Supporting Emerging Applications With Low-Latency Failover in P4

Roshan Sedar
Université catholique de Louvain
Belgium

Michael Borokhovich
Independent Researcher
USA

Marco Chiesa
KTH Royal Institute of Technology
Sweden

Gianni Antichi
Queen Mary University of London
UK

Stefan Schmid
University of Vienna
Austria

ABSTRACT

Emerging applications expect fast turn-around from in-network failover mechanisms. This paper starts exploring the design space for supporting high availability and low latency using fast reroute in programmable data planes. In particular, we present a primitive for supporting well-known fast reroute mechanisms that is both efficient in terms of packet processing latency, memory requirements, and switch throughput.

CCS CONCEPTS

• **Networks** → **Network algorithms**; *Network design principles*; *Network protocols*;

KEYWORDS

Fast failover, Low latency, High availability, Programmable data planes

ACM Reference Format:
Roshan Sedar, Michael Borokhovich, Marco Chiesa, Gianni Antichi, and Stefan Schmid. 2018. Supporting Emerging Applications With Low-Latency Failover in P4. In *NEAT'18: ACM SIGCOMM 2018 Workshop on Networking for Emerging Applications and Technologies , August 20, 2018, Budapest, Hungary.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3229574.3229580

1 INTRODUCTION

Novel applications, e.g., in the context of industrial or tactile networks, or applications expected to emerge around 5G, require very low latency and deterministic packet delivery and high availability. *Fast Re-Route* (FRR) is a widely deployed networking mechanism providing such ultra-reliable and low-latency packet delivery, hence increasing the robustness of a network to unexpected failures, by pro-actively provisioning the switches with backup forwarding rules. When a switch detects a failure, i.e., defecting link or port, it quickly detours the affected packets using its own local backup rules. This allows a network to preserve connectivity without having to wait for the slower control plane to reconverge.

NEAT'18, August 20, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5907-8/18/08…$15.00
https://doi.org/10.1145/3229574.3229580

Lately, programmable data planes [6], PDP, emerged as a promising solution to further enrich the capabilities of networks by allowing network managers to deploy customized packet processing algorithms. Recent studies and industry startups have indeed shown the benefits of arming network managers with data plane programming capabilities for a variety of use cases including monitoring [17], traffic load-balancing [12], and more [3]. Yet, very little is known today about how to implement FRR mechanisms in such systems, as the P4 programming language does not provide in-built support for FRR, as confirmed to us on the P4 mailing list [1].

In this paper, we explore the design space for supporting fast reroute in PDP. We argue that an FRR primitive should meet at least the following four requirements: (1) *efficient rerouting*, i.e., once a failure is locally detected, packets affected by that failure should be rerouted to an alternate active port as fast as possible so as to avoid packet drops, (2) *no unintended throughput degradation*, i.e., packets should only be eventually dropped if the incoming rate of packets is higher than the outgoing port rate, (3) *low forwarding latency*, i.e., packet processing latency should not increase when ports fail, which we see as a critical requirement for latency-critical applications, and (4) *low switch memory requirements*. In this context, we devise and implement an FRR primitive for a P4 switch that supports multiple types of existing OpenFlow-based FRR mechanisms while meeting all the above four requirements. Our solution detects the first active port in a parallel manner by relying on *just one* lookup in a TCAM. This is particularly critical in those latency-critical networks where switches have high port density, such as rack-scale ones [13]. As TCAMs are expensive resources, we observe that our FRR primitive in a k-port switch only requires $2k-1$ TCAM entries, each with $3k-1$ matchable bits, to support most FRR mechanisms. This is just 95 TCAM entries in a 48-port switch. We implement our primitive on a P4 software switch and we show that it outperforms a naïve approach based on packet recirculation. We, therefore, perform microbenchmarks on a physical switch with TCAM support and we evaluate our P4-based implementation of existing FRR mechanisms on data center and empirically-derived ISP networks in terms of path lengths.

