Research Article

Scalable Topological Forwarding and Routing Policies in RINA-enabled Programmable Data Centres

Sergio Leon Gaixas*,[1] Jordi Perelló,[1] Davide Careglio,[1] Eduard Grasa,[2] Diego R. López,[3] and Pedro A. Aranda[4]

[1] *Advanced Broadband Communications Center (CCABA), Universitat Politècnica de Catalunya (UPC), Barcelona, Spain*
[2] *Fundació Privada i2CAT, Barcelona, Spain*
[3] *Telefónica I+D, Madrid, Spain*
[4] *Universidad Carlos III de Madrid (UC3M), Madrid, Spain*

Correspondence: *Sergio Leon Gaixas. Email: slgaixas@ac.upc.edu

Received 21 June 2017; Revised 25 August 2017; Accepted -

Summary

Given the current expansion of cloud computing, the expected advent of the Internet of Things (IoT) and the requirements of future 5G network infrastructures, significantly larger pools of computational and storage resources will soon be required. This emphasises the need for more scalable data centres, capable of providing such an amount of resources in a cost-effective way. A quick look into today's commercial data centres shows that they tend to rely on variations of well-defined leaf-spine/Clos Data Centre Network (DCN) topologies, offering low latency, ultra-high bisectional bandwidth and enhanced reliability against concurrent failures. However, DCNs are typically restricted by the use of the TCP/IP protocol suite, suffering limited routing scalability. In this work, we study the benefits that replacing TCP/IP with the Recursive InterNetwork Architecture (RINA) can bring into commercial DCNs, focusing on forwarding and routing scalability. We quantitatively evaluate the benefits that RINA solutions can yield against those based on TCP/IP and highlight how, by deploying RINA, topological routing solutions can improve even more the efficiency of the network. To this goal, we propose a rule-and-exception forwarding policy tailored to the characteristics of several DCN variants, enabling fast forwarding decisions with merely neighbours' information. Upon failures, few exceptions are necessary, whose computation can also profit from the known topology. Extensive numerical results show that the proposed policy requirements depends mainly on the number

of neighbours and concurrent failures in the DCN rather than its size, dramatically reducing the amount of forwarding and routing information stored at DCN nodes.

Keywords: Data centre network, RINA, topological routing, scalability

1 Introduction

Seeking the highest efficiency, uptime and scalability, nowadays' commercial Data Centres (DCs) tend to rely on small variations of well-defined leaf-spine Data Centre Network (DCN) topologies. These topologies offer low latency and ultra-high bandwidth for server-to-server and server-to-edge communication. In addition, the high connectivity of these networks also yield enhanced reliability against multiple concurrent link and node failures across the DCN. The reported Google's and Facebook's DCN topologies, available in references [1] and [2] respectively, are good examples of this tendency. With the increasing usage of cloud computing and moving towards the future Internet of Things (IoT) [3] and 5G network scenarios [4], a plethora of emerging new cloud services are expected to proliferate. To properly face them DCs will be required to grow up even larger in terms of computing and storage resources. Fortunately, leaf-spine and Clos DCN topologies deployed in many DCs to date can be scaled to accommodate these requirements. However, routing and forwarding solutions in DCNs, typically based on the TCP/IP protocol suite, do not scale well. That is, large and unmanageable forwarding tables (at least in the order of several tens of thousands of entries in highly-optimized configurations [5]) should have to be properly handled at DCN nodes. Moreover, IP routing protocols incur in a high communication cost (information exchanged to populate routing tables and re-converge upon changes in the DCN topology). Such limitations of the TCP/IP protocol suite for efficient routing inside DCs have been known for long time, being not originally designed for well-defined (e.g., leaf-spine) DCNs, while being very inflexible to improvements [6]. In contrast to the rigidness of the TCP/IP protocol suite, the clean-slate Recursive InterNetwork Architecture (RINA)[7,8] enables a programmable environment where the network administrator can freely configure Quality of Service (QoS), forwarding, routing or security policies. This opens the path to the deployment of policies tailored to the specific characteristics and needs of any network environment. For example, in a RINA-enabled DCN, forwarding and routing policies can be programmed for superior scalability in leaf-spine topologies, outperforming solutions based on TCP/IP, whose protocols were designed to deliver traffic over any arbitrary topology in a best-effort manner.

In this work, we aim at quantifying the benefits that RINA can bring in large-scale DCs thanks to its programmable behaviour, giving the possibility to deploy customized forwarding and routing policies. In particular, we propose a rule-and-exception forwarding policy that leverages DCN topology knowledge to forward packets to any (or to a subset) of the neighbour devices closest to their destination based on programmable rules. In the non-failure scenario,

this approach requires a minimal amount of information to be stored at any forwarding device as only adjacent neighbour reachability is necessary. This represents a large improvement versus traditional forwarding tables as in IP networks, which can even contain more than one entry per network node (e.g., nodes with a private DC address plus multiple public addresses to allow access to virtual machines running there). When failures occur in the DCN some forwarding rules may fail to successfully deliver packets to destination. In these cases, few exceptions overriding those rules need to be stored at forwarding devices. This is only the time when additional forwarding information is required.

Besides the huge forwarding table size reduction, we also illustrate that knowing the DCN topology characteristics, which can be summarized by merely a few parameters in the case of leaf-spine ones, can substantially reduce the routing communication cost and the path computation burden. Indeed, when all nodes know the DCN topology characteristics, network changes due to failures and repair actions are the only information that must be disseminated. Taking this into account, we propose routing policies that reduce the amount of information shared between nodes. In fact, in our rule-and-exception policy, routing information is only needed for the computation of new exceptions instead of the full forwarding table (or to remove them when a failure has been repaired). This allows bounding the computation of exceptions to only destination nodes in the neighbourhood of failures, which results in a computational cost dependent on the number of concurrent failures in the DCN, rather than its size.

This work extends our previous work in [9], by contemplating additional leaf-spine DCN variations and quantifying the benefits of our forwarding and routing policies in a significantly broader way. Instead of the highly tailored forwarding policies presented in [9], a generalized rule-and-exception forwarding policy is proposed here. This opens its implementation in generic hardware, enabling its deployment in any network that could benefit from the approach proposed here. Finally, in addition to the extended distributed routing approach, a centralized approach to exceptions computation is also introduced in this work.

The remainder of this paper continues as follows. Routing solutions for DCNs available in the literature are reviewed in section 2. Next, in section 3 we introduce the RINA model and its key benefits against the TCP/IP protocol suite. In section 4, we introduce the assumed RINA-enabled DCN scenarios and their characteristics. The proposed forwarding and routing policies are introduced for each of them in sections 5 and 6, respectively. In section 7, we provide numerical results to compare the performance our forwarding and routing policies against current solutions based on TCP/IP in the contemplated DCN scenarios. Finally, section 8 summarises the key achievements of this work.

2 Related work

DCN topologies have been always designed to ensure the maximum cost-efficiency. Even so, common routing and forwarding techniques cannot take profit from their specific topological characteristics. To address these issues, deterministic routing [10] was the scheme initially deployed in many DCs. In such a scheme, nodes are assigned addresses based on their specific topological properties, so that the route between any pair of nodes is known beforehand and does not change over time. These routes are usually encoded in the same packets, in the form of a bit-stream or coordinates (e.g., see [11]). While highly scalable, this rigid scheme has two major drawbacks: the lack of automation in defining addresses, and thus the setup of routes, and the inability to leverage multiple paths (given the identity relationship address=path), preventing any possible recovery actions upon DCN failures.

In fact, any node in a DCN typically communicates with only a small sub-set of neighbours. This may become a favourable scenario for using solutions based on source routing. These solutions can alleviate both routing and forwarding requirements, as only paths to this small sub-set of neighbours have to be computed. Within this family, the valiant routing scheme [12] was proposed as a solution to overcome the shortcomings of deterministic routing, bringing multipath support and load balancing, while still being highly scalable. In this scheme, for a communication between any pair of nodes, a random intermediate address A is firstly selected, and the path is composed by routing packets from source to A and then from A to the destination. This multipath approach provides an easy and direct solution for avoiding non-valid routes in failure scenarios, simply replacing the intermediate node when either the source-to-A or A-to-destination sub-paths became invalid. However, this is achieved at expenses of longer paths, while still lacking an automation process for addressing and having load balancing restricted to the selection of the intermediate node A.

While the valiant routing scheme does not take full profit from the high connectivity of DCNs, alternative source routing solutions also exist to this end. For instance, the Line Speed Publish/Subscribe Inter-Networking (LIPSIN) [13] provides a more robust solution for DCNs with support for multipath, once the flow has been allocated. Unlike in the valiant routing scheme, LIPSIN assigns unique names to the different links in the network (as directed links). When allocating a flow, the forwarding tree from source to destination is computed. Given this tree, a Bloom filter [14] is computed, containing the links in the tree. This is added to the header of the packet, and each node in the path selects the next hop from one of the links within the encoded bloom-filter. To avoid false-positives produced by bloom-filters, when allocating a flow, special entries are added to the nodes in the tree that could generate invalid paths. LIPSIN solves some of the shortcomings of the valiant routing scheme regarding to load balancing. However, it also introduces additional complexity, like the requirement of adding extra entries at intermediate nodes to avoid false-positives.

Currently, given the low-cost of IP-based commodity servers and existing forwarding devices, many large scale DCs have adopted more generic solutions based on the TCP/IP protocol suite. To mitigate the inherent limitations of IP routing solutions, initially designed for an Internet with arbitrary topology, modifications to link-state and path-vector routing have been introduced in order to better accommodate to more specific scenarios. For example, Facebook uses BGP-4 to disseminate routing information in its DCNs [15], which was initially designed for Internet backbone networks. In this way, the need for an address per interface (as commonly required in IP) is avoided by assigning an Autonomous System Number (ASN) per node, while routing on the one node instead of the interface [16]. Nonetheless, designed for more dynamic and heterogeneous networks, BGP-4 suffers from many limitations when facing highly regular DCN topologies with high nodal degrees (e.g., path exploration upon failures, manual configuration of timers, the need to setup TCP connections between any pair of connected ASNs, and so on) Eventually, these schemes imply a high communication cost and require many entries in the routing and forwarding tables to take optimal routing decisions and allow for route recovery upon failures.

Moreover, Software Defined Networking (SDN) [17] is an approach that has been spreading lately, especially in tightly managed networks. SDN builds upon the separation of transport and control planes, enabling programmable networks and flexible management, which allows administrators to better adapt to their particular network requirements. With SDN, most forwarding decisions are centralized, requiring only a few nodes to know the state of the full network. Given the large number of nodes in DCNs, this centralized approach has been lately considered as a substitute of more traditional distributed approaches. For example, Google DCs use a SDN-based approach to control packet forwarding within the DCN [1]. Although this strategy allows taking efficient decisions at low communication cost, it also has multiple drawbacks, like a: full dependency on centralized management to perform any forwarding decision for new flows, or; potential scalability issues since the computational cost of computing centralized decisions increases with the network size or introducing single points of failure.

Among these approaches for DCN routing, we found that each solution has its pros and cons. Source routing solutions, like valiant routing or LIPSIN, provide forwarding decisions that do not require almost any forwarding information to be stored in the network nodes (except some exceptions), but increase the complexity of flow allocation and path recovery. In addition, they require considerably longer packet headers to encode the path information, which has an impact on the resource usage. In contrast, IP-based solutions rely on the use of forwarding tables and generic routing solutions (e.g., BGP-4), resulting in solutions that do not take profit from the network topology. These generic approaches can benefit from cheap commodity hardware, but result in costly routing operations and large forwarding tables. Finally, SDN-like solutions take a more centralized approach where only few powerful nodes manage the entire network. This allows for a more precise management of the network and removes most of the

information required at intermediate nodes. However, they also show potential scalability issues, since everything is managed by the central authorities.

