Distributing connectivity management in Cloud-Edge infrastructures using SDN-based approaches

David Espinel Sarmiento

DOCTORAT BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPERIEURE MINES-TELECOM ATLANTIQUE
BRETAGNE PAYS DE LA LOIRE -IMT ATLANTIQUE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
David ESPINEL SARMIENTO

Distributing connectivity management in Cloud-Edge infrastructures using SDN-based approaches

Thèse présentée et soutenue à IMT Atlantique - Campus de Nantes, le 07 septembre 2021
Unité de recherche : LS2N
Thèse N° : 2021IMTA0250

Rapporteurs avant soutenance :

Vania MARANGOZOVA-MARTIN	Professeure (HdR) - Université Grenoble-Alpes
Stefano SECCI	Professeur (HdR) - CNAM

Composition du Jury :

Président :	Thomas LEDOUX	Professeur (HdR) - IMT Atlantique
Examinateurs :	Vania MARANGOZOVA-MARTIN	Professeure (HdR) - Université Grenoble-Alpes
	Stefano SECCI	Professeur (HdR) - CNAM
	Isabelle CHRISMENT	Professeure (HdR) - Université de Lorraine
	Lucas NUSSBAUM	Professeur associé - INRIA
	Abdelhadi CHARI	Ingénieur chercheur - ORANGE
Dir. de thèse :	Adrien LEBRE	Professeur (HdR) - IMT Atlantique

The art of victory is learned in defeat.

SIMON BOLIVAR

Acknowledgments

I would first like to thank my thesis advisor, Adrien Lebre, who always pushed me to give my best in all situations. Thanks for all your questions and your invaluable support and assistance all along with this thesis. I could not imagine a better advisor than you.

Also, I want to thank my academic tutor, Lucas Nussbaum, for his valuable guidance during my thesis. My sincere thanks to my enterprise advisor, Abdelhadi Chari, for all his cooperation.

Otherwise, I wish also to thank jury members, Vania Marangozova-Martin and Stefano Secci for having accepted to be rapporteurs of this thesis, Thomas Ledoux for having accepted being the jury president, and Isabelle Chrisment for her presence as examiner.

I would also like to thank Maria Luisa Guerra Feliz de Vargas, NAVI team manager, for allowing me to do a thesis at Orange Labs. I would like to acknowledge my colleagues from the NAVI team, for their professionalism and expertise.

Throughout the thesis, I have received a great deal of assistance without which nothing would have been possible. A special thanks to my former colleague, Ali Sanhaji, who assisted me more than once with his technical expertise and suggestions every time I had problems. Thanks for all the gym recommendations and all the dissertations at coffee time.

I would like to thank the STACK team, for the warm welcome during my time at Nantes and every event, we did together. I thank all the professors and fellow students with whom I was able to exchange points of view and ideas. I would particularly like to single out Marie Delavergne, I want to thank you for all your assistance and patience while helping me in the development of my experiments. Thanks also to Ronan-Alexandre Cherreau and Mathieu Simonin without which understanding Grid'5000 would have been a nightmare.

I want also to thank all the colleagues, Ph.D. fellows, and interns that I have crossed in this time at Orange. Thanks to Pierre-Léo Begay, Sylvain Barthomeuf, Pierre Mahé, Anas El Ankouri, Flavio Sampaio, Gael Simon, Rémi Rigal, Minqi Wang, and Tanguy Lé Gléau. Thanks to the colleagues of Cyclo Detente Ploulec'h with whom I shared the passion for cycling. Thanks for all the rides and amazing views of Britany.

Despite the distance, I would like to thank my parents, Alvaro and Lucia, and my grandmother, Nohora, for their constant support and understanding. Thanks to my broth-

ers, Camilo and Ronald, for all the laughs and fights. Life wouldn't have been the same without you. How to forget my thanks to my faculty friends, Pedro Estupiñan, Christian Jimenez, and Sebastian Valencia who have been there for me all these years.

Finally, my special thanks go to Deisy. For your neverending support, for all the talks, for all the tears, for all the joys, for all the trips, for all the smiles. Because you have been at my side in all situations with your unconditional love. This one is for you.

Abstract

The evolution of the cloud computing paradigm in the last decade has amplified the access of on-demand services (economical attractive, easy-to-use manner, etc.). However, the current model built upon a few large datacenters (DC) may not be suited to guarantee the needs of new use cases, notably the boom of the Internet of Things (IoT). To better respond to the new requirements (in terms of delay, traffic, etc.), compute and storage resources should be deployed closer to the end-user. In the case of telecommunication operators, the network Point of Presence (PoP), which they have always operated, can be inexpensively extended to host these resources. The question is then how to manage such a massively Distributed Cloud Infrastructure (DCI) to provide end-users the same services that made the current cloud computing model so successful.