The main contributions of the paper are the following:

- We explore the design space alongside the trade-offs of implementing FRR mechanisms in PDP.
- We propose a new primitive that can be adopted as building block for implementing a number of FRR algorithms, such as Rotor-Router (RR), Depth First Search (DFS), and Breadth First Search (BFS) [4].

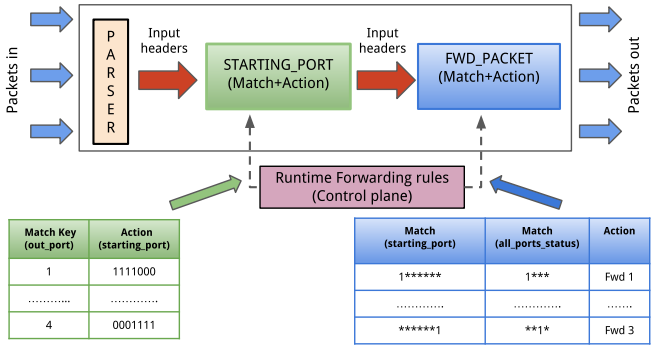


Figure 1: PISA and efficient FRR implementation.

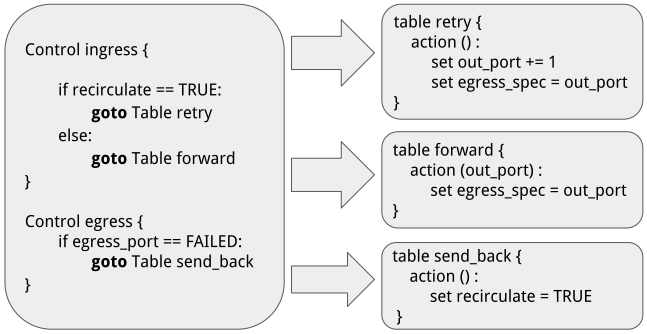


Figure 2: Naïve approach with recirculation.

- We implement the primitive alongside the mentioned FRR algorithms and evaluate them in both a data center and empirically-derived ISP network scenarios.

2 TOWARDS A UNIFIED PRIMITIVE

P4 [6] is a programming language specifically designed to program data plane packet pipelines based on a match/action architecture. While the P4 language is target-independent, i.e., it abstracts from the specific hardware characteristics of a switch, a P4 compiler translates high-level P4 programs into target-dependent switch configurations.

The top part of Figure 1 represents the typical life of the packet inside a programmable device. Packet data alongside metadata information, both switch-specific and user-defined, follow such a pathway. The parser first identifies the various headers of interest for the processing. Then, a set of match/action tables dictate the operations to be performed. Upon the first match within a table, the table executes its corresponding action(s) on the packet [10]. It is worth noting that P4 does not dictate how match tables are mapped to TCAMs, SRAMs, and DRAMs. Clearly, different memory types strike different trade-offs in terms of cost, energy consumption, and latency. The P4 compiler is responsible to map match tables to physical memories based on the target switch.

So far, such an architecture demonstrated a lot of flexibility by allowing customized packet processing algorithms. However, P4 does not have in-built support for *FRR groups*, where the action associated to a group consists of a sequence of ports such that the packet is forwarded through the first non-failed port. Instead, from our discussion with P4 developers, the implementation of FRR groups is left to the operator [1].

2.1 Challenges and Naïve Approaches

Implementing an FRR primitive in P4 is far from being trivial. Without specific built-in FRR hardware support within the switch devices, operators have to rely only on the match/action processing to enable quick packet detouring upon failure detection.

