# Programmable Residues Defined Networks for Edge Data Centres

Magnos Martinello*, Alextian B. Liberato*, Arash Farhadi Beldachi†, Koteswararao Kondepu†,
Roberta L. Gomes*, Rodolfo Villaca*, Moisés R. N. Ribeiro*, Yan Yan†, Emilio Hugues-Salas†, Dimitra Simeonidou†
*Informatics Dept, Federal University of Espirito Santo, Brazil,
Emails: {magnos.martinello, alextian.liberato, roberta.gomes, rodolfo.villaca, moises.ribeiro}@ufes.br
†High Performance Networks Group, University of Bristol, Bristol, United Kingdom,
Emails: {arash.beldachi, k.kondepu, yan.yan, e.huguessalas, dimitra.simeonidou}@bristol.ac.uk

***Abstract*—Edge Data Centres (EDC) are often managed by a single administrative entity with logically centralized control. 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. However, in the design of an SDN architecture, there is no clear distinction between functional network parts such as core and edge elements. It means that all switches require to support lookups over hundreds of bits with complex actions that have to be specified by multiple tables. In this paper, we propose a new programmable architecture for EDC networks, named *Residues Defined Networks (RDN)*. In RDN, a controller defines a network policy (e.g. connectivity protection) setting flow entries at the edges. Based on these entries, the edge switches assign route-IDs to flows. A route is defined as the remainder of the division (Residue) between a route-ID and a set of switch-IDs within RDN core. In case of failures, emergency routes are compactly encoded as programmable residues forwarding paths written into the packets. RDN scalability is evaluated considering 2-tier Clos topologies which cover mostly EDC deployments supporting up to 2304 servers. A RDN proof-of-concept prototype is implemented in Mininet for network emulation. Also, to increase the accuracy on latency measures, we implement RDN in NetFPGA that is validated in a testbed with 10Gbps Ethernet boards. RDN offers ultra-fast failure recovery (sub-milliseconds carrier grade), achieves low latency with RDN switching time per hop ($\approx 0.6\mu$s) and no jitter within the RDN core.**

## I. Introduction

Current networks have evolved toward complex systems that are widely agreed to be expensive, complicate to manage and susceptible to vendor lock-in. Although the network research community has focused on "clean-slate" designs of the overall Internet architecture, there is an urgent set of problems that remain in the design of the underlying *network infrastructure*.

The network infrastructure is composed essentially by two components [1]: (i) the underlying hardware representing the networking data plane and (ii) the software that controls the overall behavior of the network, representing the control plane. The Software Defined Networking (SDN) fundamental point is to decouple the networking data plane from the control plane; the latter holds the network intelligent decisions while the former merely hosts executive tasks based on tables to process incoming flows.

However, in the design of an SDN architecture, there is no clear distinction between the functional network parts such as core and edge elements. In fact, an OpenFlow (the common standard for SDN) enabled switch is clearly far from a simple design, requiring to support lookups over hundreds of bits with complex actions that have to be specified by multiple tables [2]. Moreover, there is no decoupling between the routing and the network policy, when an SDN controller decides the functions a flow needs, it also decides the path the flow has to go through and the setup states on all the intermediate switches.

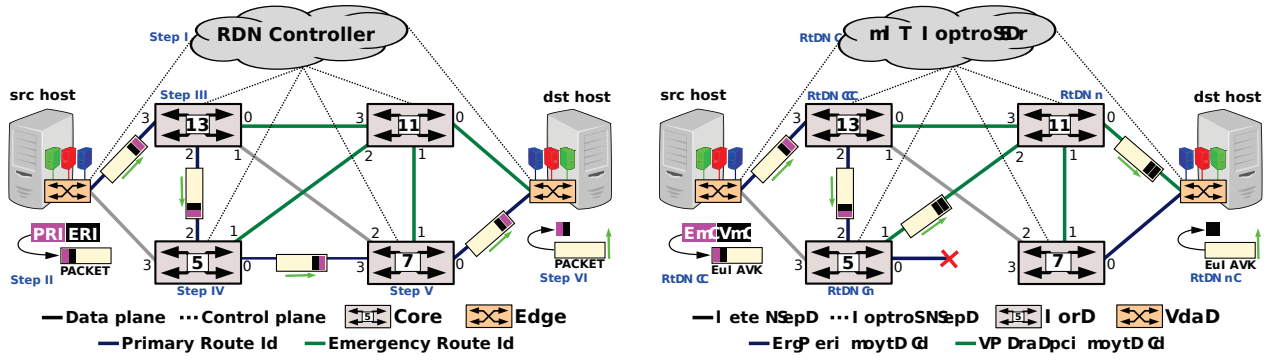
We propose the concept of programmable *Residues Defined Networks* (RDN). We argue that the decoupling between edge and core switches builds foundational blocks towards a pragmatic SDN design pattern. Although this idea of designing a network pushing the complexity to the network edge and keeping the core extremely simple (e.g. fabric model [1]) is not new in the network community, there is a real need to implement SDN design patterns to evolve the SDN architecture.

Thus, we advance the state-of-art demonstrating the feasibility of RDN through a programmable resilient routing approach. Differently from previous works (e.g. MPLS), the network core is designed by exploiting special mathematical properties from Residue Number System (RNS) [3]. Our focus is to provide a protection mechanism along the entire route with an extremely fast failure reaction by using programmable residues forwarding paths. A route between a pair of hosts in the RDN is defined as the remainder of the division between a route-ID and a set of local switch-IDs on the path. Emergency routes are similarly defined, but they are compactly encoded using one single route-ID. As soon as a core switch detects a link failure, it triggers an emergence route precomputed into the packets that lead the packets to their destination.

The proposed RDN protection mechanism allows tableless switches to reroute packets directly at the data plane as a faster alternative to controller-based route restoration. The source routing approach simplifies the forwarding task in every switch that processes packets based on simple modulo operation, rather than a routing entry per potential destination.

RDN approach is topology independent and since an Edge Data Centre (EDC) is often managed by a single administrative

978-3-901882-98-2 




(a) (b)

Figure 1. Architecture Design of Programmable Residues Defined Networks: (a) routing system, (b) fast failure reaction.

entity [4], it leads to the adoption of a logically centralized controller that makes programmable decisions within the EDC. In this work, we carry out a RDN scalability analysis considering topologies as a design reference based on 2-tier Clos networks which cover the majority of EDC deployments and support thousands of physical servers [5].

