SlickFlow: Resilient Source Routing in Data Center Networks Unlocked by OpenFlow

Ramon Marques Ramos
and Magnos Martinello
Federal University of Espírito Santo (UFES)
Vitória - ES - Brazil
Email: ramonmramos@inf.ufes.br, magnos@inf.ufes.br

Christian Esteve Rothenberg
School of Electrical and Computer Engineering (FEEC)
University of Campinas (UNICAMP)
Campinas - SP - Brazil
Email: chesteve@dca.fee.unicamp.com.br

***Abstract*—Recent proposals on Data Center Networks (DCN) are based on centralized control and a logical network fabric following a well-controlled baseline topology. The architectural split of control and data planes and the new control plane abstractions have been touted as Software-Defined Networking (SDN), where the OpenFlow protocol is one common choice for the standardized programmatic interface to data plane devices. In this context, source routing has been proposed as a way to provide scalability, forwarding flexibility and simplicity in the data plane. One major caveat of source routing is network failure events, which require informing the source node and can take at least on the order of one RTT to the controller. This paper presents SlickFlow, a resilient source routing approach implemented with OpenFlow that allows fast failure recovery by combining source routing with alternative path information carried in the packet header. A primary and alternative paths are compactly encoded as a sequence of segments written in packet header fields. Under the presence of failures along a primary path, packets can be rerouted to alternative paths by the switches themselves without involving the controller. We evaluate SlickFlow on a prototype implementation based on Open vSwitch and demonstrate its effectiveness in a Mininet emulated scenario for fat-tree, BCube, and DCell topologies.**

I. Introduction

Data Center Networks (DCN) are an essential component of today's cloud-oriented Internet infrastructure. Proper planning of the data center infrastructure design is critical, where performance, resiliency, and scalability under certain cost constraints require careful considerations [6]. Data centers are growing in size and importance as many enterprises are migrating their application and data to the cloud. To achieve the benefits of resource sharing, cloud data centers must scale to very large sizes at marginal cost increases.

Continued growth of DCN requires a reliable infrastructure because interruptions in services can have significant consequences to enterprises [2], [19]. An increasingly portion of Internet traffic is based on the data communication and processing that takes place within data center networks. However, traditional fixed network approaches are simply not designed to meet the current needs of a virtualized environment and are not flexible enough to support the changing demands.

Recently, a promisingly networking technology referred to as Software Defined Network (SDN) is emerging. SDN is an architectural proposition based on some key attributes, including: separation of data and control planes; a uniform vendor-agnostic interface to the forwarding engine (i.e. OpenFlow [21]); a logically centralized control plane; and the ability to virtualize the underlying physical network [13].

This paper proposes SlickFlow, an OpenFlow-based fault tolerent routing design that allows switches to recover from failures by specifying alternative paths in the packet headers. The approach relies on defining at the source the primary route and alternative paths, enabling packets to skip from failures without the intervention of centralized controllers. The key idea of SlickFlow is to represent the paths as a sequence of segments that will be used by each switch to perform the forwarding operation. A segment carries the information of the next node of the primary path, and an alternative path related to the current node. If the primary path is not available, then the current switch rebuilds the header and forwards the packet to the alternative path.

Route computation is performed by the controller which has a centralized view of the network, and is able to compute all the routes among endpoints. When a new flow arrives at the network edge, the controller installs rules at source switches (ingress) to embed the SlickFlow header into the packet and at destination switches (egress) to remove the header in order to deliver the packet to the destination endpoint.

The proposed protection mechanism allows switches to reroute packets directly in the dataplane as a faster alternative to controller-based path restoration. The source routing approach simplifies the forwarding task in every switch, since it only requires local knowledge of its neighbors, rather than a routing entry per potential destination. SlickFlow can be deployed on top of recently proposed DCN topologies, such as fat-tree [1], DCell [16], and BCube [14]. A prototype was implemented as a proof of the concept using software-based OpenFlow switches and evaluated under fat-Free, BCube and DCell arrangements. The experimental results point to significant gains in failure reaction time.

The rest of the paper is organized as follows. Section II discusses related work. Section III presents the design and implementation of SlickFlow. Section V evaluates our proposal in terms of packet header overhead, link failure recovery times, packet loss, and forwarding performance. Finally, Section VI concludes the paper and outlines avenues for future work.

II. Related Work

A. Data Center Networks

Recently, there have been many proposals for network architectures specifically targeting the data center.