Packet recirculation. One simple way to implement FRR is to recirculate a packet until an active outgoing port is found. This operation entails storing in the packet metadata information about the failed port through which a packet could not be forwarded. Consider for instance an FRR mechanism that indexes all the switches' ports and each switch first tries to send a packet through a port with index i and, if that port is failed, tries with port $i+1$, $i+2$, and so on, modulo the number of ports. We show in Figure 2 how this FRR can be implemented in P4 using packet recirculation. In the CONTROL INGRESS flow, we verify whether the incoming packet has been recirculated or not.[1] If the packet has not been recirculated, we apply the action from table FORWARD, which simply sets the outgoing port for that packet according to the primary forwarding tables. If the packet has instead been recirculated, we update the egress port of the packet that has been recirculated through table RETRY with the next one in the order. Regardless of whether the packet has been recirculated or not, we try to send the packet to its designated egress port. If that port is not active, then the packet is recirculated through table SEND_BACK.

There are few drawbacks to this implementation: when a packet is recirculated, it will add an additional bandwidth overhead on the switch capacity and increase the packet processing latency. In the former case, it can also result in sort of self-induced incasts leading to network performance degradation [19]. In the latter case, the processing might cause delays in the order of μs, well in the range which has been shown to have negative impact on performance for several common datacenter applications [20].

Multiple stages. Another approach to implementing the above "circular" FRR in P4, would be to iteratively check through a specific sequence of outgoing ports which one is the first active. Such a solution would increase the latency of the packet forwarding. The overhead would be proportional to the number of the backup ports specified in the FRR groups, which can be as high as the total number of switch ports for graph exploration and spanning-tree based FRR mechanisms. The drawback would be then similar to the previous case.[2]

[1] For the sake of space, we assume we can test the state of a variable.
[2] We do not have access to any proprietary Tofino P4 compilers. Yet, we believe that any naïve parallelization of "circular" FRR group would require a TCAM space that is quadratic in the number of ports instead of linear space.

To start from port **1**, set metadata to starting_port = **1111000**
To start from port **2**, set metadata to starting_port = **0111100**
To start from port **3**, set metadata to starting_port = **0011110**
To start from port **4**, set metadata to starting_port = **0001111**

Figure 3: Primitive in action for a 4-port switch

2.2 An FRR Primitive for P4

To overcome the limitations described in the previous section, we design and implement a novel FRR primitive in P4. Our primitive does not incur any bandwidth overhead and computes the first active outgoing port with a single ternary-match search in a TCAM, thus achieving low latency. Specifically, our primitive is expressive enough to support a variety of FRR mechanisms without requiring any packet recirculation or use of multiple stages and using limited TCAM space (linear in the number of ports). We now discuss in more details how the primitive works. For the sake of simplicity, we assume an operator only wants to specify FRR groups that are *circular*, i.e., there exists a unique circular ordering of the ports where each port is assigned a specific index (as in Sect. 2.1). We discuss how to implement arbitrary FRR groups later. The design choice to implement circular FRR groups draws inspirations from previous work in FRR [4, 8].

Primitive design. Our primitive relies on two tables: STARTING_PORT and FWD_PACKET (see bottom part of Figure 1). Through STARTING_PORT (green table), we match the designated outgoing port and generate the necessary state that we will use in FWD_PACKET to find the first active outgoing port that follows the starting port in a circular order. Namely, given a switch with k outgoing ports, we map the switch ports to $2k-1$ bits as follows: the ordered ports to the first k bits and then the same ordered ports, except the last one, to the remaining $k-1$ bits.

The content of the $2k-1$ bits depends on the starting port from which a packet should be sent. Namely, we set to 1 the first bit mapped to the starting port and its subsequent $k-1$ bits. Intuitively, this allows us to denote which ports should be used for forwarding and to identify the starting port. Through FWD_PACKET (blue table), we identify the first available active port starting from the selected initial port. To do this, our FRR primitive assumes that the switch stores the port status in a register with k bits, where each bit determines whether a port is active (set to 1) or not (set to 0). The FWD_PACKET matches packets based on the starting_port output and the port status as shown in Table 1. Namely, for each bit of starting_port output, we have one entry in the table that verifies if that bit is set to 1, i.e., a packet should be sent to its associated port, and verifies if the status of that port is active. These entries are ordered by priority in a TCAM, with the top entries having higher priority, i.e., the first bit of the starting_port output has the highest priority.