For experimental evaluation, a RDN prototype was implemented in Mininet emulated environment. Also, RDN core switches were implemented in NetFPGA devices to increase the accuracy on latency measures. The experimental results obtained from real testbed point out to ultra-fast failure reaction (sub-milliseconds carrier grade), achieving low latency with RDN switching time per hop ($\approx 0.6\mu$s) and no jitter within the RDN core.

## II. Programmable Residues Defined Network

Figure 1 shows our architecture design. Building upon the principles of SDN, the proposed architecture makes a clear separation of the edge and core switches to form two foundational blocks towards a pragmatic SDN design pattern. This separation allows to push the complexity to the network edge while keeping the core extremely simple.

RDN introduces an explicit distinction between the RDN Controller, the network edge and core switches. The RDN Controller is a central entity in charge of (i) selecting the network routes, (ii) computing their specific route-IDs, and (iii) sending these identifications to the edge switches. As the RDN Controller knows the network topology, which is discovered by using e.g. Link Layer Discovery Protocol (LLDP), this component is able to (i) define the policy assigning functions (e.g. route protection) to specific flows installing their states at the edge switches, and to (ii) calculate all route identifications (route-IDs) between each pair of hosts (or even virtual machines).

Route-IDs are actually sent to the edge switches by the RDN Controller that installs per-flow states in OpenFlow enabled switches. Such a fine-grained control enables the edge switches to map specific fields of the packets into route-IDs. Either a reactive or a proactive approach can be used here. In the first case, the rules are installed when the ingress edge switch forwards the packet to the RDN Controller via *packet_in* requesting the route mapping. In the second case, the RDN Controller pre-computes the route and installs in advance the incoming per-flow state via *flow_mod* for a route end-to-end between the source and the destination.

When packets enter the core network, ingress edge switches inspect them and then, based on the flow-table rules, they embed route-IDs into packets (e.g OpenFlow action to write from a MAC address fields to route-IDs). Route-IDs are later removed from packets at egress edge switches (e.g. rewriting route-IDs to original mac-addresses). It is important to notice that edge functionality can be built on software, i.e., in vSwitchs. The vSwitch occupies a unique position in the networking stack as it can easily modify packets without requiring any changes to customer VMs or transport layers. Functionality built into the vSwitch can be made aware of the underlying hardware offload features presented by the NIC and Operating System.

Concerning the RDN core switches, these are actually built just to deliver packets to the destination simply using the remainder of the division of route-IDs and its own ID. RDN route-IDs have meaning only within the core and are completely decoupled from the host protocol (e.g. IPv4, IPv6 or MAC).

### A. Routing based on Tableless Residues Switches

Let us suppose that $src$ host in Figure 1 wishes to communicate with $dst$ host through an intra-domain network operator that supports a programmable RDN architecture. The network architecture is split between RDN Controller, edge and core switches. For the core switches, the idea is to replace the traditional lookup table operation by tableless switches that operate only using residues operations, i.e. remainder of the division. The core switches are assigned to co-prime numbers that are in this example $\{5, 7, 11, 13\}$. Thus, in order to establish the communication between the pair of hosts, the RDN Controller needs to calculate what is the number (*route-ID*) for which the modulo operations results will lead the packets to their destination.

By using a routing algorithm, the RDN Controller selects an end-to-end path across the network according to the EDC policies, as presented in Figure 1(a) (**Step I**). For instance, it chooses the route to be set through the switches $S = \{13, 5, 7\}$ composing what we call the Primary Route. In this case, the

switches' output ports are $P = \{2, 0, 0\}$. Then, it computes a Primary Route Identification (PRI), e.g. $PRI = 210$. The RDN Controller sends the route-ID to the edge switches which then install the respective flow-table rule(s). Subsequently, the ingress edge switch is responsible for embedding PRI into each packet (e.g. into one of its header fields) coming from $src$ host to $dst$ host (**Step II**).

Once the packet has entered into the core, at every switch the packet arrives to, the remainder of the division (denoted as $< a >_b \equiv a \ modulo \ b$) between the packet's PRI ($R = 210$) and the respective *switch-ID* is computed in order to define the appropriate output port to forward it. Thus, as shown in Figure 1(a), when $S_{13}$ receives a packet with route-ID ($R = 210$), it forwards the packet to port $< 210 >_{13} = 2$ (**Step III**); then, $S_5$ forwards it to port $< 210 >_5 = 0$ (**Step IV**); after, $S_7$ forwards it to port $< 210 >_7 = 0$ (**Step V**), reaching the egress edge switch that removes the *route-ID* from the packet (**Step VI**) and delivers it to $dst$ host (as can be seen in Figure 1(a)).

### *B. Programmable Residues Resilient Routing*

In the case of link failure, one traditional approach is **route restoration**. It consists on notifying the controller, which recalculates the route excluding the faulty link from the available paths. The problem is how to react quickly after a failure detection, avoiding the latency to communicate with the controller. A typical mechanism for fast failure reaction is **route protection** through packets deflection. Deflection routing techniques are conceptually simple and allow every switch to independently decide which packets to forward to any available link [6].

Deflection routing is a probabilistic routing technique that may form transient loops [6]. To overcome this drawback, our approach offers deterministic resilient routing with network protection mechanism along the whole route. Our fast failure reaction mechanism uses an Emergency Route Identification (ERI). ERI is computed so as to represent a set of switches necessary to bypass a failure in the primary route, reducing the number of bits required to safeguard the end-to-end path.

To illustrate this concept, consider the scenario shown in Fig 1(b). As in the previous scenario, $src$ wishes to communicate with $dst$. So, steps I, II and III are repeated. However, in this case, there is a failure link between $S_5$ and $S_7$. Before packet forwarding, the switch $S_5$ checks the connection link. As it is not available, the switch must overwrite the current PRI with the ERI. Notice that, as part of the **Step I**, the RDN Controller has calculated the ERI as a unique value that composes a protection route for the whole primary route. In this example, we have $ERI = 1716$, which sets the switches $S = \{13, 5, 7, 11\}$ (with their output ports $P = \{0, 1, 1, 0\}$) as the protection route.

Then, using the overwritten PRI, $S_5$ calculates again the remain of the division in order to properly forward the packet. In this example, using route-ID ($R = 1716$), $S_5$ forwards packet to port $< 1716 >_5 = 1$ (**Step IV**). When $S_{11}$ receives a packet with route-ID ($R = 1716$), it forwards the packet to port $< 1716 >_{11} = 0$ (**Step V**). Note that $S_{11}$ is unaware of the re-routing, it works just like an ordinary packet forwarding. Eventually, the packet reaches the egress edge switch, which removes the *route-ID* from the packet (**Step VI**) and delivers it to $dst$ host.