VL2 [11] provides the illusion of a large L2 network on top of an IP network, using a logically centralized directory service. VM hosts add a shim layer to the networking stack called the VL2 agent which encapsulates packets into a IP-in-IP tunnel. The are two classes of IP addresses, Locator Addresses (LA) with topological significance, and Application Addresses (AA). LA addresses are assigned to the switches, which run a link state routing protocol to disseminate these LAs and provide a global view of the topology. VL2 assigns servers AA addresses and when a server sends a packet, the shim-layer on the server invokes a directory system to encapsulate the packet with the LA address of the destination ToR (Top of Rack) switch. Once the packet arrives at the LA of the destination ToR, the switch decapsulates the packet and delivers it to the destination AA carried in the inner header. Link failures are handled by assigning the same LA address to all intermediate switches that are all exactly three hops away from any source host. ECMP takes care of load balancing the traffic encapsulated with the anycast address of the active intermediate switches. Upon switch or link failures, ECMP reacts without needing to notify agents while ensuring scalability.

Like VL2, PortLand [23] also employs two classes of addresses at L2: Actual MAC (AMAC) addresses, and pseudo MAC (PMAC) addresses, which encodes hierarchical location information in its structure. PortLand employs a centralized Fabric Manager to resolve ARP queries, and to simplify multicast and fault tolerance. The Fabric Manager assigns to each end host a PMAC representing its location on the topology. The ARP requests made by end hosts are answered with the PMAC of the destination. Thus, packet forwarding process is based solely on the PMAC. The egress switches are responsible for PMAC to AMAC mapping and packet re-writing to maintain the illusion unchanged MAC addresses. Therefore, end hosts remain unchanged. In presence of failures, the fabric manager informs all affected switches of the failure, which then get their forwarding tables recalculated based on the new version of the topology.

B. Source Routing

Not new to DCN or specific to SDN is the previous work devoted to source routing under different flavours. Source routing is an appealing approach that uncouples the routing logic from and provides data plane scalability by turning forwarding elements stateless. In contrast to conventional hop-by-hop routing, switches just forward packets based on the routing information carried in the packet headers. We now fast forward from early work on source routing [26] to a subset of recent work relevant to the scope of this paper.

OpenFlow programmability provides a convenient way to define the source routing rules at the network edges. In SiBF [25], the packet's source route is represented into a in-packet Bloom Filter (iBF). The Bloom filter bits are carried by the MAC fields and the MAC rewriting occurs at the source and destination OpenFlow-enabled ToR switches. SecondNet proposes *Port-Switching based Source Routing* (PSSR) that represents a routing path as a sequence of output ports of switches. In both SiBF and SecondNet, the failure recovery mechanism is based on a centralized controller that reroutes the traffic. In SecondNet [15], when a link or switch fails, the controller tries to allocate a new path for that VM-pair. SiBF proposes the installation of lower priority back-up flow entries with an alternative link-disjoint iBF path. Upon a failure at intermediate switches, the controller removes the active (high priority) flow entries at ToRs affected by the failure.

Slick Packets [22] have been proposed to achieve fast data plane failure reaction by embedding alternative routes within the packet headers. The packet header contains a directed acyclic graph called the forwarding subgraph (FS). Each router along the packet's path may choose to forward it along any of the outgoing links at that router's node in the FS (optionally preferring a path marked as the primary). This allows packets to 'slip' around failures in-flight while retaining the flexibility of source route control. However, Slick Packets is a theoretical network design without an implementation and validated using analytic results.

Related work with focus on datapath resilience includes MPLS Fast Reroute [3], SafeGuard [18], and OpenFlow fast failover [27]. Common to these proposal is computing alternative path to each destination for intradomain routing, so a router can locally switch to the alternative path without waiting for a control-plane convergence process.

C. SlickFlow contributions to the state-of-the-art

This paper elaborates on our previous work on Encoded Paths [24] extending it to support arbitrary topologies beyond fat-trees and addressing data plane fault tolerance with the proposed SlickFlow mechanisms, experimentally validated for a number of assumptions and DCN scenarios. SlickFlow, as reflected in the name choice, is inspired by Slick Packets [22] but fundamentally differs on the alternative paths encoding. Rather than using forwarding subgraph that can increase the dataplane complexity due to its variable size header, SlickFlow relies on a sequence of fixed-size segments that could be easily supported in hardware-based datapath devices as suggested by our OpenFlow prototype implementation that re-uses existing headers of the TCP/IP stack.

Our work aims at combining the flexibility of source routing, where sources specify an explicit route in the packet header rather than a destination as traditional network controlled routing protocols, and the fast failure reaction of alternative routes embedded within the packets. Besides, our proposal does not require end point modifications nor shim-header overheads, being able to recover from failures without recalculating forwarding tables, but using the alternative path information carried in the packet header.

