Decentralized SDN Control Plane for a Distributed Cloud-Edge Infrastructure: A Survey

David Espinel Sarmiento, Adrien Lebre, Lucas Nussbaum, Abdelhadi Chari

Decentralized SDN Control Plane for a Distributed Cloud-Edge Infrastructure: A Survey

David Espinel Sarmiento*, Adrien Lebre†, Lucas Nussbaum‡, Abdelhadi Chari§
* *Orange* Lannion, France davidfernando.espinelsarmiento@orange.com
† *IMT-Atlantique - Inria - LS2N* Nantes, France adrien.lebre@inria.fr
‡ *Université de Lorraine - Inria - LORIA* Nancy, France lucas.nussbaum@loria.fr
§ *Orange* Lannion, France abdelhadi.chari@orange.com

***Abstract*—Today's emerging needs (Internet of Things applications, Network Function Virtualization services, Mobile Edge computing, etc.) are challenging the classic approach of deploying a few large data centers to provide cloud services. A massively distributed Cloud-Edge architecture could better fit these new trends' requirements and constraints by deploying on-demand infrastructure services in Point-of-Presences within backbone networks. In this context, a key feature is establishing connectivity among several resource managers in charge of operating, each one a subset of the infrastructure. After explaining the networking management challenges related to distributed Cloud-Edge infrastructures, this article surveys and analyzes the characteristics and limitations of existing technologies in the Software Defined Network field that could be used to provide the inter-site connectivity feature. We also introduce Kubernetes, the new de facto container orchestrator platform, and analyze its use in the proposed context. This survey is concluded by providing a discussion about some research directions in the field of SDN applied to distributed Cloud-Edge infrastructures' management.**

***Index Terms*—IaaS, SDN, virtualization, networking, automation**

I. Introduction

Internet of Things (IoT) applications, Network Function Virtualization (NFV) services, and Mobile Edge Computing (MEC) [1] require to deploy IaaS services closer to the end-users in order to respect operational requirements. One way to deploy such a distributed cloud infrastructure (DCI) is to extend network points of presence (PoPs) with dedicated servers and to operate them through a cloud-like resource management system [2].

Since building such a DCI resource management system from scratch would be too expensive technically speaking, a few initiatives proposed to build solutions on top of OpenStack (RedHat DCN [3] or StarlingX [4] to name a few). These proposals are based either on a centralized approach or a federation of independent Virtual Infrastructure Managers (VIMs)[1]. The former lies in operating a DCI as a traditional single data center environment, the key difference being the wide-area network (WAN) found between the control and compute nodes. The latter consists in deploying one VIM on each DCI site and federate them through a brokering approach to give the illusion of a single coherent system as promoted by ETSI NFV Management and Orchestration (MANO) framework [5].

Due to frequent isolation risks of one site from the rest of the infrastructure [6], the federated approach presents a significant advantage (each site can continue to operate locally). However, the downside relates to the fact that resource management systems code does not provide any mechanism to deliver inter-site services. In other words, VIMs have not been designed to peer with other instances to establish inter-site services but rather in a pretty stand-alone way in order to manage a single deployment.

While major public cloud actors such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure already provide a globally deployed infrastructure to provide cloud services (i.e., In 2020, more than 170 global PoPs for Azure [7], more than 90 global PoPs for Google [8], more than 210 global PoPs for AWS [9]), they are limited in terms of the actions users can do concerning the management of networking resources. For instance, in the AWS environment, a virtual private cloud (VPC) can only exist at a region scope. Although a VPC spans all the availability zones in that region, subnetworks belonging to that VPC must reside entirely within a single Availability Zone.

Several academic studies investigated how this global vision can be delivered either through a bottom-up or top-down approaches. A bottom-up collaboration aims at revising low-level VIM mechanisms to make them collaborative, using, for instance, a shared database between all VIM instances [1], [2], [10]. A top-down design implements the collaboration by interacting only with the VIMs' API leveraging, for instance, a P2P broker [11].

In addition to underlining the inter-site service challenges, these studies enabled us to identify key elements a resource management for DCIs should take into account:

- Scalability: A DCI should not be restricted by design to a certain amount of VIMs.
- Resiliency: All parts of a DCI should be able to survive network partitioning issues. In other words, cloud service capabilities should be operational locally when a site is isolated from the rest of the infrastructure.
- Locality awareness: VIMs should have autonomy for local domain management. It implies that locally created data should remain local as much as possible and only

[1]Unless specified, we used the term VIM in the rest of the article for infrastructure management systems of any kind such as OpenStack.

shared with other instances if needed, thus avoiding global knowledge.
- Abstraction and automation: Configuration and instantiation of inter-site services should be kept as simple as possible to allow the deployment and operation of complex scenarios. The management of the involved implementations must be fully automatic and transparent for the users.

In this article, we propose to extend these studies by focusing on the inter-site networking service, i.e., the capacity to interconnect virtual networking constructions belonging to several independent VIMs. For instance, in an OpenStack-based DCI (i.e., one OpenStack instance by POP), the networking module, Neutron [12], should be extended to enable the control of both intra-PoP and inter-PoP connectivity, taking into account the aforementioned key elements.

We consider, in particular, the following inter-site networking services [13]:

- *Layer 2 network extension*: being able to have a Layer 2 Virtual Network (VN) that spans several VIMs. This is the ability to plug into the same VN, virtual machines (VMs) deployed in different VIMs.
- *Routing function*: being able to route traffic between a VN A on VIM 1 and a VN B on VIM 2.
- *Traffic filtering, policy, and QoS*: being able to enforce traffic policies and Quality of Service (QoS) rules for traffic between several VIMs.
- *Service Chaining*: Service Function Chaining (SFC) is the ability to specify a different path for traffic in replacement of the one provided by the shortest path first (SPF) routing decisions. A user needs to be able to deploy a service chaining spanning several VIMs, having the possibility to have parts of the service VMs placed in different VIMs.

Technologies such as Software Defined Networking (SDN), which proposes a decoupling among control and data plane [14], can be leveraged to provide such operations among VIMs [15]. This paradigm has been applied, for instance, to provide centralized control of lower infrastructure layers of WANs (e.g., physical links or fabric routers and switches) in several proposals [16]–[18], including the well-known Google B4 controller [19] and Espresso [20], and commonly referred nowadays as Software Defined WAN [21]. Similarly, VIMs specialized in the management of WANs, usually called WAN infrastructure managers (WIMs) [22], have been proposed as SDN stacks capable of controlling all the links and connectivity among multiple PoPs [23]–[25]. Moreover, the literature is rich on the study of SDN aspects such as the controller placement problem (CPP) [26] or the optimization of computational paths [27].

The approach in which this survey is interested is called SDN-based cloud computing [28]. As such, SDN-based cloud computing has been applied to implement specific applications at a cloud-scale, such as load balancing [29], [30] or policy enforcement [31] functionalities. Our objective is to study how IaaS networking services can be delivered in a DCI context leveraging as much as possible existing solutions. To the best of our knowledge, this is the first study that addresses this question. Concretely, our contributions are (i) an analysis of the requirements and challenges raised by the connectivity management in a DCI operated by several VIMs taking OpenStack as IaaS management example, (ii) a survey of decentralized SDN solutions (analyzing their design principles and limitations in the context of DCIs), and (iii) a discussion of recent activities around the Kubernetes [32] ecosystem, in particular three projects that target the management of multiple sites. Finally, we conclude this survey by (iv) discussing the research directions in this field.

The rest of this paper is organized as follows. Section II defines SDN technologies in general and introduces the OpenStack SDN-based cloud computing solution. A review of previous SDN surveys is provided in Section III. Challenges related to DCI are presented and discussed in Section IV. Definitions of selected design principles used for the survey are explained in Section V. Section VI presents properties of the studied SDN solutions. Section VII introduces Kubernetes and its ecosystem. The discussion and future work related to the distribution of the SDN control plane for DCIs are presented in Section VIII. Finally, Section IX concludes and discusses future works.

II. SDN and VIMs networking: Background

In this section, we present background elements for readers who are not familiar with the context. First, we remind the main concepts around Software-Defined-Network as they are a strong basis for most solutions studied in this survey. Second, we give an overview of how the network is virtualized in OpenStack, the de facto open-source solution to use cloud computing infrastructures.

A. Software-Defined-Network

The Software-Defined-Network paradigm offers the opportunity to program the control of the network and abstract the underlying infrastructure for applications and network services [14]. It relies on the control and the data plane abstractions. The former corresponds to the programming and managing of the network (i.e., it controls how the routing logic should work) The latter corresponds to the virtual or physical network infrastructure composed of switches, routers, and other network equipment that are interconnected. These equipment use the rules that have been defined by the control plane to determine how a packet should be processed once it arrives at the device. While the idea of control and data plane separation is present in IETF ForCES Working Group works [33] and even earlier with the concept of programmable and active networks [34], [35], the work achieved in 2008 around OpenFlow [36] is considered as the first appearance of Software Defined Networks in modern literature [37]. In this initial proposal, the control plane is managed through a centralized software entity called the SDN controller. To communicate with every forwarding device or lower-level components, the controller uses standardized application program interfaces (APIs) called southbound interfaces. In addition to OpenFlow, the most popular southbound APIs are Cisco's OpFlex ones [38].

Controllers also expose a northbound API, allowing communication among the controller and the higher-level components like management solutions for automation and orchestration. A generic SDN architecture with the aforementioned elements is presented in Figure 1. Overall, an SDN controller abstracts the low-level operations for controlling the hardware, allowing easy interaction with the control plane, as developers and users can control and monitor the underlying network infrastructure [39]. The authors refer to [40] for fundamentals in a more detailed SDN history. Moreover, SDN principles have been successfully applied at the industry level with controllers such as VMware NSX [41], Juniper Contrail [42], or Nuage [43].

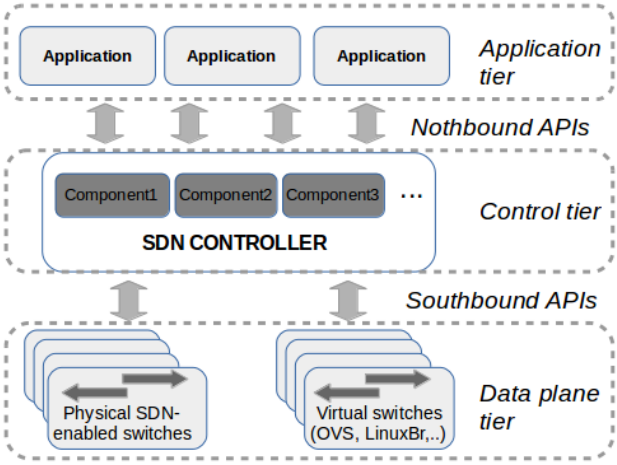

Fig. 1: SDN general architecture

One of the possible application domains of the SDN technology lies in the cloud computing paradigm, which exposes to end-users software, applications, or virtualized infrastructures in a simple and easy-to-use way [44].

By delivering network programmability, SDN abstractions provide the control and management necessary for cloud computing solutions to expose network resources to end-users. Referred to as SDN-based cloud networking (SDN-clouds or SDN-enabled clouds), this approach enables the configuration and provisioning of virtualized network entities using programmable interfaces in cloud computing infrastructures. It is used, for instance, to assure the multi-tenancy needs and the sharing of network resources among end-users [16], [28], [45], [46]. The Neutron OpenStack service described in the next section is a concrete example of such an SDN-based cloud networking service.

B. SDN-based Cloud networking: The Neutron example

To illustrate how the network resources are managed in a cloud computing infrastructure using SDN technologies, we discuss in this section Neutron, the "Network connectivity as a Service" of OpenStack, the de facto open-source solution to manage and expose Infrastructure-as-a-Service platforms [47]. Neutron provides on-demand, scalable, and technology-agnostic network services. Neutron is generally used with Nova [48], the OpenStack project to manage virtualized computational resources, in order to provide VMs with networking capabilities.

1) Neutron architecture:

Neutron is a modular and pluggable platform. The reference Neutron architecture, as shown in Figure 2 is composed of the following elements:

- *Representational state transfer (REST) API*: Neutron's REST API service exposes the OpenStack Networking API to create and manage network objects and passes tenant's requests to a suite of plug-ins for additional processing. It requires indirect access to a persistent database where related information to network objects is stored.
- *Network plug-ins*: While Neutron API exposes the virtual network service interface to users and other services, the actual implementation of these network services resides in plug-ins. The Neutron pluggable architecture comprises Core plug-ins (which implements the core API) and Service plug-ins (to implement the API extensions). Core plug-ins primarily deal with L2 connectivity and IP address management, while Service plug-ins support services such as routing. The Modular Layer 2 (ML2) is the main OpenStack Core plug-in. It supports type drivers to maintain type-specific network state (i.e., VLAN, VXLAN, GRE, etc.) and mechanism drivers for applying configuration to specific networking mechanisms (i.e., OpenvSwitch [49], LinuxBridges [50], etc.). Since 2020, Neutron's main ML2 mechanism driver changed from OVS to Open Virtual Network (OVN) [51]. OVN is a sub-project in OpenvSwitch that provides native in-kernel support for virtual network abstraction and a better control plane separation than an OpenvSwitch.
- *Network Agents*: Agents implement the actual networking functionality closely associated with specific technologies and the corresponding plug-ins. Agents receive messages and instructions from the Neutron server on the message bus. The most common Agents are L3 (Routing functions), L2 (Layer 2 functions), and Dynamic Host Configuration Protocol (DHCP).
- *Messaging queue*: Used by most Neutron installations to route information between the Neutron-server and various Agents.

2) Neutron networking:

Neutron divides the networking constructions that it can manage as Core and extension resources. *Port*, *Network*, and *Subnetwork* are the basic Core object abstractions offered by Neutron.

Each abstraction has the same functionality as its physical counterpart: *Network* is an isolated L2 segment, *Subnetwork* is a block of IPv4 or IPv6 addresses contained inside a Network, *Port* is a virtual switch connection point used to attach elements like VMs to a virtual network. More extension network objects can be defined and exposed, extending the Neutron API. Some examples are *Router*, *floatingIPs*, *L2GWs* [52], *BGP-VPNs* [53] or Virtual Private Network as a Service (VPNaaS) [54]. A more complete list of OpenStack networking objects can be found in [55]. Agents configure

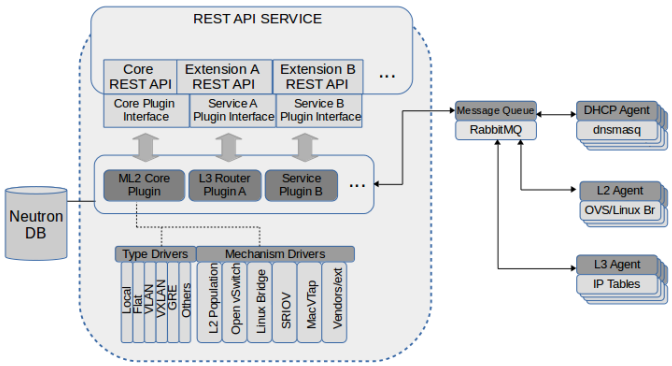

Fig. 2: Neutron deployment components.

the mechanisms allowing these constructions to communicate among them (i.e., a tunnel between two OpenvSwitches to communicate VMs belonging to the same Subnetwork) or with the Internet (i.e., Network Address Translation (NAT) capabilities normally implemented by a Router to route traffic externally).

3) Multi-instance Neutron:

Conceptually speaking, a single instance of the Neutron module could manage all network resources of a DCI. As long as it is possible to communicate at the IP level between all equipment, Neutron can configure the different network devices and deliver the expected network abstractions. Technically speaking, leveraging a single centralized module cannot satisfy DCI properties (i.e., Single Point of Failures (SPOF), network bottleneck, intermittent connectivity between network equipment, etc.) and the deployment of several collaborative instances (in the worst case, one per locations) is necessary. Unfortunately, the software architecture of Neutron does not allow collaborations between multiple instances. For instance, the Neutron database belongs to a single Neutron entity; that Neutron can only manage the resources created at one site. It is not possible for a distant Neutron to have knowledge nor access to the objects present in a database of another instance. Because information of resources is not shared among Neutrons, the notion of a virtual network spanning different VIMs does not exist today in the Neutron database. Further, operations like identifiers assignation, IP, and MAC address generation and allocations, DHCP services, or security group management are also handled internally at each Neutron instance. Such kind of constraints takes out the possibility to manage Neutron resources in a distributed way.