In this thesis realized in an industrial context with Orange Labs, we study the inter-site connectivity management issue in DCIs leveraging the Software-Defined Networking (SDN) principles. More in detail, we analyze the problems and limitations related to centralized management, and then, we investigate the challenges related to distributed connectivity management in DCIs.

We provide an analysis of major SDN controllers indicating whether they are able or not to answer the DCI challenges in their respective contexts. Based on this detailed study, which is a first contribution on its own, we propose the DIMINET solution, a service in charge of providing on-demand connectivity for multiple sites. DIMINET leverages a logically and physically distributed architecture where instances collaborate on-demand and with minimal traffic exchange to provide inter-site connectivity management. The lessons learned during this study allows us to propose the premises of a generalization in order to be able to distribute in a non-intrusive manner any service in a DCI.

Résumé en Français

L'évolution du paradigme d'Informatique en nuage au cours de la dernière décennie a permis de démocratiser les services à la demande de manière significative (plus simple d'accès, économiquement attrayant, etc.). Cependant, le modèle actuel construit autour de quelques centres de données de très grande taille ne permettra pas de répondre aux besoins des nouveaux usages liés notamment à l'essor de l'Internet des Objets. Pour mieux répondre à ces nouvelles exigences (en termes de latence, volumétrie, etc.), les ressources de calculs et de stockages doivent être déployées à proximité de l'utilisateur. Dans le cas des opérateurs de télécommunications, les points de présence réseau qu'ils opèrent depuis toujours peuvent être étendus à moindre cout pour héberger ces ressources. La question devient alors : comment gérer une telle infrastructure nativement géo-distribuée (référencée dans le manuscrit sous l'acronyme DCI pour Distributed Cloud Infrastructure) afin d'offrir aux utilisateurs finaux les même services qui ont fait le succès du modèle actuel d'Informatique en nuage.

Dans cette thèse réalisée dans un contexte industriel avec Orange Labs, nous étudions le problème de la gestion distribuée de la connectivité entre plusieurs sites d'une DCI et proposons d'y répondre en utilisant les principes des réseaux définis par logiciel (connus sous les termes "Software Defined Network"). De manière plus précise, nous rappelons les problèmes et les limitations concernant la gestion centralisée, et ensuite, examinons les défis pour aller vers un modèle distribué, notamment pour les services liés à la virtualisation réseaux.

Nous fournissons une analyse des principaux contrôleurs SDN distribués en indiquant s'ils sont capables ou non de répondre aux défis des DCIs. Sur cette étude détaillé, qui est une première contribution en soi, nous proposons la solution DIMINET, un service en charge de fournir une connectivité à la demande entre plusieurs sites. DIMINET s'appuie sur une architecture distribuée où les instances collaborent entre elles à la demande et avec un échange de trafic minimal pour assurer la gestion de la connectivité. Les leçons apprises durant cette étude nous permettent de proposer les prémisses d'une généralisation afin de pouvoir "distribuer" d'une manière non intrusive n'importe quels services en charge de gérer une infrastructure géo-distribuée.

Résumé Etendu

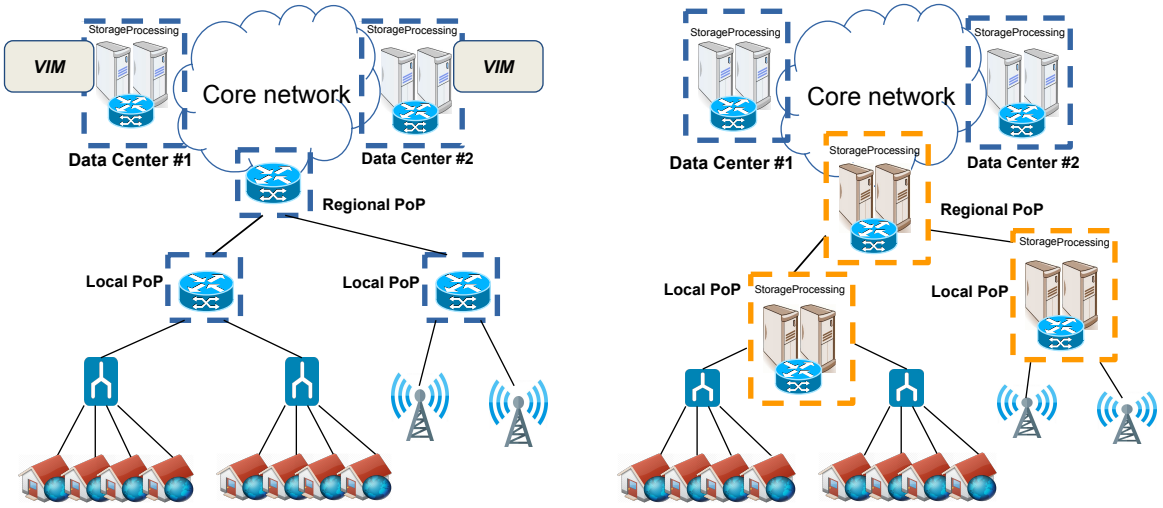
Depuis les quinze dernières années, le paradigme cloud computing a complètement changé la manière dont nous accédons et utilisons les technologies de l'information. En proposant un accès en libre-service aux ressources, services et applications disponibles en ligne, les utilisateurs ont le pouvoir de payer à la demande pour ce qui est réellement utilisé. En général, les services cloud sont réunis en trois catégories : SaaS qui fournit des logiciels prêts à être utilisés, PaaS qui fournit des plateformes de développement, et Iaas, qui fournit des ressources d'infrastructure telles que capacité de calcul, stockage sur disque ou connectivité réseau.