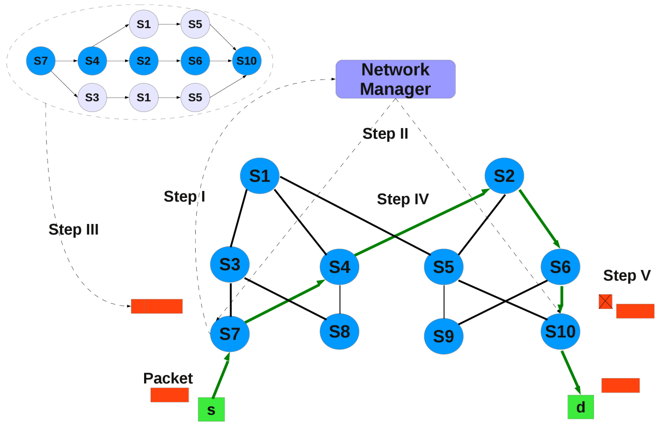


Fig. 1: Overview of the design

III. Design and Implementation

Building upon the principles of SDN, one key characteristic of the proposed architecture is adopting a clear separation of the route computation (on the deployed topology) from failure handling (when link or node status changes) [5]. Thus, the core idea is compute every potential path on a given deployed topology. That is, the routing function ignores all state changes and only recomputes paths when the topology changes. Since these topology changes are rare and often known in advance [5], the computation of routes can be done in a centralized fashion. This flexibility, in both the length and location of the computation, allows a wide variety of path computation according to various criteria.

A data center is often managed by a single logical entity which leads us to adopt a logically centralized controller (NM - Network Manager) that makes all the decisions within the data center. As the controller knows the network topology, we can remove from the switches the task of making routing decisions which allows to employ specific routing techniques such as source routing, increasing scalability of the data plane because intermediate switches become stateless. Basically, the NM discovers its network map by using e.g. Link Layer Discovery Protocol (LLDP) to (*i*) calculate all routes between each pair of hosts, and (*ii*) install the forwarding states at the intermediate switches based solely on the neighboring switch information.

Thus, for new flows at the network edge, the controller commands the source (ingress) switches to embed a primary and (link-disjoint) alternative path into the packet in form of a SlickFlow header. This redundant protection mechanism allows the traffic to be switched to this alternative path, when a failure is detected under the primary path. It contrasts with reactive path restoration mechanisms, in which the alternative path is established by the controller when it receives the failure notification from the OpenFlow switches [27], using for instance LLDP messages.

Figure 1 shows an example to illustrate our proposal design. Suppose the host s wishes to communicate with destination d. As the flow table of source ToR switch S7 is empty, the packet misses a rule and is forwarded to the NM (Step I). From a list of the precomputed existing paths, NM selects the primary path ($S7 - S4 - S2 - S6 - S10$) and alternative paths and installs one rule (OpenFlow entry) at the source and destination ToRs (Step II). The rule at source ToR $S7$ instructs the switch to (*i*) embed the selected paths into the packet header fields (Step III), and (*ii*) forward the packet via the outport port to $S4$. At $S4$, $S2$ and $S6$, all the forwarding decisions are based on the contents of the SlickFlow header carried by the packet (Step IV). Finally, the rule in the destination ToR $S10$ removes the SlickFlow header and delivers the packet to the destination d (Step V). In case of a failure event, the current switch looks for alternatives paths in the SlickFlow header, if available, the packet is rerouted to the backup path without intervention of the NM.

It is worth to note that, SlickFlow forwarding does not use IP address for packet flow matching within the DCN. This way, unlike the traditional hierarchical assignment of IP addresses, there are no restrictions on how addresses are allocated. In essence, we separate host location from host identifier so that this is transparent to end hosts and compatible with existing commodity switches hardware. Moreover, this approach does not introduce additional protocol headers such as MAC-in-MAC or IP-in-IP [11], [12].

In the following, we describe the details of the underlying assumptions and mechanisms.

A. *Topology Discovery and Route Computation*

In order to construct source routes, the NM needs to know the network topology at boot time without manual intervention. For this, we use LLDP messages for topology inference. LLDP packets are sent from each interface of switches and then, according to the received packets in neighboring switches, the connections are discovered.

The route calculation module can be configured to calculate all routes between all pairs of hosts, or a predetermined number of k paths. This parameter can be set by the administrator based on the size of the network. For best performance, the routes can be selected based on disjoint paths [20], or paths selecting randomly from a probability distribution [29].

B. *Path Selection*

Once the controller needs to install rules to a new flow, it selects, among the previously calculated paths between source and destination, the path along which you want the packet to be sent, and also select alternative link-disjoint paths.

The paths the controller will choose can be defined according to various criteria. It can select paths, for example, to avoid single link failures along a primary route, to anticipate node failures, to optimize the network according metrics such as bandwidth. In this work, we consider paths selection to minimize the latency in the primary path and to offer alternative routes to avoid single link failures.