However, some projects propose Neutron's management in a more distributed way, such as the Tricircle project [56]. Tricircle achieves network automation in a multi-instance deployment of OpenStack using a Central Neutron instance managing a series of Local Neutrons with modified Neutron Core plug-ins. To allow communication with the Central Neutron when querying for an element that is not locally present, Tricircle uses the Tricircle Local Core plug-in, a modified version of the ML2 Core plug-in. Once this Local Core plug-in receives a request, it will internally send the same request to the Central Neutron that will answer using standard REST API calls. With the answer, the resource will be locally created using the same information present in the Central Neutron database, and then, the initial request will be treated by the real Core plug-in. For the Central Neutron, which gathers and maintains data consistency across the infrastructure, Tricircle provides a Tricircle Central plug-in that allows the Tricircle to effectively manage and map central created resources and their respective locally created copies, storing this information in the Tricircle Database. Tricircle's major flaw lies in its hierarchical architecture as the Central Neutron becomes a SPOF and a bottleneck. The user is only allowed to use this central API to manage or control networking resources. Since Local Neutrons always need to communicate with Central Neutron in order to answer local requests, network partitioning cases become the principal blocking point as isolated Local Neutrons will remain useless as long as the disconnection remains. Finally, Tricircle does not leverage real collaboration among Local Neutron. Indeed, there is neither communication nor knowledge of each other's presence since all the management is done at the Central Neutron.

4) Summary:

Because VIMs such as OpenStack have not been designed to peer with other instances, several projects have investigated how it might be possible to deliver inter-site services. These projects aim at exposing multiple Openstacks as a single entity. Unfortunately, these solutions have been designed around a centralized architecture and face important limitations (scalability, network partitions, etc.).

Thus, how to decentralize the management of multiple clusters is an important question that our community must deal with. We propose to initiate the debate by focusing on the OpenStack Neutron service in the following. However, we underline that our study is valuable more generally since it provides an abstract enough analysis of DCI management's different challenges.

Before analyzing in detail those challenges, we discuss previous works and surveys on SDN technologies. In addition to underlining major studies that might be relevant to better understand SDN concepts, it enables us to position our current work with respect to previous surveys.

III. Reviews on work & surveys on SDN and SDN-based cloud computing

SDN Technology has been a hot topic for the last few years. In that sense, several studies have already been published. In this section, we underline the major ones. First, we discuss papers that discuss SDN technologies in general. Second, we review SDN-based cloud activities. By introducing these studies, we aim to underline that the use of multi-instance SDN technologies in a DCI context has not been analyzed in the literature yet.

In [57] the authors presented a general survey on SDN technologies presenting a taxonomy based on two classifications: physical classification and logical classification. For every classifications, multiple subcategories were presented and explained, and the surveyed SDN solutions were placed

according to their architectural analysis. The work finished by presenting a list of open questions such as scalability, reliability, consistency, interoperability, and other challenges such as statistics collection and monitoring.

In [58], and [59], the authors focus on the scalability criteria. More precisely, the work done by Karakus et al. [58] provided an analysis of the scalability issues of the SDN control plane. The paper surveyed and summarized the SDN control plane scalability's characteristics and taxonomy through two different viewpoints: topology-related and mechanisms-related. The topology-related analysis presents the relation between the topology of architectures and some scalability issues related to them. The mechanism-related viewpoint describes the relation between different mechanisms (e.g., parallelization optimization) and scalability issues. This work's limitation is that the analysis is done by only considering the throughput measured in established flows per second and the flow setup latency.

In [59], Yang et al. provided a scalability comparison among several different types of SDN control plane architectures by doing simulations. To assign a degree of scalability, the authors proposed to measure the flow setup capabilities and the statistics collection. Although comparisons among controller architectures are made in these two articles, there is no analysis nor mention of the DCI context and related challenges.

Among other available studies in traditional SDN technologies, the work presented in [60], and [61] are probably the most interesting ones concerning DCI objectives. In their article [60], Blial et al. give an overview of SDN architectures composed of multiple controllers. The study focuses on the distribution methods and the communication systems used by several solutions to design and implement SDN solutions able to manage traditional networks. Similarly, the survey [61] discusses some design choices of distributed SDN control planes. It delivers an interesting analysis of the fundamental issues found when trying to decentralize an SDN control plane. These cornerstone problems are scalability, failure, consistency, and privacy. The paper analyses pros and cons of several design choices based on the aforementioned issues. While these two studies provide important information for our analysis, they do not address the cloud computing viewpoint as well as the DCI challenges.

In the field of SDN applied specifically to cloud computing, the works of Azodolmolky et al. [28], [45] provide information about the benefits, and potential contributions of SDN technologies applied for the management of cloud computing networking. While these works represent an interesting entry point to analyze SDN-based cloud networking evolution, they mostly analyzed the networking protocols and implementations (e.g., VLAN, VXLAN, etc) that may be used in order to provide networking federation among a few data centers. More recently, Son et al. [46] presented a taxonomy of SDN-enabled cloud computing works as well as a classification based on their objective (e.g., energy efficiency, performance, virtualization, and security), the method scope (e.g., network-only, and joint network and host), the targeted architecture (e.g., Intra-datacenter network (DCN), and Inter-DCN), the application model (e.g., web application, map-reduce, and batch processing), the resource configuration (e.g., homogeneous, and heterogeneous), and the evaluation method (e.g., simulation, and empirical). Additional metrics, such as data center power optimization, traffic engineering, network virtualization, and security, are also used to distinguish the studied solutions. Finally, the paper provides a gap analysis of several aspects of SDN technologies in cloud computing that have not been investigated yet. Among them, we can cite the question related to the extension of cloud computing concepts to the edge of the network (i.e the DCI we envisioned).

To summarize, prior surveys and works neither analyze the challenges of decentralized virtualized networking management in the context of DCIs nor the characteristics (pros/cons) of SDN solutions that could be used to execute this management between multiple instances of the same VIM. The present survey aims to deliver such a contribution.

IV. Distributed network control management Challenges

As discussed in Section II, the control of the network elements of a DCI infrastructure should be performed in a distributed fashion (i.e., with multiple VIM networking services that collaborate together to deliver the same network capabilities across multiple sites). Obviously, decentralizing a controller such as Neutron brings forth new challenges and questions. We choose to divide them into two categories: the ones related to the organization of network information and the ones related to the implementation of the inter-site networking services. These challenges are summarized in Table I. The key words column is used to introduce the name of the challenges that will be used in the rest of the document. Finally, the term VIM refers to the VIM network service in the following.

A. Network information's challenges

Giving the illusion that multiple VIMs behave like a global SDN-based Cloud networking service requires information exchange. However, mitigating as much as possible data communications while being as robust as possible (w.r.t network disconnection or partitioning issues) requires to consider several dimensions as discussed in the following.

*1) Identifying how information should be shared (**information granularity**):* The first dimension to consider is the organization of the information related to the network elements. As an example, the provisioning of an IP network between two VIMs will require to share information related to the IPs that have been allocated on each VIM. A first approach may consist of sharing the information between the two VIMs each time an IP is allocated to one resource. This way will prevent possible conflict, but with an overhead in terms of communications (the global knowledge base is updated each time there is a modification). A second approach would be to split the range of IP addresses with the same Classless Inter-Domain Routing (CIDR or network prefix) between the two VIMs at the creation of the network (i.e., each VIM has a subset of the IPs and can allocate them without communicating with other controllers). This way prevents facing IP conflicts even in the case of network partitioning without exchanging

Challenge	Key words	Summary
Network information's challenges		
Identifying how information should be shared	**information granularity**	Propose good information sharding strategies
Sharing networking information on-demand and in an efficient manner	**information scope**	Avoid heavy synchronization by contacting only the relevant sites
Facing network disconnections	**information availability**	Continue to operate in cases of network partitioning and be able to recover
Technological challenges		
Standard automatized and distributed interface	**automatized interfaces**	Well-defined and bridged vertical and horizontal interfaces
Support and adaptation of networking technologies	**networking technologies**	Capacity to configure different networking technologies

TABLE I: DCI Challenges summary

information each time a new IP is allocated to a particular resource.

Understanding the different structures that are manipulated will enable the definition of different information sharding strategies between multiple VIMs and identify the pros and cons of each of them.

Additionally, other elements related to local domain networking management that may be attached to a virtual network as local router gateways, external gateways, DHCP ports, DNS servers, fixed host routes, or floating IPs may not be likely to be shared with remote sites. In consequence, depending on the inter-site service, the granularity of the shared objects' information needs to be well specified to avoid conflicts among the networking management entities. If, in any case, the joint management of a networking construction is strictly required, the management entities should have the necessary mechanisms to do management coordination in order to provide some kind of data consistency.

*2) Sharing networking information on-demand and in an efficient manner (**information scope**):* The second dimension to consider is related to the scope of a request. Networking information should stay as local as possible. For instance, network information, like MAC/IP addresses of ports and identifiers of a network related to one VIM, does not need to be shared with the other VIMs that composed the DCI. Similarly, information related to a Layer 2 network shared between two VIMs as depicted in Figure 3 does not need to be shared with the 3rd VIM. The extension of this Layer 2 network could be done later. That is, only when it will be relevant to extend this network to *VIM 3*.

Taking into account the scope for each request is critical since sharing information across all VIMs should be avoided due to the heavy synchronization and communication needs. In other words, contacting only the relevant sites for a request will mitigate the network communication overhead and the limitations regarding scalability as well as network disconnections.

Obviously, the information-sharing protocol needs to be fast and reliable to avoid performance penalties that could affect the deployment of the network service.

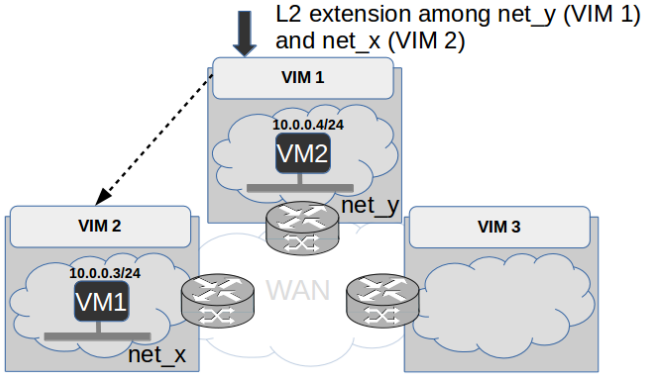

Fig. 3: Layer 2 extension Request

*3) Facing network disconnections (**information availability**):* Each VIM should be able to deliver network services even in case of network partitioning issues. Two situations must be considered: (i) the inter-site network resource (for instance, a Layer 2 network) has been deployed before the network disconnection and (ii) the provisioning of a new inter-site network resource. In the first case, the isolation of a VIM (for instance *VIM 2* in Figure 4) should not impact the inter-site network elements: *VIM 2* should still be able to assign IPs to VMs using the "local" part of the inter-site Layer 2 network. Meanwhile, *VIM 1* and *VIM 3* should continue to manage inter-site traffic from/to the VMs deployed on this same shared Layer 2 network.

In the second case, because the VIM cannot reach other VIMs due to the network partitioning issue, it is impossible to get information that is mandatory to finalize the provisioning process. The first way to address such an issue is to simply revoke such a request. In this case, the *information availability* challenge is only partially addressed. The second approach is to provide appropriate mechanisms in charge of finalizing the provisioning request only locally (e.g., creating temporary resources). However, such an approach implies integrating

mechanisms to recover from a network disconnection.

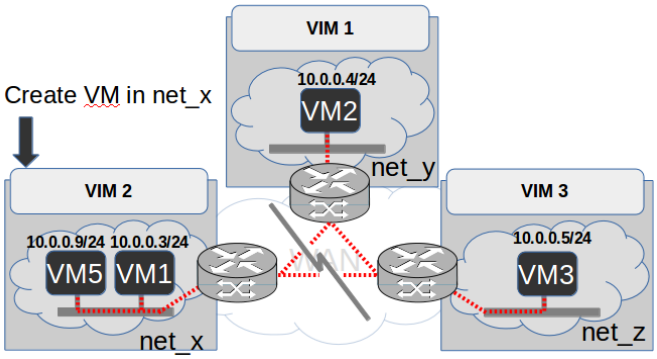

Fig. 4: Operate in a local any mode

Depending on the way the resource has been created during the partitioning, the complexity of the re-synchronization procedure may vary.

In the aforementioned scenario, the VIM may do the provisioning of a new VM in *VIM 2* using an IP address already granted to a VM in *VIM 1* or that belongs to another CIDR. Once the network failure is restored, *VIM 1* will face issues to forward traffic to *VIM 2* either because of the overlapping addresses or because there are two different CIDRs.

To satisfy the availability property, the inter-site connectivity management should be able to address such corner cases.

B. Technological challenges regarding inter-site networking services

Technological challenges are related to the technical issues that could be presented when trying to implement DCI networking services.

*1) Standard automatized and distributed interfaces (**automatized interfaces**):* A first challenge is related to the definition of the vertical and horizontal APIs to allow the provisioning of inter-site services from the end-users viewpoint but also to make the communication/collaboration between the different VIMs possible. This means that the interface which faces the user (user-side or north-side as traffic flows in a vertical way) and the interface which faces other networking services (VIMs-side or east-west-side as traffic flows in a horizontal way) have to be smoothly bridged among them. This integration needs to be done in order to provide the necessary user abstraction and the automation of the VIMs communication process. Consequently, this necessitates the specification and development of well-defined north- and east-west-bound interfaces presenting to the user and to remote instances an abstract enough API with the available networking services and constructions. Thus, how to design efficient APIs for both northbound and east-west-bound communication is another problem to address in the case of inter-site connectivity management tools.

Within the framework of OpenStack, Neutron (see Section II-B) only presents a user-oriented interface to provide local services due to its centralized nature.

The Tricircle project [56] partially address this interface automation leveraging a hierarchical architecture where an API gateway node is used as an entry point to a geo-distributed set of OpenStack deployments. Neutron deployments are not aware of the existence of other local Neutrons but instead always communicate with the Neutron gateway, which is also the only interface exposed to the user.

*2) Support and adaptation of networking technologies (**networking technologies**):* Along with the initial networking information exchanges among VIMs to provide inter-site connectivity (MAC/IP addresses, network identifiers, etc.), the identification of the mechanism to actually do the implementation will be needed. Although there are many existing networking protocols to rely on to do the implementation (VLANs on an interconnection box, BGP-EVPN/IPVPN, VXLAN ids, GRE tunnels, etc.), they will need adaptation in the DCI case. Since the configuration of the networking mechanisms needs to be known by all the participant VIMs in a requested inter-site service, the exchange of additional implementation information will be required among the sites in an automatized way. This automation is required due to the fact that the user should not be aware of how these networking constructions are configured at the low-level implementation. Since a Cloud-Edge infrastructure could scale up to hundreds of sites, manual networking stitch techniques like [52] [53] will be simply not enough.

Depending on the implementation, the solution needs to be able to do the reconfiguration of networking services at two different levels:

- At the overlay level which implies the ability to configure virtual forwarding elements like GoBGP instances [62], OpenvSwitch switches or Linux bridges.
- At the underlay level, which implies the ability to talk or communicate with some physical equipment like the Edge site gateway. As not all physical equipment is OpenFlow-enabled, the possibility to use other protocols may be an advantage when it is necessary to configure heterogeneous components or when internal routes should be exposed to allow traffic forwarding at the data plane level.

Additionally, in the context of the challenge described in IV-A3, the mechanisms used for the implementation need to be capable of reconfiguring themselves in order to re-establish the inter-site traffic forwarding.

V. Distributed SDN design principles

In addition to highlight the main challenges that should be addressed by a DCI network service management, our study aims at describing the state of the art of major SDN solutions and the analysis on how each controller may answer the different DCI proposed challenges. This section presents an overview of the major differences we identified between the solutions we have studied.

A. Architecture

The first point that distinguishes one solution from another is the way controllers are interconnected with each other [57]–[61]. Figure 5 presents the connection topologies we identified

during our analysis. We discuss in the following the pros and cons of each approach.

Centralized: Architecture presenting a single centralized controller with a global view of the system. It is the simplest and easiest architecture to manage, but at the same time, the less scalable/robust one due to the well-known problems of centralized architectures (SPOF, bottlenecks, network partitioning, etc.).

Hierarchical: Tree-type architecture composed of several layers of controllers. Most solutions present a two-level tree consisting of local controllers and a "root" controller. As the names indicate, local controllers handle local operations such as intra-site routing. On the opposite, the "root" controller deals with all inter-site operations. While local controllers only have their own local view and are not aware of other local controllers' existence, the root controller should maintain a global knowledge of the infrastructure to communicate with local controllers each time it is mandatory. While this approach tackles the scalability challenge w.r.t. the centralized approach, it only increases the robustness partially as the root controller is still a centralized point.

Distributed but logically centralized: Architecture where there is one controller per site, managing both intra and inter-site operations. Each time a controller creates or updates a network resource, it broadcasts all other controllers' modifications. This way enables controllers to maintain an up-to-date copy of the global knowledge, thus acting as a single logical entity. This design stands close to the initial SDN proposition [14] as several controllers share global network information to present themselves as one single controller.

Fully distributed: Architecture similar to the previous one but without communicating all creations/modifications to other controllers. In this approach, locally-created data remains in the instance where it has been created and shared with other instances only when needed. In such a case, explicit communications between controllers are instantiated in order to exchange technical information to establish, for instance, inter-site services. This way of interconnecting controllers increases the robustness w.r.t network disconnections as a network disconnection, or a node failure only impacts a subpart of the infrastructure.