As previously mentioned, this approach allows fast recovery for the whole primary route. For example, if a failure occurs in the link between $S_{13}$ and $S_5$ instead, $S_{13}$ executes **Step IV**, thus, overwriting PRI and forwarding the packet to port $< 1716 >_{13} = 0$. Then, $S_{11}$ forwards it to the edge switch (**Step V**). But if the failure occurs in the link between $S_7$ and the edge switch, then $S_7$ executes **Step IV**, using route-ID ($R = 1716$) to forward the packet to its port $< 1716 >_7 = 1$ (**Step IV**), allowing the packet to reach $S_{11}$, which then forwards it to the edge switch.

It is worth mentioning that the RDN architecture does not really require core switches to be SDN enabled. The modulo operation and replacing PRI by ERI are in fact the key functions to be supported. Nevertheless, SDN would enable core switches to notify failures to controllers or to support a dynamic switch-IDs registration and reconfiguration.

### *C. Encoding the Primary and Emergency Routes*

The RDN architecture is underpinned by the concept of "programmable" packets. Both Primary and Emergency routes are in fact programmed into each packet. When a RDN switch receives a packet, it only needs to know its own switch-ID and to read the packet's route-ID in order to determine where it should send the packet to. At each switch, the output port is the result of the modulo operation between the route-ID and the respective switch-ID. This is possible due to the properties of the RNS.

Let $S$ be a set $S = \{s_1, s_2, \ldots, s_N\}$ of the $N$ switch-IDs on the desired path, in which all elements are pairwise co-prime numbers. Let $P$ be a set of outgoing ports $P = \{p_1, p_2, \ldots, p_N\}$, where $p_i$ is the outgoing port for the packet at the switch $s_i$.

Let $M$ be

$$M = \prod_{i \in S} s_i \tag{1}$$

A number $R \in \mathbb{N} | 0 \leq R < M$ can be represented by a residue set given a basis modulo set S:

$$R \xrightarrow{RNS} \{p_1, p_2, \ldots, p_N\}_S \tag{2}$$

, where

$$p_i \ = \ R \ modulo \ s_i \tag{3}$$

The RDN Controller must find out the value of $R$ (the explicit route-ID), given a modulo set $S$ (the switch-IDs), and its RNS representation $P$ (the switch output ports).

The Chinese Remainder Theorem [7] states that it is possible to reconstruct $R$ through its residues in a RNS as follows:

$$R = < \sum_{i \in S} p_i \cdot M_i \cdot L_i >_M \tag{4}$$

where

$$< a >_b \equiv a \ modulo \ b \quad (5)$$

$$M_i = \frac{M}{s_i} \quad (6)$$

$$L_i = < M_i^{-1} >_{s_i} \quad (7)$$

Eq. (7) means that $L_i$ is the modular multiplicative inverse of $M_i$. In other words, $L_i$ is an integer number such that:

$$< L_i \cdot M_i >_{s_i} = 1 \quad (8)$$

Returning to the example of this section, the computation of PRI from $src$ to $dst$ is obtained as follows:

$switches = \{s_1, s_2, s_3\} = \{13, 5, 7\}$
$ports = \{p_1, p_2, p_3\} = \{2, 0, 0\}$
$M = 13 \cdot 5 \cdot 7 = 455$
$M_1 = 35, M_2 = 91, M_3 = 65$
$L_1 =< M_1^{-1} >_{s_1} =< 35^{-1} >_{13} = 3$
$L_2 =< 91^{-1} >_5 = 1, \ L_3 =< 65^{-1} >_7 = 4$
$R =< L_1 \cdot M_1 \cdot p_1 + L_2 \cdot M_2 \cdot p_2 + L_3 \cdot M_3 \cdot p_3 >_M$
$R =< 210 + 0 + 0 >_{455} = 210$

With the ERI, the route-ID is computed as follows:

$switches = \{13, 5, 7, 11\}$
$ports = \{0, 1, 1, 0\}$
$M = 13 \cdot 5 \cdot 7 \cdot 11 = 5005$
$M_1 = 385, M_2 = 1001, M_3 = 715, M_4 = 455$
$L_1 =< 385^{-1} >_{13} = 5, \ L_2 =< 1001^{-1} >_5 = 1$
$L_3 =< 715^{-1} >_7 = 1, \ L_4 =< 455^{-1} >_{11} = 3$
$R =< 0 + 1001 + 715 + 0 >_{5005} = 1716$

It can be noticed in Eq. (4) that the route-ID does not depend on the order of the switches, as the finite summation is commutative. And this is also true for the output port computation (Eq. 3). This property is particularly interesting for computing the ERI. To do so, the RDN Controller needs to consider an alternative route for each possible link failure in the primary route. For example, back to Fig. 1(a), where the primary route is $S = \{13, 5, 7\}$, the alternative routes are $S = \{13, 11\}$, $S = \{13, 5, 11\}$ and $S = \{13, 5, 7, 11\}$, for failures on the links $S_{13} \rightarrow S_5$, $S_5 \rightarrow S_7$ and $S_7 \rightarrow dst$, respectively. The RDN Controller aggregates these routes into the resulting $S = \{13, 5, 7, 11\}$. Thus the ERI is computed simply considering the switches from the primary route and the additional switches composing the alternative routes, but using different port IDs for each switch. Due to this aggregation, the size of S is bounded by the number of switches, and not by the number of alternative routes.

## III. RDN Implementation

For validation purposes, a RDN prototype was implemented and experimentally evaluated. In this prototype, the RDN Controller was developed as an application on top of Ryu [8]. Regarding the edge switches, OpenFlow 1.3 enabled switches were used in the Mininet [9] emulation platform.

For the core switches, two prototypes of RDN forwarding mechanism were implemented in two different platforms:

- Open vSwitch [10] (OvS), modifying its version 2.5, in the Mininet;
- Field-Programmable Gate Array (FPGA) hardware using the NetFPGA-SUME platform.

### A. RDN on Open vSwitch and Mininet Platform

When a packet arrives at the ingress switch, the packet is sent to the RDN Controller which selects the primary and emergency routes among all pre-calculated paths between the source and destination. Then, the RDN Controller sends messages (OF messages) to install the necessary rules at the ingress and egress switches.