In contrast to the reviewed routing solutions, in this work we propose a different approach. We assume that we can know the entire DCN topology and configure addressing schemes that easily gives us the location of nodes in that topology, which can easily be achieved in RINA as will be discussed in the following section. Taking some knowledge as granted, the amount of information stored at nodes becomes minimal, only requiring to store exceptions to the primary forwarding rules when the DCN topology changes for any reason (e.g., a link or node failure). In order to compute these exceptions, either distributed or centralized routing solutions can be used. These solutions can also take profit from the static knowledge of the network, resulting in fewer and smaller routing updates (lower communication cost) and simpler computations.

3 Recursive InterNetwork Architecture

The Recursive InterNetwork Architecture (RINA) is a clean-slate architecture for computer networking based on the idea that networking is distributed inter-process communication (IPC) and only IPC [18]. As shown in Figure 1, RINA presents a single type of layer, called Distributed IPC Facility (DIF), which repeats as many times and levels as needed by the network designer. This contrasts with the TCP/IP model, in which different layers perform different functions (end-to-end transport, packet forwarding, link management, etc.), offering a different set of services and application programming interfaces (APIs), sometimes repeated at more than one layer. As opposed to TCP/IP, RINA defines its DIF as a programmable layer capable of performing any of the functions needed to provide IPC services to applications or higher level DIFs, offering at each level a common API. All RINA layers use the same two protocols: a data transfer protocol called EFCP (Error and Flow Control Protocol) and an object-oriented application protocol called CDAP (Common Distributed Application Protocol) that carries all the information exchanged by layer management tasks (usually known as control plane in TCP/IP terms). Both protocols are adapted to the different requirements of each layer via policies [19]. Policies are a set of variable behaviours that can customise the different mechanisms available in the two protocols. For example, acknowledgements are a mechanism built into the data transfer protocol (EFCP), but when to acknowledge is a policy. Different forwarding functions can also be plugged into EFCP. CDAP allows all layer management functions (enrolment, routing, namespace management, flow allocation, security management, resource allocation, etc.) to specify its data model as objects, providing a naming scheme and a set of callbacks that are executed in layer management tasks when a particular action on an object is invoked remotely. This programmability allows network administrators to properly configure each DIF with the policies that better adapt to its scope, operating environment and offered levels of service.

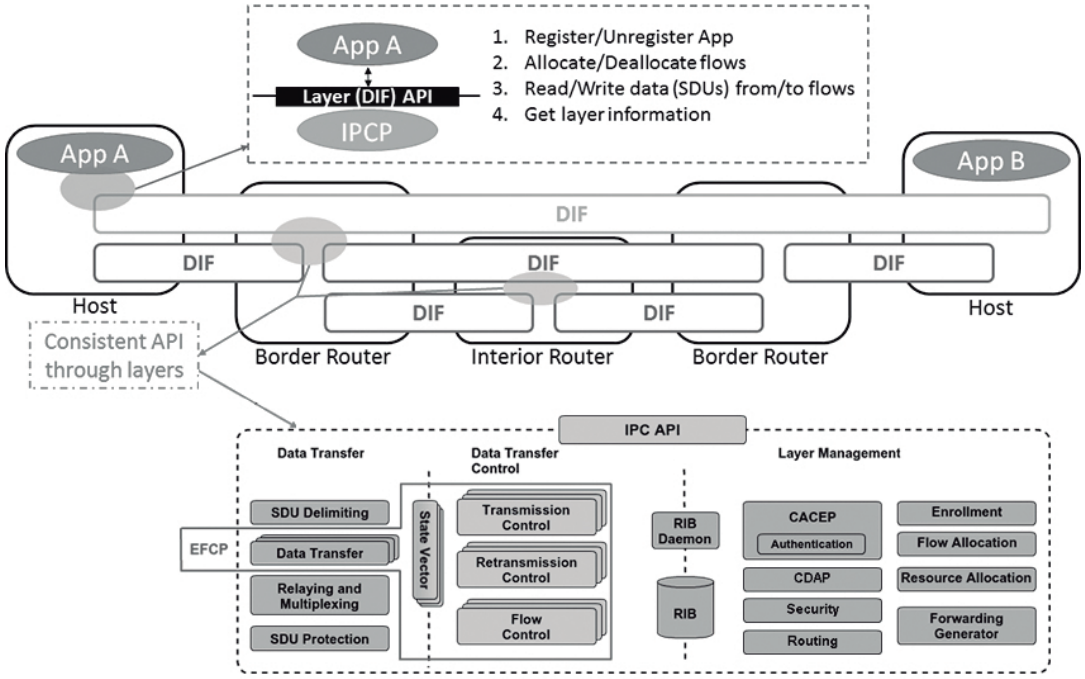

Figure 1: RINA architecture overview.

A DIF is a distributed application, formed by a collection of Application Processes (APs) that collaborate to provide distributed IPC services over a certain scope. The APs that are members of the DIF are called IPC Processes (IPCPs). Addressing within DIFs is based on Jerry Saltzer's idea [20] that a good addressing scheme clearly distinguishes between naming (who), addressing (where) and routing (how), so that a node name does not imply its address or the way to reach it. Therefore, all application processes have names, which uniquely distinguish them in a certain application namespace. These names are location-independent, so that APs can move without losing their identity. Since IPCPs are APs they have an application name, but they also have synonyms that are only unique within a layer (a DIF) and are structured to facilitate routing and forwarding. These synonyms (called addresses) are location dependent but route independent: they encode the information of where the IPCP is located with respect to an abstraction of the connectivity graph of the layer, without impairing how to get there. Our work extensively builds on this property to minimize the size of forwarding tables and routing overhead. This naming and addressing approach avoids the use of default ports or the need to tie addresses to forwarding interfaces, providing a higher degree of scalability and facilitating multi-homing and mobility of IPCP [21]. Moreover, security is improved, as now flows between nodes in an N-level DIF must be requested to, and allocated via, N-1 DIFs to know the N-1 address and port, which also facilitates the monitoring of flows [22].

While all DIFs provide the same kind of service to upper processes, the characteristics of the offered service may vary between DIFs, as mentioned above. In RINA, each DIF defines a set of supported QoS Cubes, namely QoS classes, for flows, providing statistical bounds on metrics like data rate, latency, losses, and so on. With these QoS

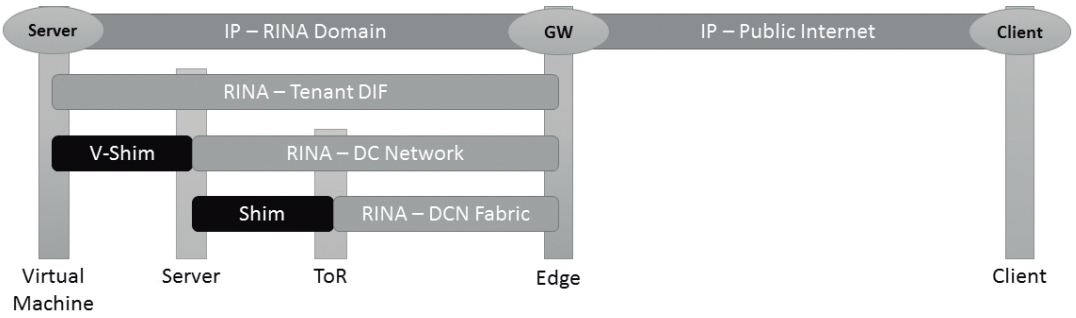

Figure 2: RINA-enabled DC scenario providing IP services.

Cubes, applications are able to request specific bounds in the experienced end-to-end QoS of their flows, for which the DIF will automatically assign the QoS cube that suits them best. In addition, given the recursiveness of DIFs, upper IPC processes are also capable of requesting specific QoS levels for flows to their lower level DIF, making it easier for them to ensure their own QoS Cubes.

The creation and management of DIFs is performed by the Network Management System (NMS), a distributed application composed of one or more Manager(s), together with the Management Agents (MA) located at each RINA device. The NMS is responsible for creating RINA IPCPs in the devices, configuring them according to the requirements of each DIF, triggering neighbour discovery and enrolment, monitoring their status, etc. With the recursive layering of DIFs and the full control of the NMS, most changes in the connectivity between IPCPs in one layer can be completely hidden to upper layers or at least their negative effects mitigated. The commonality across DIFs and the use of a common protocol for management (i.e., the CDAP) greatly simplifies the management of RINA networks compared to legacy architectures [23].

Despite the clear differences between RINA and the TCP/IP model, both are compatible enough to allow for a progressive migration between them at little expense. Indeed, the migration towards RINA does not require turning off the entire Internet overnight to be deployed. Conversely, it can start either by replacing lower layers with RINA, while keeping IP services on top, or by using any existing network protocol (IP, Ethernet, UDP, WDN, etc.) as a bearer for RINA. The use of RINA over existing technologies is done through the introduction of "wrapper" DIFs, referred as shim-DIFs. Shim-DIFs are a special kind of DIF, with limited functionality, which uses IPCPs designed to provide a RINA API for a specific non-RINA technology [24]. Figure 2 shows a simplified example of how RINA could be introduced as a networking model within a DC connected to the current TCP/IP Internet. In this example, a RINA-enabled DC is set and an additional DIF provided to its tenants, offering connectivity between their nodes and some edge routers. Over some of those tenant IPC processes, the IP connectivity between servers and edge routers (gateway) is established, providing IP access from the public Internet, independently from the internal structure of the DCN or where the servers are located.

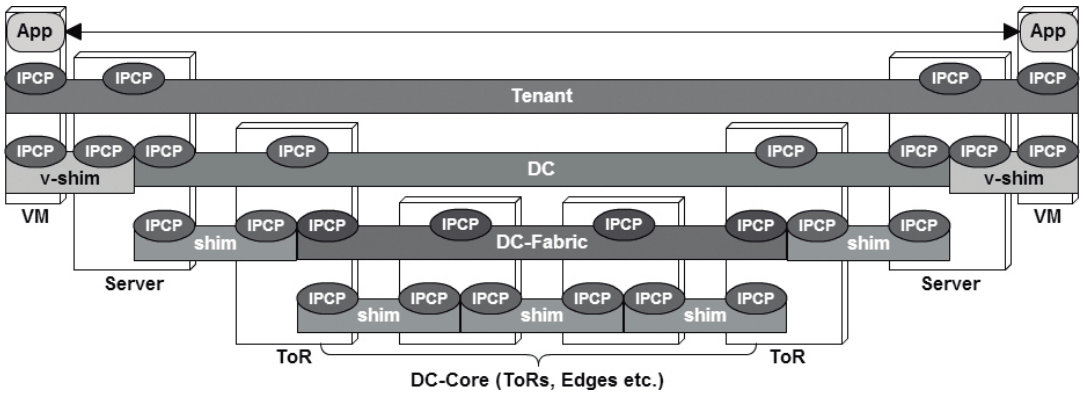

Figure 3: Example of recursive DIF layering in a typical DCN.

4 Scenarios under study

While having the same two protocols and mechanisms at each layer (i.e., DIF), it is possible to configure each DIF instance with policies specifically tailored to its scope (operating environment). This enables an easy and cheap deployment of scenario-optimized solutions, outperforming any generic solution. In this work, we investigate the benefits of a RINA deployment inside a DC, following the DIF setup depicted in Figure 3. Such a RINA-enabled DC is partitioned into three main types of DIFs, covering three different scopes:

1. One DC-Fabric DIF, acting as a large distributed switch that connects all ToR switches and edge routers.
2. One DC DIF, connecting all servers in the DC as a single pool of computation and storage resources.
3. Multiple tenant DIFs, isolated and customized as per each tenant requirements.

Note in the figure that underlying point-to-point links are abstracted as shim (or v-shim) DIFs. This allows abstracting any legacy technology or physical media (e.g., Ethernet, hypervisor VM communication [25], etc.).