Pour proposer les ressources d'infrastructure, nous faisons l'utilisation des technologies de virtualisation, qui permettent de faire l'abstraction des objets physiques dans des objets virtuels qui sont proposés à l'utilisateur, dont l'exemple le plus connu sont les machines virtuelles (VMs) . Ces infrastructures virtualisées sont très importantes pour le cloud computing, mais ajoutent une couche supplémentaire de complexité d'un point de vue de la gestion. Pour centraliser le management ainsi que le monitoring, nous allons nous appuyer sur des gestionnaires d'infrastructure virtualisé (VIM en anglais pour virtual infrastructure managers), dont OpenStack est un des gestionnaires open-source le plus populaire et utilisé pour le management des infrastructures cloud.

OpenStack est une plateforme composée par 29 services au total dont les trois les plus importants correspondent aux trois types de virtualisation les plus connus : Nova pour le compute, Cinder pour le stockage et Neutron pour le réseau. Ces services vont utiliser un service commun d'authentification qui s'appelle Keystone. L'effort collectif d'OpenStack se compose de plus de 10 millions de lignes de code en Python dont Orange est un contributeur sur les aspects de la virtualisation réseau. Plus de détails du background de cette thèse sont présentes dans le Chapitre 2.

Des gestionnaires, comme OpenStack, sont utilisés aujourd'hui dans l'industrie pour gérer des infrastructures cloud tel que montré dans la Figure 1(a). Le schéma représente l'infrastructure d'accès d'un opérateur des télécommunications comme Orange. Il y a quelques grosses data centers depuis lesquels l'operateur va fournir des services cloud aux utilisateurs à l'échelle d'un pays voir d'un continent. Ensuite, il existe une série de points de présence (PoP) régionaux et locaux qui sert à interconnecter plusieurs réseaux d'accès des utilisateurs. La popularité du cloud computing a permis la naissance des services et

des applications comme l'internet des objets (IoT), le mobile edge computing (MEC), les functions réseau virtualisés (NFV) ou le cloud gaming pour lesquelles le modèle traditionnel du cloud n'a pas été conçu. Aujourd'hui l'opérateur doit faire face aux exigences de latence de l'ordre de la milliseconde, nous connaissons une explosion de trafic vers les data centers et de plus en plus les utilisateurs sont concernés pour que leur données soient stockés le plus proche d'eux.



(a) Infrastructure cloud traditionnelle

(b) Infrastructure cloud distribué

Figure 1 – Evolution de l'infrastructure cloud vers le DCI.

En conséquence, il s'avère nécessaire de déployer des ressources cloud plus proches de l'utilisateur tel que montré sur la Figure 1(b). Pour cela, nous pouvons profiter des PoPs déjà existants dans l'infrastructure pour en déployer des ressources de calcul et de stockage et créer une infrastructure composée par des centaines voir des milliers de micro data centers. Cette infrastructure cloud distribuée (DCI) pourra mieux répondre aux besoins et exigences des nouvelles applications.

Cette nouvelle infrastructure nous pose la question de comment effectuer son management. Pour cette action, il y a deux techniques qui peuvent être utilisées : un management centralisé similaire à celui du modèle traditionnel ou un management consistant à déployer une instance d'un gestionnaire d'infrastructure par micro data center dans une approche distribuée. La première approche consiste à manager la totalité de l'infrastructure comme un seul data center. Dans cette approche, le gestionnaire doit faire face aux problèmes connus des architectures centralisés (SPOF) ou encore les partitionnements réseau. La

deuxième solution peut mieux correspondre à la nature distribuée de l'infrastructure et diminuer les risques associés à une architecture centralisée.

Pour fournir ce management distribué, une grande question s'est pose : comment interconnecter et manager plusieurs gestionnaires d'infrastructure d'une manière distribuée nativement ? Le problème avec les gestionnaires déjà utilisés dans l'industrie comme OpenStack est qu'ils ont été conçus d'une manière centralisée et nativement il n'y a pas de communication entre gestionnaires différents. L'objectif serait d'étendre les gestionnaires pour permettre une collaboration native entre eux.