Hybrid: Two-layer architecture mixing the distributed and the hierarchical architectures. The control plane consists of several root controllers at the top layer. Each one of the roots manages multiple local controllers who are in charge of their respective sites. These root controllers are organized in a distributed fashion, gathering global network state information among them.

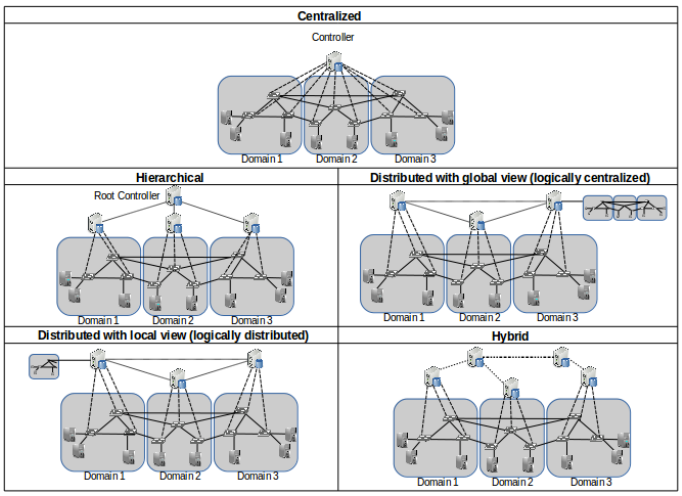

Fig. 5: SDN topologies

B. Leader-based operations

When implementing a DCI network service, it is important to consider two kinds of operations: *leaderless vs. leader-based*. Leaderless operations such as creating an only-local network and its sub-networks are "simple" operations that should not lead to network information inconsistencies [61] and thus do not require leadership mechanisms. On the opposite, leader-based operations, such as the assignment of an IP in an inter-site network, require a dedicated approach to avoid issues such as IP collisions. For those operations, there should be a leader to take consistent decisions among all controllers. Leaderships can be either given in two ways [63]: in a static manner to a controller (i.e., the root node in a hierarchical approach) or by using consensus protocols. Consensus can be divided in two general types: leaderless consensus (i.e., such as EPAXOS [64] or Alvin [65]), and leader-based consensus (i.e., such as PAXOS [66] or RAFT [67]).

Leader-based consensus protocols such as the aforementioned ones are used for several tasks such as leader election, group membership, cluster management, service discovery, resource/access management, consistent replication of the master nodes in services, among others [68]. Consensus typically involves multiple instances agreeing on values. Moreover, consensus can be reached when any majority of the instances is available; for instance, a cluster of 5 instances can continue to operate even if two nodes fail. However, applying consensus protocols to a distributed SDN controller may present some problems. In RAFT, for instance, network failures can seriously impact the performance of the protocol: in the best case, the partitioning may reduce the normal operation time of the protocol; in the worst case, they render RAFT unable to reach consensus by failing to elect a consistent leader [69].

To avoid the limitation imposed by a single leader election, leaderless consensus protocols allow multiple nodes to operate as a leader at-a-time [70]. This is achieved by dividing conflicting and non-conflicting operations. Non-conflicting operations can be executed without synchronization, while for the conflicting ones, the nodes proposing the operation assume the leadership. The per-operation-leader then collects

the dependencies from remote nodes to compute the order among conflicting operations. However, as the system size gets bigger, leaderless protocols may present scalability problems. In EPAXOS, for instance, as the system size increases, more nodes could propose transactions generating more conflicting operations. As a consequence of this possibility, there is a higher probability of different nodes viewing different dependencies, which can fail to deliver fast decisions.

C. Internal communication protocols

Depending on the selected topology, the communication between controllers occurs either vertically (centralized and hierarchical) or horizontally (distributed). Those communications can be handled through different manners like polling information from other controllers periodically, using a publish/subscribe approach to send notifications automatically, or through explicit communication protocols between controllers.

D. Database management system

As largely discussed in Section IV, storing and sharing the state of the DCI network service would be an important challenge. Surveyed SDN solutions rely either on relational (SQL) or NoSQL databases.

1) SQL Databases: SQL databases are based on the relational data model; they are also known as Relational Database Managing System (RDBMS). In most cases, they use Structured Query Language (SQL) for designing and manipulating data and are normally deployed in a centralized node [71]. Relational databases can generally be vertically scalable, which means that their performance could be increased using more CPU or RAM. Some SQL databases such as MySQL Cluster [72] proposes to scale horizontally, generally sharding data over multiple database servers (a.k.a. "shared nothing" architecture).

2) NoSQL Databases: NoSQL database is a general term that gathers several kinds of databases that do not use the relational model. NoSQL databases can be gathered in four main types [73]: document-based (e.g., MongoDB [74]), key-value pairs (e.g., Redis [75]), graph databases (e.g., Neo4j [76]) or wide-column stores (e.g., Apache Cassandra [77]). This kind of databases are by nature horizontal scalable as the unstructured data scheme allows information to be sharded in different sites, thus allowing different entities to access it simultaneously in a geographically distributed way [78], [79].

More generally, the database management system would be a key element of a DCI network service. It could be used as the means to share information between controllers, thus eliminating the need for a dedicated communication protocol as discussed in the previous paragraph.

E. SDN interoperability and maturity

The studied SDN controllers should be capable of achieving a good performance in heterogeneous and dynamic network environments. For this reason, the capacity to configure a different kind of equipment and the maturity of the solution will be explained in this section.

1) Network types targeted: The popularity of virtualization technologies leads to the abstraction of the physical network (a.k.a. the underlay network) into multiple virtual ones (a.k.a. overlay networks).

- *Underlay network*: Is a physical infrastructure that can be deployed in one or several geographical sites. It is composed of a series of active equipment like switches or routers connected among them using Ethernet switching, VLANs, routing functions, among other protocols. Due to the heterogeneity of equipment and protocols, the Underlay network becomes complex and hard to manage, thus affecting the different requirements that must be addressed like scalability, robustness, and high bandwidth.
- *Overlay network*: Virtual network built on top of another network, normally the underlying physical network, and connected by virtual or logical links. Overlay networks help administrators tackle the scalability challenge of the underlay network. For instance, overlay networks leverage the use of encapsulation protocols like VXLAN because of its scalability (e.g., VXLAN provides up to 16 million identifiers while VLAN provides 4096 tags).

Because of these two levels of complexity, SDN controllers could be designed to deal with both levels or just one. Obviously, the richer the operations offered by controllers, the more difficult it would be to distribute the DCI network service.

2) Supported Southbound protocols: The reference SDN architecture exposes two kinds of interfaces: Northbound and Southbound. Northbound interfaces reference the protocol communication between the SDN controller and applications or higher layer control programs that may be automation or orchestration tools. Southbound interfaces are used to allow the SDN controller to communicate with the network's physical/virtual equipment. OpenFlow [80] is an industry-standard considered as the de facto southbound interface protocol. It allows entries to be added and removed to the switches and potentially routers' internal flow-table, so forwarding decisions are based on these flows. In addition to OpenFlow, SDN controllers may use other protocols to configure network components like NETCONF, LISP, XMPP, SNMP, OVSDB, BGP, among others [14]. The Border Gateway Protocol (BGP), for example, allows different Autonomous Systems (ASes) to exchange routing and reachability information between edge routers.

More generally, as not all physical equipment is OpenFlow-enabled, the possibility to use other protocols may be an advantage when it is necessary to configure heterogeneous components or when internal routes should be exposed to allow communication at the data plane level.

3) Readiness Level: The Technological Readiness Level (TRL) scale is an indicator of a particular technology's maturity level. Due to the complexity of the mechanisms we are dealing with in this survey, it is important to consider the TRL of technology in order to mitigate as much as possible development efforts. This measurement provides a common understanding of technology status and allows us to establish the surveyed SDN solutions' status, as not all solutions have the same maturity degree. To this end, the TRL proposed by

the European Commission presented in [81] has been used to classify the different solutions we studied.

4) Additional considerations: OpenStack compatibility: As we also propose to take the example of an OpenStack-based system to explain how SDN solutions could be used in a multi-VIM deployment, the capability to integrate with Neutron is introduced as an illustrative example. Indeed, some SDN controllers may be able to integrate with Neutron to implement networking services or add additional functionalities, consuming the Neutron core API or its extensions. Therefore, having a driver to pass network information to the controller.

c

VI. Multi-controller SDN Solutions

Although it would be valuable, delivering a survey of all SDN controller solutions that have been proposed [82], [83] is beyond the scope of this article. We limited our study to literature major solutions' state of the art and selected the best candidates that may fit the Distributed Cloud Infrastructure we are investigating. For the sake of clarity, we present the solutions we studied into two categories:

- Network-oriented SDN: Solutions designed to provide network programmability to traditional or virtualized network backbones. The controllers gathered in this category have not been designed to provide SDN capabilities for cloud computing networking environments.
- Cloud-oriented SDN: Solutions that proposed an SDN way to manage the networking services of cloud computing infrastructures (as explained in Section II-A). While some of the controllers gathered in this category have been initially designed to manage traditional networks, they propose extensions to provide SDN features within the cloud networking services.

For each selected proposal, we present a qualitative analysis and summarize their characteristics and whether they address the DCI challenges, respectively in Table II and Table III.

A. Network-oriented (controllers for SDN domains)

In this first part, we give an overview of the seven SDN solutions we investigated, namely DISCO [84], D-SDN [85], Elasticon [86], FlowBroker [87], HyperFlow [88], Kandoo [89], and Orion [90].

DISCO

DISCO (Distributed SDN Control Plane) relies on the segregation of the infrastructure into distinct groups of elements where each controller is in charge of one group using OpenFlow as Control plane protocol. Each controller has an *intra-domain* (or intra-group) part that provides local operations like managing virtual switches, and an *inter-domain* (or inter-group) part that manages communication with other DISCO controllers to make reservations, topology state modifications, or monitoring tasks. For the communication between controllers, DISCO relies on an Advanced Message Queuing Protocol (AMQP) message-oriented communication bus where every controller has at the same time an AMQP server and a client. The central component of every controller is the database where all intra- and inter-domain information is stored. We underline that there is no specific information on how the database actually works in the article that presents the DISCO solution.

DISCO can be considered to have a fully distributed design because every local controller stores information of its own SDN domain only and establish inter-domain communication with other controllers to provide end-to-end services only if needed. DISCO controllers do not act as a centralized entity and instead work as independent entities peering among them. It has a leader-less coordination because of its logically distributed condition (each controller is in charge of a subgroup, so there is no possible conflict). DISCO's evaluations have been performed on a proof of concept. For this reason, we assigned a TRL of 3 to DISCO.

Addressing the challenges:

- *Information granularity*: Addressed - due to the segregation of the infrastructure into distinct groups.
- *Information scope*: Addressed - thanks to its per-group segregation. When an inter-domain forwarding path is requested, DISCO controllers use the communication channel to only contact the relevant sites for the request. Thus, avoiding global information sharing.
- *Information availability*: Addressed - in case of network disconnections, each controller would be able to provide intra-domain forwarding. Besides, controllers that can contact each other could continue to deliver inter-site forwarding. Finally, a recovery mode is partially provided, given that disconnected sites only need to contact the remote communication channels to retake the inter-domain forwarding service when the connectivity is reestablished. As aforementioned, we underline that due to its implementation and the information that is manipulated, DISCO is conflict-less. This makes the recovery process rather simple.
- *Automatized interfaces*: Addressed - thanks to the bridge presented among the northbound and east-west interfaces to do inter-controller communication.
- *Networking technologies*: Not addressed since it does not integrate other networking technologies aside from OpenFlow.

D-SDN

D-SDN (Decentralized-SDN) distributes the SDN control into a hierarchy of controllers, i.e., Main Controllers (MCs) and Secondary Controllers (SCs), using OpenFlow as control plane protocol. Similar to DISCO, SDN devices are organized by groups and assigned to one MC. One group is then divided into subgroups managed by one SC (each SC requests one MC to control a subgroup of SDN devices). We underline that the current article does not give sufficient details regarding how states are stored within Main and Secondary Controllers. The paper mainly discusses two protocols. The first one is related to communications between SCs and MCs using *D-SDN's MC-SC* protocol for control delegation. The second one, entitled *D-SDN's SC-SC*, has been developed to deal with fail-over

TABLE II: Classification of SDN solutions.

Proposals	Model					Implementation						Interoperability & maturity				Extra consideration
	Centralized (Single) Controller Designs	Distributed Designs				Coordination Strategy		Internal Communication Protocols			Database management system	Network types targeted		Southbound Protocols	Readiness Level	OpenStack compatibility
		(Flat) Logically centralized	(Flat) Logically distributed	Hierarchical	Hybrid	Leader-based	Leader-less	Among local nodes	With higher layers	Among root nodes		Underlay	Overlay			
Network-oriented solutions																
DISCO			✓				✓	AMQP	-	-	?		✓	OpenFlow & RSVP-like	PoC (TRL 3)	✗
D-SDN				✓		✓		SC-SC Protocol	MC-SC Protocol	-	-		✓	OpenFlow	PoC (TRL 3)	✗
ElastiCon		✓				✓		DB-in/ TCP channel	-	-	NoSQL DB		✓	OpenFlow	PoC (TRL 3)	✗
FlowBroker				✓		✓		-	FlowBroker control channel	?	?		✓	OpenFlow	PoC (TRL 3)	✗
HyperFlow		✓					✓	WheelFS	-	-	WheelFS		✓	OpenFlow	PoC (TRL 3)	✗
Kandoo				✓		✓		-	Simple message channel	-	-		✓	OpenFlow	PoC (TRL 3)	✗
Orion					✓	?	?	Not needed	TCP channel	Pub/Sub	NoSQL DB		✓	OpenFlow	PoC (TRL 3)	✗
Cloud-oriented solutions																
DragonFlow		✓		✓		✓		DB-in	DB-in	DB-in	NoSQL DB/ others		✓	OpenFlow	Demonstrated (TRL 6)	
Onix		✓				✓		NoSQL DB	?	-	SQL DB NoSQL DB	✓	✓	OpenFlow & BGP	System prototype (TRL 7)	✗
ONOS		✓				✓		DB-in	-	-	Atomix (NoSQL framework)	✓	✓	OpenFlow, NetConf& others	Proven system (TRL 9)	
ODL (Fed)	✓		✓				✓	AMQP	-	-	In-memory	✓	✓	OpenFlow, BGP & others	Proven system (TRL 9)	
Tungsten	✓			✓		✓		BGP	IFMAP	DB-in	NoSQL DB	✓	✓	XMPP, BGP & others	Proven system (TRL 9)	

Proposals	Organization of network information			Inter-site networking services implementation	
	Information granularity	Information scope	Information availability	Automatized interfaces	Networking technologies
Network-oriented solutions					
DISCO	✓	✓	✓	✓	✗
D-SDN	✓	~	?	✗	✗
ElastiCon	~	?	?	~	✗
FlowBroker	✗	✗	✓	✗	✗
HyperFlow	✓	~	~	~	✗
Kandoo	✗	✗	✓	✗	✗
Orion	✗	✗	?	✗	✗
Cloud-oriented solutions					
DragonFlow	~	?	?	~	~
Onix	~	?	?	~	~
ONOS	~	?	?	~	✓
ODL (Fed)	✓	✓	~	✓	~
Tungsten	✗	✗	?	✗	✓

[1] ✓Challenge completely addressed.
[2] ~ Challenge partially addressed.
[3] ✗Challenge not addressed.
[4] ? Undefined.

TABLE III: Summary of the analyzed SDN solutions.

scenarios. The main idea is to have replicas of SCs in order to cope with network or node failures.

As stated in the proposition, D-SDN has a hierarchical design: the MC could be seen as a root controller and SCs as local controllers. It has a leader-based coordination, with MC being the natural leader in the hierarchy. As D-SDN is presented as a proof of concept, we defined a TRL of 3.

- *Information granularity*: Addressed - due to the segregation of the infrastructure elements into distinct groups.
- *Information scope*: Not addressed - the MC gathers global knowledge, and the communication between SC appear just for fault tolerance aspects.
- *Information availability*: Undefined - in case of disconnection, SCs controllers can continue to provide forwarding within its local domain at first sight. However, the article does not specify how the MC deals with such a disconnection. Besides, the controller does not provide any type of recovery method as D-SDN does not consider network partitioning issues.
- *Automatized interfaces*: Not addressed - D-SDN proposes an interface for SC-SC communication only for fault tolerance issues. Moreover, there is no information regarding MC-MC communication patterns.
- *Networking technologies*: Not addressed - since it does not integrate any other networking technologies nor the capacity to provide inter-group service deployment.