In DC and Tenant DIFs, there is only one "eligible" path between any pair of IPCPs. This makes forwarding decisions straightforward and routing unnecessary (e.g., to go from server A to server B in a distinct rack in the DC DIF, traffic must be forwarded to its ToR switch, next to the ToR switch of the rack where server B is located, so as to finally deliver it to server B). In contrast, DC-Fabric DIFs are specifically designed in a way that there exist multiple redundant paths between any pair of nodes, in order to ensure resiliency upon multiple concurrent failures, as well as effective load balancing. Therefore, DC-Fabric DIFs tend to follow well-designed topologies with certain properties, not exploited by generic forwarding and routing solutions as the ones employed in the TCP/IP protocol stack.

Specifically, many large-scale data centres currently built their DCNs relaying on leaf-spine and Clos topologies, with good scalability properties in terms of hardware, reliability and load balancing. In view of this, we focus in this work on quantifying the benefits that forwarding and routing policies specifically tailored to the DCN topological characteristics can bring into DC-Fabric DIFs in a RINA-enabled DC. To this end, we contemplate five different

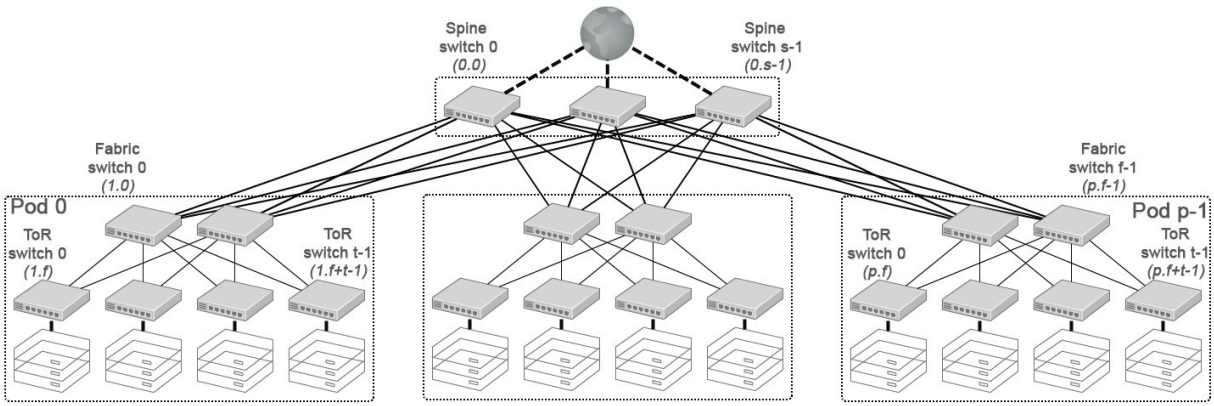

Figure 4: Generic leaf-spine (GLS) DCN topology.

DC-Fabric DIFs, mimicking the topological characteristics of a nowadays' widely accepted generic DCN design (Clos and leaf-spine) besides specific DCN design solutions made publicly available by large corporations, as Google and Facebook, and also a variation of one of them (referred as modified Clos DCN topology).

4.1 Generic leaf-spine (GLS) DCN

The leaf-spine topology (Figure 4) is one of the simplest and more straightforward DCN topology available nowadays. In this topology, we have pods forming a full bipartite graph, with ToR switches at one side and fabric switches at the other. Then, another bipartite graph connects all fabric switches with the spine switches, acting as edge routers at the same time. Interestingly, DCNs following this topology can be fully described by only 4 parameters:

- p : Number of pods in the DCN
- t : Number of ToR switches per pod
- f : Number of fabric switches (aggregators) per pod
- s : Number of spines switches (edge routers) in the DCN

Hence, we propose to use the following possible addressing scheme in a DC-Fabric DIF following this topology, where addresses encode the type of node, as well as its location:

- Spine switch : $0\ .\ spine_{id}$
- Fabric switch : $(1 + pod_{id})\ .\ fabric_{id}$
- ToR switch : $(1 + pod_{id})\ .\ (f + tor_{id})$

With this addressing scheme, for example, the address of ToR switch 5 in pod 3, when having 4 fabric switches per pod, would be <4.9>. It has to be noted that, in all DCN scenarios, we consider all identifiers (for pods, ToRs, etc.) starting at 0. This is the reason why pod 3 takes addresses 4.* instead of 3.*, as it is really the 4th pod in the DC.

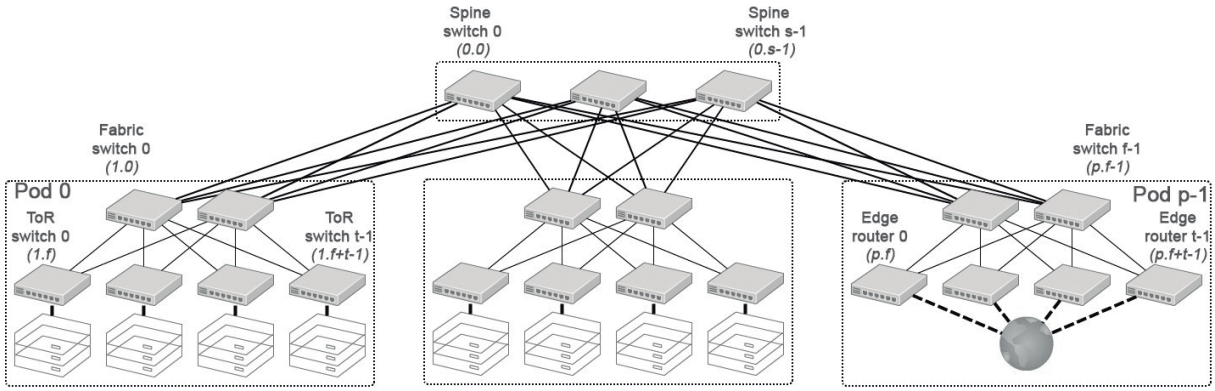

Figure 5: Google's (GO) DCN topology.

We could have used any other topological addressing scheme showing a simple relation between address and location. Even so, we decided on this one, not only given its simplicity, but also that addresses can be encoded with only $log_2(1+p) + log_2(f+t)$ bits. With this in mind, we get that, even for DCNs with twice the number of pods and ToR switches per pod as the largest DCN to date, we would only require merely 2 bytes for addressing within the DCN, an important reduction compared to the 4 and 16 byte-length addresses of IPv4 and IPv6, respectively. In addition, these 2-byte addresses perfectly match one byte per address coordinate, allowing for improved performance.

When an upgrade on the number of fabric switches per pod is planned in the near future, the expected f parameter value should be considered. This will allow a graceful upgrade, without requiring full node renaming in the DC-Fabric DIF, although that could still be easily managed in RINA.

While this topology is widely spread, it is hardly scalable to nowadays' DCN sizes. Since the number of edge routers (spine switches) rises as the network grows up, this requires increasing the node degree of fabric switches dramatically.

4.2 Google's (GO) DCN

As reported in [1], Google decided to deploy a modified version of the generic leaf-spine topology in its DCNs (Figure 5). With the same connectivity between ToR, fabric and spine switches as in the generic leaf-spine DCN topology, Google moves edge routers to their own pod-like sets, instead of locating them at spine switches. This modification entails some benefits and drawbacks against traditional leaf-spine DCNs. It solves the scalability problems of the leaf-spine topology by moving edge routers out of spine switches. In addition, it fosters load balancing and relieves the responsibility of ensuring reliability from the high loaded spine switches. However, this comes at the cost of slightly increasing the path length of such flows to/from the outside of the DC premises. Note that the parameters describing this topology are the same as for the generic leaf-spine DCN. Moreover, the addressing scheme proposed before also fits this DCN topology.

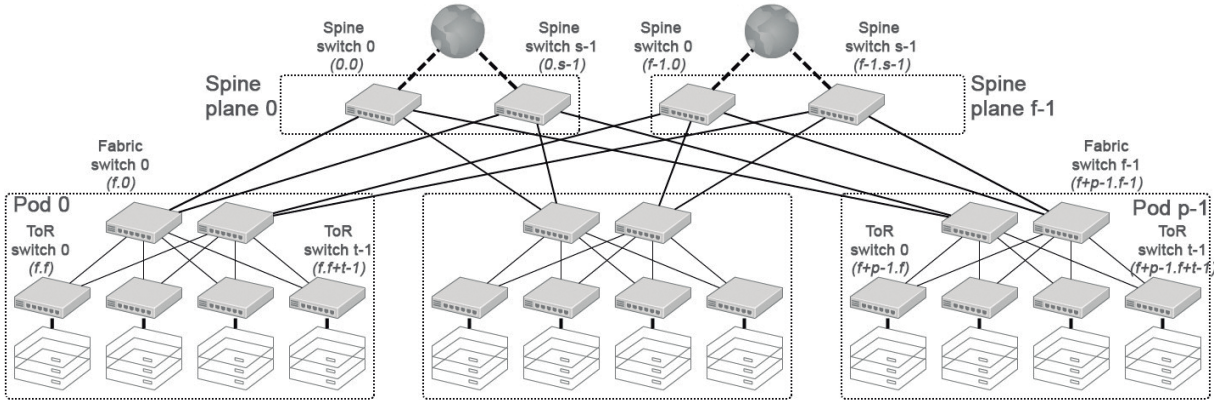

Figure 6: Facebook's Clos based (FB1) DCN topology.

4.3 Facebook's Clos based (FB1) DCN

In contrast to Google, Facebook [2] deploys a Clos DCN topology inside its DCs (Figure 6). In this case, pods follow the same bipartite graph as generic leaf-spine DCNs. However, instead of a unique plane of spine switches connecting all fabric switches, multiple spine planes are deployed, one per fabric switch in the pods (each fabric switch is connected to one and only one spine plane). Compared to the generic leaf-spine topology, Clos DCNs foster better scalability, allowing to increase the number of fabric switches per pod without requiring an important upgrade in terms of ports at spine switches. DCNs following this topology can be described by 4 parameters, similarly as the generic leaf-spine DCN:

- p : Number of pods in the DCN
- t : Number of ToR switches per pod
- f : Number of spine planes = fabric switches per pod
- s : Number of spine switches (edge routers) per spine plane

Given this parametrization, we propose to use the following addressing scheme in the DCN (also similar to that previously proposed for leaf-spine topologies). As before, these addresses encode both the type of node and its location:

- Spine switch : $spine\_plane_{id}$. $spine_{id}$
- Fabric switch : $(f + pod_{id})$. $fabric_{id}$
- ToR switch : $(f + pod_{id})$. $(f + tor_{id})$

With this addressing scheme, for example the address of ToR switch 5 in pod 3, when having 4 fabric switches per pod/spine set, would be <7.9>. Remember that pod, ToR, etc. identifiers start at 0, hence being pod 3 the 4th one and ToR 5 the 6th one in the pod.

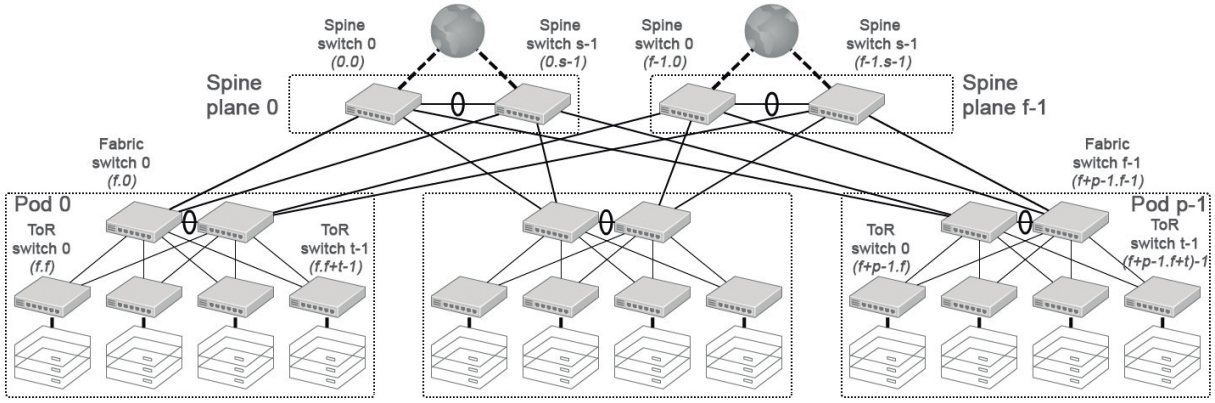

Figure 7: Previous Facebook (FB2) DCN topology.