First, NM selects the primary path P from a source s to a destination d, then for each hop (or a subset of hops) of P, it selects an alternative path p_i to be used if the next hop on

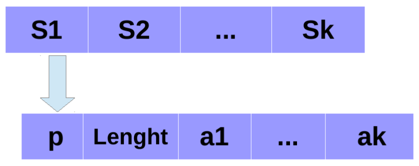


Fig. 2: Encoding format layout

the primary path is not available. An alternative path p_i is a path that starts at node i and goes up to destination d avoiding the link that connects i to the next node of the primary path. To reduce the latency over the primary path, the NM selects the shortest path between s and d. In case of multiple shortest paths to the destination, the NM arbitrarily selects one of these paths.

Despite the presented path selection covering only single link failures, multiple link failures can be recovered with the assistance of the centralized controller. The controller installs an alternative path on the ToR's after it receives the link failure notification from the switches.

C. *Packet Header Encoding*

After path selection, the NM must encodes the paths into the SlickFlow header as a sequence of segments that will be used by each hop along the route. As shown in Figure 2, a segment corresponding to a node k has 3 pieces of information: (1) p, the label of next hop on the main path, (2) the length of the alternative path of k, and (3) the alternative path of k represented by a sequence of labels $(a1, ..., ak)$. Therefore, when a node has no alternative path, the length is 0.

The header also contains an additional piece of information, a bit field called $alternative$, which specifies whether the packet is on the primary or alternative path. This encoding is based on the studies in [22], where the authors compared different encoding formats to represent networks as directed acyclic graphs. Our specific choice is based on the fact that the encoding via graphs results in less compact headers and increase the complexity of the forwarding operation. Also note that the form of encoding could be generalized to allow more than one alternative path to the same node, or even allow alternative paths to hops on alternative paths. However, the investigation of enhanced encoding formats and other optimizations have been left for further work. Next, we explain how switches use this information to forward packets.

D. *Forwarding*

Upon receiving a packet, the switch extracts the bits contained in the first segment (leftmost) and uses it as key to search the output port in the forwarding table. After that, the switch checks if the corresponding link is available and checks the $alternative$ bit to know if it is on the primary or alternative path.

1) Primary path: If the switch is on the primary path, and the output link to the next hop is available, the switch updates the header, removing the first segment by shifting the rest of the header to the left, so that the second segment becomes the outer most bits, and so on. The packet is then forwarded to the next hop on the primary path.

If the connection to the next hop on the primary path is unavailable, the switch checks the length of its alternative path. If it is 0 –there is no alternative path– the packet is dropped. Otherwise, the switch reads the label of the next hop of the alternative path a_1. If the link corresponding to a_1 is not available, the packet is dropped. If the link is available, the switch rebuilds the header by replacing the labels of the primary path by the labels of the alternative path (a_2, ..., a_k). It also sets the $alternative$ bit to 1. The packet is then forwarded to the next hop a_1.

2) Alternative path: If the value of the $alternative$ bit is 1, the switch knows the packet is being forwarded through an alternative path. The switch reads the label of the next hop, and if the corresponding link is not available, the packet is dropped. If the link is available, the switch shifts the corresponding segment bits to the left and forwards the packet to the next hop.

When the failed link becomes available again, the flow is switched back to the primary path automatically. As the switch verifies that the next hop of the primary path is available again, it just send the packet to it, without any additional signaling.

IV. Prototype Implementation

We implemented the SlickFlow forwarding mechanism on OpenFlow software switches (Open vSwitch), and the Network Manager (NM) was developed as an application on top of NOX [13], an OpenFlow network controller.

When a packet arrives at the first hop ToR and lacks of a matching entry, the packet is sent to the NM which selects the primary and alternative paths among all pre-calculated paths between the source and destination. The NM then install the necessary rules at the originating and destination ToR switches. In the source ToR, the OpenFlow rule includes an action to add the SlickFlow header to the packet and in the destination ToR, the flow actions instruct the switch to remove the SlickFlow header and deliver the packet to the destination endpoint. Flow entries at the edge switches are based on the destination IP address (plus optional VLAN or tenant identifiers).

Forwarding at intermediate switches is based on the fixed rules installed by the NM during booting up and topology discovery. These OpenFlow entries are based on the next hop label of the SlickFlow header and simply indicate the output port to which matching packets should be sent. We modified the default OpenFlow switch software to, before sending a packet out, make the SlickFlow header processing described in the previous section.

The concept itself is agnostic to the particular location in the packet header used to carry the bits. It may reside in a shim header between the IP and MAC layers, in an IP option, or in a novel header format in a next-generation Internet protocol. Our key goal in designing the encoding format is to ensure simple data plane forwarding that can be deployed over existing hardware (e.g., commercial OpenFlow switches). However, we chose to use existent headers to avoid additional overhead and simplify the implementation.