ElastiCon

Elasticon (i.e., elastic controller) is an SDN controller composed of a pool of controllers. The pool can be expanded or shrunk according to the size of the infrastructure to operate. Each controller within the pool is in charge of a subset of the SDN domain using OpenFlow as control plane protocol. The elasticity of the pool varies according to a load window that evolves over time. A centralized module triggers reconfigurations of the pool like migrating switches among controllers or adding/removing a controller based on the computed value. While decisions are made centrally, it is noteworthy to mention that actions are performed by the controllers. To do so, each controller maintains a TCP channel with every other one creating a full mesh. This protocol enables controllers to coordinate themselves if need be. The states of Elasticon are shared through the Hazelcast distributed data store [91], which can be accessed by all controllers. The use of a shared backend by the pool gives, as a result, a physically distributed but logically centralized design. As stated in Elasticon's work, the solution has been implemented as a prototype, and thus a TRL of 3 has been assigned to it.

- *Information granularity*: Partially addressed - Elasticon has been designed to distribute the control of infrastructure over several controllers. If the Hazelcast data store can be deployed across several sites, it is possible to envision distributing the pool of controllers between the different sites. By accessing the same database, controllers will be able to add information to the database and fetch the others' information to establish inter-site services. However, the consistency of the data store might be another issue to deal with.
- *Information scope*: Undefined - it is linked to the database capabilities (in this case, to the way the Hazelcast data store shards the information across the different sites of the infrastructure). However, it is noteworthy to mention that most advanced database systems such as CockroachDB only favor data-locality across several geo-distributed sites partially.
- *Information availability*: Undefined - similarly to the previous challenge, it is linked to the way the database services deals with network partitioning issues. In other words, intra/inter-domain forwarding paths that have been previously established should go on theoretically (network equipment has been already configured). Only the recovery mechanism to the DB is unclear.
- *Automatized interfaces*: Partially addressed - because each controller already has a TCP channel to commu-

nicate with the other controllers. However, this communication channel is only used for coordination purposes.
- *Networking technologies*: Not addressed - since it only operates in OpenFlow-based scenarios.

FlowBroker

FlowBroker is a two layers architecture using OpenFlow as a control plane protocol. It is composed of a series of broker agents and semi-autonomous controllers. The broker agents are located at the higher layer. They are in charge of maintaining a global view of the network by collecting SDN domain-specific network state information from the semi-autonomous controllers deployed at the bottom layer. Semi-autonomous controllers do not communicate among them, so they are not aware of other controllers' existence in the network. These controllers are only aware of interfaces in the controlled switches, thus, providing local-domain forwarding. By maintaining a global view, the broker agents can define how semi-autonomous controllers should establish flows to enable inter-domain path forwarding.

FlowBroker presents a hierarchical design clearly, with broker agents acting as root controllers and semi-autonomous domain controllers as local controllers. Although semi-autonomous controllers can establish forwarding paths inside their own domain, communication with the broker agents is mandatory for inter-domain forwarding. Because of this reason, FlowBroker presents a leader-based coordination, where brokers act as leaders. However, we underline that there is not any information describing how the information is shared between the different brokers.

Regarding maturity, we assigned a TRL of 3 to FlowBroker because only a proof-of-concept has been implemented.

Addressing the challenges:

- *Information granularity*: Not addressed - the segregation into semi-autonomous controllers enables the efficient sharing of the information per site. However, the global view of the information that is maintained by the brokers does not enable the validation of this property.
- *Information scope*: Not addressed - although the global view maintained by each broker allows them to contact only the semi-autonomous controllers that are involved in the inter-service creation, the result of each operation is forwarded to each broker in order to maintain the global view up-to-date.
- *Information availability*: Addressed - as aforementioned, semi-autonomous controllers can continue to provide local-domain forwarding without the need of the brokers. In the hypothetical case of a network disconnection and the subsequent reconnection, interconnected controllers can still forward the traffic among them. Actually, they only need to contact brokers in order to request the inter-site forwarding configuration. Once the configuration of network equipment has been done, controllers do not need to communicate with brokers. Regarding the loss of connectivity with brokers, the recovery process is quite simple because the information shared between all brokers and semi-autonomous controllers is conflict-less.
- *Automatized interfaces*: Not addressed - because semi-autonomous controllers do not have an east-west interface to communicate among them, but only communicate with brokers. Moreover, the way brokers exchange network knowledge to gather global network view is not discussed.
- *Networking technologies*: Not addressed - since its use is only intended with OpenFlow protocol.

HyperFlow

Hyperflow is an SDN NOX-based [92] multi-controller using OpenFlow as control plane protocol. The publish/subscribe message paradigm is used to allow controllers to share global network state information and is implemented using WheelFS [93]. Each controller subscribes to three channels: data channel, control channel, and its own channel. Events of local network domains that may be of general interest are published in the data channel. In this way, information propagates to all controllers allowing them to build the global view. Controller to controller communication is possible by publishing into the target's channel. Every controller publishes a heartbeat in the control channel to notify about its presence on the network.

As global networking information is shared by all participant controllers, the controller topology presents a physically distributed but logically centralized design. Every controller manages its own domain. In the case of network partitions, traffic forwarding can continue inside each controller domain and between the controllers that can contact each other. However, the dependency with respect to WheelFS is not discussed. In other words, the behavior of a controller that cannot contact WheelFS is undefined. More generally, the publish/subscribe paradigm enables Hyperflow to be leaderless. As this proposition has been implemented as a proof-of-concept, a TRL of 3 has been assigned to HyperFlow.

Addressing the challenges:

- *Information granularity*: Addressed - thanks to WheelFS, it is possible to deploy one controller per site. Each one uses WheelsFS to share networking information in order to create inter-domain forwarding paths.
- *Information scope*: Partially addressed - HyperFlow presents both a general information data channel and the possibility to communicate directly to specific controllers using their respective channel. Unfortunately, the paper does not clarify whether the establishment of inter-site forwarding is done by contacting the relevant controllers or if, instead, the general channel is used. In the former case, the exchange is efficient; in the latter, the information will be shared through all controllers.
- *Information availability*: Partially addressed - in case of a network partitioning, every controller can continue to serve their local forwarding requests. Regarding inter-forwarding, the dependency w.r.t. to WheelFS is unclear. Theoretically speaking, inter-forwarding channels should survive disconnections (at least among the controllers that can interact). Moreover, WheelFS provides a recovery method to deal with network partitioning issues. Such a feature should enable controllers to request new inter-forwarding paths after disconnections without implementing specific recovery mechanisms. Similar to previous

solutions, this is possible because the information shared through WheelFS is conflict-less.
- *Automatized interfaces*: Partially addressed - since WheelFS is used as both communication and storage utility among controllers. Thus, it is used as the east-west interface. However, HyperFlow's authors underlined that the main disadvantage of the solution is the use of WheelsFS: WheelsFS can only deal with a small number of events, leading to performance penalties in cases where it is used as a general communication publish/subscribe tool among controllers.
- *Networking technologies*: Not addressed - since it does not integrate other networking technologies besides OpenFlow.

Kandoo

Kandoo is a muti-controller SDN solution built around a hierarchy of controllers and using OpenFlow as control plane protocol. At the low level, local-domain controllers are in charge of managing a set of SDN switches and processing local traffic demands. At the high-level, the single root controller gathers network state information to deal with inter-domain traffic among the local domains. The Kandoo proposal authors claim that there are only a few inter-domain forwarding requests and that a single root controller is large enough to deal with. Regarding the local controllers, they do not know about the others' existence, thus only communicating with the root controller using a simple message channel to request the establishment of inter-domain flows. Unfortunately, the authors did not give sufficient information to understand how this channel works and how the inter-domain flows are set up.

By its two-level hierarchical design, Kandoo presents a leader-based coordination (the root controller being the architecture's natural leader). As the solution had been implemented as a proof-of-concept, a TRL of 3 has been assigned to Kandoo.

Addressing the challenges:
- *Information granularity*: Not addressed - the root controller is used to get information to do inter-domain traffic forwarding, thus gathering the global network view.
- *Information scope*: Not addressed - similarly to the previous challenge, there is no direct communication between controllers: the single root controller is aware of all inter-domain requests.
- *Information availability*: Addressed - similarly to FlowBroker solution, the root controller is only required to configure the inter-domain traffic. Once network equipment has been set up, there is no need to communicate with the root controller. The recovery process between local controllers and the root is simple: it consists of just recontacting the root once the network connectivity reappears (similarly to FlowBroker is conflict-less).
- *Automatized interfaces*: Not addressed - there is not an east-west interface to communicate among local controllers.
- *Networking technologies*: Not addressed - since Kandoo does not implement other protocols besides OpenFlow.

Orion

Orion is presented as a hybrid SDN proposition using OpenFlow as a control plane protocol. The infrastructure is divided into domains that are then divided into areas. Orion leverages area controllers and domain controllers. Area controllers are in charge of managing a subset of SDN switches and establish intra-area routing. Domain controllers, at the top layer, are in charge of synchronizing global abstracted network information among all domain controllers and to establish inter-area routing paths for their managed area controllers. Synchronization of network states between domain controllers is done using a scalable NoSQL database. Moreover, a publish/subscribe mechanism is used to allow domain controllers to demand the establishment of inter-area flows among them. Finally, it is noteworthy to mention that area controllers are not aware of other area controllers' existence and only communicate with their respective domain controller. This communication is done via a simple TCP channel.

Orion is the only solution that presents a hybrid design: each domain follows a two-level hierarchy, and all domain controllers are arranged in a P2P way, using a NoSQL database to share information between each other. Unfortunately, the paper does not give details regarding the NoSQL database nor the coordination protocol among the domain controllers. Hence, it is not clear whether Orion uses a leader-based coordination in its P2P model. As the solution had been implemented as a proof-of-concept, a TRL of 3 has been assigned to Orion.

Addressing the challenges:
- *Information granularity*: Not addressed - although the infrastructure is divided into domains (each domain controller maintains its own view of the information), each area controller should notify its domain controller about all changes that occur at the low level.
- *Information scope*: Not addressed - first, area controllers cannot contact directly other controllers to set up inter-site forwarding services, and second, we do not know how information is shared between domain controllers (i.e., it is related to the database system, see Elasticon for instance).
- *Information availability*: Undefined - in case of network disconnections, area controllers can continue to forward intra-domain traffic and inter-domain traffic on paths that have been previously established. In other words, domain controllers are used only for inter-domain path forwarding establishments. In the case of network disconnection, the area controller only needs to reconnect to its domain controller when needed and when the connection reappears. There is no need for a specific recovery protocol because the information shared between area controllers and their respective domain controller is conflict-less. Only the recovery mechanism related to the DB that is used to share information among domain controllers is unclear.
- *Automatized interfaces*: Not addressed - due to the fact that local controllers do not present an east-west interface to communicate among them.
- *Networking technologies*: Not addressed - since it does

not integrate other networking technologies aside from OpenFlow.

B. Cloud-Oriented

In this second part, we present the five solutions we selected from the cloud computing area, namely DragonFlow [94], Onix [95], ONOS [96], ODL [97], and Tungsten [98].

DragonFlow

DragonFlow is an SDN controller for the OpenStack ecosystem, i.e., it implements the Neutron API and thus can replace the default Neutron implementation (see Section II-B). DragonFlow relies on a centralized server (i.e., the Neutron server) and local controllers deployed on each compute node of the infrastructure from the software architecture. Each local controller manages a virtual switch, providing switching, routing, and DHCP capabilities using entirely OpenFlow. A DragonFlow ML2 mechanism driver and a DragonFlow service plug-in are activated in Neutron Server in order to provide system network information to all local controllers. Communication between the plug-ins at the Neutron server side and local controllers is done via a pluggable distributed database (currently supporting OVSDB [99], RAMCloud [100], Cassandra [101], and using etcd [102] as default back-end).

Local controllers periodically fetch all information of network state through this database and update virtual switches, routes, etc., accordingly.

By maintaining a global knowledge of the network elements through its distributed database, DragonFlow can be considered as a distributed but logically centralized controller (see Section V-A) at first sight. However, the fact that there is a root controller (i.e., the Neutron server-side) in charge of the management layer (i.e., updating configuration states in the distributed database) and local controllers that implement the control plane makes DragonFlow more a hierarchical solution than a distributed one. In other words, for all leader-based operations, the Neutron plug-in deployed at the server-side acts as the leader. In conclusion, although DragonFlow is presented as a distributed SDN controller, its design does not allow the management of a geo-distributed infrastructure (i.e., composed of multiple SDN controllers).

From the maturity viewpoint and according to its activity, we believe DragonFlow has reached a TLR 6. Initially supported by Huawei, the project is rather inactive right now.

Addressing the challenges:

- *Information granularity*: Partially addressed - similarly to Elasticon, if the distributed database service can be deployed across several sites, we can envision an infrastructure composed of several DragonFlow Neutron plug-in. Each one will add information to the database, and all local controllers will be capable of fetching the necessary information to provide end-to-end services.
- *Information scope*: Undefined - it is linked to the way the distributed database system shards the information across the different sites of the infrastructure.
- *Information availability*: Undefined - similarly to the previous challenge and to the Elasticon solution, it is linked to the way the distributed database services deals with network partitioning issues.
- *Automatized interfaces*: Partially addressed - DragonFlow controllers do not present an east-west interface to communicate with remote sites. Instead, the distributed database is used as a communication tool.
- *Networking technologies*: Partially addressed - the controller incorporates the adaptation and reconfiguration of networking services, but it lacks the heterogeneity of networking protocols. Currently, DragonFlow does not support BGP dynamic routing [103].

Onix

Onix is a multi-controller SDN platform. In other words, Onix presents several building blocks to develop network services in charge of operating either overlay (using OpenFlow by default) or underlay (using BGP if needed) networks.

Onix's architecture consists of multiple controller instances that share information through a data store called Network Information Base (NIB). The infrastructure is divided into domains, each domain being managed by one instance. Depending on durability and consistency, a network service may use a specific database to implement the NIB module. If durability and strong consistency are required, a replicated transactional SQL database should be used among the instances. Otherwise, it is possible to use any kind of NoSQL system.

Regarding coordination aspects, the system leverages ZooKeeper [104] to deal with instance failures (using the Zookeeper Atomic Broadcast protocol for leader election). The responsibility of the SDN equipment is then determined among the controllers.

By using multiple controllers, and a global network database, the Onix architecture corresponds to a physically distributed but logically centralized one.

As Onix was built as a basis for Nicira's SDN products but was not really a commercial product, a TRL of 7 was assigned.

Finally, the Onix platform integrates some applications, including the management of multi-tenant virtualized DCs. This service allows the creation of tenant-specific Layer 2 networks establishing tunnels among the hypervisors hosting VMs in one single deployment. However, this module works in a stand-alone mode and does not interact with the OpenStack Neutron service.

Addressing the challenges:

- *Information granularity*: Partially addressed - similarly to solutions such as Elasticon or DragonFlow, it is related to the database used to share the information between the instances.
- *Information scope*: Undefined - similarly to the previous challenge, it is related to the database. In case of strong consistency, the information should be synchronized across all instances. In the case of a NoSQL system, it depends on how the DB shards the information across different instances.
- *Information availability*: Undefined - established services can go on, and disconnected sites can continue to operate in isolated mode. The main issue is related to the NIB

that should provide the necessary consistency algorithms to allow recovery in case of network disconnection.
- *Automatized interfaces*: Partially addressed - similarly to DragonFlow, the use of distributed DB to share information among instances can be seen as an east-west interface allowing communication among controllers.
- *Networking technologies*: Partially addressed - the solution has been designed to use several networking technologies and protocols. Although the initial Onix proposition only supported OpenFlow, Onix design does not impose a particular southbound protocol but rather the use of the NIB as an abstraction entity for network elements.

ONOS

ONOS (Open Network Operating System) is a modular and distributed SDN framework consisting of several network applications build on top of Apache Karaf OSGi container [105]. It supports the use of multiple control plane protocols like OpenFlow, NetConf, and others. ONOS has been created for overlay and underlay networks of service providers. Network states' information is stored using the Atomix database [106], a NoSQL framework developed for ONOS, which is also used for coordination tasks among controllers.

Similar to other proposals, the infrastructure is divided into domains with one controller per domain. Considering the shared back-end and the multiple controller instances, ONOS presents a physically distributed but logically centralized design. As aforementioned, ONOS has a leader-based coordination approach, leveraging the Atomix DB (more precisely, it uses the RAFT algorithm). Considering that ONOS is one of the most popular SDN open-source controllers and is used by several key actors in telecommunications [107], a TRL of 9 has been assigned to ONOS.

Finally, the modular design of ONOS allows the implementation of the Neutron API. Concretely, there are three applications, which consume Neutron API and provide ML2 drivers and Services plug-ins: SONA (Simplified Overlay Network Architecture), VTN (Virtual Tenant Network), and CORD (Central Office Re-architected as a Datacenter) VTN. Each application has been designed with different targets [108], [109]. SONA provides an ML2 driver and an L3 service plug-in implementation. VTN provides service function chaining capabilities. CORD VTN extends VTN with its own interfaces for switching and routing configuration [110].