Note that these addresses can be encoded with only $log_2(f+p)+log_2(f+t)$ bits. If an upgrade to increase the number of spine planes is already planned in the near future, the f value should also be set as expected in order to avoid a full DCN renaming process in the DCN (i.e., as also suggested for the GLS DCN).

4.4 Previous Facebook (FB2) DCN

The DCN topology used at previous Facebook's DCs [26] follows the same Clos topology as for the current ones (Figure 7), but with an added extra layer of protection against failures. In this variation of the Clos topology, all fabric switches within a pod are connected describing a ring topology, and the same is done for spine switches at spine planes. These extra links should not be used in non-failure scenarios. Conversely, they enable short secondary paths between ToR and spine switches when the link that connects the spine switch with the fabric switch of their pod fails, which would otherwise require rerouting the traffic across other pods. These links increase the reliability of the network and avoid using resources of other pods upon failures. Nevertheless, they are protection resources that remain unused most of the time. Having this DCN topology almost the same structure as FB1, it can be described with the same parameters and the same addressing scheme can be used.

4.5 Modified Clos (MC) DCN

This modified Clos DCN topology (Figure 8) follows the idea of Google to move edge routers at the same level as ToR switches, while taking advantage of the enhanced scalability (upgradability) that the Clos topology provides. However, it also carries some of their drawbacks. For instance, it solves one of the Clos topology main problems, namely, the loss of a direct route between a spine switch (that also acts as edge router) and all ToR switches of a pod, when the link between the spine switch and the connected fabric switch of the pod fails. In addition, it yields enhanced load balancing for outgoing flows. As a drawback, the path length of these flows is slightly increased. This

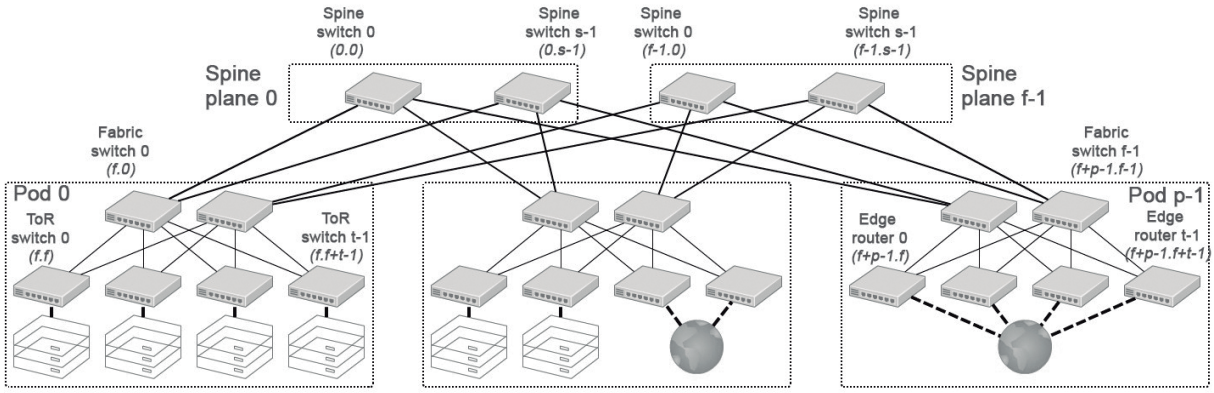

Figure 8: Modified Clos (MC) DCN topology.

DCN topology can be described by the same parameters as the FB1 DCN. Moreover, the same addressing scheme proposed for that DCN topology can be used.

In addition to the benefits of moving edge routers at the ToR level, given that RINA allows for a clean separation between DIFs, addresses in the DC-Fabric DIF are not related to those used in the upper DC DIF. This removes the need for grouping edge routers (to be able to aggregate public IP addresses), thus making possible to place some edge routers directly at pods, thus enabling the possibility of maintaining most outgoing flows within the same pod.

5 Rule-and-exception forwarding policy

A key requirement of any forwarding policy is the ability to quickly decide on the neighbouring node (or nodes) to which a packet has to be forwarded in order to reach its destination. Traditional forwarding tables, allow the aggregation of destination addresses per entry with address masks. However, they do not scale well as the network size grows up. In DCNs deploying the TCP/IP protocol suite, this is even more problematic, as the relation between nodes and addresses requires a whole bunch of public addresses for both tenants with their own IPs and for enabling the mobility of servers and VMs.

RINA inherently removes the need of extra addresses, as no public address is required (IPCP addresses can be independent, i.e., private within each DIF). Nevertheless, the use of conventional forwarding tables in IPCPs with a flat addressing scheme (per DIF) would not solve the scalability problem of regular DCN topologies as the ones reviewed in previous section. Luckily, programmable forwarding policies in RINA are not restricted to traditional forwarding tables. Instead, any forwarding function capable of quickly performing accurate forwarding decisions can be used. To this avail, we leverage on the topological characteristics of typical DCN topologies to design a minimalistic forwarding function that can take profit of them, and be used as a generic substitute to (or upgrade of) generic forwarding tables.

Being aware of a specific leaf-spine or Clos DCN topology like the ones presented before, and the location of the node (encoded in its address), merely storing forwarding information to neighbouring nodes is enough to forward any packet to its destination with simple forwarding rules. When failures occur across the DCN, some routes may become invalid. In these cases, exceptions to primary rules may be required, being the only moment when additional information is needed. Even so, the number of exceptions should be considerably smaller than the number of entries in a traditional forwarding table, as many communications across the DCN remain unaffected by specific link or node failures. Therefore, only storing exceptions to primary rules upon failures can yield a large reduction in terms of memory usage and computational cost compared to a traditional forwarding table. The full Rule-And-Exception (R&E) forwarding policy can be described by the pseudo-code in Algorithm 1.

Algorithm 1 Full Rule-and-exception policy pseudo-code

```
1  *Forward(addr)*
2      if Connected_Neighbour (addr)
3          Forward_to (addr);
4      else
5          Exception e = Search_Exception (addr);
6          if e != null
7              Use Exception (e);
8          else
9              Use Rule (addr);
```

In order to do a lookup for the next hops, the policy firstly checks if the destination is a directly connected neighbour. If not, it searches if an exception is present; if so, the exception is executed to forward the packet; if not, the default rule is applied. This pseudo-code seems more complex than a simple lookup of an entry in a forwarding table. Nonetheless, its primary benefit comes from the fact that searches are only among neighbours plus a small number of exceptions (if any), while rules only consist in a few instructions, as will be detailed later on.

5.1 Groups

In order to reduce the complexity of R&E, our policy uses groups of nodes in order to perform an abstraction of neighbour node addresses and ports. These groups can be seen as named arrays of neighbour pointers (references to connected neighbour nodes) that can be used by R&E to easily define the set of valid ports to reach a destination, independently of the physical interfaces, connected nodes, etc. Moreover, they do not require large lists of valid ports at exception entries, which would be hard to update and would occupy excessive size in memory. The definition of groups is performed by the same policy that populates exceptions, and could be modified as desired depending on

Table 1: Definition of groups for leaf-spine DCN topologies (GLS and GO)

At ToR switches	*A*: Fabric switches valid to reach nodes in the same pod. *B*: Fabric switches valid to reach nodes in other pods.
At fabric switches	*A*: ToR switches. *B*: Spine switches.
At spine switches	*A*: Fabric switches ordered by pod_{id} followed by $fabric_{id}$ position = $fabric_{id} + f * pod_{id} - 1$

Table 2: Definition of groups for Clos DCN topologies (FB1, FB2 and MC)

At ToR switches	*A*: Fabric switches valid to reach nodes in the same pod. *B*: Fabric switches valid to reach nodes in other pods, ordered by $fabric_{id}$ position = $fabric_{id}$
At fabric switches	*A*: ToR switches. *B*: Spine switches.
At spine switches	*A*: Fabric switches ordered by pod_{id} position = $pod_{id} - f$

the current network status and stored exceptions. Tables 1 and 2 depict possible definitions of groups in the DCN scenarios presented in previous section (used to define the forwarding rules presented hereafter).

Apart from these groups, an extra neighbour group is introduced when defining exceptions, where each neighbour is given a fixed index in the array (for spine switches, group A is a synonym of neighbour group). It has to be noted that, while in some cases the order within a group does not matter, in others it can be really important, like in the case of the neighbours group where a neighbour pointer must always be in the same position in the array. In any case, it is possible to have null positions within a group. In these cases, null positions will be simply skipped when executing rules or exceptions.

5.2 Rules

The key elements of the proposed R&E forwarding policy are the forwarding rules. They are simple rules that, given the expected topology of the network, current location and destination address, provide a list of valid neighbours to where a packet can be forwarded. Rules are designed for the non-failure DCN topology. Thus they are neither affected by changes in the network, nor can route around failures by themselves outside the same node (except if the defined groups are changed). Nonetheless, for destinations where primary rules are valid, they provide fast forwarding decisions with minimal information. Tables 3 and 4 show the rules used in the considered DCN topologies to forward packets toward a destination address "a.b", returning in each case the list of valid neighbours either, as a whole group (GROUP X), a range within a group (RANGE $X[min, max]$) or a unique node in a group (NODE $A[index]$).

Table 3: Definition of rules for leaf-spine DCN topologies (GLS and GO)

At ToR switches	if $a = pod_{id}$: return GROUP A else : return GROUP B
At fabric switches	if $a = pod_{id}$: return GROUP A else : return GROUP B
At spine switches	if $a = 0$: return GROUP A else : return RANGE $A[(a-1) * f, a * f - 1]$

Table 4: Definition of rules for Clos DCN topologies (FB1, FB2 and MC)

At ToR switches	if $a = pod_{id}$: return GROUP A else if $a < f$: return NODE $A[a]$ else : return GROUP B
At fabric switches	if $a = fabric_{id}$: return GROUP B else if $a = pod_{id}$ or $a < f$: return GROUP A else : return GROUP B
At spine switches	if $a < f$: return GROUP A else : return NODE $A[a - f]$

As can be seen, forwarding rules are really simple, requiring only a few lines to define how to route packets across the whole DCN. In addition, given their simplicity, they are quite simple to generate as programmable instructions, or even as rule entries for their use in a specialized forwarding hardware. Forwarding rules make full use of the previously defined groups in Table 1 to decide the list of valid neighbours to reach a destination. Specifically, we can obtain two kinds of decisions from them: either to use any node from a group, or a range of nodes from a group (a specific neighbour if the range has a length equal to 1). In fact, rules do not know the content of groups. This has no effect on their usage, though, as unreachable neighbours are removed from groups (leaving a null position in its place if the order is important or redefining the group otherwise). Recall that any null position in a group is not considered when the rule is executed. This has some really great advantages, as a large number of routes that become invalid upon failures can be avoided with only small changes in the group definitions.

Note that the parametrizations presented in the previous section assume the same number of ToR and fabric switches at each pod, and the same number of spine switches at each spine planes (if there is more than one). In other cases, while the proposed solutions would still be valid, possible modifications of rules might be required to accommodate such changes. For example, at spine switches in leaf-spine DCNs, with a varying number of fabric switches per pod, we could define f as the maximum number of fabric switches among all pods, filling the unused positions with null pointers (remaining then valid the current rules). Alternatively, if the hardware allows it, it would be simpler to define a group per pod and replace the rule as "Any fabric switch of group pod_{id}". Also, note that, as rules contemplate non-failure scenarios, they are not affected by failures that do not affect primary paths.

5.3 Exceptions

Upon failures across the DCN, some of the primary rules may become invalid to reach a specific destination (or range of destinations). Sometimes, a simple redefinition of a group should be enough to avoid failed paths, but this may not always be possible. In such cases, we need to record a specific exception for this destination (or destinations, e.g., a pod).