We place the SlickFlow header in the VLAN and MAC Ethernet fields. Whereas the next hop label of the first segment is placed in the $Vlan_{VID}$ field and the remaining of the header is placed in the 96 bits of the Ethernet source and destination MAC address fields and the *alternative* bit is placed in the $Vlan_{PCP}$ field. For this configuration, each label and length fields have a fixed size of 4 bits, so each switch is limited to 16 neighbors and the alternative path can have at most 16 hops.

We reckon that this proof of concept configuration is insufficient for real data center fabrics which may contain switches with more than 100 ports. It is however sufficient to demonstrate our idea and prove the feasibility of implementation.

A more appropriate label size would be 8-bit long, allowing switches to have up to 256 neighbors, but the header bits in the Ethernet MAC address space would quickly exhaust, not allowing to encode alternative paths (or very short paths). One scenario we are considering for future work leveraging the latest OpenFlow protocol versions is using additional packet bit space (e.g., PBB, MPLS, and/or IPv6 basic and extended headers) to allow for larger SlickFlow headers.

One obvious possibility to reduce the size of the header is to not include the alternative path of the originating switch where the SlickFlow encoding rules are installed. Instead, the alternative path relative to the first hop switch could be established by the NM as an additional flow entry with lower priority. When a failure on an outgoing link from the first switch is detected, the switch can disable the affected outgoing port(s)traffic and automatically switched to the alternative path. In case of OpenFlow 1.1+ capable switches, group tables with fast-failover actions can be used [7].

We envision offering the SlickFlow forwarding service to a fraction of the traffic (or applications, or VMs groups), while the rest of the traffic can be handled according to a legacy network model or by another controller operating in parallel, e.g., using a virtualization layer like FlowVisor [28].

V. Evaluation

We evaluate SlickFlow using our OpenFlow-based prototype running on the Mininet [17] emulation platform. Our main evaluation goals include exploring (*i*) how arbitrary topologies can be supported and behave in SlickFlow, and (*ii*) how fast the data plane can recover from network link failures. We consider common DCNs topologies varying their configuration and analyzing how many bits would be required to support the proposed fault tolerent source path encoding.

As a proof of the concept, SlickFlow is implemented and evaluated encoding one alternative path for every hop on the primary path (Sec. III-B) measuring the failure recovery time, packet loss, and forwarding performance.

A. Encoding Size

We start by evaluating the SlickFlow header sizes of the basic encoding format presented in §III-C. The encoding size depends mainly on the diameter of the topology, and the switch fan-out (i.e. number of outgoing ports). Thus, considering a network with diameter d, each switch with p ports, and the length of each alternative path with up to l bits, the SlickFlow header will have d segments, each of size $\log_2(p) + l$, plus the size of the alternative path. To simplify the equation, we assume that for each path between the nodes, there is an alternative path of the same length. So, the alternative path for the first hop has a length equal to network diameter d, the next alternative path for the next hop contains $d-1$ segments, and so on. The redundancy level n is defined as the number of alternatives paths for every segment. These assumptions lead to the following equation on the SlickFlow header size:

Redundancy level *n*	**1** (alternative path)			**2** (alternative paths)		
Diameter *d*	**5**	**7**	**9**	**5**	**7**	**9**
p=16	80	140	216	120	224	360
p=32	95	168	261	145	273	441
p=64	110	196	306	170	322	522
p=128	125	224	351	195	371	603

TABLE I: Header sizes (in bits) for different configurations.

Topology	**Ports *p***	**Hosts**	**Diameter *d***	**Avg. path length**
Fat-tree-3	24	3,456	6	5.9
Fat-tree-4	8	4,096	8	7.6
DCell-1	58	3,422	5	4.9
DCell-2	7	3,192	11	8.2
BCube-1	58	3,364	4	3.9
BCube-3	5	3,125	10	8.0

TABLE II: Topology configurations.

$$S = d * (\log_2(p) + l) + n * \sum_{i=1}^{d} i * \log_2(p) \qquad (1)$$

$$S = d * (\log_2(p) + l) + n * \frac{d^2 + d}{2} * \log_2(p) \qquad (2)$$

The first component of this equation refers to the segment structure (p and length l in Fig. 2) and the second component refers to the redundant information contained in the segments (a_i of Fig. 2). Assuming that the label corresponding to the length of the alternative path has 4 bits (which means that alternative paths are maximum 16-hop long), Table I shows the size (in bits) of the SlickFlow header for some topologies and varying network diameter d, number of switch ports p, and n alternative paths per hop (redundancy level).

As we can see, the diameter of the topology is the dominant factor, as it was expected by analyzing equation 2. Also, we can observe that for redundancy level n=2, i.e., two alternative paths per segment, the header size grows faster. As the redundancy level in SlickFlow design is general to be configured, we assume a redundancy level equal to 1, for validation purpose.

Now, we evaluate the required SlickFlow header sizes for the topologies used by the main data center architectures found on the current literature [8], fat-tree, DCell, and BCube, shown in Figure 4, with variations in the parameters to keep the number of end servers constant and close to 3,500.