Addressing the challenges:

- *Information granularity*: Partially addressed - similarly to previous solutions that are composed around several instances and a global shared database.
- *Information scope*: Undefined - it is linked to the way the Atomix database system shards the information across the different instances.
- *Information availability*: Undefined - similarly to the previous challenge, it is linked to the Atomix system.
- *Automatized interfaces*: Partially addressed - ONOS controllers use the Atomix framework for coordination tasks among controllers and to communicate among them.
- *Networking technologies*: Addressed - ONOS includes several networking technologies.

OpenDayLight

OpenDayLight (ODL) is a modular SDN platform supporting a wide range of protocols such as OpenFlow, OVSDB, NETCONF, BGP, among others. Originally, OpenDayLight was developed as a centralized controller to merge legacy networks with SDN in data centers, but its modularity allows users to build their own SDN controller to fit specific needs [111]. The internal controller architecture is composed of three layers: The southbound interface, which enables communication with network devices. The Service Adaptation Layer adapts the southbound plug-in's functions to higher-level application/service functions. Finally, the northbound interface provides the controller's API to applications or orchestration tools. Network states are stored through a tree structure using a dedicated in-memory data store (i.e., developed for ODL). While the default implementation of ODL can be used in cluster mode for redundancy and high availability, its modularity allows the introduction of features aiming to allow different instances of the controller to peer among them like the SDNi [112] or the more recent Federation [113] projects. ODL Federation service facilitates the exchange of state information between multiple OpenDayLight instances. It relies on AMQP to send and receive messages to/from other instances. A controller could be at the same time producer and consumer.

The Federation project of ODL corresponds to a physical and logical distributed design (each instance maintains its own view of the system). Moreover, it is a leader-less coordination approach because there is a flat on-demand communication between controllers, and no leader is needed for these exchanges.

The modularity of the controller allows multiple projects to implement the Neutron API. For instance, ODL comes with the OpenStack Neutron API application. This application provides the abstractions that are mandatory for the implementation of the Neutron API inside the controller. Among those implementations, we found: Virtual Tenant Network Manager (VTN), Group-Based Policy (GBP), and OVSDB-based Network Virtualization Services (NetVirt) [114].

By leveraging the Federation and NetVirt projects, it is possible to create virtual network resources spreading across several OpenStack instances. When the Federation manager receives a request to create an inter-site service between two OpenStack instances, it realizes the interconnection at the ODL level (i.e., creating shadow elements, etc.) and performs the matching with the OpenStack Neutron resources on the different sites. Although this enables to interconnect multiple instances of OpenStack, it is noteworthy to mention that Neutron instances remain unconscious of the information shared at the ODL level. In other words, there is no coordination mechanism that will prevent overlapping information at the Neutron level. This is rather critical as it may lead to consistency issues where an IP, for instance, can be allocated on each site without triggering any notification.

Since ODL is a community leader and industry-supported framework presented in several industrial deployment and

continuous development, a TRL of 9 has been assigned to ODL [115].

Addressing the challenges:

- *Information granularity*: Addressed - through the Federation project, it is possible to leverage several controllers to operate an infrastructure (each controller maintains its own view).
- *Information scope*: Addressed - each controller can interact with another one by using AMQP. In other words, there is not any information that is shared between controllers unless needed.
- *Information availability*: Partially addressed - in case of network disconnection, ODL instances can satisfy local networking services (including the VIM ones). At the same time, the non-disconnected controllers can continue to provide the inter-site services. Since the inter-site services are proposed outside the VIM networking module's knowledge, ODL assumes that there are no conflicts between networking objects when establishing the service. Actually, ODL cannot provide a recovery method in case of incoherence since it is not the entity in charge of the networking information management. This is an important flaw for the controller when it needs to recover from networking disconnections.
- *Automatized interfaces*: Addressed - thanks to the use of AMQP as east-west interface among the controllers.
- *Networking technologies*: Addressed - ODL implements several networking technologies allowing to reconfigure each controller's networking services.

Tungsten (Open-Contrail)

Tungsten Fabric (previously known as Juniper's Open-Contrail) is the open-source version of Juniper's Contrail SDN controller, an industry leader for commercial SDN solutions targeting overlay and underlay Networks. Tungsten has two main components: an SDN controller and a virtual Router (vRouter). The SDN controller is composed of three types of nodes:

- Configuration nodes that are responsible for the management layer. They provide a REST API [116] that can be used to configure the system or extract operational status. Multiple nodes of this type can be deployed for HA purposes. Note that configuration states are stored in a Cassandra database (NoSQL).
- Control nodes are in charge of implementing decisions made by the configuration nodes. They receive configuration states from the configuration nodes using the IF-MAP protocol and use IBGP to exchange routing information with other control nodes. They are also capable of exchanging routes with gateway nodes using BGP.
- Analytic nodes are responsible for collecting, collating, and presenting analytic information.

The vRouter is a forwarding plane of a distributed router that runs in a virtualized server's hypervisor. It is responsible for installing the forwarding state into the forwarding plane. It exchanges control states such as routes and receives low-level configuration states from control nodes using XMPP.

Although there is no constraint on how the different nodes should be deployed, Tungsten architecture can be considered a two-level hierarchical design. Configuration nodes could be seen as root controllers and control nodes as local controllers (hence the configuration nodes can be considered as the leaders). Given the fact that the solution is used by several of the most important actors in the industry and that anyone can test the code, a TRL of 9 has been assigned to Tungsten. Tungsten integrates closely with Neutron consuming its API. Since Tungsten supports a large set of networking services, it is configured as a Core plug-in in Neutron.

Addressing the challenges:

- *Information granularity*: Not addressed - although multiple configuration nodes can share the network information through Cassandra, the internal design of Tungsten prevents the deployment of different configuration nodes across different sites. An extension has been proposed to handle multi-region scenarios [117]. However, the extension exposes a centralized entity to orchestrate remote controllers.
- *Information scope*: Not addressed - the configuration nodes share a global knowledge base. One operation is visible by all configuration nodes.
- *Information availability*: Undefined - because Tungsten has been designed for a single deployment, the impact of network disconnections between the configuration and control nodes has not been discussed in detail. It is unclear what could happen if a control node cannot interact with the site that hosts the configuration nodes for a long period.
- *Automatized interfaces*: Not addressed - although control nodes can interact with each other, there is no east-west interface to communicate among configuration nodes of different Tungsten deployments.
- *Networking technologies*: Addressed - Tungsten incorporates several networking technologies and is able to configure a different kind of network equipment.

C. Summary

This section described twelve solutions that propose leveraging on several controllers to manage virtualized networking infrastructure. Solutions such as FlowBroker, D-SDN, Tungsten, and Kandoo use a hierarchy of controllers to gather networking states and maintain a global view of the infrastructure. To avoid the SPOF issue of the root controller (see Section V-A, most of these systems propose to deploy multiple instances. By deploying as many root controllers as local ones, it is possible to transform such a hierarchical architecture into a distributed one and envision direct communication between each root controller when needed. The pending issue is related to the global view of the system that needs to be maintained by continuously exchanging messages among the root controllers (i.e., distributed but logically centralized architecture).

To deal with such an issue, solutions such as Elasticon, HyperFlow, Orion, DragonFlow, Onix, and ONOS, use a distributed database, which enables controllers to easily maintain

and share global networking information. While it is one more step to fulfill the system's requirements, the efficiency of these systems depends on the capabilities offered by the database system. Even if dedicated systems have been designed for some of them (e.g., ONOS), they do not cope with the requirements we defined in terms of data locality awareness or network partitioning issues.

The two remaining systems, i.e., DISCO and ODL, propose a fully distributed architecture (i.e., without the need for a global view). More precisely, DISCO respects the principle of locality awareness and independence of every group composing the infrastructure: Each controller manages its respective group and peers with another only when traffic needs to be routed to it, thus sharing only service-concerning data and not necessarily global network information. This way of orchestrating network devices is also well fitted in cases of network partitions as an isolated DISCO controller will be capable of providing local domain services. The flaw of DISCO is to provide networking services without the scope of the VIM (i.e., it delivers mainly domain-forwarding operations, which includes only conflict-less exchanges). Offering the VIM expected operations (such as dynamic IP assignment) is prone to conflict and thus might be harder to implement in such an architecture. We discussed this point for ODL, which has a lot of similarities with DISCO (data locality awareness, AMQP to communicate among controllers, etc.). Through the Federation and Netvirt projects, ODL offers premises of a DCI networking service but at a level that does not enable it to solve conflicts. Leveraging the DISCO or ODL architecture and investigating how conflicts can be avoided is a future direction of this survey, as underlined in the following section.

VII. *New cloud ecosystems: the Kubernetes case*

Although OpenStack still remains the de facto opensource solution to operate private cloud platforms, it is noteworthy that the popularity of VMs as the main unit to execute workloads has been decreasing in favor of lighter technologies such as Docker-based containers [118]. By promising low-overhead virtualization and improved performance, containers have become the new center of interest of DevOps [119]. Consequently, a couple of new frameworks in charge of managing the life cycle of container-based applications have been developed [120]. Released by Google in 2016, Kubernetes [32] (a.k.a. K8s) has become the default solution to manage containers on top of a distributed infrastructure. In addition to help DevOps create, deploy, and destroy containers, K8s proposes several abstractions that hide the distributed infrastructure's complexity. The way it manages network resources, for instance, differs from the OpenStack solution. K8s does not propose means to virtualize and control multiple network segments but rather expose services that relieve DevOps with the burden of managing network low-level aspects (e.g., IP assignments, L2 and L3 management, load balancing, etc.). While Kubernetes significantly from its network abstractions, the challenges at the low level remain close to the ones discussed previously in this article. After summarizing the Kubernetes architecture and networking capabilities, we discuss three projects that aim to deliver inter-site connectivity between multiple K8s instances. By introducing these proposals, we aim to underline that the management of multiple K8s clusters in a DCI context is still an ongoing study field that can benefit from a further discussion around distributed management.

A. *Kubernetes overview*

According to the ETSI standards, K8s can be considered as a Container Infrastructure Service (CIS), a service that provides a runtime environment for one or more container technologies [121]. [2].

Concretely, K8s divides a cluster into two parts: a set of worker machines called *Nodes*, where containerized applications are executed, and a set of control plane machines called the *Master nodes*, in charge of managing the aforementioned Nodes. Figure 6 depicts the K8s default architecture. Each Node has an agent called *kubelet* that is in charge of creating and configuring containers according to the Master orders, an agent called *kube-proxy* that is used to define networking rules. Finally, a container runtime such as Docker [123], Linux Containers [124], or any other implementation of Kubernetes Container Runtime Interface (CRI) [125] to effectively start and execute containers. The Master is composed of the API server, the scheduler that assigns workloads to Nodes, the controller managers that maintain the cluster's expected state using control loops, and etcd, a key-value store used as Kubernetes backend.

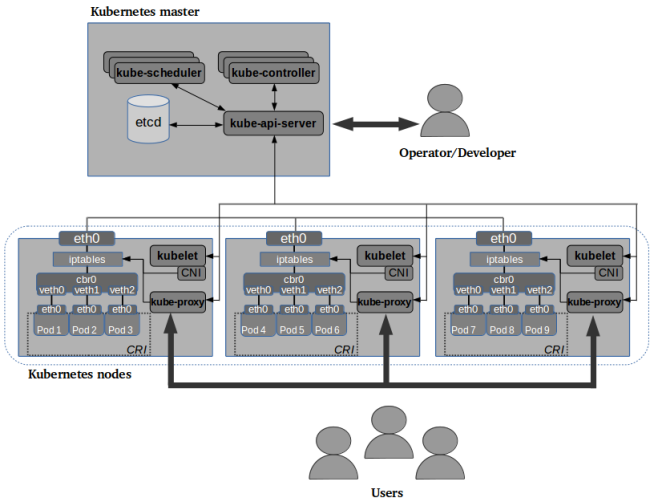

Fig. 6: Kubernetes cluster components.

K8s does not directly deal with containers but works at the granularity of *Pods*. A Pod is a group of one or more containers with shared networking and storage resources and a specification defining how to run the workload (number of replicas, etc.).

[2]This interpretation is open to debate since it can also be defined as a Container Infrastructure Service Management (CISM), a function that manages one or more CISs [122]

In addition to basic operation on Pods (creation, deployment, etc.), K8s proposes several abstractions (objects or resources in the K8s terminology) to hide the architecture's distribution. In other words, DevOps do not have to deal with low-level aspects of the infrastructure but use K8s predefined objects. For instance, *Volumes* are storage units accessible in a Pod wherever they are deployed. Similarly, *Services* are used to logically abstract a Pods group with a DNS name and a virtual IP. Finally, a *Namespace* enables DevOps to isolate Pods within a cluster. Additional objects have been built on top of these abstractions in order to offer supplementary functionalities. For instance, ReplicaSet enables DevOps to define many replicas for a Pod and let the K8s controller maintain this number. Because of the modularity of K8s, more objects can be exposed by the API (for an up-to-date list, please refer to [126]). This philosophy of using predefined abstractions is a major change concerning the OpenStack solution where DevOps should deal with many infrastructure details.

B. Kubernetes networking

From the network point of view, there are four types of communications in K8s: (i) Highly coupled container-to-container, (ii) Pod-to-Pod, (iii) Pod-to-Service, and (iv) External-to-Service.

Rather than imposing a single implementation and in order to leverage modularity, K8s supports its networking functionality through the Container Network Interface (CNI) [127], an open-source project proposing a specification and libraries for writing plug-ins to configure network interfaces on Linux containers. A CNI plug-in is responsible for inserting a network interface into the container network, making necessary changes on the host, assigning an IP address to the interface, and configuring routes for the traffic exchange. A second plug-in called the IP Address Management plug-in (IPAM) is used for the IP and routes configurations. Several CNI as well as IPAM plug-in implementations have been proposed to abstract the network within a K8s cluster [128]–[135]. It is noteworthy to mention that the split between the CNI plug-in and the IPAM module provides more flexibility as it is possible to use a combination of two different solutions.

By default, implementing a plug-in should deliver the four types of communications with the following properties. Regarding Pod-to-Pod communications, a Pod on a Node can communicate with all Pods on all Nodes without NAT communications. Regarding Pod-to-Service communications, the virtual IP associated with a Service needs to be correctly translated and load-balanced to Pods registered to this Service (using technologies such as iptables or IP virtual servers (IPVS) [136]). Finally, for the External-to-Service exchanges, the configuration of the different routes exposing Services should be achieved (implementing the logic of the K8s *NodePort*, *Load* or *Ingress controller* objects).

C. Multi-Cluster Kubernetes

Like OpenStack, Kubernetes has been designed in a standalone manner: it exposes the resources of a single cluster. Hence, being able to execute container-based applications across multiple sites (i.e., a DCI as we defined in this article) raises different questions in the Kubernetes ecosystem also [137]. From the network viewpoint, a K8s Multi-Cluster architecture should deliver means to provide the aforementioned communications. More precisely, since Container-to-Container communication is limited at Pod's scope, solutions for the three latest communications are required. We discuss in the following three projects, namely Kubernetes Federation [138], Istio Multi-Cluster service mesh [139] and Cilium MultiCluster CNI implementation [140]. From our point of view, these projects are the most advanced among a large number of proposals [141]–[145]. In addition to a brief description, we present whether they address the DCI challenges introduce in Section IV.

1) Kubernetes Federation: Kubernetes Federation (a.k.a. Kubefed) is a project developed by the multi-cluster working group of K8s, providing an easy-to-use way to manage multiple K8s clusters. Kubefed does not target the inter-site networking connectivity but rather a global approach for deploying container-based applications across multi-sites through a common API. In other words, it does not leverage nor integrate SDN technologies but rather proposes a broker-like approach to partially deal with the DCI challenges.

In detail, Kubefed relies on a two-layer architecture where a central cluster called *host cluster* will propagate its application configuration to a series of independent clusters called *member clusters*. To make this possible, the *host* leverages a federation API using custom resource definitions (CRDs), an object provided by the vanilla Kubernetes that allows DevOps to define their own data types. These new federated objects are then used to wrap basic objects. For example, *FederatedService* and *FederatedDeployment* objects are abstractions to wrap the vanilla Service and Deployment objects.

When a *FederatedService* is created, Kubefed creates matching K8s Services in the selected member clusters. To propagate each cluster's DNS records, Kubefed gathers all the locally generated DNS records in the host cluster and then pushes the records to each of the concerning clusters. This implies that Services must be exposed using a publicly available IP address. From the network point of view, Kubefed can only provide cross-cluster Pod-to-Service and External-to-Service communications, relying on the public routable IP addresses in both cases. Since it proposes management at the API level, no coordination is possible at the low-level networking implementation. In consequence, cross-cluster Pod-to-Pod communication is not proposed. More generally, Kubefed presents important flaws for the DCI context. In addition to the host cluster's limitation that is the only entry point for federated resources (SPOF), there is no collaboration among the different K8s instances. In other words, there is no mechanism to propagate modifications done on one particular K8s object to the other sites, even if this object has been created through a federated abstraction.