These exceptions, while being similar to traditional forwarding entries, are only required upon certain failure scenarios. Its number is considerably smaller than the number of entries that a traditional forwarding table would require, since most communications across the DCN remain unaffected by specific link or node failures. In addition, we can also reduce the amount of required exceptions even more if we consider that end-to-end flows across the DC fabric DIF are only between ToR switches or ToR switches and Edge routers.

Only storing exceptions to primary rules upon failures can yield a large reduction in terms of memory usage and computational cost, compared to a traditional forwarding table. Additionally, taking profit from the defined groups and taking into consideration the high number of valid ports for some exceptions, we consider 7 possible types of exceptions in order to minimize the space required to encode them:

- UNREACHABLE: The destination cannot be reached.
- ANY: Any of the neighbours group.
- COMMON: List of valid positions from the neighbours group.
- REVERSE: List of invalid positions from the neighbours group.
- GROUP ANY: Any of the specified group.
- GROUP COMMON: List of valid positions from the specified group.
- GROUP REVERSE: List of invalid positions from the specified group.

The most important variation compared to a traditional forwarding table probably is the use of reverse entries (i.e., REVERSE, GROUP REVERSE). These entries have a large effect in reducing the size of exceptions in nodes where most available neighbours are still valid to reach a destination, allowing the generation of exceptions such as "To reach X use any of G except Y". In addition, while all types of exceptions could be expressed as "COMMON" or "GROUP REVERSE", we have intentionally considered all of them to clearly show all possible variations.

5.4 Example

Now let us see a small example to see how the R&E forwarding policy works. To this goal, let us consider the small network in Figure 9, that is, a reduced example of a MC DCN with few failures in it.

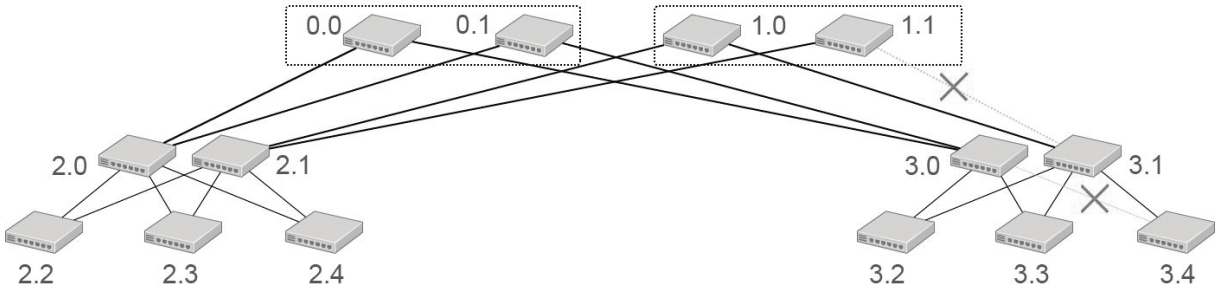

Figure 9: Modified Clos DCN topology with failures.

Firstly, let us see how failures affect to the definition of the neighbour groups in some nodes. In this case, no node of pod 0 (2.*) and spine plane 0 (0.*) is affected, as they are far from the failure points. On the other hand, all nodes with link failures (1.1, 3.0, 3.1 and 3.4) get some of their groups modified, removing the disconnected neighbour from them (e.g. node 3.4 redefines groups A and B as 3.1).

Regarding the exceptions, we see that ToR switches in both pods require exceptions to ToR 3.4 (except for itself). As for ToRs in pod 0 (2.2, 2.3 and 2.4), they need to forward the traffic to 2.1, and for ToRs in the same pod, to fabric switch 3.1. At fabric switches, we get a more varied scenario. In this case, node 2.0 may either get an unreachable exception for destination 3.4 or one using any ToR neighbour to reach it (Group A), as ToR 3.4 is disconnected from its fabric switch in that spine plane. On the other hand, at 2.1, the failure between 3.0 and 3.4 does not affect, but, as the fabric switch 3.1 is disconnected from the spine switch 1.1, it requires an exception to the whole pod 1 (3.*) directing to 1.0. In contrast, fabric switches in pod 1 (3.0 and 3.1) do not require any exception thanks to their modified groups of neighbours. Finally, spine switches do not store any exception.

Considering the state of the forwarding policy in this scenario, let us see how forwarding would be done between ToR nodes in the same or different pods.

From 2.2 to 2.3:

- At 2.2 (ToR). Destination in same pod, use any of group A $\{2.0, 2.1\}$.
- At 2.0 or 2.1 (Fabric). Destination is a connected neighbour, forward.

From 3.2 to 3.4:

- At 3.2 (ToR). There is an exception to node 3.4, forward to 3.1.
- At 3.1 (Fabric). Destination is a connected neighbour, forward.

From 2.2 to 3.2:

- At 2.2 (ToR). Destination in another pod, use any of group B $\{2.0, 2.1\}$.
- If chosen 2.0
  - At 2.0 (Fabric). Destination in another pod, use any of group B $\{0.0, 0.1\}$.

  - At 0.0 or 0.1 (Spine). Forward to Fabric switch of destination pod {3.0}.
  - At 3.0 (Fabric). Destination is a connected neighbour, forward.

- If chosen 2.1
  - At 2.1 (Fabric). There is an exception to pod 3.∗, forward to 1.0.
  - At 1.0 (Spine). Forward to Fabric switch of destination pod {3.1}.
  - At 3.1 (Fabric). Destination is a connected neighbour, forward.

From 2.2 to 3.4:

- At 2.2 (ToR). There is an exception to node 3.4, forward to 2.1.
- At 2.1 (Fabric). There is an exception to pod 3.∗, forward to 1.0.
- At 1.0 (Spine). Forward to Fabric switch of destination pod {3.1}.
- At 3.1 (Fabric). Destination is a connected neighbour, forward.

5.5 Extending the policy

So far, we have presented the complete forwarding policy, with some exemplary configurations for the defined scenarios. However, most implementation decisions are left open, allowing the policy to be extended in multiple ways. For example, given the fast computation requirements of forwarding policies, they could easily be implemented in hardware. A benefit of our approach is that, while taking advantage of the DCN topology knowledge, it is generic enough to allow using the same hardware, independently of the scenario (even as a replacement of a traditional forwarding table) and only relying on exceptions. This makes the presented policy an interesting substitute of traditional forwarding tables for future RINA-based hardware.

When it comes to a potential implementation, there are multiple extensions that can be considered, either to enhance its behaviour or to extend the range of supported scenarios. An example can be load balancing decisions between all available neighbours. It may be important for a flow to always follow the same path if possible. To this end, two options can be considered, either storing the results in a small cache to follow always the same paths, or taking load balancing decisions based on a hash value of the flow identifier. Another extension that can be considered is the addition of QoS metrics into rules and exceptions. While this does not affect the current scenarios, as we only consider reachability, it is something to consider in other cases where a generic implementation can be used (e.g., routing urgent QoS classes with stationary latency as metric and the rest based on the number of end-to-end hops of the routes).

6 Routing policies. Computation of exceptions

The forwarding policy described in previous section requires knowledge about the affected routes to destinations upon failures and how to alternatively reach them. Routing policies are responsible for providing enough information to populate the exceptions to primary forwarding rules. This does not substantially differ from what would be done with a traditional forwarding table. In order to compute exceptions, we could use a generic link-state or distance-vector routing protocol, and simply compare the resulting entries with those that forwarding rules would provide. Even so, in the same way as how rules allow us to reduce the amount of information required for forwarding, we could also enhance routing by exploiting the complete knowledge about the DCN topology that all nodes have. Indeed, there is no need for all nodes to propagate the state of operational resources across the network or to compute routes to all nodes, but only propagate and compute failure information. To this end, we propose distributed and centralized routing policies that take advantage of the topology of the DCN and failure information to reduce both computational and communication costs required to compute the exceptions to primary rules.

6.1 Distributed routing

Generic link-state algorithms assume no knowledge about the network graph and even from the expected connectivity at the current node. Therefore, nodes need to share all their connectivity information in order to compute how packets should be forwarded. When considering a network where the non-failure network graph is known beforehand by all nodes, then these restrictions disappear, becoming only necessary to disseminate network changes. In order to propagate failure information, we propose a variation of link-state routing, based on the propagation of failed links informations solely.

In this routing policy, we assign to each link a unique name (e.g., link name = src.dst). In order to sort updates, each link has an update-sequence-Id, synchronized between link endpoints, incremented each time the link status changes from ON-OFF and vice versa. When a link status change occurs, both endpoints propagate the new status to all their neighbours that, not knowing the new status already, continue propagating it until all nodes are aware of the change. Therefore, there is no big difference with any generic link-state routing. The policy propagates link instead of node status, halving the amount of updates in the best case (i.e., the status of one link instead of two nodes is propagated). The bigger improvement comes from the fact that we only need to propagate and store failure-related information. Hence, the network can run without specific routing, provided that (single- and multi-) failure situations are not too common over time.

Nevertheless, there exist some requirements for the proposed routing policy. The first one is that, the bootstrapping of the routing policy has to be slightly delayed from the initialization of the network. This has to be done in order

to avoid all nodes sending updates of disconnected neighbours for those not yet connected links. This should not be a problem, but something to be taken into account.

Another issue is to decide at what moment an old update can be discarded. This is the case when a failure is repaired and an ON update is propagated in the network to notify the change in the topology; as the network is back to the non-failure state, this information can be discarded. In order to allow all nodes to get this ON update, this information is stored during a limited period of time. Nonetheless, it could happen during a failure that a node or a group of nodes remains disconnected from the rest of the network, which causes that an previously discarded update never reaches them. In this case, there are two possible solutions: nodes with one or more failures may not drop old updates, avoiding ON updates to be lost before reaching the entire network; or a reactive approach can be taken, in which a new update is propagated when an old update arrives at its source.

With the expected knowledge of the network graph and shared knowledge about link failures, computing either the forwarding table or, in this case, the exceptions, may be something as trivial as using the Dijkstra algorithm to compute the shortest path to each destination. However, we are only interested in computing those exceptions for currently unreachable destinations. Instead of re-computing the entire forwarding table using Dijkstra we propose algorithms that take advantage from the DCN topology knowledge, focusing the search on such problematic DCN regions (where unreachable destinations are), while allowing to add some routing restrictions if desired.

Algorithm 2 Exception computation pseudo-code at a ToR switch in the MC topology (distributed routing)

```
1  Clean old exceptions (e.g., all from pod if has new changes)
2  Parse and sort new link failures:
3      Pod -> {ToR switch -> Fabrics switch, Fabric switch -> ToR switch, Fabric switch -> Spine switch}
4      Fabric switch -> {Pod -> Spine switch, Spine switch -> Pod}
5  Initialise groupA and groupB as a list of alive neighbours.
6  Check in-pod failures (if changes in pod):
7      Remove from groupA all fabric switches of the same pod with problems reaching all other ToR
           switches or edge routers.
8      For Each other ToR switch or edge router with problems in the same pod, create exception if cannot
           be reached through all groupA.
9  Check out-pod failures (for changes in pods):
10 For each alive neighbour:
11     If disconnected from all spine switches, remove from groupB and check next.
12         Check pods with problems at the same fabric switch and mark as unreachable through the current
               neighbour if there are no shared spine switches available or the destination fabric switch
               is not connected to any ToR switch or edge router.
13     Create pod exceptions to pods unreachable by all fabric switches in groupB.
14     For each pod with problems at Tor switches or edge routers:
15         Initialize valids as groupB or the list of valids if it has an exception.
16         For each ToR switch and edge router with problems in the pod:
17             Generate an exception if cannot be reach from all valids for pod.
18 Update groups A and B in forwarding policy as groupA and group B respectively.
19 Encode exceptions and replace update the forwarding policy.
```

In Algorithm 2, we find a simple example of how exceptions could be computed at ToR switches and edge routers in a DCN describing the MC topology (with similar algorithms being possible for other nodes and scenarios). In order to compute exceptions, we first do a fast pass cleaning the previous state, clearing old exceptions (e.g. towards pod new failure/recovery) and restarting neighbour groups, as well as parsing the known failures. Then, we search for reachability problems within our same pod (if any failure), removing from group A those neighbours disconnected from the rest of the pod and creating exceptions to ToRs with failures. Then, if there are failures outside our pod, first we check if our neighbours are connected to the other pods (removing them from group B if not). Now, for each pod with failures, we search for problems reaching either the whole pod or specific ToRs in it. Finally, with the groups and exceptions recomputed, we update the forwarding policy with the new information.