Example. Consider Figure 3, where we assume to apply our mechanisms to a switch with four ports. We need 7 bits for the starting_port field, where the first four bits index ports from 1 to 4 and the remaining 3 bits index ports from 1 to 3. If a packet should be sent to port 3, then we will store in the metadata of the packet the value 0011110. This value and the port status is fed as input to the FWD_PACKET table. The first entry of that table matches any packet that has the first bit of the starting_port output set to one only if port 1 is active. The second entry matches any packet that has the second bit of the starting_port output set to one only if port 2 is active, and so on. If we assume both ports 3 and 4 have failed, then our packet will match the $5th$ entry (Table 1) and the packet will be forwarded through port 1.

Match Key {starting_port}	Match Key {all_ports_status}	Action
1******	1***	Fwd 1
*1*****	*1**	Fwd 2
1**	**1*	Fwd 3
1	***1	Fwd 4
****1**	1***	Fwd 1
*****1*	*1**	Fwd 2
******1	**1*	Fwd 3

Table 1: Forwarding table for 4-port switch.

Discussion. Our FRR primitive has several advantages with respect to the naïve approaches described before. First, it does not require packet recirculation, thus avoiding bandwidth overhead and latency increase. Second, it requires just one atomic search in a wildcard match table, which is a fast operation that can be performed with one clock-cycle with a TCAM. Third, the required TCAM space is limited. In a switch with 48 ports, we would require 95 TCAM entries with 143 matchable bits, i.e., starting_port + all_ports_status. While the reroute expressiveness of our FRR primitive is limited to circular FRR groups, we note that: (1) if an operator only specifies one backup port, instead of a circular order of all ports, then our FRR primitive can fully support any possible backup port, even without following the circular order, (2) our primitive supports a large variety of FRR mechanisms (see next section), and (3) we can eventually add a group identifier that can represent different mappings, thus allowing an operator to implement any arbitrary FRR group.

2.3 One Primitive – Many Mechanisms

This section shows how some FRR mechanisms based on graph-exploration techniques [4, 5] use circular FRR groups and can be realized with our primitive.

1) The Rotor-router (RR) mechanism: performs simply "deterministic random walk" upon detecting a failure in the network. If a packet hits a failed port, an alternative forwarding port is sought in modulo (round-robin) fashion. RR leverages header tags to store, for each switch in the network, the last link (i.e., port index) through which the packet has been sent. When a packet returns to the same switch, it is forwarded to the next non-failed outgoing port, that is, a circular FRR. Namely, for each switch, RR stores a counter value (*modulo counter*) $c(v)$: whenever a packet arrives at switch v with the counter $c(v)$, the packet is forwarded to the port $(c(v)+1)$

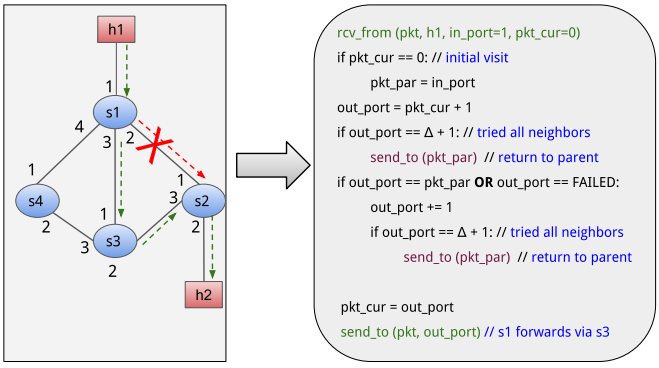


Figure 4: Packet traversal (from h1 to h2) in DFS FRR mechanism. The green arrows indicate the DFS failover path.