Addressing the challenges:

- *Information granularity*: Not addressed - the segregation of information into each cluster enables the efficient sharding of the information per site. However, the host

cluster gathers global information about federated resources.
- *Information scope*: Partially addressed - while Kubefed only contacts the relevant sites when deploying a federated resource, this is only done by the host cluster.
- *Information availability*: Addressed - in case of network disconnection, each cluster is fully independent of the others, with the worst-case scenario being the isolation of the host cluster. Since a federated resource is deployed on the concerned clusters, the local resources' information remains locally available.
- *Automatized interfaces*: Not addressed - because member clusters do not have an east-west interface to communicate among them, but only receive requests from the host cluster.
- *Networking technologies*: Not addressed - Kubefed relies on a broker-like approach. Consequently, no connectivity at the networking level is established among the member clusters, and no networking technology is used.

2) Istio Multi-Cluster service mesh: Istio is an open-source implementation of a service mesh that provides traffic management, security, and monitoring. In the microservices context, a service mesh is a complementary layer to the application, and it is responsible for traffic management, policies, certificates, and service security [146]. To provide this, a service mesh introduces a set of network proxies that will be used to route requests among services. The idea is to inject a special sidecar container in the Pod of every microservice and route traffic through these sidecars instead of doing it through the Service. A central authority will then exerts control over the proxies to route traffic at the application layer (L7 of the OSI model). Hence, a service mesh follows a design pattern familiar to the SDN principles [147].

From the architecture viewpoint, an Istio deployment is logically composed by a control plane, which manages and configures the proxies and elements such as gateways to route traffic, and a data plane, which is composed of a set of intelligent proxies (Envoy [148]) deployed as sidecars. Istio also proposes an Ingress Gateway object that acts as a load balancer at the mesh's edge receiving incoming or outgoing traffic.

This concept of service meshes can be extended to take account of multiple clusters. The idea is then to have a logical service mesh composed of several clusters [149]. For this to be done, Services from remote clusters are created locally via *ServiceEntries*, an Istio resource that is not proposed by vanilla Kubernetes. The Istio Ingress Gateway is then used as an entry point for requests to the cluster's Services.

The Istio service mesh operates at the application layer. Offering its functionalities at this level implies that Istio specific resource definitions need to be used in deployments. Besides, considering all hops, a request must go through from containers to sidecars in a DCI context along with all the ServiceEntry rules treatment and processing requests on remote gateways could greatly add latency and potentially add a performance overhead [150], [151].

Regarding the three communication types, since Istio is a service mesh-oriented solution, it can only provide cross-cluster Pod-to-Service and External-to-Service communications using the replication of ServiceEntries. In the case of Pod-to-Service communication, Services with private virtual IPs are reachable through the Istio Ingress gateways.

Addressing the challenges:

- *Information granularity*: Addressed - due to the segregation of the infrastructure in independent clusters while proposing strategies to share the information related to external Services using the replication of remote Services as ServiceEntries.
- *Information scope*: Addressed - since Istio proposes the creation of ServiceEntries to reference remote Services, only the relevant clusters are taking into account to do the information exchange.
- *Information availability*: Partially addressed - in case of network disconnections clusters remain locally operative. However, considering that ServiceEntries are replicated on demand, Istio does not provide mechanisms to ensure the consistency between K8S Services and the information related to the ServiceEntries.
- *Automatized interfaces*: Not addressed - Istio does not implement an east-west interface allowing cluster collaboration. Instead, the user is in charge of mirroring the Istio configuration between the different clusters in order to deliver a Multi-Cluster service mesh.
- *Networking technologies*: Not addressed - although the network routing logic is implemented at the Envoy proxy and at the Istio Ingress Gateway, Istio is independent of the low-level network technology used by the cluster.

3) Cilium Multi-Cluster: Cilium is a CNI plug-in that implements the Kubernetes networking API by leveraging the extended Berkeley Packet Filter (eBPF). EBPF is used to forward data within the Linux kernel. It is an alternative of IP Tables, which delivers better performance [152], [153].

In addition to the Cilium plug-in, each K8s node executes a Cilium agent. (This agent is in charge of interacting with the CRI to setup networking and security for containers running on the node. Finally, a key-value store, global to the cluster, is used to share data between Cilium Agents deployed on different nodes.

Cilium proposes a multi-cluster implementation called *ClusterMesh*. It provides Pod IP routing among multiple Kubernetes clusters thanks to tunneling techniques through eBPF (i.e., without dedicated proxies or gateways). Concretely, the Cilium key-value store of each cluster must be exposed as a public Service. This enables Cilium agents to collect information from all stores in order to create tunnels with all other clusters. Moreover, Cilium allows the creation of a Global Service by creating at each cluster a Service with an identical name and namespace. An additional Cilium annotation defining the Service as global is mandatory on each cluster. Cilium uses this annotation to automatically load-balance requests throughout all Pods exposing the Service in the different clusters.

Thanks to its relation with K8S at the CNI level, Cilium effectively provides the three aforementioned communications types. However, such a model's scalability is questionable

as ClusterMesh initiates a tunnel (e.g., VXLAN or Geneve) between each worker nodes pair.

Addressing the challenges:

- *Information granularity*: Addressed - Cilium ClusterMesh leverages several independent clusters operating the infrastructure, each one only managing its own deployment.
- *Information scope*: Not addressed - because Cilium ClusterMesh requires to connect all clusters before deploying applications, agents create tunnels to all remote Nodes at the cluster setup. In this sense, information is exchanged even before a user requires an inter-site resource to be deployed.
- *Information availability*: Addressed - due to clusters being independent among them. In the case of networking partitioning, the isolated sites continue to propose their services locally, and sites still connected can continue to provide the ClusterMesh capabilities.
- *Automatized interfaces*: Addressed - Cilium proposes to expose the local database to remote clusters to exchange information. The way remote clusters consume this information could inspire more propositions leveraging their databases' exposition as an east-west interface among clusters. It is noteworthy to mention that the user still needs to deploy applications and Services in each cluster to provide a Multi-Cluster Service.
- *Networking technologies*: Partially addressed - While the solution leverages eBPF as networking technology incorporating its adaptation and reconfiguration, it lacks the heterogeneity of networking protocols.

D. Summary

Building a system capable of providing a native distributed DCI management from scratch is not an easy task. As we have demonstrated for OpenStack, Kubernetes also needs to be extended.

While some ongoing works try to propose ways to make independent Kubernetes collaboration, we consider that there is still plenty of innovation and improvement opportunities. Kubernetes Federation proposes to leverage a "master" cluster exposing a federated API that will translate the application deployment to a series of worker clusters. The problem with this approach is that worker clusters are not aware of each other presence and the "master" cluster is the only entry point for federated resources. Other projects such as Istio and Cilium require the user to deploy applications independently in each cluster, which can be costly when needed in hundreds or thousand clusters [150], [154].

VIII. Directions for DCI networking research

In the previous sections, we have studied major decentralized SDN technologies in the DCI context. While we identified that DISCO and ODL are good candidates, several open questions and challenges need further investigation. We discuss the most important ones in the following.

A. East-West SDN-based interfaces

While the East-West interface has been considered as the principal element to provide inter-controller communications [155], there is still no standard to communicate between controllers, and some proposals co-exist [156]–[159]. This is an issue as such a standard is critical for delivering collaborations between networking modules of multiple VIM instances of a DCI. Moreover, this lack of standardization impacts the communication and automation between North and East-West interfaces. This leads to multiple ad-hoc solutions [40], [157].

As we outlined in the last section, the East-West interface proposed by DISCO and ODL provides some references to design an efficient horizontal interface for inter-VIMs networking modules communications. Although the analyzed solutions leveraged AMQP as technology to do the East-West interface implementation, other technologies such as REST APIs could be used to provide synchronization and information exchanges among VIMs.

If we consider the model proposed by DISCO and ODL, the use of independent and local databases implies managing consistency at the application level (i.e., between the different controllers). This entails that the East-West interface should deliver additional calls to resolve conflicts depending on the controllers' inter-site service logic. Since neither DISCO nor ODL proposes a way to manage conflicts at the East-West interface level, this remains an open question, as already highlighted. Another solution could consist of leveraging distributed databases.

B. Leveraging new kinds of databases

As we highlighted in the summary of Section VI, solutions such as Elasticon, HyperFlow, Orion, DragonFlow, Onix, and ONOS, use a distributed database to share global networking information among controllers. While this approach is useful as it is intended to avoid a single point of failures and bottlenecks by logically splitting the database, it does not respect the principles of data locality and is not resilient enough in case of network partitions.

To illustrate this point, one can consider the Cassandra database [101]. Cassandra is based on a distributed hash table (DHT) principles to uniformly distribute states across multiple locations based on the computed hash. This means that one specific controller's states are not necessarily stored in the same geographical location as the controller (e.g., a controller in site A will have states stored in remote sites B, C, and so forth). Thus, the principle of locality awareness is not respected as information belonging to site A will be spread to other sites with no possibility to control the location.

Likewise, an SDN-based cloud architecture that leverages Cassandra will not be resilient to network isolation. It will be impossible for the local controller to retrieve its states, which may have been spread over non-reachable sites.

However, data-related approaches different from traditional SQL and NoSQL engines can be relevant. In the last years, the concept of NewSQL has been gaining popularity and notoriety as an approach mixing the advantages of both traditional SQL and NoSQL systems. These kinds of engines search to propose

the same scalability of NoSQL systems while keeping the relational model of traditional SQL (i.e., maintaining the ACID guarantees) engines [160]. NewSQL engines generally try to leverage different memory storage modes, modes of partitioning, and concurrency control to provide the aforementioned properties.

While historically, database engines have used disk-oriented storage, today, NewSQL engines can profit from memory cost reduction and leverage memory-oriented approaches. Using this approach, new engines can get better performance because slow reads and writes to disk can be avoided [161]. Moreover, almost all engines used to scale out splitting a database up into subsets, called partitions or shards.

NewSQL engines can be gathered in three main classes [162]: new architectures, transparent sharding middlewares, and Databases-as-a-Service.

- *New architectures*: This group gathers engines build from scratch and that mostly use a distributed *shared-nothing* architecture [163]. Most of them are also capable of sending intra-query data directly between nodes rather than having to route them to a central location. In this group we find solutions such as Clustrix [164], CockRoachDB [165], Google Spanner [166] and its related solutions such as Google F1 [167], VoltDB [168], or HyPer [169].
- *Transparent sharding middleware*: This group gathers engines that split a database into multiple shards that are stored across a cluster of single node instances. In this sharding, each node runs the same database management system. Each one has only a portion of the overall database, and data does not mean being accessed and updated independently by separate applications. Then, a centralized middleware component routes queries, coordinates transactions, and manages data placement, replication, and partitioning across the nodes. In this group we find solutions such as AgilData Scalable Cluster [170] or MariaDB MaxScale [171].
- *Database-as-a-Service (DBaaS)*: While there are already DBaaS propositions, there are only a few NeWSQL DBaaS engines available. In this group we find solutions such as Amazon Aurora [172] or ClearDB [173]. For instance, Amazon Aurora does a decoupling between the engine storage and compute. In this sense, the network becomes the constraint because all input and outputs (I/O) will be written over it. In order to do this operation, Aurora relies upon a log-structured storage manager capable of improving I/O parallelism [174].

In order to measure the value of these new engines in the DCI context, an SDN-based application needs to be built on top of the selected solution to analyze the pros and cons of each one and verify if they can satisfy the DCI requirements highlighted in Section I.

On the other hand, solutions such AntidoteDB [175] or Riak [176] that have been designed on top of conflict-free replicated data type (CRDT) [177] could be leveraged by SDN controllers in order to address the aforementioned DCI challenges while respecting the principal characteristics of DCI architectures such as data locality awareness. A CRDT is a data structure that can be replicated across multiple nodes in a network. Each replica can be updated independently and concurrently. This means that each replica will be locally accessible and ready to use in the case of network partitions. The richness of the CRDT structure is that it is possible to eventually resolve the inconsistency between multiple replicas. However, CRDTs come with two important limitations. First, it requires to replicate the state of every site of the DCI infrastructure. Second, only elementary values can be structured as CRDT right now. For instance, it is not sure that a CIDR can be modeled as a CRTD. If solutions for these two problems might be found, CRDT may represent a step forward to provide a distributed solution while respecting the DCI properties.

C. Data plane technologies

The ecosystem to deliver traffic forwarding between virtual instances is old and extremely rich, with multiple proposals since the initial OpenFlow protocol [36] (BGP [178], SoftRouter [179], RCP [180], as well as RouteFlow [181] to name a few). This eco-system continues to grow with more recent solutions [182]–[184]. Since heterogeneity in networking protocols may be present in a DCI, the possibility to agree on which mechanisms to use when establishing an inter-site networking service needs to be considered in future works. While being only a theoretical proposition, the works presented in [185] for Neutron to Neutron Interconnections proposes such kind of mechanism agreement. In this sense, two Neutron instances will identify the mechanism to use and the corresponding parameters that will be exchanged (e.g., VLANs IDs, BGPVPN RT identifiers, VXLAN IDs, or L2GWs addresses).

D. Performance analysis

This survey focuses on defining a general distributed model for DCI networking management based on SDN technologies. While it gives valuable information, it would be relevant to evaluate the selected solutions under the performance perspective. Depending on the analyzed element (e.g., East-West interface, database, or networking protocols) of new proposals, the metrics and analyzed characteristics may vary:

- *East-West interfaces:* The use of a horizontal interface implies that besides the time expended by the system to answer a user request locally (e.g., CPU consumption, memory consumption, threads utilization), the time needed to synchronize with remote sites and provide a final result needs to be taken into account [159]. Such delay will impact the total inter-site service creation time and may depend on the technical implementation of the solution (e.g., protocol used). In the architectural design, the quantity of messages needed to establish a service, needs to be optimized to minimize the overhead. Thus, future works should analyze the impact on the performance of their inter-site communication model and the technology used to implement it.
- *New kinds of databases:* Like the East-West interface, the use of new kinds of database engines will add an extra delay to communicate with remote sites. NewSQL

benchmarking studies such as the ones proposed in [186], [187] could be extended to take into account the DCI case. Additionally, a comparison can be made w.r.t. traditional SQL database replication systems and distributed databases in two different aspects: the local execution time spent by the database executing CRUD actions (e.g., CPU consumption, memory consumption,...), and the time needed to communicate with remote sites in order to replicate data or synchronize them. Moreover, an important analysis should be done to clarify how the database will deal with conflicts or inconsistencies in network partitions. Although new database engines provide models based on theoretical assumptions, there are side effects that can be identified only by conducting experimental activities The aforementioned studies [1], [10] can be taken as a good example of such kind of experimental-driven research needed in this context.

- *Networking protocols:* Since the networking route exchange and traffic forwarding and routing will also impact the time needed for an inter-site service to be effectively deployed, the performance of the data plane technologies needs to be taken into account. For instance, in the case of BGP VPN routes exchanges, prior works analyzed the benefits and disadvantages of their use in several contexts [188], [189]. Although the implementation of several different networking protocols is a very challenging and complex task, solutions supported by large communities such as OpenStack, ODL, or ONOS could promote such development to perform further tests and analysis. Work such as [28], [190] proposing the comparison of virtual networking implementations in SDN-based scenarios can be used as guidelines to identify major indicators (i.e., throughput, CPU utilization, latency, packet loss, etc.).
- *Consensus protocols application:* If a consensus protocol is mandatory, further investigations will be needed in order to quantify the overall performance of the protocol. While works such as [191]–[194] already analyzed the use of consensus protocols in SDN, they mostly targeted traditional SDN networks and not DCI architectures. Indeed, the round-trip times in a geo-distributed infrastructure, composed by hundreds or thousands of nodes, could affect the protocol performance by adding latency for which the consensus protocol may not be designed [68]. Moreover, tasks such as leader election may create traffic overload due to the number of messages exchanged. For this reason, compared with a static master election or leaderless protocols will be needed to understand the different trade-offs that may be involved.

E. Security

Some of the open questions in SDN-based cloud computing solutions rely on security issues. While security in cloud data centers has been explored in the last decade [195], [196], more research in security for SDN and SDN-based solutions is needed [46]. Obviously, the decentralization reduces the impact of having a single point of failure into an architecture (e.g., DoS attacks), but some other components need to be revised. For instance, the inter-site communication needs to be secure enough to avoid uncontrolled or non-desired intrusions. The protocols allowing the database distribution may also be deeply studied in order to evaluate the impact of encryption technologies in the overall performance of future solutions. Finally, as a great number of tenants may share the same network medium to access DCI networking services, the isolation and security provided by the networking protocols used will need further studies.