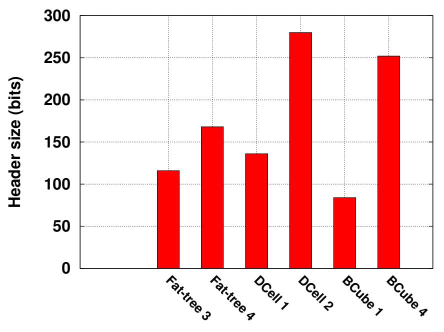


Fig. 3: SlickFlow header size

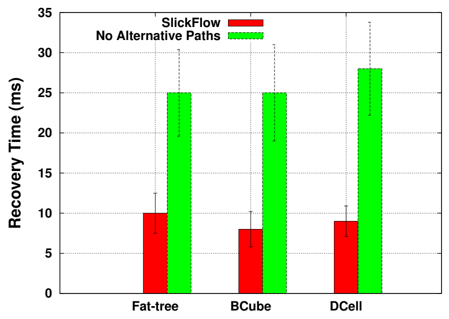


Fig. 5: Recovery Time

Table II shows the parameters of the topologies as well as the corresponding number of ports per switch and the total number of hosts. The table also includes network properties such as diameter and the average length of the shortest paths between servers.

It is worth noting that although some of these topologies could operate partially complete, we consider only the use of the complete network topologies, i.e., with all the network interfaces of servers and switches in use. To build the SlickFlow header, we consider the worst-case scenario in terms of distance, that is, we choose the source and destination nodes so that the distance between them is the diameter of the network.

Figure 3 shows the results of the resulting packet header sizes to encode. As it can be seen, BCube1 has the smallest size an requiring an encoding size of 84 bits, while in DCell2, the header is about 3 times larger.

An interesting result by comparing topologies of the same type with different parameters, such as BCube and DCell, shows that topologies with larger number of ports per switch have larger header sizes. The reason is because the source-destination pairs have longer primary paths, requiring alternative paths for a larger number of nodes. On the other hand, these topologies are more resilient, because they offer a higher number of different paths between every pair of nodes.

B. Failure Recovery Time

In order to evaluate the performance of the SlickFlow failure recovery, we measure the recover time after link failures and compare with a mechanism that employs the controller to restore the path of packet flows. In this recovery process, the alternative path is installed on the ToR's switches by the controller when it receives the link failure notification from the OpenFlow switches (`port-down` event). Note that the link failure *detection* time is not considered in these experiments since we want to isolate the contribution of the failure recovery time from the specific controller-based (e.g. LLDP) or dataplane-based (e.g. BFD, Ethernet OAM, OSPF Hello) detection mechanism.

For the experimental validation purposes, we used the fat-tree, BCube and DCell of Figure 4. The switches inside Mininet connect to an OpenFlow controller in the same machine. The OpenFlow protocol version used in our implementation is 1.0.

In the first experiment, we measure the time to re-establish the traffic flow for a single link failure. UDP traffic of 10Mbps is started between two arbitrary hosts. We monitor the network activity at the receiver while we inject link failures according to an uniform distribution and measure the time required to re-establish the communication, i.e., to continue receiving the UDP stream.

Figure 5 shows the average recovery time for both topologies for 10 experiment runs. Total recovery time in SlickFlow is about 10ms for both topologies, while the time for recovery process involving the controller is about 150% higher. Note that in the case of more than one single path failure, re-routing is still possible by using the controller intelligence.

C. Packet loss

We now measure the loss of packets during the failure recovery process, varying the transfer rates of the previous experiment between 1Mbps, 5Mbps, and 10Mbps. Figures 6 plots the average packet loss, for fat-tree and DCell respectively, across 10 runs as a function of the transfer rate. Since the results are similar for the three topologies, due to lack of space we omitted the result referring to BCube. As it can be seen, the packet loss is smaller in SlickFlow compared to the controller-based restoration process.

Ideally, SlickFlow should have no packet loss, since every packet is processed by the switches, and if the primary link is not available, the packet should be sent to the alternative path. In our proof of concept software-based implementation, however, zero-packet loss only occurs for the lower transmission rates (1Mbps). This packet loss behavior can be explained due to the time it takes for a switch to detect port status changes and locally switch to an alternative path.

Note that the recovery mechanism without alternative paths depends of the RTT between the switches and the controller. For larger networks with heavier control traffic loads, the effective RTT tends to be higher, decreasing the performance of the restoration mechanism [9].