Table I details the flow rules installed at the edges switches with the corresponding actions for setting a flow between host $src$ and host $dst$, as previously illustrated in Figure 1(a). In the ingress edge switch, the OpenFlow rule includes an action to add the route-IDs to packets' headers, and an action to forward the packets to the core network. Notice that in this prototype, MAC address fields have been used to embed PRI and ERI into packets' headers. So, in the egress edge switch, the rule includes an action to rewrite the original MAC addresses, and an action to forward the packets to the $dst$ host. Flow entries at the edge switches are based on the destination IP address (plus optional VLAN or tenant identifiers).

In terms of modifications to OvS, we present the pseudo-code 1 taken from *xlate_ff_group* function. Actually, we take advantage of the OF1.3 Fast Failover [11] structure for the core switches (line 1). Thus, when the switch receives a packet, it checks if the port is available, then PRI is used (line 3). However, if the switch port is not available, the core switch checks if PRI is different from ERI (line 6). If it is the case, it replaces PRI by ERI (line 7). From now on, the new route-ID is used to bypass the failed link. For the subsequent switches along the route, they just keep doing the modulo operation until the packets reach the edge. Only if a second failure occurs, then the packets will be forwarded to the controller (line 9).

**Pseudocode 1** Forwarding packets within core switches

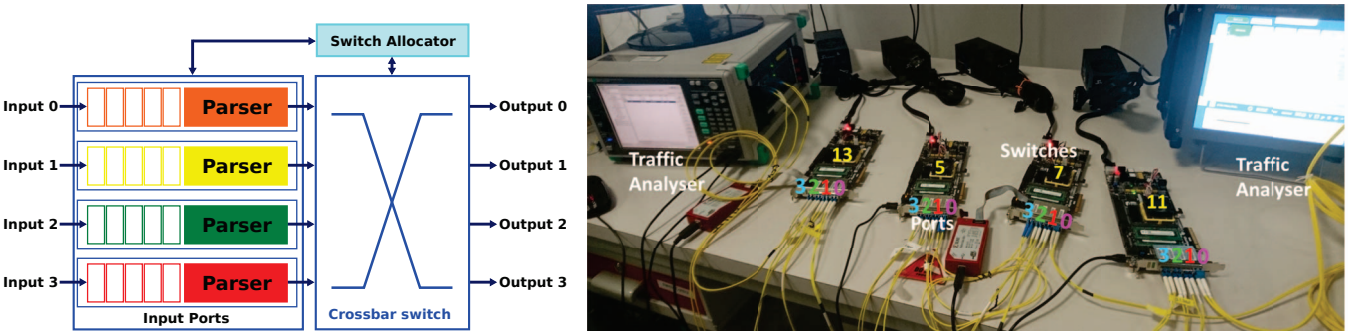
```
1:  function XLATE_RW_GROUP(struct xlate_ctx ...)
2:      ...
3:      if bucket then                          ▷ Primary route available
4:          send pkt next hop
5:      else                                    ▷ Replaces the primary route
6:          if MAC_DST != MAC_SRC then
7:              copy(MAC_SRC to MAC_DST)
                        ▷ Call xlate_rw_group with new MAC_DST
8:          else
9:              send pkt to Centre Control
10:     return -1
11: end function
```

### B. RDN Implementation on FGPA Platform

Fig. 2(a) shows the RDN router architecture with four input and output ports implemented on NetFPGA-SUME platform. Here, the RDN FPGA-based router supports wormhole (WH) routing [12] which is a packet switching technique to reduce buffer space and latency. In WH switching, packets arriving at the input port are routed immediately to the output port as soon as the port is free. In this scenario, switch allocator is used to set the output ports based on route-ID (i.e., PRI or ERI)

Table I
FLOW TABLE ENTRIES IN THE EDGE SWITCHES.

| Flows Direction | Edge switches | Match | Action |
|---|---|---|---|
| Forward Flow | Ingress | MAC_DST: 00:00:00:00:00:02 and MAC_SRC: 00:00:00:00:00:01 | SetField(eth_dst=00:00:00:00:00:D2, eth_src=00:00:00:00:06:B4), output = 4 |
| | Egress | MAC_SRC: 00:00:00:00:06:B4 and IP_DST: 10.0.0.2 | SetField(eth_dst=00:00:00:00:00:01, eth_src=00:00:00:00:00:02), output = 1 |
| Backward Flow | Ingress | MAC_DST: 00:00:00:00:00:01 and MAC_SRC: 00:00:00:00:00:02 | SetField(eth_dst=00:00:00:00:01:6F, eth_src=00:00:00:00:06:28), output = 4 |
| | Egress | MAC_SRC: 00:00:00:00:06:28 and IP_DST: 10.0.0.1 | SetField(eth_dst= 00:00:00:00:00:02, eth_src= 00:00:00:00:00:01), output = 5 |




(a) (b)

Figure 2. (a) router architecture, (b) Experimental testbed setup 10Gbps.

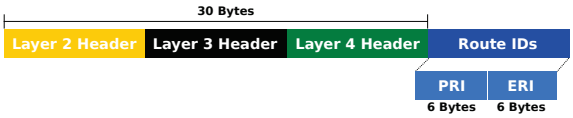



Figure 3. Header format for FPGA experiments.

extracted from the packets by input ports parsers. Note that 10Gbps Ethernet MAC and PCS/PMA module blocks (Xilinx IP cores) are not represented in Fig. 2(a).

The experimental setup is shown in Fig. 2(b). It comprises 4 NetFPGA-SUME boards to evaluate the core network in the proposed programmable RDN architecture, as previously illustrated in Fig. 1. Each SUME FPGA board supports 4 optical small-form factor pluggable (SFP+) transceivers at 10Gbps line rate. Moreover, each board is configured to perform the steps described in Sec. II based on the received PRI or ERI value within each packet, with the corresponding defined switch co-prime value. As it is also shown in Fig. 2(b), two different Anritsu traffic analyzers (MD1230B, MT1100A) were used to generate and monitor the traffic. The considered traffic analyzers perform the edge switch operations (i.e., ingress and egress) on the programmable packet.

Fig. 3 shows the extended high-level packet header format used in this experimental setup. Instead of using MAC address fields for embedding PRI and ERI, here we insert the route-IDs from the $31^{st}$ byte in order to make it evident that the RDN architecture is actually protocol agnostic. In this case, it is necessary to wait for 30 bytes to be received (which takes 4 clock cycles in FPGA logic, i.e. 25.6ns) to be able to read the route-IDs and to compute the modulo operation.