Cette manière de manager les gestionnaires doit être en mesure de garantir les propriétés suivantes :

— **Passage à l'échelle** : Nous devons être en capacité de déployer autant des gestionnaires que nécessaire pour l'infrastructure.
— **Résilience** : Tous les gestionnaires doivent pouvoir résister aux pannes et partitionnement réseaux.
— **Localité des données** : Les données crées par un gestionnaire doivent rester aussi locales que possible.
— **Abstraction et automatisation** : Permettre le déploiement des scénarios complexes d'une manière simple pour l'utilisateur.

Dans cette thèse nous nous concentrerons plus particulièrement sur comment manager et interconnecter des constructions réseau appartenant aux gestionnaires indépendants. Puisque le réseau est le moyen de communication pour tous les autres objets comme les machines virtuelles et le stockage, les réseaux virtuels jouent un rôle clé dans une infrastructure cloud distribuée. L'idée est d'être en mesure de proposer les mêmes ressources réseau que nous connaissons dans le modèle traditionnel du cloud mais d'une manière distribuée. Nous considérons particulièrement les quatre ressources suivantes :

— **Routage de niveau 3** : Être capable de router du trafic entre des réseaux virtuels appartenant aux gestionnaires indépendants.
— **Extension du niveau 2** : Être capable d'avoir un réseau virtuel étendu sur plusieurs gestionnaires indépendants.
— **Politiques de filtrage de trafic et de qualité du service** : Être capable de dicter de règles de filtrage et de qualité pour le trafic acheminé entre des gestionnaires indépendants.
— **Service Function Chainning** : Être en mesure de définir un chemin pour le trafic notamment pour traverser des fonctions réseau virtualisés.

Pour être capables de fournir ces ressources réseau nous arrivons à la première contribution de cette thèse qui est la définition des défis de management réseau pour une

DCI. Nous avons choisi de diviser ces défis en deux catégories, ceux correspondant à l'organisation de l'information réseau et ceux correspondant à l'implémentation technique de ce management. L'intégralité du contexte de cette thèse se trouve dans le Chapitre 3.

— **Partage de l'information réseau** : Le premier défi concerne le partage d'information réseau. Si nous prenons l'exemple d'un réseau virtuel partagé entre deux gestionnaires, il faudrait penser comment partager les adresses IP. Une première manière serait de communiquer entre les différents sites chaque fois qu'une IP est attribuée. Ça permettra d'éviter les conflits, mais avec la conséquence d'une surcharge de communication et de l'établissement d'une connaissance globale. Une deuxième manière pourrait être de diviser la totalité des adresses disponibles entre les deux sites dès la création de la ressource. Donc, nous devons analyser et comprendre les différentes ressources réseau concernées pour savoir quelles sont les informations à partager entre les gestionnaires. Cela nous permettra de proposer des stratégies de partage de l'information spécifiquement définies pour chaque type de ressource.
— **Identifier comment communiquer avec les sites distants** : Le deuxième défi concerne la portée d'une requête pour communiquer avec les sites distants. Comme nous avions évoqué pour la propriété de localité des données, il est nécessaire que les données créées par un gestionnaire restent aussi locales que possible. Si nous avons un réseau virtuel qui existe seulement dans un site 1, ce n'est pas nécessaire pour le site 2 de connaître les adresses IP et MAC de ce réseau distant. Si l'utilisateur demande la création d'un réseau L2 entre ces deux sites, alors c'est là qu'il faudra communiquer avec le site 2. Considérer la portée d'une requête est essentielle pour éviter une surcharge dans la quantité de trafic demandé pour la synchronisation d'une DCI. Le fait de contacter seulement les sites pertinents pour une requête permettra de diminuer la surcharge de communication.
— **Face aux déconnexions du réseau** : Le troisième défi concerne la disponibilité des gestionnaires en cas de panne ou déconnexion réseau. C'est-à-dire que chaque gestionnaire isolé doit être en mesure de fournir ces services cloud localement. Par ailleurs, les gestionnaires non affectés doivent pouvoir continuer à proposer les ressources inter-sites.
— **Interfaces standard automatisées et distribuées** : Ce défi d'implémentation technique concerne la définition des interfaces permettant de communiquer avec les utilisateurs, mais permettant la communication entre gestionnaires. Ces interfaces doivent être couplées entre elles pour permettre l'automatisation de la procédure de communication entre les gestionnaires pour fournir les ressources réseau. Alors, il est nécessaire de définir la liste des abstractions réseau qui seront proposées à

l'utilisateur.

— **Support et adaptation des technologies de mise en réseau** : Le dernier défi concerne le support et l'adaptation des technologies réseau. En complément de l'information réseau partagé dans le premier défi, il est nécessaire aussi de partager l'information concernant les mécanismes réseau permettant l'implémentation des ressources dans l'infrastructure virtualisée. Bien qu'il y a plusieurs protocoles et technologies réseau existant ils devront être adaptés au contexte DCI. Les gestionnaires doivent être en mesure d'échanger d'une manière automatisée ces informations, car l'utilisateur ne doit pas être au courant de cet échange d'information.