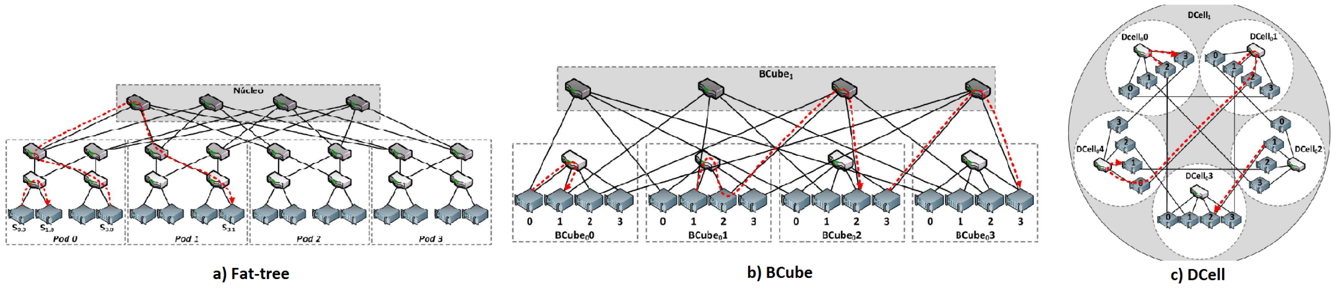
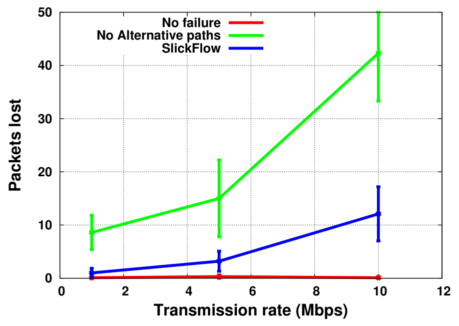
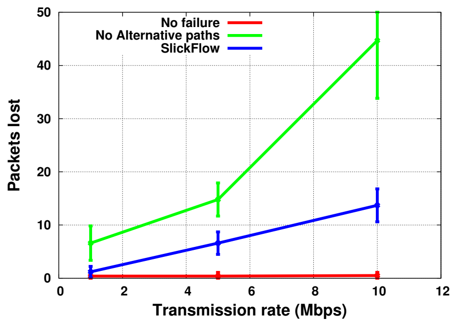


Fig. 4: Topologies



(a) Fat-tree Topology



(b) DCell Topology

Fig. 6: Packet loss during network link failure events.

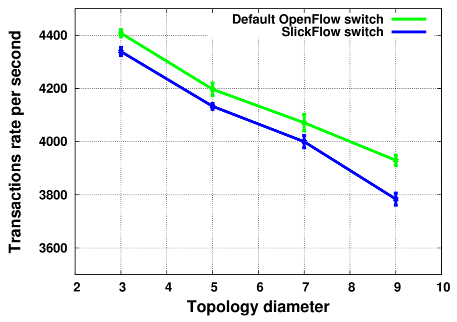


Fig. 7: Request/response performance

The results also show that the topologies have not significant influence on the recovery times or the number of lost packets. This is explained by the fact that the switch implementation and the failure recovery mechanism is identical regardless the topology of the network.

D. *Forwarding performance*

To implement SlickFlow forwarding, we modified the OpenFlow switch implementation to, before sending a packet out, process the packet header as described in Sec. III-D. We now measure the impact of this additional processing compared to the performance of unmodified switches. Using linear topologies with different diameters, we used *netperf* [4] to measure request/response transactions between a sender and a receiver, where a transaction is defined as the exchange of a single application-level request and a single response. From the observed transaction rate, one can infer one way and round-trip average latencies. As shown in Figure 7, the average transaction rate for the modified OpenFlow switch is less than 5% lower than the default OpenFlow switch in the worst case, when the diameter is 9 hops.

We reckon that the purpose of this evaluation is solely for a straw-man comparison between traditional and SlickFlow forwarding in a software-based implementation. There is no conceptual barrier to SlickFlow forwarding being supported in hardware given its simple, fixed size match and packet-header re-write operations.

VI. Conclusion and Future Work

In this paper, we present SlickFlow, a proposal to add datapath fault tolerance in DCN based on source routing and augmented with alternative paths carried in packet headers and programmed via OpenFlow primitives. SlickFlow is independent of the network topology and has shown potential benefits in emulated DCN topologies. Results in a virtualized testbed show that the proposed approach can achieve efficient failure recovery on the data plane without involving the controller.

As future work we plan to improve our implementation in the following points. We intend to explore features of newer versions of the OpenFlow protocol, like the fast-failover mechanism, and using push/pop tags and IPv6 headers to validate SlickFlow at scale. We will also devote efforts to consider the hypervisor vSwitch as the first networking hop where initial flow matching and header rewriting takes place. Moreover, taking into account that the failures in data centers occur more frequently in the ToR-to-Aggregation links [10], in future revisions of our design we intend to extend the encoding to increase the level of redundancy by allowing more than one alternative path over these critical links.