## IV. VALIDATION METHODOLOGY

The validation methodology is structured in two parts. The first part (described in Sec. IV-A) consists of an analytical evaluation of the RDN scalability for EDC topology design. The main goal is to analyze how many bits are needed for RDN encoding with regards to different fan-outs of 2-tier Clos networks topologies that cover mostly EDC deployments [5].

The second part (described in Sec. IV-B, IV-C, IV-D) is devoted to experimental results using our RDN prototype implemented both in the network emulation environment and using the NetFPGA platform. The main evaluation goals include (i) the impact of fast failure reaction on TCP throughput, (ii) a comparison of failure recovery time, and (iii) RDN latency measures in netFPGA with ultra-fast failure reaction.

### A. RDN Scalability for EDC Topology Design

Assuming that Ethernet standard at the core is essentially for framing purposes, we have analyzed the RDN scalability for EDC network topologies considering it as a design choice embedding PRI and ERI into the 48 bits destination and source MAC addresses, respectively.

As PRI and ERI are embedded in the packet (e.g. into header fields), its bit length affects the packet overhead. The maximum number of bits required by route-IDs can be computed as follows:

$$bit\_length(R) = \lceil log_2(M-1) \rceil \quad (9)$$

Eq. (9) states that the higher the value of $M$, the larger the maximum required bit length. Remember that $M$ is the multiplication of the switch-IDs along the primary and alternative routes. Therefore, as more switches are used to build the route, more bits are required to represent the route-IDs (PRI and ERI). This constraint should be considered for implementation purposes. Notice that in the case of ERI, the gain resulting from the aggregation of alternative routes (described in Sec. II-C) is now evident, as it ends up by limiting the number of factors of M.

Since the multi-stage Clos networks are topologies commonly found in enterprise datacentres supporting tens of

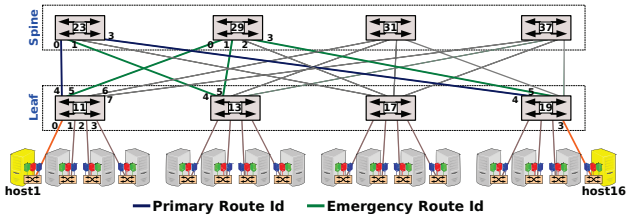


Figure 4. Example of RDN Architecture for EDC 2-tier Clos topology.

Table II
NUMBER OF BITS FOR DIFFERENT CONFIGURATIONS OF 2-TIER CLOS NETWORKS.

| 2-tier setting | Ports | Physical Hosts | PRI | ERI |
|---|---|---|---|---|
| **Spine = 06 Leaf = 06** | 24 | 108 | 19 | 31 |
| | 48 | 252 | 20 | 33 |
| | 96 | 540 | 22 | 36 |
| **Spine = 12 Leaf = 12** | 24 | 144 | 22 | 35 |
| | 48 | 432 | 23 | 37 |
| | 96 | 1008 | 24 | 39 |
| **Spine = 16 Leaf = 16** | 24 | 128 | 23 | 36 |
| | 48 | 512 | 23 | 39 |
| | 96 | 1280 | 25 | 40 |
| **Spine = 24 Leaf = 24** | 48 | 576 | 25 | 41 |
| | 96 | 1728 | 26 | 43 |
| **Spine = 36 Leaf = 36** | 48 | 432 | 27 | 44 |
| | 96 | 2160 | 27 | 45 |
| **Spine = 48 Leaf = 48** | 96 | 2304 | 29 | 47 |

thousands of physical servers [5], our focus in this analysis is on EDC based on 2-tier Clos networks. For example, in a 2-tier Clos network with $v$ spine switches, the RDN controller can easily allocate $v$ disjoint routes (load balancing and traffic engineering) by having each route through a unique spine switch [13].

Fig. 4 shows the RDN architecture for EDC 2-tier Clos topology. For sake of clarification, we removed the RDN Controller. Every switch is identified with its own ID, where its ID must be greater than the number of switch ports (modulo operation constraint). The $src$ is host1 and the $dst$ is host16. PRI is set using the switches $S = \{11, 23, 19\}$ with out ports $P = \{4, 3, 3\}$ and ERI is set using the switches $S = \{11, 29, 19, 23, 13\}$ with out ports $P = \{5, 3, 3, 1, 5\}$.

If a failure happens at the link between $S_{11}$ and $S_{23}$, $S_{11}$ will use port 5 (ERI), then $S_{11}{\rightarrow}S_{29}$, $S_{29}{\rightarrow}S_{19}$ and $S_{19}{\rightarrow}dst$, which corresponds to one of the emergency routes coded by ERI ($S = \{11, 29, 19\}$). If a failure occurs at the link between $S_{23}$ and $S_{19}$ , $S_{23}$ will use port 1, then $S_{23}{\rightarrow}S_{13}$, $S_{13}{\rightarrow}S_{29}$, $S_{29}{\rightarrow}S_{19}$ and $S_{19}{\rightarrow}dst$ which corresponds to another emergency route coded by ERI ($S = \{11, 23, 13, 29, 19\}$). Clearly, there is an aggregation gain for computing ERI if different emergency routes are composed by common switches, such as in this example ($S_{19}$ and $S_{29}$).

We use Eq. 9 to compute the maximum bit length of PRI and ERI. For instance, PRI is set using $S = \{11, 23, 19\}$, and ERI is set using $S = \{11, 23, 19, 13, 29\}$. Thus, PRI and ERI need 13 and 21 bits to be encoded, respectively.

In Table II, we present an analysis of the RDN scalability for different EDC 2-tier Clos topologies (with different fan-outs). Assuming the protection of the entire route and how many bits we need for encoding it, this analysis shows that it is possible to achieve a network with up to 96 switches (96 ports per switch), where the maximum lengths for both PRI and ERI fit in the 48 bits destination and source MAC addresses.

### *B. Impact of Fast Failure Reaction on TCP Throughput*

In order to evaluate the efficiency of the recovery process to link failures, we select different recovery techniques including reactive/proactive protection mechanisms. Our aim is to evaluate the granularity of failure recovery time and its impact on the TCP throughput. The experiments are carried out in the emulated environment where the network topology illustrated by fig. 4 was implemented. During the experiments, links are disconnected using the Linux *ifdown* command.