mod Δ (assuming this port is active and if not, the next one is tried) and at the same time, counter $c(v)$ is updated with the new value.
2) The Depth First Search (DFS) mechanism: it provides similar results to rotor-router for long rerouting paths. DFS instead reduces routing path lengths, by constraining the destination search to a depth-first spanning tree[3]. Concretely, upon a link failure, DFS explores alternative routes in a depth-first fashion, implicitly constructing a spanning tree. Towards this goal, for each node v_i, DFS reserves a certain part of the packet header $pkt.v_i.par$, to store the parent of v_i: the node from which the packet was first received. DFS also reserves a place in the header to store $pkt.v_i.cur$, which is used to remember what neighbor has been traversed.

Figure 4 illustrates packet traversal in the DFS FRR mechanism. Upon detecting a port failure, the switch enters the DFS failover routing mode. Whenever a packet reaches a switch for the first time, the value of *pkt_cur* for that switch is zero, and the incoming port is assigned as parent (*pkt_par*) for that switch. In the successive visits, the value of *pkt_cur* is being incremented gradually to ensure that all the other neighbors are visited one by one. If a link towards a neighbor is failed, then the next neighbor (unless it is the parent node) is chosen until an active link is found. This circular search can be implemented with a circular FRR group (but the parent port must be visited last). Upon traversing all the neighbors, the packet then is sent back to the parent i.e., *pkt_par*. We defer the reader to the pseudo-code presented in Algorithm 2 in [4] for further details.
3) The Breadth First Search (BFS) mechanism: is also based on spanning trees that are computed on-the-fly after failures happen. However, BFS employs breadth-first traversal technique. BFS controls the depth at which the destination d is searched in each phase. Each node v (other than the source) behaves as a leaf at the first time a packet reaches it: v returns the packet right back to its parent (i.e., sends it to the port from which it was received). On any other occasion, v behaves as a DFS node, i.e., trying all its neighbors and only then returning the packet to the parent. When testing all its neighbors, BFS leverages a circular FRR group to find the first active neighbor. We defer the reader to the pseudo-code presented in Algorithm 3 in [5] for further details.

[3]The cover time of rotor-router is $\Theta(Edges \times Diameter)$ (see [11]) while DFS will take only $O(Edges)$ since each edge is traversed at most twice.

3 PROOF-OF-CONCEPT

In this section, we describe how the previously mentioned graph-exploration FRR mechanisms can be implemented in P4 using our primitive. Our implementations are based on the P4_14 [10] specification and the evaluation was done on the P4-BMv2 [9] software switch. To the best of our knowledge, today there is no P4 abstraction for the detection of link/port failures. Hence, we rely on P4 registers for representing ports status and use them in our programs. How such registers are updated depends on the specific target packet-processing system and it is outside the scope of this paper. In our evaluation, we modify the state of these registers, called *all_port_status*, using the P4 local agent. The value of these registers is matched in the FWD_PACKET table, as explained in the previous section (§2.2).

3.1 Enabling Fast Reroute Algorithms

In this section, we only, for the sake of space, discuss the DFS mechanism implementation and explain how to incorporate the FRR primitive in the implementation.
Implementing DFS with our primitive. Our primitive cannot be applied naïvely since we need to guarantee that a packet is sent to a parent node only after having been sent through all the other links. This is non-trivial since the parent node is only known after the failures happen. Our DFS implementation starts from extracting the header fields *pkt_start*, *pkt_cur*, and *pkt_par* into the metadata fields. These extracted header fields are then passed to ingress match/action pipeline, where the packet processing enters the DFS failover mode (if *pkt_start* is set to 1), otherwise operates in default routing mode (line 2-3 in Fig. 5b). The P4 program for DFS in Figure 5 illustrates the packet processing pipeline at the ingress, which determines the outgoing port that the incoming packet is destined to. Whenever a packet arrives at a switch for the very first time, while in DFS failover mode (line 1-24 in Fig. 5a), *pkt_cur* is equal to zero. This is handled by the `curr_eq_zero` table that matches on the *pkt_cur* field and sets the incoming port as *pkt_par* for this switch. The following visits of the same packet, due to failures, will be forwarded through the next working port. This is performed by the `curr_neq_zero` table which increments the current value of *pkt_cur* by one. This incremented *pkt_cur* value stores the *out_port* variable as the next forwarding port. This repeats until the *out_port* value surpasses the number of ports represented by *PORTS* (line 6 and 14 in Fig. 5a); subsequently, the packet is sent back to the parent where it came from for the first time.