In this example, we can see that, while algorithms required in these cases are more complex than general solutions and are completely dependent on the topology as well, they significantly reduce the computational cost of computing exceptions. Another benefit of the algorithms used to compute exceptions is that they can be designed with some routing rules in mind. For example, considering that only flows to ToR switches and edge routers will be established (apart from the point-to-point ones), we may contemplate only exceptions toward entire pods and specific ToR switches and edge routers. In this way, we can skip exceptions to fabric and spine switches, avoiding at the same time paths through fabric switches not connected to any ToR switch or edge router.

In order to compute exceptions in a fast way, those algorithms do not consider finding the best path under any possible failure scenario. Instead, they limit the resulting paths to only those close to the optimal ones. Such constraints may be restrictive in some cases (e.g., the pseudo-code in Algorithm 2 considers only optimal paths, while it could have also considered sub-optimal paths within the same pod). However, they still do not represent a true reduction of the forwarding capabilities of the DCN, given the high available connectivity. In contrast, these constraints allow us to define what we consider as invalid paths, namely, paths for which their performance would be under our expectations, introducing unacceptably large latency and extra bandwidth consumption. In those really infrequent cases where two nodes become unreachable, it would certainly be more cost-effective to move a virtual machine to an alternative reachable node.

6.2 Centralised routing

As previously discussed, the SDN paradigm [17] fosters centralisation of network control to relieve the computational requirements of the forwarding devices. Following on this approach, we also propose a variation of the previous routing policy that takes advantage from the computational resources available in the DC to centralize the computation of exceptions. In Figure 10, we can see a small example of this solution that could be deployed in a DCN describing the MC topology.

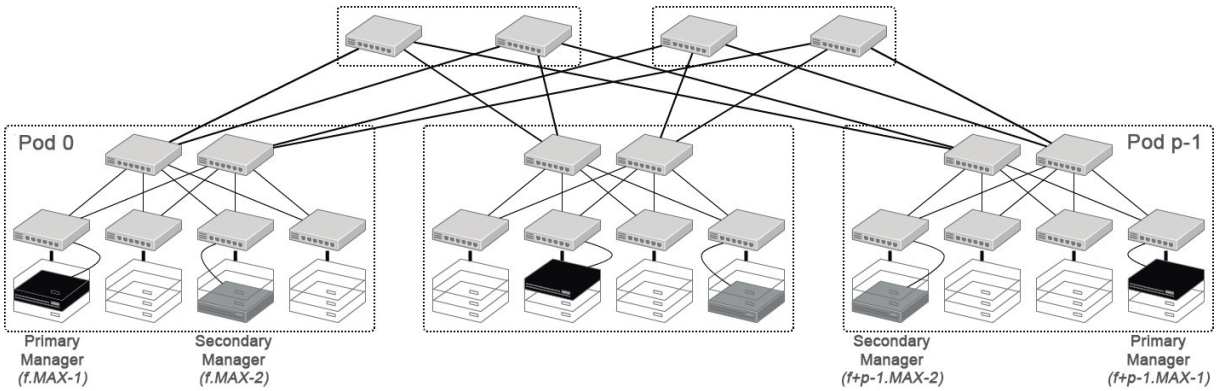

Figure 10: Location of managers for centralised routing in a DCN describing the MC topology.

In this solution, we have two or more servers per pod acting as primary (P) or secondary (S) routing managers. These managers are the ones in charge of configuring groups and exceptions for all ToR switches, edge routers and fabric switches in the pod. In contrast, the rest of nodes only setup their forwarding policies and perform quick modifications to groups (e.g., when a link goes down, before advertising the event to the managers).

Managers are assigned addresses in the form of "$pod_{id}.manager_{id}$", similar to ToR switches, so that forwarding toward them can be done by forwarding rules. In addition, an anycast address "$pod_{id}.MAX$" may also be used as an alias to the primary manager (or the next reachable secondary one), so that the destination of routing updates does not depend on the specific addresses of the managers. As nodes within the pod cannot rely on the managers to compute the paths toward them, a limited distance-vector routing algorithm is used to compute the paths to all managers in the pod, and specific exceptions are added if needed.

When a node in the pod detects a failure or recovery event, it informs the pod primary manager about the change. Once the manager gets the update, it synchronizes its knowledge with the rest of managers in the same pod. Then, it computes a baked update with information about how to reach ToR switches and edge routers, whose reachability is affected, and sends it to the primary managers of other pods. Finally, it computes and propagates the exceptions for all the nodes in the pod. Figure 11 depicts a simplified version of the message propagation between nodes and managers in case of a failure.

When a failure situation is detected at spine switches, they inform the managers of all connected pods, but no exception is computed for them. This is a small simplification, based on the fact that there are no end-to-end flows directed to any spine switch (apart from possible ACKs to routing updates). Given that, any possible exception at a spine switch would only be used for a small period of time before a new stable state avoiding its usage is established. This also avoids possible forwarding loops, until reaching that new stable state.

Apart from the benefits that moving most of the routing computation to a few servers entails in terms of requiring less computational resources at forwarding devices (thus making them cheaper), this solution employs available

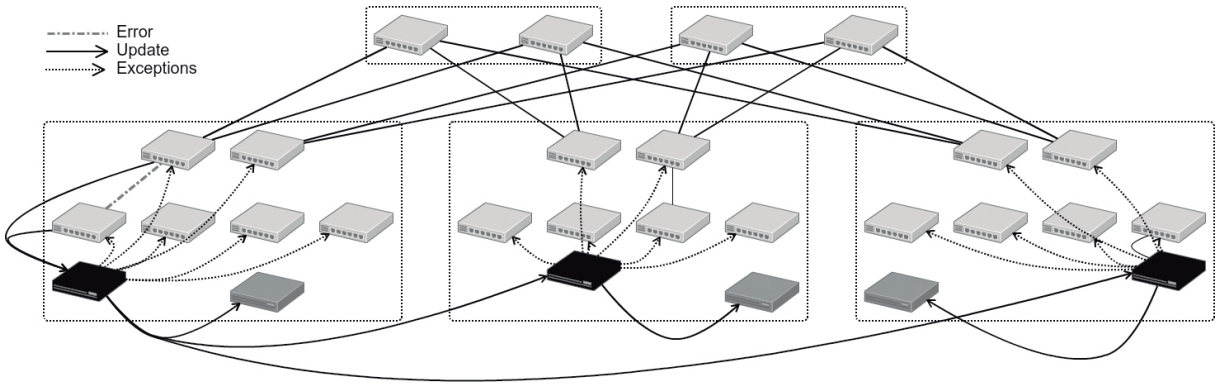

Figure 11: Location of managers for centralised routing in a DCN describing the MC topology.

resources. Also, given that updates only need to be shared between managers, and that not all failure situations require new exceptions, the total number of routing messages ends being significantly reduced compared to any distributed solution, where any change could end being propagated through the entire network.

In Algorithm 3, there is a small example of the baked data shared with managers in other pods and how exceptions and groups can be computed after changes in a DCN describing the MC topology (similar algorithms could be used for the other topologies). In this example, each manager manages a unique pod and, therefore, shares failures and computes exceptions for its nodes. To compute exceptions, it firstly computes default groups A and B for ToR switches, given the connectivity of fabric switches, and specifies them for each ToR depending on their failures. Given the already computed distance matrix, it searches for exceptions within the same pod. Then, for each other pod with known failures, it searches for disconnected fabric switches in those pods. Next, it uses the distance between ToRs and fabric switches in each side to compute exceptions to the other pods and their ToRs. Regarding the updates, we share baked information about the connectivity of edge routers, ToR and fabric switches in the pod. Instead of the raw list of failures, we precompute if there are unreachable fabric switches and how far ToR switches and edge routers are from fabric switches. These baked exceptions per pod allows us to greatly reduce the computational cost compared considering the entire network connectivity at the same time, computing exceptions to each pod once at a time.

Unlike with distributed routing, we here relax the restrictions on valid paths. While we do not allow using intermediate pods when forwarding, we allow the use of any secondary path within both source and destination pods. Even so, as exceptions are extracted from already computed distance matrices of pods, the resulting complexity is not higher than considering only optimal paths. In addition, as failures in a pod do not affect the way to reach other pods, only exceptions to the affected pod have to be recalculated upon changes.

Algorithm 3 Exception computation pseudo-code at a pod manager in a DCN describing the MC topology (centralised routing)

```
--Baking updates--
From failures at fabric switches
    List of fabric switches disconnected from all spine switches and or ToR switches and  edge routers.
    List of failures towards spine switches from connected fabric switches, if any.
Compute the distance matrix of nodes in the pod:
    For ToR switches and edge routers without the same distance to all connected fabric  switches,
        share the distance towards all the pod fabric switches.

--Computing exceptions--
For each fabric switch:
    Check connectivity with ToR switches and edge routers and compute group A.
        If not disconnected, add to default A.
    Check connectivity with spine switches and compute group B.
        If not disconnected, add to default B.
For each ToR switch and edge router:
    Compute groups A and B from the neighbours with minimum distance to any of default A and B
        respectively.

=Compute in-pod exceptions=
Exceptions from the distance matrix comparing to groups.

=Compute out-pod exceptions for pods with failures=
For each pod with some failures:
    Compute the list of fabric switches that reach a connected fabric switch at the destination pod.
    If all disconnected, create an unreachable pod exception at all nodes and go to next problematic
        pod.
    For each ToR switch and edge router:
        Check if neighbours of group B takes the same minimum distance to reach any of the fabric
            switches connected to the destination pod, if not, create an exception.
    For ToR switches and edge routers with failures at destination
        Compute the distance from our fabric switch to the node from connectivity and shared distances.
        Add unreachable exception at fabric switch if shared distance for it is INFINITE.
    From each ToR switch and edge router:
        Check if neighbours of group B take the same minimum distance to reach the node (distance to
            fabric switch + from fabric switch to node), if not, create an exception.
```

Note that in this scenario we assumed ToR switches (and therefore servers) at each pod. If not, this solution would be still possible by adding a manager server to those pods with only edge routers therein. This happens in DCNs describing the GO topology.

Another solution could be to have manager servers connected directly to fabric switches and interconnected between them, as can be seen in Figure 12, where few primary and backup managers can be used to manage the full network in a scalable way. This solution does not use computational resources initially available in the DCN, but works well with any of the considered topologies, reducing at the same time the cost of routing updates. In this case, the number of managers can be reduced, as each manager can compute the exceptions of more than one pod. Moreover, only

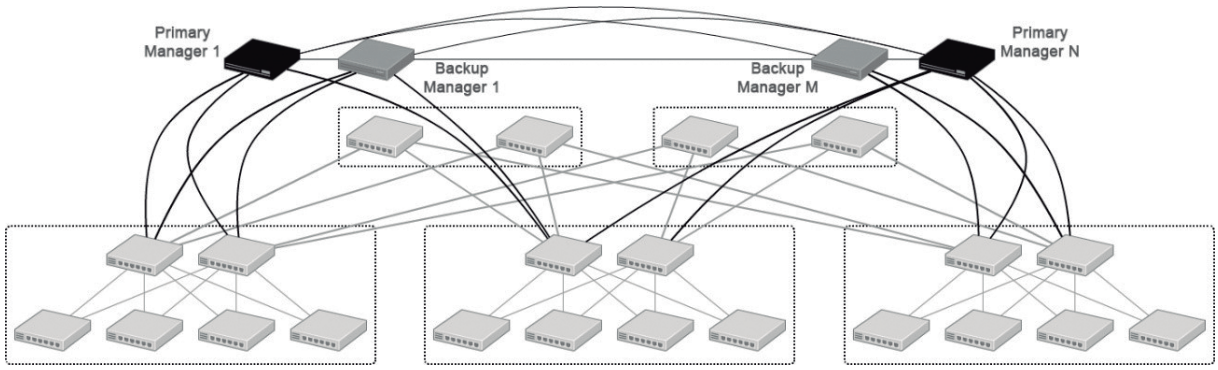

Figure 12: Alternative manager placement for centralized routing at a DCN describing the MC topology.

status updates from fabric switches are required to know whether a link or node is down (considering in this case as down a fabric switch non-reachable through any manager). As a side note, since the management layer would be something fully separated from the forwarding across the DCN, it is not required for managers to have its own addresses within the DCN address space.