IX. Conclusions

Leveraging each Point of Presence (PoPs) in Telcos' architectures as part of a Distributed Cloud Infrastructure (DCI) is an approach that will be mandatory soon to address the requirements posed by trends like NFV, IoT, and MEC. Although some resource management systems for this kind of infrastructure have been proposed, providing inter-site networking services in a distributed fashion is still a challenge to be addressed.

This paper explained the problems of centralized networking management in VIMs and how decentralized SDN approaches could represent an opportunity to tackle the inter-site connectivity management challenges. We then presented a survey of several multi-controller SDN solutions and their possible use and limitations as a DCI network management tool.

Solutions such as FlowBroker, D-SDN, Tungsten, or Kandoo propose maintaining a global view of the system through a hierarchy of controllers that does not enable to address the identified challenges.

The use of a distributed database to share global information among the controller proposed by Elasticon, HyperFlow, Orion, DragonFlow, Onix or ONOS does not entirely address the proposed challenges as the use of the database in a DCI context and under network partitioning is unclear and do not respect the general requirements. We, however, do not eliminate such an approach as new database engines have been recently proposed [175], [186]. In particular, we should better understand how these systems behave according to the DCI challenges (data granularity, scope, and network partitioning issues) while respecting the DCI general requirements (such as the locality awareness).

Solutions following fully distributed architectures address several challenges while guaranteeing the general requirements. In particular, we would like to underline the potential of a solution like DISCO or ODL to distribute the connectivity management in a scalable, locality-aware, resilient, and easy manner. The work we should address in the short term is to investigate how VIM-related operations can be developed. In particular, a proposal should include creating on-demand Layer 2 extensions and Layer 3 to span overlay networks among the requested sites at the VIM-level, avoiding or solving the possible management conflicts.

We concluded our survey by analyzing the Kubernetes ecosystem. Kubernetes has gained important popularity in the DevOps community, in particular, due to its way of hiding low-level technical details thanks to high-level abstractions (e.g., DevOps no longer have to deal with IP assignments).

Although Kubernetes significantly differs from IaaS solutions, it has been designed to operate a single cluster. Similarly to OpenStack, the development of inter-site networking services at a low level is mandatory to enable K8s abstractions to be visible and consistent between multiple sites. Like DISCO or ODL, some proposals have been proposed. For instance, the Cilium ClusterMesh proposes such inter-site capabilities for the network in a decentralized manner. However, the requirement of interconnecting all worker nodes affects the scalability.

Whether we consider a resource management system for IaaS or CaaS solutions, the implementation of a distributed tool in which network devices are automatically configured, provisioned, and managed may represent a huge contribution to favor the advent of DCIs such as envisioned in Fog and Edge Computing platforms. Through this survey, our goal was to identify key elements that can be used to guide the design and implementation of such a tool.

REFERENCES

[1] A. Bousselmi, J. F. Peltier, and A. Chari, "Towards a Massively Distributed IaaS Operating System: Composition and Evaluation of OpenStack," *IEEE Conference on Standards for Communications and Networking*, 2016.

[2] A. Lebre, J. Pastor, A. Simonet, and F. Desprez, "Revising OpenStack to Operate Fog/Edge Computing Infrastructures," *IEEE International Conference on Cloud Engineering*, 2017.

[3] "Deploying Distributed Compute Nodes to Edge Sites." https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/13/html/deploying_distributed_compute_nodes_to_edge_sites/deploying_distributed_compute_nodes_to_edge_sites. Accessed: 02/2020.

[4] "StarlingX, a complete cloud infrastructure software stack for the edge." https://www.starlingx.io. Accessed: 02/2020.

[5] D. Sabella, A. Vaillant, P. Kuure, U. Rauschenbach, and F. Giust, "Mobile-Edge Computing architecture: The role of MEC in the Internet of Things," *IEEE Consumer Electronics Magazine*, vol. 5, pp. 84–91, Oct 2016.

[6] OpenStack Foundation, "Cloud Edge Computing: Beyond the Data Center (White Paper)." https://www.openstack.org/assets/edge/OpenStack-EdgeWhitepaper\-v3-online.pdf, Jan 2018. (Accessed: 2020-02-10).

[7] Microsoft Azure, "Azure Global Network." https://azure.microsoft.com/en-ca/global-infrastructure/global-network/, 2020. Accessed: 06/2020.

[8] Google Cloud, "Google Cloud Locations." https://cloud.google.com/cdn/docs/locations, 2020. Accessed: 06/2020.

[9] AWS, "AWS global infrastructure." https://aws.amazon.com/about-aws/global-infrastructure/?nc1=h_ls, 2020. Accessed: 06/2020.

[10] R.-A. Cherrueau, "A POC of OpenStack Keystone over CockroachDB." https://beyondtheclouds.github.io/blog/openstack/cockroachdb/2017/12/22/a-poc-of-openstack-keystone-over-cockroachdb.html, 2017.

[11] J. Soares, F. Wuhib, V. Yadhav, X. Han, and R. Joseph, "Re-designing Cloud Platforms for Massive Scale using a P2P Architecture," *IEEE 9th International Conference on Cloud Computing Technology and Science*, 2017.

[12] OpenStack, "Neutron - Openstack Networking Service." https://docs.openstack.org/neutron/latest/, 2020. Accessed: 02/2020.

[13] A. Chari, T. Morin, D. Sol, and K. Sevilla, "Approaches for on-demand multi-VIM infrastructure services interconnection," Tech. Rep. 2489-1, Orange Labs Networks, Lannion, France, 2018.

[14] S. Subramanian and S. Voruganti, *Software-Defined Networking (SDN) with OpenStack*. Packt, 2016.

[15] ETSI, "Network Functions Virtualisation (NFV) Ecosystem, Report on SDN Usage in NFV Architectural Framework," Tech. Rep. DGS/NFV-EVE005, European Telecommunications Standards Institute, 2015.

[16] M. Mechtri, I. Houidi, W. Louati, and D. Zeghlache, "SDN for Inter Cloud Networking," in *Proceedings of the 2013 IEEE SDN for Future Networks and Services*, pp. 1–7, 2013.

[17] I. Petri, M. Zou, A. Reza-Zamani, J. Diaz-Montes, O. Rana, and M. Parashar, "Software Defined Networks within a Cloud Federation," in *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 179–188, 2015.

[18] A. Sanhaji, P. Niger, P. Cadro, C. Ollivier, and A.-L. Beylot, "Congestion-based API for cloud and WAN resource optimization," *2016 IEEE NetSoft Conference and Workshops*, 2016.

[19] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-deployed Software Defined Wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pp. 3–14, 2013.

[20] K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat, "Taking the edge off with espresso: Scale, reliability and programmability for global internet peering," 2017.

[21] Z. Yang, Y. Cui, B. Li, Y. Liu, and Y. Xu, "Software-Defined Wide Area Network (SD-WAN): Architecture, Advances and Opportunities," *28th International Conference on Computer Communication and Networks*, 2019.

[22] ETSI, "Network Functions Virtualisation (NFV); Management and Orchestration ," Tech. Rep. DGS/NFV-MAN001, European Telecommunications Standards Institute, 2014.

[23] P. Marsch, Ömer Bulakci, O. Queseth, and M. Boldi, *5G System Design: Architectural and Functional Considerations and Long Term Research*. John Wiley & Sons, 2018.

[24] M.-A. Kourtis, M. McGrath, G. Gardikis, G. Xilouris, V. Riccobene, P. Papadimitriou, E. Trouva, F. Liberati, M. Trubian, J. Batallé, H. Koumaras, D. Dietrich, A. Ramos, J. Ferrer, J. Bonnet, A. Pietrabissa, A. Ceselli, and A. Petrini, "T-NOVA: An Open-Source MANO Stack for NFV Infrastructures," *IEEE Transactions on Network and Service Management*, 2017.

[25] T. Soenen, S. V. Rossem, W. Tavernier, F. Vicens, D. Valocchi, P. Trakadas, P. Karkazis, G. Xilouris, P. Eardley, S. Kolometsos, M.-A. Kourtis, D. Guija, S. Siddiqui, P. Hasselmeyer, J. Bonnet, and D. Lopez, "Insights from SONATA: Implementing and integrating a microservice-based NFV service platform with a DevOps methodology," *2018 IEEE/IFIP Network Operations and Management Symposium*, 2018.

[26] T. Das, V. Sridharan, and M. Gurusamy, "A Survey on Controller Placement in SDN," *IEEE Communications Surveys Tutorials*, vol. 22, no. 1, pp. 472–503, 2020.

[27] M. A. Togou, D. A. Chekired, L. Khoukhi, and G. Muntean, "A Hierarchical Distributed Control Plane for Path Computation Scalability in Large Scale Software-Defined Networks," *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 1019–1031, 2019.

[28] S. Azodolmolky, P. Wieder, and R. Yahyapour, "SDN-based Cloud Computing Networking," *ICTON 2013*, 2013.

[29] Patel, Parveen and Bansal, Deepak and Yuan, Lihua and Murthy, Ashwin and Greenberg, Albert and Maltz, David A. and Kern, Randy and Kumar, Hemant and Zikos, Marios and Wu, Hongyu and Kim, Changhoon and Karri, Naveen, "Ananta: Cloud Scale Load Balancing," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, (New York, NY, USA), p. 207–218, Association for Computing Machinery, 2013.

[30] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2014.

[31] X. Leng, K. Hou, Y. Chen, K. Bu, L. Song, and Y. Li, ""A lightweight policy enforcement system for resource protection and management in the SDN-based cloud"," *Computer Networks*, vol. 161, pp. 68 – 81, 2019.

[32] Linux Foundation, "Kubernetes." https://kubernetes.io/docs/home/, 2020.

[33] L. Yang, R. Dantu, T. Anderson, and R. Gopal, "Forwarding and Control Element Separation (ForCES) Framework," RFC 3746, RFC Editor, April 2004.

[34] J. E. van der Merwe, S. Rooney, L. Leslie, and S. Crosby, "The tempest-a practical framework for network programmability," *IEEE Network*, vol. 12, no. 3, pp. 20–28, 1998.

[35] D. Tennenhouse and D. Wetherall, "Toward an active network architecture," *Proceedings of SPIE - The International Society for Optical Engineering*, pp. 2–16, 03 1996.

[36] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, JenniferRexford, S. Shenker, and J. Turner, "OpenFlow: Enabling

innovation in campus networks," *ACM SIGCOMM Computer Communications Review*, 2008.

[37] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: an intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[38] Cisco, "OpFlex: An Open Policy Protocol White Paper," Tech. Rep. 1538025281906783, Cisco, San Jose, California, 2014.

[39] B. Medeiros, M. S. Jr., T. Melo, M. Torrez, F. Redigolo, E. Rodrigues, and D. Cristofoleti, *Applying Software-defined Networks to Cloud Computing*. 33rd Brazilian Symposium on Computer Networks and Distributed Systems, 2015.

[40] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014.

[41] VMware, "VMware NSX Data Center." https://www.vmware.com/products/nsx.html, 2020.

[42] Juniper networks, "Contrail SDN controller." https://www.juniper.net/us/en/products-services/sdn/contrail/, 2020.

[43] Nuage networks, "Nuage SDN controller." https://www.nuagenetworks.net/solutions/telco-cloud/, 2020.

[44] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[45] S. Azodolmolky, P. Wieder, and R. Yahyapour, "Cloud Computing Networking: Challenges and Opportunities for Innovations," *IEEE Communications Magazine*, 2013.

[46] J. SON and R. BUYYA, "A Taxonomy of SDN-enabled Cloud Computing," *ACM Computing Surveys*, 2017.

[47] OpenStack, "OpenStack." https://docs.openstack.org/latest/, 2020.

[48] OpenStack, "OpenStack Nova Project." https://docs.openstack.org/nova/latest/, 2020.

[49] Linux Foundation, "OpenvSwitch." https://www.openvswitch.org/, 2018.

[50] Linux Foundation, "Linux Bridges." https://wiki.linuxfoundation.org/networking/bridge, 2018.

[51] OpenStack Foundation, "Open Virtual Network." https://docs.openstack.org/networking-ovn/latest/, 2020.

[52] OpenStack, "Neutron Networking-L2GW." https://docs.openstack.org/networking-l2gw/latest/readme.html, 2018.

[53] OpenStack, "Neutron BGPVPN Interconnection." https://docs.openstack.org/networking-bgpvpn/latest/, 2018.

[54] OpenStack, "Neutron VPNaaS." https://docs.openstack.org/neutron-vpnaas/latest/, 2019.

[55] OpenStack Foundation, "Networking API v2.0." https://docs.openstack.org/api-ref/network/v2/, 2020.

[56] OpenStack, "Tricircle Project." https://wiki.openstack.org/wiki/Tricircle, 2018.

[57] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN Control: Survey, Taxonomy and Challenges," *IEEE Communications Surveys & Tutorials*, 2018.

[58] M. Karakus and A. Durresi, "A survey: Control plane scalability issues and approaches in Software-Defined Networking (SDN)," *Computer Networks 112*, 2016.

[59] H. Yang, J. Ivey, and G. F. Riley, "Scalability Comparison of SDN Control Plane Architectures Based on Simulations," *International Performance Computing and Communications Conference*, 2017.

[60] O. Blial, M. B. Mamoun, and R. Benaini, "An Overview on SDN Architectures with Multiple Controllers," *Hindawi*, vol. 2016, 2016.

[61] Y. E. Oktian, S. Lee, H. Lee, and J. Lam, "Distributed SDN controller system: A survey on design choice," *Computer Networks 121*, 2017.

[62] OSRG, "GoBGP." https://osrg.github.io/gobgp/, 2018. Tokyo, Japan.

[63] Z. Li, Z. Duan, and W. Ren, "Designing Fully Distributed Consensus Protocols for Linear Multi-agent Systems with Directed Graphs," *IEEE Transactions on Automatic Control 60*, 2014.

[64] I. Moraru, D. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," pp. 358–372, 11 2013.

[65] A. Turcu, S. Peluso, R. Palmieri, and B. Ravindran, "Be general and don't give up consistency in geo-replicated transactional systems," pp. 33–48, 12 2014.

[66] L. Lamport, "The Part-Time Parliament," *ACM Transactions on Computer Systems 16*, 1998.

[67] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," *USENIX Annual Technical Conference*, 2014.

[68] A. Ailijiang, A. Charapko, and M. Demirbas, "Consensus in the Cloud: Paxos Systems Demystified," *25th International Conference on Computer Communication and Networks*, 2016.

[69] Y. Zhang, E. Ramadan, H. Mekky, and Z.-L. Zhang, "When Raft Meets SDN: How to Elect a Leader and Reach Consensus in an Unruly Network," *Proceedings of the First Asia-Pacific Workshop on Networking*, 2017.

[70] R. Palmieri, "Leaderless consensus: The state of the art," pp. 1307–1310, 05 2016.

[71] S. Binani, A. Gutti, and S. Upadhyay, "Sql vs. nosql vs. newsql- a comparative study," *Communications on Applied Electronics*, vol. 6, pp. 43–46, 10 2016.

[72] M. Ronstrom and L. Thalmann, "Mysql cluster architecture overview," *MySQL Technical White Paper*, vol. 8, 2004.

[73] T. Khasawneh, M. Alsahlee, and A. Safia, "Sql, newsql, and nosql databases: A comparative survey," pp. 013–021, 04 2020.

[74] K. Banker, *MongoDB in action*. Manning Publications Co., 2011.

[75] M. Paksula, "Persisting objects in redis key-value database, white paper," 2010. University of Helsinki, Department of Computer Science, Helsinki, Finland.

[76] J. Webber, "A programmatic introduction to neo4j," pp. 217–218, 10 2012.

[77] A. Lakshman and P. Malik, "Cassandra — a decentralized structured storage system," *Operating Systems Review*, vol. 44, pp. 35–40, 04 2010.

[78] A. Davoudian, L. Chen, and M. Liu, "A survey on nosql stores," *ACM Comput. Surv.*, vol. 51, Apr. 2018.

[79] M. Stonebraker, "Sql databases v. nosql databases," *Commun. ACM*, vol. 53, pp. 10–11, 04 2010.

[80] Open Networking Foundation, "OpenFlow Switch Specifications," tech. rep., Open Networking Foundation, 2015.

[81] European Commission, "Horizon work programme 2020." https://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf, 2014.

[82] SDXCentral, "SDN Controller Comparison Part 1: SDN Controller Vendors (SDN Controller Companies)." https://www.sdxcentral.com/sdn/definitions/sdn-controllers/sdn-controllers-comprehensive-list/, 2014.

[83] SDXCentral, "SDN Controller Comparison Part 2: Open Source SDN Controllers." https://www.sdxcentral.com/sdn/definitions/sdn-controllers/open-source-sdn-controllers/, 2014.

[84] K. Phemius, M. Bouet, and J. Leguay, "DISCO: Distributed Multi-domain SDN Controllers," *Network Operations and Management Symposium*, 2014.

