latency` and `latency_offset` values. However, this comes with the cost of packets may still be forwarded to destinations even though they have exceeded their latency requirements. For existing IP/MPLS networks, Taproot may also be implemented (approximately) in the data plane using the new Segment Routing (SR)[7] forwarding paradigm [3, 9] to support resilient routing. Exploring these possibilities will be topics for future research.

10 CONCLUSION

We have presented *Taproot*, a resilient diversity routing algorithm with bounded latency, based on the novel notion of *Latency-Complete* PrOGs. Given a network represented as a graph, we showed how Latency-Complete PrOGs can be constructed in $O(n^3)$ for all source-destination pairs, and established the correctness of the construction algorithms. We demonstrated how Taproot can be realized in SDN, where pre-computed Latency-Complete PrOGs are pre-installed in SDN switches as a set of *match-action* rules. We also detailed the SDN data plane operations in terms of the forwarding state maintained by each switch, the failure (and recovery) handling mechanisms via the *deactivation* (and *activation*) processes, the (*local*) information exchange process by neighboring switches to update the forwarding state in response to changes in the network, and packet forwarding mechanisms where switches select *eligible* outgoing links for packet forwarding based on the latency information carried in packet headers, and update the latency information as packets traverse them. We have implemented Taproot in OVS and conducted extensive simulations and experiments to demonstrate its superior performance over existing solutions. As a use case, we showed that Taproot can be employed to provide a "hardened" SDN control network against arbitrary failures.

ACKNOWLEDGMENTS

The research was supported in part by NSF under Grants CNS-1617729, CNS-1814322, CNS-1831140, CNS-1836772, CNS-1901103, CNS-2106771 and CCF-2123987.

[7] We note that just like SDN which is a new networking paradigm, SR introduces a new source-routing-based forwarding paradigm using MPLS label stacks and node SIDs for IP/MPLS networks. In itself, SR does not provide any resilient routing solutions.

REFERENCES

[1] 2021. *INOVISWITCH.* http://noviflow.com/products/noviswitch/, Last Accessed: 2021-08-25.

[2] 2021. Mininet. http://mininet.org/. Last Accessed: 2021-08-25.

[3] 2021. *Segment Routing Architecture - IETF Draft.* https://www.segment-routing.net/ietf, Last Accessed: 2021-08-25.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, Vol. 38. ACM, 63–74.

[5] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM*.

[6] Daniel O Awduche. 1999. MPLS and traffic engineering in IP networks. *IEEE Communications Magazine* 37, 12 (1999), 42–47.

[7] Michael Borokhovich, Liron Schiff, and Stefan Schmid. 2014. Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms. In *HotSDN*.

[8] C. Boutremans, G. Iannaccone, and C. Diot. 2002. Impact of link failures on VoIP performance. In *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*. ACM New York, NY, USA, 63–71.

[9] Dennis Cai, Anna Wielosz, and Songbin Wei. 2014. Evolve carrier Ethernet architecture with SDN and segment routing. In *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on*. IEEE, 1–6.

[10] Jeff Dean. 2009. Google: Designs, lessons and advice from building large distributed systems. *Keynote Speech at LADIS'09* (2009).

[11] Joan Feigenbaum, Brighten Godfrey, et al. 2012. On the resilience of routing tables. *arXiv preprint* (2012).

[12] Pierre François and Olivier Bonaventure. 2005. An evaluation of IP-based Fast Reroute Techniques. In *CoNEXT*.

[13] Eli M Gafni and Dimitri P Bertsekas. 1981. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Transactions on Communications* (1981).

[14] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM Computer Communication Review*, Vol. 41. ACM, 350–361.

[15] Keqiang He, Eric Rozner, Kanak Agarwal, et al. 2015. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *SIGCOMM*.

[16] G. Iannaccone, C. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. 2002. Analysis of link failures in an IP backbone. In *Proc. of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, 242.

[17] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 3–14.

[18] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. 2015. What You Need to Know About SDN Flow Tables. In *Passive and Active Measurement*, Jelena Mirkovic and Yong Liu (Eds.). Springer International Publishing, Cham, 347–359.

[19] Amund Kvalbein, Audun Fosselie Hansen, et al. 2006. Fast IP network recovery using multiple routing configurations. In *INFOCOM*.

[20] Kin-Wah Kwong, Lixin Gao, Roch Guérin, and Zhi-Li Zhang. 2011. On the feasibility and efficacy of protection routing in IP networks. *IEEE/ACM Transactions on Networking (ToN)* 19, 5 (2011), 1543–1556.

[21] Karthik Lakshminarayanan, Matthew Caesar, et al. 2007. Achieving convergence-free routing using failure-carrying packets. *ACM SIGCOMM CCR* (2007).

[22] S. Lee, Y. Yu, S. Nelakuditi, Z.-L. Zhang, and C.-N. Chuah. 2004. Proactive vs Reactive Approaches to Failure Resilient Routing. In *INFOCOM*.

[23] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. 2013. Ensuring Connectivity via Data Plane Mechanisms.. In *NSDI*. 113–126.

[24] Mahesh K Marina and Samir R Das. 2001. On-demand multipath distance vector routing in ad hoc networks. In *Network Protocols,*.

[25] Mahesh K Marina and Samir R Das. 2005. Routing in mobile ad hoc networks. In *Ad Hoc Networks*.

[26] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, and Christophe Diot. 2004. Characterization of failures in an IP backbone. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, Vol. 4. IEEE, 2307–2317.

[27] Srihari Nelakuditi, Sanghwan Lee, et al. 2005. Blacklist-Aided Forwarding in Static Multihop Wireless Networks. In *SECON*.

[28] Srihari Nelakuditi, Sanghwan Lee, et al. 2007. Fast local rerouting for handling transient link failures. *Transactions on Networking* (2007).

[29] Srihari Nelakuditi, Sanghwan Lee, Yinzhe Yu, and Zhi-Li Zhang. 2003. Failure Insensitive Routing for Ensuring Service Availability. In *IWQoS*.

[30] Radhika Niranjan Mysore, Andreas Pamboris, et al. 2009. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM CCR*.

[31] Ping Pan, George Swallow, and Alia Atlas. 2005. Fast reroute extensions to RSVP-TE for LSP tunnels.

[32] Eman Ramadan, Hesham Mekky, Braulio Dumba, and Zhi-Li Zhang. 2016. Adaptive resilient routing via preorders in SDN. In *Proceedings of the 4th Workshop on Distributed Cloud Computing*. ACM, 5.

[33] Aman Shaikh, Chris Isett, Albert Greenberg, Matthew Roughan, and Joel Gottlieb. 2002. A case study of OSPF behavior in a large enterprise network. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, 217–230.

[34] M. Shand and S. Bryant. 2009. *IP Fast Reroute Framework*. Internet Draft. Internet Engineering Task Force. (Work in progress).

[35] Ankit Singla, Chi-Yao Hong, et al. 2012. Jellyfish: Networking Data Centers Randomly. In *NSDI*.

[36] Neil Spring, Ratul Mahajan, and David Wetherall. 2002. Measuring ISP topologies with Rocketfuel. *ACM SIGCOMM Computer Communication Review* 32, 4 (2002), 133–145.

[37] Jack Tsai and Tim Moors. 2006. A review of multipath routing protocols: From wireless ad hoc to mesh networks. In *ACoRN*.

[38] Xiaowei Yang, David Clark, and Arthur W Berger. 2007. NIRA: a new inter-domain routing architecture. *ACM Transactions on Networking* (2007).