The failure recovery mechanisms considered for this analysis are the following:

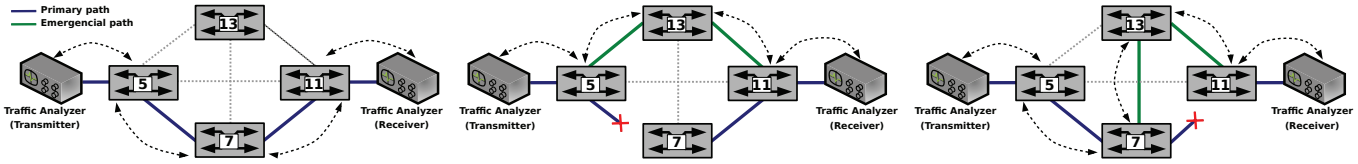
- Reactive: The controller waits for a switch link failure notification; after notification, it updates the flow table rules for all the switches that belong to the selected path; This is a restoration baseline case.
- OpenFlow1.3 Fast Failover [14] (FF): The controller installs FF entries in the switches 11 and 23; but in the case of failure a new entry should be added to switches 29 and 13 via controller.
- Full proactive: FF rules are installed configuring all the switches to have an emergency route to protect the flow of a failure in any link of the entire route. This is the upper bound scenario on failure recovery time.
- RDN: Flows are installed only at edge switches. Core switches react to a link failure just by replacing PRI by ERI in each packet. Output ports are then computed according to the modulo operation of the ERI, steering the flow through the emergency route.

Fig. 6 presents the TCP throughput results (rate at 1 Gbps) for a failure in the $S_{11}{\rightarrow}S_{23}$ link (see fig. 4) for the different failure recovery mechanisms. Clearly, the worst case scenario is the reactive recovery, as it depends on the communication with the control plane, resulting in the poorest TCP performance. Despite the reduction on failure recovery time by the OF Fast Failover, TCP throughput is drastically affected because the route had not been entirely protected. Moreover, although fast failover is an interesting local protection mechanism, there is additional burden to the administrators who have to add specific entries at every switch along the route, for both forward and backward flows.

In contrast, the *Full proactive* leads to a significant reduction on recovery time, keeping the performance of TCP throughput. It is clear that RDN has provided the same TCP performance as Full proactive protection mechanism, but with much less complexity, since there is no need to add flow entries in every switch along the route. This result has shown that (i) using packet rewrite actions at edge switches (insert and remove PRI and ERI) and (ii) triggering emergency routing only by the switch which has detected its local link failure, enable RDN to offer equivalent carrier grade on failure recovery time.

### *C. Failure Recovery Time*

In order to compare the failure recovery time, we evaluate the full proactive mechanism versus RDN protection. To




(a) (b) (c)

Figure 5. Topology full mesh with 4 core switches: (a) without failure (3 hops); (b) link failure $S_5$ to $S_7$ (3 hops) and (c) link failure $S_7$ to $S_{11}$ (4 hops).

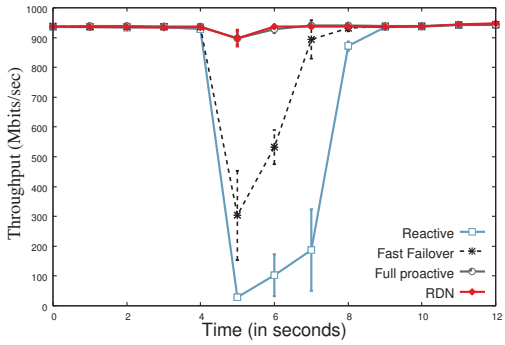



Figure 6. Analysis of TCP throughput in Mbps for recovery time with different techniques and 95% confidence intervals.

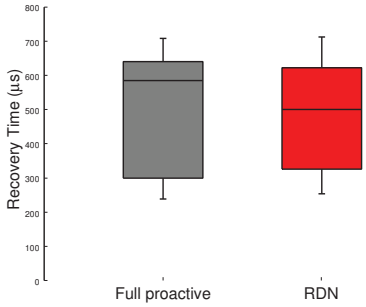



Figure 7. Recovery time for Full proactive protection versus RDN.

measure network with micro-second grade accuracy, we used a tool named Metherxis [15]. A workload is configured to send a traffic at rate of 100Mbps during 30 seconds from host 1 to host 16 for 30 rounds. The time-stamped packets were stored in file for posterior analysis. A link failure is injected to the link between the switches 11 and 23 with *ifdown* command.

Figure 7 shows the recovery time for full proactive protection versus RDN protection in the emulated environment. As can be seen, although there is a larger variability on full proactive than RDN, both techniques are able to provide similar recovery time guarantees (sub-milliseconds, around 550$\mu$s). They can be considered equivalent statistically speaking.

However, RDN does not dependent on the flow table occupation. For instance, to ensure a flow protection it is necessary to install 4 OpenFlow rules in each core switch (2 rules to forward and 2 rules to backward flow). Taking the example of the HP ProVision 10Gbps Ethernet switch [16], [17], which supports 1500 flows in TCAM, just 375 flows could be fully protected. Beyond that, the flow rules will be stored out of the TCAM in slower memories (SRAM), significantly impacting the latency variability [18]. On the other hand, RDN reduces packet forwarding state by embedding the route information in the packet itself. This limits micro-flow state to the edge switches, eliminating the need to maintain any micro-flow state in the core switches.

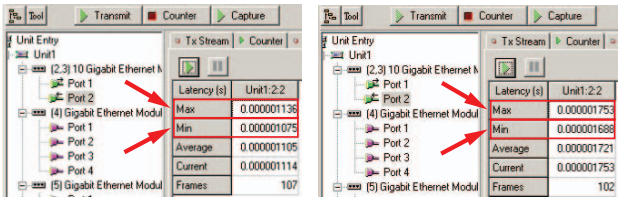
### *D. RDN Low Latency and Fast Failure Reaction*

In order to improve the accuracy on latency measures, a RDN prototype has been implemented in FPGA platform. Each FPGA is composed by 4 Ethernet interfaces of 10G (see Fig. 2(b)) and the validation scenario used is a full mesh topology as shown in Fig. 5. Due to the limitation on the number of FPGA cards (4), only the core network is considered in our testbed with and without link failure.