We apply the FRR primitive (red box at line 10-11 in Fig. 5a) initially within the DFS mechanism on the value *out_port*, to make sure that the chosen *out_port* is not the parent port (line 12): match+action lookups at both `set_starting_port` and `check_outport_status` tables ensure that any failed ports are skipped as well as the parent, until all the neighbours are tried (line 12-13). To this end, we keep a boolean type variable i.e., a *is_completed* metadata field, initialized to 1 at the `reached_depth` table (line 7 in Fig. 5a) to skip match+action lookups from line 9-18. Finally, we apply the FRR primitive (red box line 17-18 in Fig. 5b) where the packets are first matched on *out_port* at the `starting_port` table to retrieve "starting_port" metadata. In the

```
1 control ctrl_dfs_routing { /* start dfs */
2     if(local_metadata.pkt_cur == 0) {
3             apply(curr_eq_zero); /* set ingress as the parent */
4     }
5     apply(curr_neq_zero);
6     if(local_metadata.out_port == PORTS + 1) { /* tried all neighbours */
7             apply(reached_depth); /* out_port <- pkt_par &&& return to parent */
8     }
9     if(local_metadata.is_completed == 0) {

            /* we apply primitive here to "out_port" to avoid sending to parent before tried all neighbours */
10          apply(set_starting_port); /* set starting port metadata */
11          apply(check_outport_status); /* find matching out_port  */

12          if(local_metadata.out_port == local_metadata.pkt_par) { /*skip parent unless tried all neighbours*/
13                  apply(jump_to_next); /* try next port */
14                  if(local_metadata.out_port == PORTS+1) { /* tried all neighbours */
15                          apply(send_to_parent); /* out_port <- pkt_par &&& return to parent */
16                  }
17          }
18  }
19  if(local_metadata.is_completed == 1) { /* parent could be zero while switching from... */
20          if(local_metadata.out_port == 0) { /*...default routing to dfs failover mode */
21                  apply(out_eq_zero); /* try next forwarding port */
22          }
23    }
24 }
```

(a) DFS Mechanism.

```
1 control ingress {
2     if(local_metadata.pkt_start == 0)   {
              /* apply default routing mode */
3             apply(default_route);
4             if(local_metadata.out_port == 0) {
                  /* default out_port is not available */
                  /* failover kicks in (pkt_start <- 1) */
5                     apply(start_dfs);
6         }
7     }
8     else
9     {
              /* dfs contro flow */
10            ctrl_dfs_routing();
11    }
12    if(local_metadata.is_completed == 1) {
              /* tried all neighbours */
              /* return to parent */
13            apply(forward_to_parent);
14    }
15    else
16    {

              /* FRR primitive applies here to set "egress_spec" */
17            apply(starting_port);  /* set starting port metadata */
18            apply(forward_pkt); /* set egress spec to forward packets */

19    }
20 }

21  control egress {}
```

(b) Control Flow Instructions.

Figure 5: The P4 program for DFS FRR Mechanism.