Pour être en capacité de fournir les différents ressources réseau et de répondre aux défis de management réseau dans une DCI, dans cette thèse nous nous sommes appuyés sur le paradigme des réseaux défini par logiciel (SDN pour Software-Defined Networking). En effet, ce paradigme peut nous donner des pistes pour la décentralisation des fonctionnalités réseau dans les gestionnaires. Le SDN propose de programmer le contrôle du réseau en faisant l'abstraction de l'infrastructure réseau pour des applications et des services. SDN s'appuie sur la division entre le plan contrôle, chargé de manager le réseau et les règles logiques d'acheminement de trafic et le plan de données, qui correspond à l'infrastructure réseau qui applique les règles dictées par le plan de contrôle. Le plan de contrôle se trouve dans une entité appelée contrôleur SDN. Cette entité peut se trouver d'une manière distribue et pour en communiquer, les contrôleurs vont utiliser une interface de communication appelée East-West.

Pour notre étude de l'état de l'art, nous avons choisi des propositions SDN décentralisés académiques et open-source en les divisant en deux catégories : des contrôleurs orientés vers le réseau traditionnel et les contrôleurs orientés vers le cloud. Nous avons aussi étudié des solutions de distribution de management au sein des gestionnaires. L'étude de ces différents propositions est disponible dans les Chapitres 5, 6 et 7. Tout d'abord nous avons effectué une classification des propositions, selon leur modèle d'architecture, selon de caractéristiques de leur implémentation comme la stratégie de coordination, les protocoles de communication ou la base de données utilisée. Et, selon des propriétés d'interopérabilité et de maturité comme les protocoles réseau supportés, le type de réseau envisagé et le niveau de maturité. Comme nous avons pris OpenStack comme gestionnaire d'infrastructure pour tous les potentiels développements, nous avons pris en compte la compatibilité des solutions avec ce gestionnaire. Les détails de ces différents propriétés se trouvent dans le Chapitre 4 de cette thèse. Avec l'étude de ces caractéristiques nous avons effectué l'analyse de comment chaque contrôleur répondait aux défis réseau dans un contexte DCI. Malheureusement, aucune solution était en mesure de répondre à la

Propositions	Organisation de l'information réseau			Implémentation des ressources inter-site	
	Granularité de l'info	Porté de l'info	Disponibilité de l'info	Interfaces automatisés	Technologies réseau
Solutions orientés vers le réseau					
DISCO	✓	✓	✓	✓	✗
D-SDN	✓	~	?	✗	✗
ElastiCon	~	?	?	~	✗
FlowBroker	✗	✗	✓	✗	✗
HyperFlow	✓	~	~	~	✗
Kandoo	✗	✗	✓	✗	✗
Orion	✗	✗	?	✗	✗
Solutions orientés vers le Cloud					
DragonFlow	~	?	?	~	~
ODL (Fed)	✓	✓	~	✓	✓
Onix	~	?	?	~	~
ONOS	~	?	?	~	✓
Tungsten	✗	✗	?	✗	✓
Solutions d'autre type					
Kubernetes Federation	✗	~	✓	✗	✗
Kubrnetes Istio Multi-Cluster Service Mesh	✓	✓	~	✗	✗
OpenStack P2P external proxy-agents	~	~	✓	✓	✗
OpenStack Tricircle	~	?	?	~	~

[1] ✓Défi répondu.
[2] ~ Défi partiellement répondu.
[3] ✗Défi pas répondu.
[4] ? Non défini.

Table 1 – Résume des solutions analysés.

totalité des défis DCI comme résumé dans le Chapitre 8. Malgré l'impossibilité de trouver un candidat parfait, deux de ces contrôleurs répondaient à la plus grande quantité de défis : DISCO et OpenDayLight.

Nous avons décidé d'utiliser les principes de ces contrôleurs dans DIMINET : un module distribué pour le management des ressources réseau inter-site. L'explication de DIMINET et des différents tests que nous avons effectué se trouvent dans les chapitres 9 et 10.

DIMINET propose une architecture complètement distribuée dans laquelle une instance du module est déployée à côté du service réseau du gestionnaire d'infrastructure tel que montré dans la Figure 2. Comme les modules sont indépendants entre eux, des nouvelles instances de DIMINET peuvent joindre l'infrastructure sans affecter le comportement des autres. Nous avons aussi profité de l'idée de la communication East-West des contrôleurs SDN pour permettre une communication horizontale entre les modules. De cette manière les modules vont contacter les modules distants seulement quand cela soit nécessaire pour la création d'une ressource inter-site.