7 Numerical results

Current forwarding and routing policies based on TCP/IP impose multiple limitations that the use of the RINA architecture already solves. For instance, in RINA we are not forced to use 4 or 16-byte addresses as imposed by IPv4 and IPv6, respectively, but we can use scenario-specific addresses, resulting by itself in both smaller memory usage and routing updates. In addition, as RINA uses a recursive layering and a naming scheme that differentiates node location and address, servers and virtual machines addresses neither need to be known nor affect routing in the Fabric DIF. This facilitates multi-homing and mobility, reducing the scope of routing within the DCN. Such benefits of adopting RINA are enough to contemplate its usage inside DCs. Nevertheless, we also want to quantitatively evaluate in this paper the scalability of the proposed forwarding and routing policies against that of currently available solutions migrated "'as-is"' (i.e., without any change in their behaviour) into RINA.

In order to present the benefits of topological solutions in a DC scenario, we start comparing the requirements of forwarding tables and the R&E forwarding policy in both non-failure and failure scenarios. To this goal, we firstly compare the number of entries that would be required in a traditional forwarding table and their size, against those required using our R&E solution. In order to perform this comparison, we consider all DCN topologies presented in Table 5, all of them representing large DCs with 8192 ToR switches and 2048 edge routers. Note that a configuration like this one is quite unrealistic in the GLS-based DCN, being impossible to scale that topology in terms of edge routers without incrementing dramatically the size of fabric switches (number of ports). Even so, we keep this scenario in the results, as it provides us an illustrative example, where fabric switches have many neighbours.

Table 5: Assumed values for the parameters describing each of the considered DCN topologies

Scenario	p	t	f	s
Generic Leaf-Spine (GLS)	128	64	4	2048
Facebook old and new (FB1 and FB2)	128	64	8	256
Google (GO)	160 (128+32)	64	4	128
Modified Clos (MC)	160 (128+32)	64	4	64

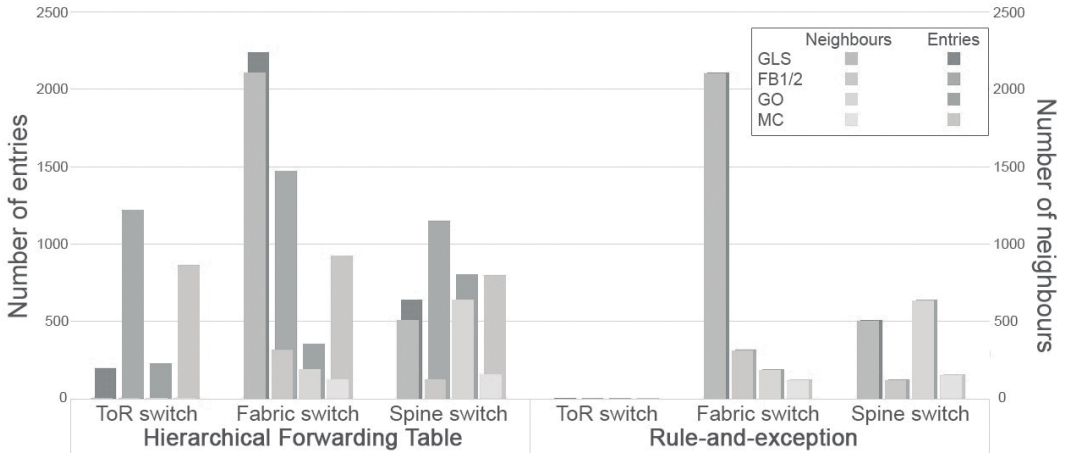

Figure 13: Number of neighbours vs. required entries in the different DCN topologies without failures.

Figure 13 shows a comparison of the number of entries required in the non-failure scenarios for the different node locations and forwarding policy in each of DCN topology. We considered 2 forwarding functions:

- Hierarchical forwarding table: Stores one entry per aggregated set of addresses (i.e., showing superior scalability than a flat addressing scheme with an entry per address in the network).
- Rule-and-exception: The proposed policy, which stores one entry per neighbour and group.

In each case, we considered the entries required for neighbour nodes and those used for forwarding packets to remote (not neighbour) nodes or groups of nodes (e.g., pods). As seen, almost no information has to be stored at ToR switches when using our R&E policy, compared to the required entries with hierarchical forwarding tables. This is really interesting, as they are the forwarding hardware most used in DCNs. For fabric and spine switches, reductions in the number of stored entries are not so remarkable, due to the large number of neighbours that these nodes have in some of the considered DCNs (e.g., in the GLS topology, fabric switches have more than 2000 neighbours each). Even so, all nodes still experience reductions from hundreds to thousands of stored entries.

As shown in Figure 13, the proposed R&E forwarding policy lowers the number of entries to be stored at DCN devices significantly. However, this evaluation does not show their real memory requirements. To illustrate this, Figure 14 compares the number of ports stored among all entries required in the non-failure DCN scenarios, independently

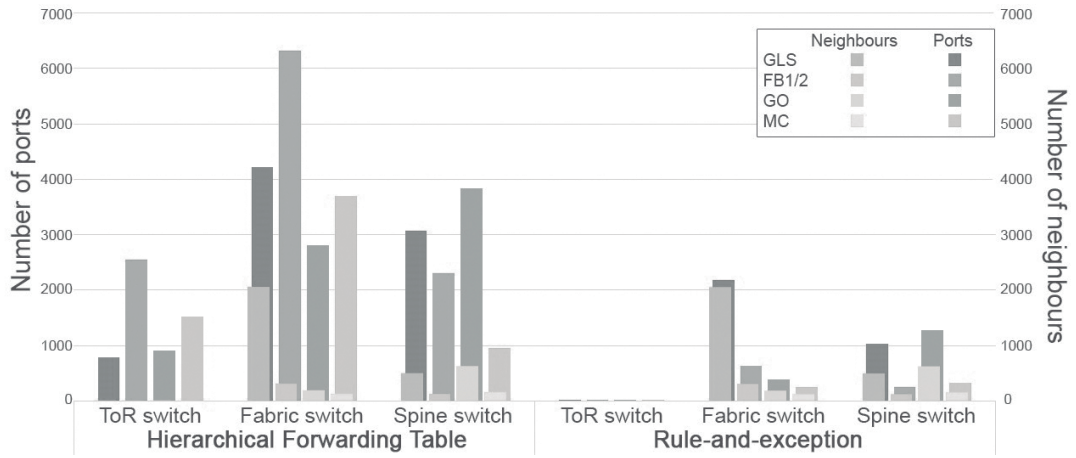

Figure 14: Number of neighbours vs. required stored ports in entries in the different DCN topologies without failures.

of the entry encoding used. As can be seen, while it is true that the number of stored entries with hierarchical forwarding tables and the proposed policy is more or less similar, memory requirements differ substantially. The R&E forwarding policy requires storing only neighbouring node information (stored at most once per defined group). In contrast, with forwarding tables, apart from neighbour information, we store not only one port per destination, but also some of the neighbours that can be used to reach it. As the number of ports eligible to reach a destination may be huge (e.g., any spine switch can be used to reach other pods from a fabric switch), we have imposed a limit of 16 ports per entry, limiting the size of forwarding entries. Even limiting at most 16 ports per entry, which has a negative impact on load balancing, the difference between forwarding tables and R&E becomes substantial..

When considering failure scenarios, our R&E forwarding policy uses either the modification of groups or exceptions to avoid the existing failures. In Figure 15, we depict the relative number of entries, with respect to the number of pods, for the MC-based DCN presented in Table 5 in different failure scenarios experiencing from 0 to 10 failures each. While the specific distribution of these failures across the DCN (node or link failure, location, etc.) could have a great effect on the resulting routes, we realized that it has little to no effect to our policy. With that in mind we decided to uniformly distribute the failures between link failure and node failure (all links attached go down). The results show, as expected, that at most one exception per failure is required by the R&E policy in all cases, resulting in a negligible stored forwarding information increment. This is an interesting property provided by the topological characteristics of the considered DCNs, which remains even in the worst case scenarios. In contrast, with hierarchical forwarding tables, certain failures can prevent the aggregation of some groups of addresses, requiring now a larger number of entries. As a result, we can identify some noticeable hierarchical forwarding tables size increments with even few failures in the DCN.

In addition to the number of entries and table size, we are also interested in comparing the scalability of the proposed R&E forwarding policy as the DCN grows up. Figure 16 shows the average number of entries and stored

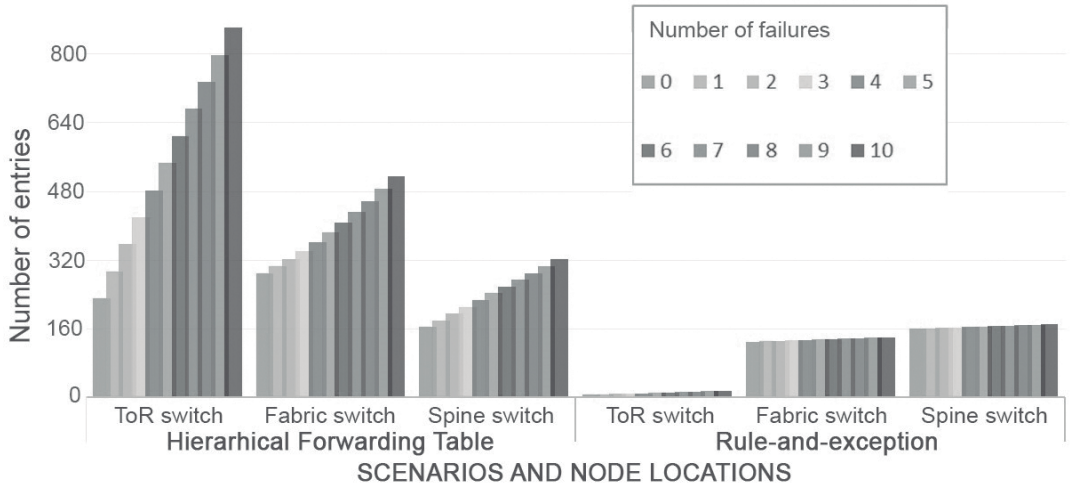

Figure 15: Comparison between the average number of entries per forwarding node in scenario MC with 1 to 10 concurrent failures.