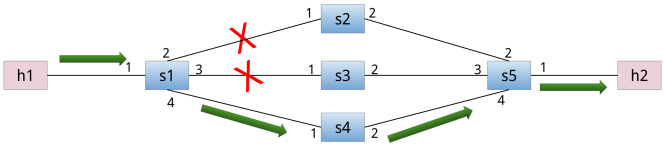
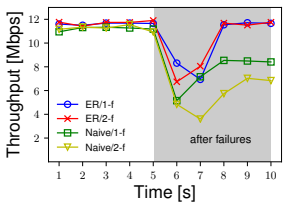
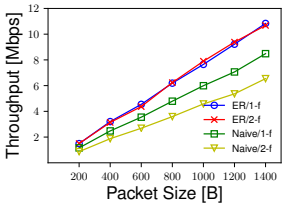


Figure 6: Topology used in evaluating efficiency between our FRR primitive and naïve approach.



(a) Throughput over time (TCP)



(b) Throughput over packet size (TCP)

Figure 7: Average throughput of our primitive (ER) and naïve approach under one (1-f) or 2 (2-f) failures.

next step, we match on both "starting_port" and "all_ports_status" at the `forward_pkt` table (line 17-18 in Fig. 5b), to set the "egress_spec". We made our source codes available to the public [2].

4 EVALUATION

We first compare our primitive to the naïve approach based on packet recirculation, showing a drastic performance improvement under one and two link failures.

Microbenchmark. We first evaluated our primitive on a NoviFlow 1132 switch [15] provisioned with TCAM.[4] We injected 10Gbps of traffic through one port and manually simulated one/two link failures. Since our approach uses a single TCAM access to find an active port, as soon as the all_ports_status metadata is updated, throughput returns to 10Gbps (packet latency is unaffected). The rest of the experiments will be performed in an emulated environment.

Setup and Scenario. For the evaluation, we built the topology of Figure 6 in Mininet, consisting of five P4-BMv2 switches (from $s1$ to $s5$) and two hosts ($h1$ and $h2$). The experiment lasts ten seconds, during which we generate a TCP flow using *iperf* from $h1$ to $h2$. During the first 5 seconds, this flow is routed through the default path (s_1, s_2, s_5). After five seconds, we either fail (s_1, s_2) or both (s_1, s_2) and (s_1, s_3) and measure the flow throughput for the remaining 5 seconds.

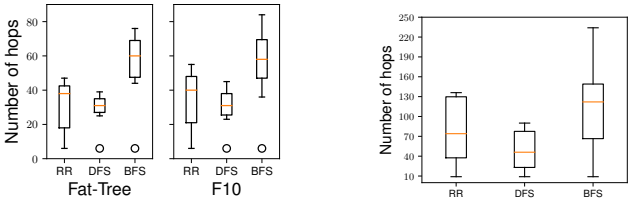
Performance Overhead. Fig. 7a shows the average traffic throughput of the two approaches for one and two links failures. We see that recirculating packets hinders performance as it creates an "incast" in the switch. Fig. 7b shows the average throughput after failures for different packet sizes, confirming the benefits of our approach.

4.1 Failover Path Lengths During Failures

Topologies. We used two datacenter network topologies, i.e., 3-tier (leaf-aggregate-spine) Fat-Tree, F10 [14], and Rocketfuel (AS3967) with 79 nodes and 149 links [18]. We used 20 switches with 4 ports and 32 links. In contrast to a FatTree, an F10 topology rewires some of the aggregate-to-spine links, which reduces the length of the backup paths. We numbered ports arbitrarily and chose random source-destination pairs in our experiments.

Performance of FRR mechanisms. Using our P4 implementation of RR, DFS, and BFS, we evaluated the trade-offs among these mechanisms in the resulting packet path lengths under different random failure scenarios. Fig. 8a shows for a FatTree and F10 topologies

[4] Even if NoviFlow supports OpenFlow FRR groups, we are only interested in evaluating our implementation in a real physical TCAM. NoviFlow switch does not support packet recirculation nor P4 packet-processing.