Nous avons implémenté DIMINET comme un module déployable à côté de Neutron, le service réseau d'OpenStack. De plus, en suivant le même pattern de Neutron et d'autres services d'OpenStack, DIMINET a été codée en Python et les interfaces North et East-West ont été construites comme des interfaces API REST. Un des éléments les plus importants de DIMINET est son cœur logique, chargé du management des ressources inter-site

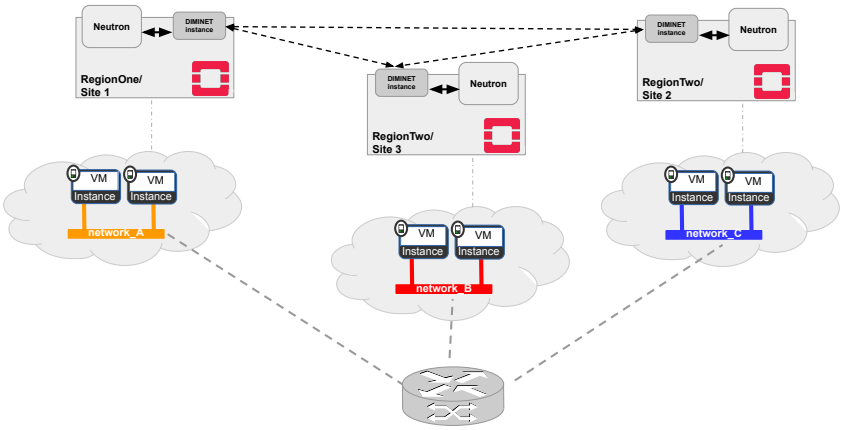


Figure 2 – Architecture de DIMINET.

y compris l'utilisation de l'interface horizontal pour requêter les modules distants quand cela s'avère nécessaire. Pour adresser le défi de granularité de l'information, nous avons conçu des stratégies de partage d'information par type de ressource qui sont implémentés dans le cœur logique. Pour cette première proposition de DIMINET nous avons conçu des stratégies pour les ressources L3 routage et L2 extension que nous allons expliquer par la suite.

La ressource de routage L3, permet de router du trafic entre des sous-réseaux appartenant à des sites différents. Par conception, les sous-réseaux ne peuvent pas se chevaucher entre eux. C'est-à-dire, l'espace d'adressage d'un sous-réseau doit être unique par rapport aux autres sous-réseaux qui sont routé. Cette validation doit être effectué lorsque la ressource inter-site est crée entre plusieurs modules de DIMINET. En prennant compte de ce besoin, notre stratégie de partage de l'information se déroule comme montré dans la Figure 3 et comme expliqué par la suite :

1. L'utilisateur demande la création d'une ressource de type routage L3 entre le réseau A du site 1 et le réseau B du site 2.
2. Le module du site 1 devient le master de la ressource.
3. Le master fait des requêtes pour avoir les informations des réseaux A et B.
4. Le master vérifie les préfixes réseau des réseaux A et B.
5. Le master envoie une requête aux module distants en demandant la création d'une ressource inter-site de type routage L3.

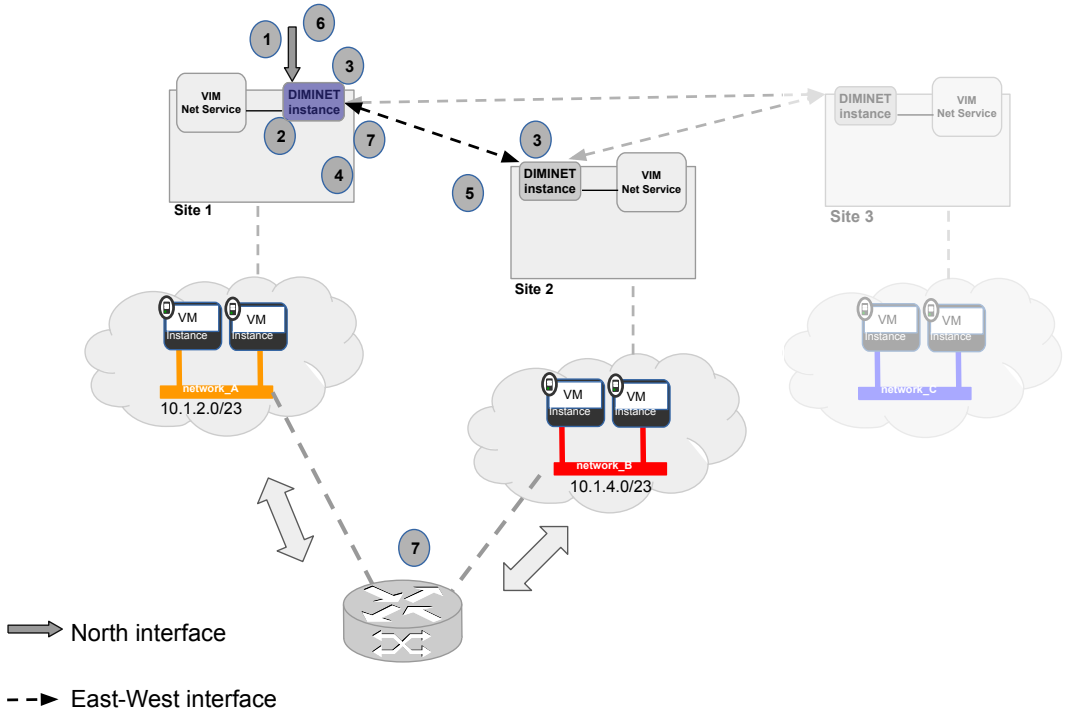


Figure 3 – Ressource de routage L3 : Stratégie de partage de l'information.