[85] M. Santos, B. Nunes, K. Obraczka, and T. Turletti, "Decentralizing SDN's Control Plane," *IEEE Conference on Local Computer Networks*, 2014.

[86] A. Dixit, F. Hao, S. Mukherjee, Lakshman, and R. K. t, "Towards an Elastic Distributed SDN Controller," *HotSDN*, 2013.

[87] D. Marconett and S. Yoo, "FlowBroker: A Software-Defined Network Controller Architecture for Multi-Domain Brokering and Reputation," *Journal of Network System Management*, 2015.

[88] A. Tootoonchian and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow," *IEEE Proceedings of the 2010 internet network management conference on Research on enterprise networking*, 2010.

[89] S. Yeganeh and Y. Ganjali, "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications," *HotSDN*, 2012.

[90] Y. Fu, J. Bi, K. Gao, Z. Chen, J. Wu, and B. Hao, "Orion: A Hybrid Hierarchical Control Plane of Software-Defined Networking for Large-Scale Networks," *IEEE 22nd International Conference on Network Protocols*, 2014.

[91] "Hazelcast Project." https://hazelcast.org/. (Accessed: 06/2020-).

[92] Nicira Networks, "NOX Network Control Platform." https://github.com/noxrepo/nox, 2009.

[93] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, F. Kaashoek, and R. Morris, "Flexible, Wide-Area Storage for Distributed Systems with WheelFS," *6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[94] OpenStack, "DragonFlow : Distributed implementation of Neutron within a large DC." https://wiki.openstack.org/wiki/Dragonflow, 2015.

[95] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A Distributed Control Platform for Large-scale Production Networks," *OSDI*, 2012.

[96] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," *HotSDN '14*, 2014.

[97] J. Medved, A. Tkacik, R. Varga, and K. Gray, "OpenDaylight: Towards a Model-Driven SDN Controller Architecture," *IEEE WoWMoM*, 2014.

[98] Juniper, "Contrail Architecture," 2015.

[99] Linux Foundation, "The Open vSwitch Database." http://docs.openvswitch.org/en/latest/ref/ovsdb.7/, 2013.

[100] J. Ousterhout, M. Rosenblum, S. Rumble, R. Stutsman, S. Yang, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. Park, and H. Qin, "The RAMCloud Storage System," *ACM Transactions on Computer Systems*, vol. 33, pp. 1–55, 08 2015.

[101] Apache Software Foundation, "Apache Cassandra." http://cassandra.apache.org/, 2016.

[102] Cloud Native Computing Foundation, "etcd." http://etcd.io/, 2016.

[103] OpenStack, "DragonFlow : BGP dynamic routing." https://docs.openstack.org/dragonflow/latest/specs/bgp_dynamic_routing.html, 2018.

[104] Apache Software Foundation, "ZooKeeper: A Distributed Coordination Service for Distributed Applications." https://zookeeper.apache.org/doc/r3.4.13/zookeeperOver.html, 2008.

[105] Apache Software Foundation, "Apache Karaf." https://karaf.apache.org/, 2010.

[106] Open Networking Foundation, "Atomix." https://atomix.io/docs/latest/user-manual/introduction/what-is-atomix/, 2019.

[107] ONOS, "ONOS - Community." https://www.opennetworking.org/onos/, 2020.

[108] ONOS, "SONA architecture ONOS." https://wiki.onosproject.org/display/ONOS/SONA+Architecture, 2018.

[109] ONOS, "CORD VTN ONOS." https://wiki.onosproject.org/display/ONOS/CORD+VTN, 2018.

[110] ONOS, "ONOS - OpenStack (Neutron) Integration." https://groups.google.com/a/onosproject.org/forum/?oldui=1#!msg/onos-discuss/NIS-m-mpp3E/dO1wHCeSAwAJ;context-place=forum/onos-discuss, 2017.

[111] The New Stack, "OpenDaylight is One of the Best Controllers for OpenStack." https://thenewstack.io/opendaylight-is-one-of-the-best-controllers-for-openstack-heres-how-to-implement-it/, 2015.

[112] OpenDayLight, "OpenDaylight SDNi Application." https://wiki.opendaylight.org/view/ODL-SDNi_App:Main, 2014.

[113] OpenDayLight, "OpenDaylight Federation Application." https://wiki.opendaylight.org/view/Federation:Main, 2016.

[114] OpenDayLight, "OpenDaylight NetVirt Application." https://wiki.opendaylight.org/display/ODL/NetVirt, 2020.

[115] OpenDayLight, "User stories OpenDaylight." https://www.opendaylight.org/use-cases-and-users/user-stories, 2018.

[116] R. Fielding, *Chapter 5: Representational State Transfer (REST). Architectural Styles and the Design of Network-based Software Architectures(Dissertation)*. UNIVERSITY OF CALIFORNIA, 2000.

[117] Juniper, "Contrail Global Controller." https://www.juniper.net/documentation/en_US/contrail3.2/topics/concept/global-controller-vnc.html, 2016.

[118] A. Randal, "The ideal versus the real: Revisiting the history of virtual machines and containers," *ACM Comput. Surv.*, vol. 53, Feb. 2020.

[119] S. Singh and N. Singh, "Containers & Docker: Emerging roles & future of Cloud technology," pp. 804–807, 01 2016.

[120] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and Virtual Machines at Scale: A Comparative Study," in *Proceedings of the 17th International Middleware Conference*, Middleware '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[121] ETSI, "Network Functions Virtualisation (NFV) Release 3; Architecture; Report on the Enhancements of the NFV architecture towards "Cloud-native" and "PaaS" ," Tech. Rep. DGR/NFV-IFA029, European Telecommunications Standards Institute, 2019.

[122] Cloud iNfrastructure Telco Task Force, "Define the place of Kubernetes in the ETSI NFV MANO stack and document the result." https://github.com/cntt-n/CNTT/issues/450, 2019. online.

[123] Docker, "Docker." https://www.docker.com/, 2020.

[124] Canonical, "Linux containers." https://linuxcontainers.org/, 2020.

[125] Linux Foundation, "CRI: the Container Runtime Interface." https://github.com/kubernetes/kubernetes/blob/242a97307b34076d5d8f5bbeb154fa4d97c9ef1d/docs/devel/container-runtime-interface.md, 2016.

[126] Linux Foundation, "Kubernetes API overview." https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.18/, 2020.

[127] Cloud Native Computing Foundation, "Container Network Interface specification." https://github.com/containernetworking/cni/blob/master/SPEC.md, 2020.

[128] Linux Foundation, "Kubernetes Network Plugins." https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/, 2019.

[129] Linux Foundation, "Cluster Networking: Kubernetes." https://kubernetes.io/docs/concepts/cluster-administration/networking/, 2019.

[130] Tigera, "Calico for Kubernetes." https://docs.projectcalico.org/v2.0/getting-started/kubernetes/, 2020.

[131] Isovalent, "Cilium." https://cilium.io/, 2019. Mountain View, California.

[132] Cisco, "Contiv." https://contiv.io/, 2019.

[133] CoreOS, "Flannel." https://github.com/coreos/flannel, 2019.

[134] Weave-Works, "Weave Net." https://www.weave.works/blog/weave-net-kubernetes-integration/, 2016.

[135] OpenStack, "OpenStack kuryr." https://docs.openstack.org/kuryr-kubernetes/latest/, 2019.

[136] Jun Du and Haibin Xie and Wei Liang, "IPVS-Based In-Cluster Load Balancing Deep Dive." https://kubernetes.io/blog/2018/07/09/ipvs-based-in-cluster-load-balancing-deep-dive/, 2018.

[137] Andrew Jenkins, "To Multicluster, or Not to Multicluster: Inter-Cluster Communication." https://www.infoq.com/articles/kubernetes-multicluster-comms/, 2019.

[138] SIG Multicluster, "Kubernetes Cluster Federation." https://github.com/kubernetes-sigs/kubefed, 2020. San Francisco, California.

[139] Istio, "Multicluster Deployments ." https://istio.io/v1.2/docs/concepts/multicluster-deployments/, 2020. Mountain View, California.

[140] Cilium, "Cilium Cluster Mesh." https://docs.cilium.io/en/v1.8/gettingstarted/clustermesh/, 2020. Mountain View, California.

[141] Admiralty, "Multi Cluster Scheduler." https://github.com/admiraltyio/multicluster-scheduler, 2020. Seattle, Washington.

[142] Linkerd, "Multicluster Kubernetes with Service Mirroring." https://linkerd.io/2020/02/25/multicluster-kubernetes-with-service-mirroring/, 2020.

[143] Rancher, "Submariner." https://submariner.io//, 2020.

[144] HashiCorp, "Multi-Cluster Federation overview." https://www.consul.io/docs/k8s/installation/multi-cluster/overview, 2020.

[145] *Submariner, connected Kubernetes overlay networks*, 2020. https://github.com/submariner-io/submariner.

[146] Istio, "Istio: What is a service mesh?." https://istio.io/latest/docs/concepts/what-is-istio/#what-is-a-service-mesh, 2020. Mountain View, California.

[147] G. Antichi and G. Rétvári, "Full-Stack SDN: The Next Big Challenge?," in *Proceedings of the Symposium on SDN Research*, SOSR '20, (New York, NY, USA), p. 48–54, Association for Computing Machinery, 2020.

[148] Envoy Project, "Envoy." https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy, 2020. San Francisco, California.

[149] Istio, "Istio Replicated Control Planes." https://istio.io/latest/docs/setup/install/multicluster/gateways/, 2020. Mountain View, California.

[150] Venkat Srinivasan, "Connecting multiple Kubernetes Clusters on vSphere with Istio Service Mesh." https://medium.com/faun/connecting-multiple-kubernetes-clusters-on-vsphere-with-istio-service-mesh-a017a0dd9b2e, 2020.

[151] Istio, "Istio Performance and Scalability." https://istio.io/latest/docs/ops/deployment/performance-and-scalability/, 2020. Mountain View, California.

[152] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture.,"

[153] Jonathan Corbet, "Extending extended BPF." https://lwn.net/Articles/603983/, 2014.

[154] Venkat Srinivasan, "Connecting multiple Kubernetes clusters on vSphere with the Cilium Cluster Mesh." https://medium.com/faun/connecting-multiple-kubernetes-clusters-on-vsphere-with-the-cilium-cluster-mesh-964f95267df4, 2020.

[155] E. Haleplidis, K. Pentikousis, S. Denazis, J. Hadi-Salam, D. Meyer, and O. Koufoupavlou, "Software-Defined Networking (SDN): Layers and Architecture Terminology," RFC 7426, RFC Editor, January 2015.

[156] H. Yin, H. Xie, T. Tsou, D. Lopez, P. Aranda, and R. Sidi, "SDNi: A Message Exchange Protocol for Software Defined Networks (SDNS) across Multiple Domains." Internet Draft, June 2012.

[157] P. Lin, J. Bi, and Y. Wang, *East-West bridge for SDN network peering*, vol. 401, pp. 170–181. Springer, 01 2013.

[158] H. Yu, K. Li, H. Qi, W. Li, and X. Tao, "Zebra: An East-West Control Framework For SDN Controllers," *International Conference on Parallel Processing*, 2015.

[159] F. Benamrane, M. B. Mamoun, and B. Redouane, "An East-West interface for distributed SDN control plane: Implementation and evaluation," *Computers & Electrical Engineering*, 2016.

[160] M. Aslett, "How will the database incumbents respond to NoSQL and NewSQL? ," Tech. Rep. 451:1–5, 451 Group, April 2011.

[161] S. Harizopoulos, D. Abadi, S. Madden, and M. Stonebraker, "Oltp through the looking glass, and what we found there," pp. 981–992, 01 2008.

[162] A. Pavlo and M. Aslett, "What's really new with newsql?," *SIGMOD Rec.*, vol. 45, p. 45–55, Sept. 2016.

[163] M. Stonebraker, "The case for shared nothing," *IEEE Database Eng. Bull.*, vol. 9, pp. 4–9, 1985.

[164] MariaDB, "A NEW APPROACH TO SCALE-OUT RDBMS." https://mariadb.com/wp-content/uploads/2018/10/Whitepaper-ANewApproachtoScaleOutRDBMS.pdf, Oct 2018. (Accessed: 06/2020-).

[165] CockRoachLab, "CockRoachDB." https://www.cockroachlabs.com/product/, 2018.

[166] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, 08 2013.

[167] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte, "F1: A distributed sql database that scales," in *VLDB*, 2013.

[168] L. VoltDB, "Voltdb technical overview," *Whitepaper*, 2010.

[169] A. Kemper and T. Neumann, "Hyper: A hybrid oltp olap main memory database system based on virtual memory snapshots," in *2011 IEEE 27th International Conference on Data Engineering*, pp. 195–206, 2011.

[170] AgilData, "AgilData Scalable Cluster for MySQL." https://www.agildate.com/product/, 2020.

[171] MariaDB, "MariaDB MaxScale." https://mariadb.com/resources/datasheets/mariadb-maxscale/, 2019.

[172] Amazon, "Amazon Aurora." https://aws.amazon.com/rds/aurora/, 2019.

[173] Navisite, "ClearDB." https://www.cleardb.com/, 2019.

[174] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, (New York, NY, USA), p. 1041–1052, Association for Computing Machinery, 2017.

[175] INRIA, "AntidoteDB." https://www.antidotedb.eu/, 2017.

[176] Riak, "Riak database." https://riak.com/, 2019.

[177] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirsk, "Conflict-Free Replicated Data Types," in *Stabilization, Safety, and Security of Distributed Systems*, vol. 6976, Springer, 2011.

[178] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)," RFC 4271, RFC Editor, January 2006.

[179] T. Lakshman, T. Nandagopal, R. Ramjee, K. Sabnani, and T. Woo, "The SoftRouter Architecture," *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networking*, 2004.

[180] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and Implementation of a Routing Control Platform," *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, 2005.

[181] M. Nascimento, C. Rothenberg, M. Salvador, C. Correa, S. de Lucena, and M. Magalhaes, "Virtual Routers as a Service: The RouteFlow Approach Leveraging Software-Defined Networks," *Proceedings of the 6th International Conference on Future Internet Technologies*, 2011.

[182] K. Thimmaraju, B. Shastry, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldmann, and S. Schmid, "Taking Control of SDN-based Cloud Systems via the Data Plane," *Proceedings of the Symposium on SDN Research*, 2018.

[183] A. Wion, M. Bouet, L. Iannone, and V. Conan, "Distributed Function Chaining with AnycastRouting," *Proceedings of the 2019 ACM Symposium on SDN Research*, 2019.

[184] A. Wang, Z. Chen, T. Yang†, and M. Yu, "Enabling Policy Innovation in Interdomain Routing:A Software-Defined Approach," *Symposium on SDN Research 2019*, 2019.

[185] OpenStack, "Neutron-Neutron Interconnections." https://specs.openstack.org/openstack/neutron-specs/specs/rocky/neutron-inter.html, 2018.

[186] K. Kaur and M. Sachdeva, "Performance evaluation of NewSQL databases," in *2017 International Conference on Inventive Systems and Control (ICISC)*, pp. 1–5, IEEE, 2017.

[187] G. A. Schreiner, R. Knob, D. Duarte, P. Vilain, and R. d. S. Mello, "Newsql through the looking glass," in *Proceedings of the 21st International Conference on Information Integration and Web-Based Applications & Services*, iiWAS2019, (New York, NY, USA), p. 361–369, Association for Computing Machinery, 2019.

[188] F. Palmieri, "VPN scalability over high performance backbones evaluating MPLS VPN against traditional approaches," *Proceedings of the Eighth IEEE International Symposium on Computers and Communication*, 2003.

[189] J. Mai and J. Du, "BGP performance analysis for large scale VPN," *2013 IEEE Third International Conference on Information Science and Technology*, 2013.

[190] A. Risdianto, J.-S. Shin, and J. Kim, "Deployment and evaluation of software-defined inter-connections for multi-domain federated sdn-cloud," pp. 118–121, 06 2016.

[191] E. Sakic and W. Kellerer, "Response time and availability study of raft consensus in distributed sdn control plane," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 304–318, 2018.

[192] T. Zhang, P. Giaccone, A. Bianco, and S. De Domenico, "The role of the inter-controller consensus in the placement of distributed sdn controllers," *Computer Communications*, vol. 113, pp. 1 – 13, 2017.

[193] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "Netpaxos: Consensus at network speed," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[194] C. Ho, K. Wang, and Y. Hsu, "A fast consensus algorithm for multiple controllers in software-defined networks," in *2016 18th International Conference on Advanced Communication Technology (ICACT)*, pp. 112–116, 2016.

[195] J. Rittinghouse and J. Ransome, *Cloud Computing Implementation, Management, and Security*. CRC Press, 2009.

[196] M. Almorsy, J. Grundy, and I. Müller, "An Analysis of the Cloud Computing Security Problem," *Proceedings of the APSEC 2010 Cloud Workshop*, pp. 1–6, 2010.