Fig. 9(a) shows the latency measurements as a function of the number of hops in the core network. To get the exact latency contribution of each hop, we have to subtract the $0.5\mu$s measured in the traffic analyzer Back-to-Back (B2B) port connection (programmable packet length of 1500 bytes) from the total cumulative latency. Thus, the average latency for one RND switch is around $0.6\mu$s. Figure 8 presents screenshots from the traffic analyzer displaying latency measures for experiment runs with one and two hops. Fig.9(a) complements the latency measures showing the latency values to 3 hops ($\approx$ $2.3\mu$s). An important observation is that the latency variability (jitter) is in order of tens of nanoseconds and can be considered as no jitter. It shows a potential packet-switched network with circuit switching guarantees.

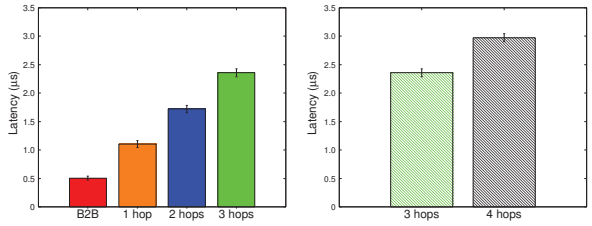
For the cases of link failures depicted in Fig. 5(b)(c), there are two different cases with 3 and 4 hops, respectively. Note that there is no link failure detection mechanism implemented internally in the FPGA. It would be necessary to use additional components (e.g. power meters) in order to detect the link failure. Instead, we have decided to emulate link failures with a special programmable packet. Along with PRI and ERI (previously shown in Fig. 3), an additional 1 byte field is used to carry the ID of the switch which will emulate the link failure. Upon receiving this special packet, the switch whose ID matches the switch-ID carried by the packet (e.g., 5 as in the example shown in Fig. 5(b)) triggers the emergency route configuration by overwriting PRI with ERI in the received packet. In this case, the switch where the link failure is emulated introduces a delay of $25.6ns$ (4 clock cycles) to perform the overwriting process.

Figure 9(b) shows how fast is the failure reaction as it just consists of replacing PRI by ERI and recomputing the modulo from ERI to steer the packets to the emergency port. The latency is around $2.33\mu$s for 3 hops and $2.93\mu$s for 4 hops. An important remark is that for the latency measured within the RDN core, we do not consider the failure detection time




(a) (b)

Figure 8. RDN core switch latency from Traffic Analyzer without failure: (a) 1 hop and (b) 2 hops.




(a) (b)

Figure 9. RDN core switch latency: (a) no failure and (b) with failure.

contribution to the latency, thus the measured latency is clearly a pure failure reaction time.

## V. Related Work

There have been many proposals in the literature to make forwarding decisions based on *flat tags* or *labels* including MPLS [19], VLANs [20] and SDN-based architectures [21], [22], [23]. MPLS [19] is a well-known source-routing protocol that forwards packets by writing and matching on labels attached to packets. Unlike RDN that is *tableless* in the core, MPLS tags instruct the packet to travel hop-by-hop along a label-switched path (LSP) with LDP[1] forwarding tables. RDN allows to steer a flow through any path and service chain while maintaining per-flow state only at the edge nodes.

Path Switching [22] proposes an alternative to MPLS for source routing which has the advantage of encoding forwarding information in a fixed amount of existing space in the packet headers. However, there is only a high-level proof-of-concept with no experimental validation or testbed deployment so that it lacks evidence of its viability.

Segment routing (SR) [24] is a proposal in which packets can be forwarded using SR forwarding tables and segment-IDs attached to packets. An ordered list of segments is encoded as a *stack of labels*. For packets processing, SR requires to rewrite the segment-ID using pop operations per SR node (top of the stack is considered the active segment-ID), whereas MPLS performs *label swaps* where the tag is swapped out for a new tag. In contrast, RDN computes a simple modulo operation without swapping or pop operations per node in the core.

A second bunch of work has been dedicated to fast *failure reaction within network-controlled routing*. In table III, we present an analysis of failure recovery time including a list of protocols that provide protection mechanisms. One of the reasons for recovery times being greater than $50ms$ is the long time to update distributed tables. Thanks to the simplicity of replacing PRI by ERI triggered only by the switch where

[1]Label Distribution Protocol

Table III
Recovery time with protection mechanisms [25].

| | Network Convergence | | | |
|---|---|---|---|---|
| **Protocol** | **>250 ms** | **Sub 250 ms** | **50 - 150 ms** | **≈550$\mu$s** |
| STP (802.1D) | X | | | |
| RSTP (802.1w) | X | | | |
| MSTP (802.1s) | X | | | |
| RPVST+ | X | | | |
| EtherChannel (LACP 802.3ad) | | X | | |
| Flex Links | | | X | |
| MPLS Fast Reroute [26] | | | X | |
| RDN | | | | X |

the failure is detected, RDN allows ultra-fast failure recovery without table updates or additional control messages.

Finally, in our previous work KAR [27], the resilient routing relies on deflection guided mechanism to drive the packets to their destination under the presence of link failure. However, to define a full protection path along the entire route, the length of the bits required to support guided deflections can increase considerably. RDN has extended it to support *deterministic routing* with programmable protection, but also differs fundamentally on the alternative paths encoding. Rather than using guided deflections, RDN relies on two segments (route-IDs) allowing to protect the entire route. Also, as suggested by our FPGA prototype implementation, RDN is easily supported in hardware-based devices. There is no need of a new protocol as it may be implemented by reusing the existing header fields to compactly encode the path attached to packets.

## VI. Conclusion

In this paper, we introduce the concept of programmable Residues Defined Networks (RDN). Programmability in this context refers to the ability to program core switches routing decisions without relying on forwarding states. RDN's main argument is that the decoupling between the edge and core switches brings foundational blocks towards a pragmatic SDN design pattern. Our contributions are on (i) the design of RDN architecture for EDC to evolve the SDN architecture and on (ii) demonstrating the RDN feasibility through realistic implementations and testbed deployment validation. To the best of the authors' knowledge, this is the first implementation of a programmable residues core *network fabric* in NetFPGA, in contrast to purely conceptual proposals [1], [22].

RDN is network topology independent and has shown its potential benefits in EDC built as tier-2 Clos topologies. Results in our testbed show that RDN achieves low latency with switching time per hop ($\approx 0.6\mu$s) and no jitter within the RDN core. In a RDN domain, a network policy can be defined, e.g. service connectivity protection, and the RDN controller is able to instruct the edge nodes in a programmable way to protect the flows. RDN offers carrier grade protection with ultra-fast failure recovery (sub-milliseconds).