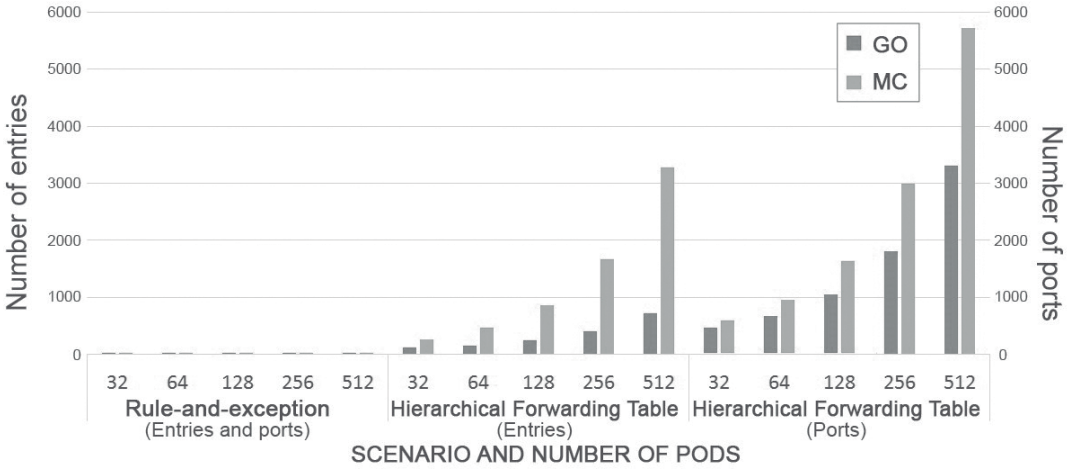

Figure 16: Average number of entries and stored ports for different number of pods in GO and MC-based DCN.

ports in the GO and MC DCN scenarios (similar results are also obtained with the other leaf-spine and Clos variants). In these scenarios, we considered the same parametrization as before, but varying the number of pods, each composed of 4 fabric switches and 64 nodes between ToR switches and edge routers. We considered a non-failure scenario and, for the scenarios using hierarchical forwarding tables, the same limit of at most 16 stored ports per forwarding entry, typical of ECMP. In the figure, we can see how, while the aggregation of addresses implies a notable improvement with respect a flat solution, the number of forwarding entries and their size grows steadily with the number of pods. In contrast, as the R&E policy only requires the storage of adjacent neighbour's information, it remains almost constant as the size of the DCN grows up.

The R&E forwarding policy clearly shows some advantages with respect to forwarding tables using ECMP. However, both of them select the next hop uniformly between the possible candidates. Other solutions like WMP [27] use extra information on the valid neighbours to improve load balancing decisions. Even so, they encounter a trade-off between

information and scalability that cannot be avoided. In these cases, the same approach as for ECMP is done, and the number of possible choices is limited. In large-scale leaf-spine and Clos DCNs, a high number of parallel paths exist between most pairs of destinations. Given that, the difference between the possible choices would not vary largely even with multiple concurrent failures in the network. Hence, we find that the solution proposed by R&E adapts better to the scenario, solving the trade-off towards scalability and enabling all possible options, instead of trying to search for optimal solutions within limited options.

The benefits of our proposal against hierarchical forwarding tables (with or without aggregation of addresses) are good enough to justify topology dependent policies. However, the computational and communication costs of searching exceptions, given the specific number of failures, should be also considered. Regarding the latter, we have a clear improvement in the sense that, as nodes know the DCN topological characteristics, they can take some knowledge as granted. With this knowledge, we can avoid most of the initial propagation of routing information flooding that any common link-state or distance-vector routing protocol needs for populating routing and forwarding tables. In addition to this, TCP/IP solutions tend to require some type of refresh of routing information to ensure that the knowledge is updated. In this regard, RINA DIFs can itself provide reliable communication between nodes and our policies work based on link status (with has to be synchronized between both extremes). This make routing refreshes unnecessary, both in distributed and centralized approaches. Given that, only upon failure and recovery events routing updates should be propagated. In this regard, the distributed approach shares a similar cost as any other link-state protocol, having to propagate the new update to the entire network. Even so, the number of messages exchanged can be halved, since the same update is propagated from both failed link endpoints, instead of the link state at both endpoints. In the centralized cases, it all depends of the approach used when locating manager servers. A quick approximation would require a status update per manager plus an exception update for any node requiring a re-population of exceptions. Anyway, the size of routing messages, both in the distributed and centralized approaches, should not exceed one packet in most if not all cases.

Finally, in terms of computational cost, in order to validate our proposed approach to compute forwarding exceptions, it is important not to exceed the cost of traditional solutions based on the Dijkstra routing algorithm. We take as an example the pseudo-code proposed for computing exceptions at ToR switch and edge routers in the MC topology in the distributed approach (Algorithm 2) and the one for computing exceptions in the centralized case of the same scenario (Algorithm 3). To simplify the results, we consider the same parametrization described in Table 5, scaling the DCN based on the number of pods (p). We also consider the number of failures (r) as a parameter to describe the complexity of each approach.

With Dijkstra-based approaches, the complexity of computing exceptions would grow linearly with p, as the number of nodes does the same. For the proposed approaches, centred on the failures in the network, there are two main

scenarios possible: processing a failure in the same pod or in another one. If the failure is in the same pod (probability $1/p$) we need to compute exceptions for all failures in the DCN, with a linear cost in the number of failures. If the failure affects a different pod (probability $1 - 1/p$), then we need to compute exceptions only to that specific pod, being that a near constant function. Since the number of concurrent failures in these types of networks is small by design, we found that those bounds carry a great improvement. Moreover, we should consider that the probability of having to take the most complex route upon a failure when computing new exceptions reduces as the DCN grows up. In addition, since these networks tend to operate in the non-failure scenario most of the time, this represents a big performance improvement as we only require the constant cost of checking that there is no failure in the network.

8 Conclusions

In this paper, we have extended the rule-and-exception based topological forwarding and routing policies for RINA-enabled large-scale DCNs proposed in [9]. These policies take the knowledge of the DCN topologies in order to provide a superior efficiency and scalability, achieving fast and successful forwarding decisions in non-failure scenarios, merely requiring information about neighbouring nodes. Upon DCN link or node failures, forwarding exceptions are computed and stored at forwarding nodes to override decisions of primary rules that have become temporally invalid. To also minimize the size of stored exceptions, a varying encoding is also proposed, allowing to store only a list of unreachable neighbours for forwarding, instead of the full list of valid ones. This yields a significant improvement given the large number of redundant paths across such large-scale DCNs. In addition, this improves the range of options for load balancing decisions, as we are not limited to storing only an arbitrary number of valid destinations as in common ECMP implementations.

The proposed forwarding policy is fully dependent on the network topology in order to take its rule-based decisions. Even so, the policy is generic enough with programmable rules to be used in any kind of topology where most routes could be easily computed from addresses. Moreover, even if the network does not follow any topology that could take complete advantage from forwarding rules, forwarding devices can still employ exceptions as traditional forwarding entries. In that case, while benefits would be reduced, node grouping and exception encoding could be still used to provide scalability improvements.

We also proposed complementary routing policies taking advantage from the known topologies, without having to compute the full forwarding table in order to largely reduce the routing communication and computational cost. These routing policies, instead of sharing all connectivity information, they disseminate only failed link information, largely reducing the communication cost. The obtained results illustrate the scalability of our topological forwarding and routing policies. The interested reader can experiment with the proposed policies in the online tutorial available in [28].

In addition to the considered leaf-spine and Clos DCNs, alternative scenarios could also profit from RINA and the R&E policy. As a future work, we plan to investigate the usage of the proposed R&E forwarding policy in alternative RINA network scenarios (not only intra-data centre but also large-scale network service provider ones), customizing it as needed to achieve maximum scalability benefits. Moreover, we find of particular interest to also prototype and test the proposed R&E policy in real RINA network test-bed scenarios, so as to also experimentally evaluate its performance.

Acknowledgements

This work is partly funded by the European Commission through the FP7 PRISTINE project (FP7-619305). Moreover, it has been partly funded by the Spanish project SUNSET (TEC2014-59583) that receives funding from FEDER.

References

1 Arjun Singh, et al., "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network". In SIGCOMM, London, United Kingdom, August 2015.

2 Alexey Andreyev, "Introducing data center fabric, the next-generation Facebook data center network", available online at: https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.

3 A Whitmore, A. Agarwal and Li Da Xu, "The Internet of ThingsâĂŤA survey of topics and trends". In Information Systems Frontiers, vol. 17, no. 2, pp. 261-274, April 2015.

4 J. G. Andrews, et al., "What Will 5G Be?". In IEEE Journal on Selected Areas in Communications, vol. 32, no. 6, pp. 1065-1082, June 2014.

5 D. Arora, T. Benson, J. Rexford, "ProActive routing in scalable datacentres with PARIS". In DCC 2014, Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing.

6 M. F. Bari, et al., "Data Center Network Virtualization: A survey".

7 J. Day, I. Matta, and K. Mattar. "Networking is IPC: A Guiding Principleto a Better Internet". In CoNEXT'08: Proceedings of the 2008 ACMCoNEXT Conference, pages 1 6, New York, NY, USA, 2008. ACM.

8 V. Maffione, et al. "A Software Development Kit to exploit RINA programmability". In IEEE ICC 2016, Kuala Lumpur, May 201.

9 S. Leon Gaixas, et al., "Benefits of Programmable Topological Routing Policies in RINA-enabled Large-scale Datacenters". In Globecom 2016, Washington DC, United States of America, December 2016.

10 M.E. Gómez, P. López, J. Duato. "A Memory-Effective Routing Strategy for Regular Interconnection Networks". In IPDPS'05 conference, 2005.

11 S. Habib, F. S. Bokhari, and S. U. Khan, "Routing Techniques in Data Center Networks," in Handbook on Data Centers, S. U. Khan and A. Y. Zomaya, Eds., Springer-Verlag, New York, USA, 2015, ISBN 978-1-4939-2091-4, Chapter 16.

12 K. Chen, et al. "Survey on routing in data centers: insights and future directions", IEEE Network, vol. 25, no. 4, pp. 6-10, July 2011.

13 Jokela, Petri, et al. "LIPSIN: line speed publish/subscribe inter-networking", In ACM SIGCOMM Computer Communication Review, Vol. 39. no 4 pp 195-206, 2009.

14 Burton H. Bloom, "Space/time trade-offs in hash coding with allowable errors", Communications of the ACM, Vol. 13. no 7, pp422-426, July 1970.

15 Y. Rekhter, T. Li, S. Hares. "A Border Gateway Protocol 4 (BGP-4)". RFC 4271, January 2006.

16 P. Lapukhov, A. Premji, J. Mitchell. "Use of BGP for routing in large-scale data centers". IETF Network Working Group, draft-lapukhov-bgp-routing-large-dc-07, February 2014.

17 B.A.A. Nunes et al., "A survey of Software-Defined Networking: past, present, and future of programmable networks". In IEEE Communications Surveys & Tutorials, vol 16, no 3, July 2014.

18 J. Day, "Patterns in network architecture: a return to fundamentals", Prentice Hall, January 2008.

19 Vincenzo Maffione, et al., "A Software Development Kit to exploit RINA programmability". IEEE ICC 2016, Next Generation Networking and Internet Symposium

20 J. Saltzer, "On the Naming and Binding of Network Destinations", IETF RFC 1498, August 1982.

21 Vatche Ishakian, et al., "On supporting mobility and multihoming in recursive internet architectures". Comput. Commun. 35, 13 (July 2012), 1561-1573

22 Eduard Grasa, et al., "From protecting protocols to protecting layers: designing, implementing and experimenting with security policies in RINA". IEEE ICC 2016, Communications and Informations Systems Security Symposium.

23 Eduard Grasa, et al., "Simplifying multi-layer network management with RINA". TNC 2016.

24 S. Vrijders, et al. "Unreliable IPC in Ethernet: migrating to RINA with the shim DIF". ICUMT 2013, Almaty.

25 S. Vrijders, et al., "Reducing the complexity of Virtual Machine Networking". IEEE Communications Magazine, Vol. 54. no 4, pp 152-158, April 2016

26 A. Roy, et al., "Inside the Social Network's (Datacenter) Network", In SIGCOMM, London, United Kingdom, August 2015.

27 Junjie Zhang et al. "Optimizing Network Performance Using Weighted Multipath Routing", In Computer Communications and Networks (ICCCN), August 2012.

28 Sergio Leon, "RINA/Sim wiki - Tutorial: Topological Routing in DC", available at: http://github.com/kvetak/RINA/wiki/Topological-Routing-in-DC.

Author Biography

Sergio Leon. Obtained the bachelor degree in computer science from the Universitat Pompeu Fabra (UPF, July 2011), and the M.Sc. degree in computing from the Universitat Politècnica de Catalunya (UPC, September 2013). Currently, he is working toward his Ph.D. degree at the Universitat Politècnica de Catalunya. His main research topics are in the field of next-generation network architectures for the future Internet, with particular focus on quality of service and routing.