(a) Datacenter networks **(b) RocketFuel**

Figure 8: Number of hops to reach the destination.

the path lengths to reach the destination when a different number of random links fail. Both BFS and RR algorithms lead to a higher number of hops to reach the destination compared to the DFS algorithm (Fig. 8a). Our primitive correctly forwarded packets upon any random failure to the designated backup ports according to the circular order described in Sec. 2. The same was observed for the RocketFuel topology experiment (Fig. 8b).

5 CONCLUSION

We presented the first design, implementation, and evaluation of a low-latency rerouting mechanism for PDP: a basic component of emerging ultra-reliable applications. Our approach relies on a simple primitive which can be used as part of many existing failover algorithms, e.g., based on arborescences [7] or based on block designs [16], allowing to port other state-of-the-art failover algorithms to PDP.

Acknowledgments. We would like to thank Amedeo Sapio, Liron Schiff, and Mathieu Jadin for the early discussions about this work. This research is (in part) supported by the UK's Engineering and Physical Sciences Research Council (EPSRC) under the EARL: sdn EnAbled MeasuRement for alL project (Project Reference EP/P025374/1).

REFERENCES

[1] 2016. P4-Dev. http://lists.p4.org/pipermail/p4-dev_lists.p4.org/2016-May/000285.html.
[2] 2018. P4-Sources. https://bitbucket.org/roshanms/p4-frr.
[3] Barefoot. 2018. In-Network DDoS Detection. https://barefootnetworks.com/use-cases/in-nw-DDoS-detection/.
[4] Borokhovich, M. et al. 2014. Provable Data Plane Connectivity with Local Fast Failover: Introducing Openflow Graph Algorithms. In *HotSDN*. ACM.
[5] Borokhovich, M. et al. 2018. The Show Must Go On: Fundamental Data Plane Connectivity Services for Dependable SDNs. In *COMCOM*. Elsevier.
[6] Bosshart, P. et al. 2014. P4: Programming Protocol-independent Packet Processors. In *CCR, Volume: 44, Issue: 3*. ACM.
[7] Chiesa, M. et al. 2016. On the Resiliency of Randomized Routing Against Multiple Edge Failures. In *ICALP*. Leibniz.
[8] Chiesa, M. et al. 2016. The Quest for Resilient (Static) Forwarding Tables. In *INFOCOM*. IEEE.
[9] P4 Language Consortium. 2018. Behavioral Model (BMv2). https://github.com/p4lang/behavioral-model.
[10] P4 Language Consortium. 2018. P4 Language Specification. https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf.
[11] Dereniowski, D. et al. 2016. Bounds on the Cover Time of Parallel Rotor Walks. In *JCSS*. Elsevier.
[12] Katta, N. et al. 2017. Clove: Congestion-Aware Load Balancing at the Virtual Edge. In *CoNEXT*. ACM.
[13] Legtchenko, S. et al. 2016. XFabric: A Reconfigurable In-rack Network for Rack-scale Computers. In *NSDI*. USENIX.
[14] Liu, V. et al. 2013. F10: A Fault-tolerant Engineered Network. In *NSDI*. USENIX.
[15] NOVIFLOW. 2013. NoviSwitch 1132 High Performance OpenFlow Switch. https://noviflow.com/wp-content/uploads/NoviSwitch-1132-Datasheet-V2_1.pdf.
[16] Pignolet, Y. A. et al. 2017. Load-Optimal Local Fast Rerouting for Resilient Networks. In *DSN*. IEEE.
[17] Sivaraman, V. et al. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *SOSR*. ACM.
[18] Spring, N. et al. 2002. Measuring ISP topologies with Rocketfuel. In *CCR, Volume: 32, Issue: 4*. ACM.
[19] Yanpei, C. et al. 2012. Understanding tcp incast and its implications for big data workloads. In *Techincal Report*.
[20] Zilberman, N. et al. 2017. Where Has My Time Gone?. In *PAM*. Springer.