6. Une fois que tous les modules concernés ont créée la ressource, le master répond à l'utilisateur que la ressource a été créée.

7. Les modules effectuent l'échange d'information des mécanismes de mise en réseau.

Pour maintenir la cohérence de l'information nous avons décidé d'utiliser un leader par opération, mais pour éviter la complexité d'utiliser un algorithme de consensus dans un contexte DCI, qui est une problematique en tant que telle, nous avons décidé d'utiliser un master choisi d'une manière statique et c'est le module qui reçoit la requête qui devient le master de cette ressource. Pour des modifications postérieures de la ressource, c'est que le module master qui pourra faire des modifications, c'est-à-dire ajouter ou retirer des sites de la ressource inter-site L3.

Ensuite, nous avons la ressource d'extension L2 qui donne la possibilité de connecter au même réseau virtuel des VMS appartenant à des sites différents. Pour appartenir au même réseau virtuel, les éléments attachés doivent avoir le même préfixe réseau et doivent avoir des adresses MAC et IP uniques. Donc l'allocation des adresses MAC et IP doivent être coordonnées entre les sites pertinents pour une requête. Dans le cas de Neutron les adresses MAC sont créées selon un pattern du gestionnaire ce qui est facile à changer entre les déploiements. Alors, nous devons effectuer la coordination des adresses

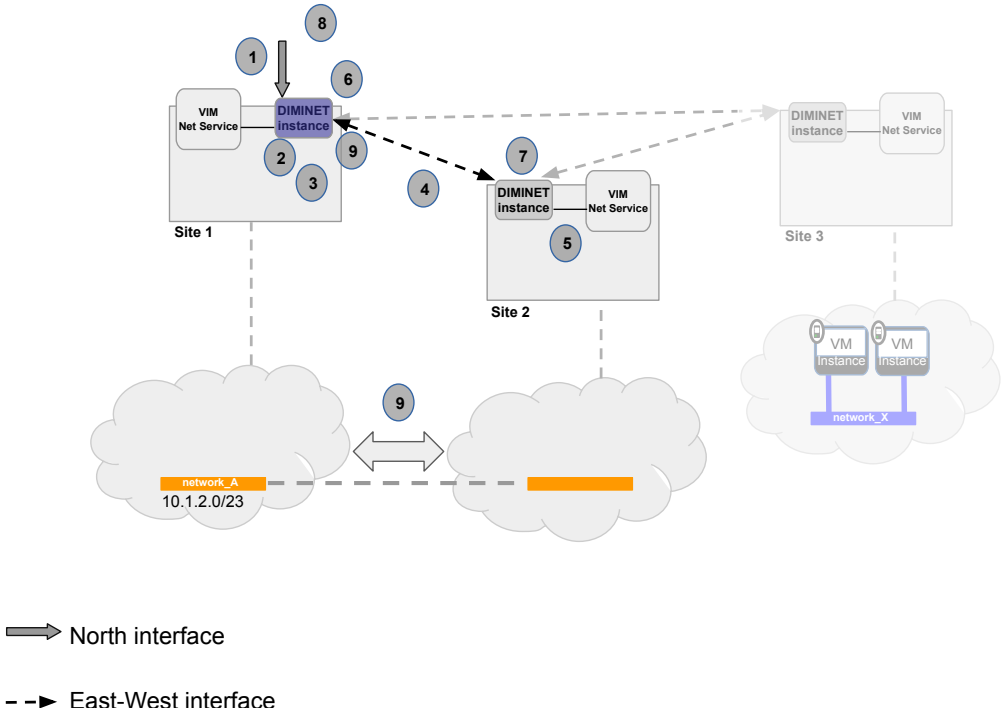


Figure 4 – Ressource d'extension L2 : Stratégie de partage de l'information.

IP. Dans DIMINET, nous avons pris l'approche d'étendre un réseau existant sur un site vers d'autres sites comme montré dans la Figure 4 et expliqué par la suite :

1. L'utilisateur demande la création d'une ressource de type extension L2 pour que le réseau A du site 1 soit étendu vers le site 2.
2. Le module du site 1 devient le master de la ressource.
3. Le master effectue la division des adresses IP disponibles dans le réseau et fait les requêtes à son gestionnaire local pour effectuer les changements nécessaires.
4. Le master envoie une requête aux module distants en demandant la création d'une ressource inter-site de type extension l2.
5. Chaque module distant demandé effectue des requêtes auprès de leurs gestionnaires locaux pour créer les réseaux virtuels avec les paramètres fournis par le master.
6. Le master fait la mise à jour de sa liste de réseaux virtuels composant la ressource inter-site de type l2 extension grâce à la réponse des modules distants.
7. Le master envoie la liste des réseaux composant la ressources à tous les modules concernés.
8. Une fois que tous les modules concernés ont fait la mise à jour de la ressource, le master répond à l'utilisateur que la ressource a été créée.