As future work, we intend to devote efforts in the application of RDN for convergent networks tailored to cloud infrastructures [28], [29] as an enabler for 5G Networks.

## Acknowledgments

This research has been supported by grants from CNPq, CAPES, FAPES, CTIC, and from European Union's Horizon 2020 under grant agreement no. 688941 (FUTEBOL), and EP/L020009/1 (TOUCAN).

## REFERENCES


[1] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian, "Fabric: A retrospective on evolving sdn," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 85–90. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342459

[2] C. Trois, M. D. D. Fabro, L. C. E. de Bona, and M. Martinello, "A survey on sdn programming languages: Toward a taxonomy," *IEEE Communications Surveys Tutorials*, vol. 18, no. 4, pp. 2687–2712, Fourthquarter 2016.

[3] H. L. Garner, "The residue number system," *Transactions on Electronic Computers*, pp. 140 – 147, june 1959.

[4] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 267–280. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879175

[5] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 503–514. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626316

[6] X. Yang and D. Wetherall, "Source selectable path diversity via routing deflections," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 159–170, Aug. 2006.

[7] C. Ding, D. Pei, and A. Salomaa, *Chinese Remainder Theorem: Applications in Computing, Coding, Cryptography*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1996.

[8] P. T. RYU, "Ryu sdn framework using openflow 1.3," http://osrg.github.io/ryu-book/en/Ryubook.pdf, Feb. 2014.

[9] 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

[10] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending Networking into the Virtualization Layer." in *8th ACM Workshop on Hot Topics inNetworks*, vol. VIII. ACM, 2009, p. 6. [Online]. Available: http://www.icsi.berkeley.edu/pubs/networking/extendingnetworking09.pdf

[11] J. Oostenbrink, N. L. M. van Adrichem, and F. A. Kuipers, "Fast failover of multicast sessions in software-defined networks," *CoRR*, vol. abs/1701.08182, 2017. [Online]. Available: http://arxiv.org/abs/1701.08182

[12] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks," *Computer*, vol. 26, no. 2, pp. 62–76, Feb 1993.

[13] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 465–478. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787507

[14] N. L. M. v. Adrichem, B. J. v. Asten, and F. A. Kuipers, "Fast recovery in software-defined networks," in *Proceedings of the 2014 Third European Workshop on Software Defined Networks*, ser. EWSDN '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 61–66. [Online]. Available: http://dx.doi.org/10.1109/EWSDN.2014.13

[15] D. R. Mafioletti, A. B. Liberatto, R. d. S. Villaça, C. K. Dominicini, M. Martinello, and M. R. N. Ribeiro, "Latency measurement as a virtualized network function using metherxis," *SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 4, pp. 14–16, Dec. 2016. [Online]. Available: http://doi.acm.org/10.1145/3027947.3027950

[16] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "Past: Scalable ethernet for data centers," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413183

[17] K. Agarwal, C. Dixon, E. Rozner, and J. Carter, "Shadow macs: Scalable label-switching for commodity ethernet," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 157–162. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620758

[18] F. Long, Z. Sun, Z. Zhang, H. Chen, and L. Liao, "Research on tcam-based openflow switch platform," in *2012 International Conference on Systems and Informatics (ICSAI2012)*, May 2012, pp. 1218–1221.

[19] E. Rosen, A. Viswanathan, and R. Callon, "RFC 3031: Multiprotocol Label Switching Architecture," IETF, Tech. Rep., 2001. [Online]. Available: www.ietf.org/rfc/rfc3031.txt

[20] "Ieee standard for local and metropolitan area networks - virtual bridged local area networks amendment 12: Forwarding and queuing enhancements for time-sensitive streams," *IEEE Std 802.1Qav-2009 (Amendment to IEEE Std 802.1Q-2005)*, pp. C1–72, Jan 2009.

[21] R. M. Ramos, M. Martinello, and C. E. Rothenberg, "Slickflow: Resilient source routing in data center networks unlocked by openflow," *IEEE Conference on Local Computer Networks*, pp. 01–08, 2013.

[22] A. Hari, T. V. Lakshman, and G. Wilfong, "Path switching: Reduced-state flow handling in sdn using path information," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '15. New York, NY, USA: ACM, 2015, pp. 36:1–36:7. [Online]. Available: http://doi.acm.org/10.1145/2716281.2836121

[23] R. MacDavid, R. Birkner, O. Rottenstreich, A. Gupta, N. Feamster, and J. Rexford, "Concise encoding of flow attributes in sdn switches," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017, pp. 48–60. [Online]. Available: http://doi.acm.org/10.1145/3050220.3050227

[24] E. C. Filsfils, E. S. Previdi, I. C. Systems, B. Decraene, S. Litkowski, Orange, R. Shakir, and J. Communications, "Segment Routing Architecture," Network Working Group, Internet-Draft Segment Routing Architecture draft-ietf-spring-segment-routing-09, 2016, standards Track. [Online]. Available: https://tools.ietf.org/html/draft-ietf-spring-segment-routing-09

[25] C. Cisco, "Deploying the resilient ethernet protocol (rep) in a converged plantwide ethernet system (cpwe) design guide," 2015. [Online]. Available: http://literature.rockwellautomation.com/idc/groups/literature/documents/td/enet-td005_-en-p.pdf

[26] G. Swallow, P. Pan, and A. Atlas, "RSVP-TE fast reroute," RFC 4090, http://www.ietf.org/rfc/rfc4090.txt, May 2005.

[27] R. R. Gomes, A. B. Liberato, C. K. Dominicini, M. R. N. Ribeiro, and M. Martinello, "Kar: Key-for-any-route, a resilient routing system," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, June 2016, pp. 120–127.

[28] M. Channegowda, R. Nejabati, and D. Simeonidou, "Software-defined optical networks technology and infrastructure: Enabling software-defined optical network operations [invited]," *Journal of Optical Communications and Networking*, vol. 5, no. 10, pp. A274–A282, 2013.

[29] J. Blendin, D. Herrmann, M. Wichtlhuber, M. Gunkel, F. Wissel, and D. Hausheer, "Enabling efficient multi-layer repair in elastic optical networks by gradually superimposing SDN," in *12th International Conference on Network and Service Management, CNSM 2016, Montreal, QC, Canada, October 31 - Nov. 4, 2016*, 2016, pp. 118–126. [Online]. Available: https://doi.org/10.1109/CNSM.2016.7818407