Acknowledgements

This work has been partially suported by grants from CNPq, CAPES and FAPES. Ramon Ramos has been supported by CAPES scholarship. This work has been partially suported by FAPES from grants of Digital Cities (CDIG) project.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1402958.1402967

[2] M. Arregoces and M. Portolani, *Data Center Fundamentals*. Cisco Press, 2003.

[3] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow, "Rsvp-te: Extensions to rsvp for lsp tunnels," United States, 2001.

[4] B. Bidulock, "Netperf utility installation and reference manual version 2.3 edition 7 updated 2008-10-31 package netperf-2.3.7."

[5] M. Caesar, M. Casado, T. Koponen, J. Rexford, and S. Shenker, "Dynamic route recomputation considered harmful," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 2, pp. 66–71, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1764873.1764885

[6] Cisco, *Cisco Data Center Infrastructure 2.5 Design Guide*. San Jose, CA: Cisco Systems, Inc, 2007.

[7] Consortium, OpenFlow, "OpenFlow 1.1 switch specification," *Can be accessed at http://openflowswitch. org/*, pp. 1–56, 2011.

[8] R. S. Couto, M. E. M. Campista, and L. H. Costa, "Uma avaliação da robustez intra data centers baseada na topologia da rede," *XXX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC'2012*, May 2012.

[9] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 254–265, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2043164.2018466

[10] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 350–361, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2043164.2018477

[11] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 51–62, Aug. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594977.1592576

[12] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Towards a next generation data center architecture: scalability and commoditization," in *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, ser. PRESTO '08. New York, NY, USA: ACM, 2008, pp. 57–62. [Online]. Available: http://doi.acm.org/10.1145/1397718.1397732

[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1384609.1384625

[14] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: a high performance, server-centric network architecture for modular data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 63–74, Aug. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594977.1592577

[15] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *Proceedings of the 6th International COnference*, ser. Co-NEXT '10. New York, NY, USA: ACM, 2010, pp. 15:1–15:12. [Online]. Available: http://doi.acm.org/10.1145/1921168.1921188

[16] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "DCell: a scalable and fault-tolerant network structure for data centers," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008, pp. 75–86. [Online]. Available: http://doi.acm.org/10.1145/1402958.1402968

[17] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466

[18] A. Li, X. Yang, and D. Wetherall, "Safeguard: safe forwarding during route changes," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, ser. CoNEXT '09. New York, NY, USA: ACM, 2009, pp. 301–312. [Online]. Available: http://doi.acm.org/10.1145/1658939.1658974

[19] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson, "F10: a fault-tolerant engineered network," in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 399–412. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482665

[20] S. Mahapatra, X. Yuan, and W. Nienaber, "Limited multi-path routing on extended generalized fat-trees," *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pp. 938–945, May 2012.

[21] McKeown, Nick and Anderson, Tom and Balakrishnan, Hari and Parulkar, Guru and Peterson, Larry and Rexford, Jennifer and Shenker, Scott and Turner, Jonathan, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[22] G. T. Nguyen, R. Agarwal, J. Liu, M. Caesar, P. B. Godfrey, and S. Shenker, "Slick Packets," in *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '11. New York, NY, USA: ACM, 2011, pp. 245–256. [Online]. Available: http://doi.acm.org/10.1145/1993744.1993769

[23] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: a scalable fault-tolerant layer 2 data center network fabric," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 39–50, Aug. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594977.1592575

[24] R. M. Ramos, M. Martinello, and C. E. Rothenberg, "Data center fault-tolerant routing and forwarding: An approach based on encoded paths," *IEEE Latin-America Symposium on Dependable Computing - LADC*, pp. 104 – 113.

[25] C. E. Rothenberg, C. A. B. Macapuna, F. L. Verdi, M. F. Magalhães, and A. Zahemszky, "Data center networking with in-packet bloom filters," *XXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, 2010.

[26] H. Saltzer, D. Reed, and D. Clark, "Source routing for campus-wide internet transport," in *ProceedingsoftheIFIPWG6.4WorkshoponLocal Networks*, 1980, pp. 25–32.

[27] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "A Demonstration of Fast Failure Recovery in Software Defined Networking," *TRIDENTCOM 2012*, pp. 411–414, 2012.

[28] Sherwood, Rob and Gibb, Glen and Yap, Kok-Kiong and Appenzeller, Guido and Casado, Martin and McKeown, Nick and Parulker, Guru, "FlowVisor: A Network Virtualization Layer," in *Technical Report OpenFlow-tr-2009-1*. Stanford University, 2009.

[29] X. Yuan, W. Nienaber, Z. Duan, and R. Melhem, "Oblivious routing for fat-tree based system area networks with uncertain traffic demands," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 1, pp. 337–348, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1269899.1254922