9. Les modules effectuent l'échange d'information des mécanismes de mise en réseau.

Concernant la dernière étape des deux stratégies à propos de l'échange d'information des mécanismes de mise en réseau, comme DIMINET a été proposé pour être implémenté à côté de neutron, nous n'avons pas implémenté l'échange d'information pour la connectivité dans le plan de données en utilisant l'interface East-West. Nous avons plutôt profité des technologies déjà existantes sur Neutron notamment le service plugin des interconnexions qui utilise la technologie BGPVPN. En attribuant des identifiants connus comme route target plusieurs sites peuvent échanger des informations des routes à l'aide du protocole BGP, une fois l'information de routage échangée le trafic du plan de données est acheminé en utilisant des protocoles d'encapsulation. DIMINET automatise la création de ces ressources dans tous les sites nécessaires pour une requête.

Cette thèse voulait contribuer avec des nouvelles approches pour la gestion distribuée de la connectivité dans des infrastructures cloud edge. Pour cela nous avons étudié et analysé la décentralisation du data center vers une infrastructure cloud distribuée composée par une grande quantité de micro data centers et nous avons défini une liste des défis à résoudre pour être en mesure de fournir les ressources réseau existants dans le modèle cloud traditionnel. Nous nous sommes inspirés sur les principes des contrôleurs SDN logiquement et physiquement distribués pour proposer DIMINET, un module capable de manager des ressources réseau inter-site pour OpenStack. Malgré plusieurs problèmes logiciels rencontrés pendant les tests, les résultats des expériences sont prometteuses.

Table of Contents

Abstract v

Résumé en Français vii

Résumé Etendu ix

List of acronyms xxiii

List of figures xxviii

List of tables xxix

Introduction 1

Research Questions and Contributions 2

Thesis Organization 3

I Background 5

2 Background on Cloud Infrastructures 7

2.1 Cloud computing 7

2.1.1 Cloud Computing Characteristics 8

2.1.2 Cloud Deployment Types 8

2.1.3 Cloud Service Types 9

2.1.4 Managing cloud virtualized infrastructures 11

2.1.4.1 OpenStack 11

2.1.4.2 Kubernetes 15

2.2 Software-Defined Networking 16

2.2.1 SDN-based Cloud Networking 18

2.2.1.1 OpenStack Neutron: Networking as a Service 18

2.2.1.2 Kubernetes Networking 19

2.3 Summary 20

3 Distributed Cloud Infrastructures: the context **21**
3.1 Distributed Cloud Infrastructures 21
3.1.1 DCI Distributed Management Characteristics 23
3.2 Why Revising Software Stacks Is Needed For DCIs 24
3.3 Networking Management Challenges in DCIs 25
3.3.1 Network Information's Challenges 27
3.3.2 Technical challenges 30
3.4 Summary 31

II State Of The Art **33**

4 Multi-instance solutions for DCI architectures: Properties **35**
4.1 SDN Solutions for DCIs 35
4.2 Architecture 35
4.3 Leader-Based Operations 37
4.4 Database Management System 38
4.5 SDN Interoperability and Maturity 39
4.6 Related Works in SDN 41
4.7 Summary 42

5 Network-oriented SDN controllers **44**
5.1 DISCO 44
5.2 D-SDN 46
5.3 Elasticon 47
5.4 Flowbroker 49
5.5 HyperFlow 51
5.6 Kandoo 53
5.7 ORION 54
5.8 Summary 56

6 Cloud-oriented SDN controllers **58**
6.1 DragonFlow 58
6.2 OpenDayLight 60
6.3 Onix 62
6.4 ONOS 64
6.5 Tungsten 66
6.6 Summary 67

7 Other decentralized propositions **69**
7.1 OpenStack P2P External Proxy-Agents . . . 69
7.2 OpenStack Tricircle . . . 71
7.3 Kubernetes Federation . . . 73
7.4 Kubernetes Istio Multi-Cluster Service Mesh . . . 75
7.5 Summary . . . 77

8 Multi-instance learned lessons and perspectives **79**
8.1 Lessons Learned on Multi-instance Cloud Controllers . . . 79
8.2 Summary . . . 82

III DCI networking: Going the distributed way **83**

9 Distributing connectivity management with DIMINET **85**
9.1 Leveraging Retained SDN Principles . . . 85
9.2 DIMINET's Architecture Overview . . . 86
9.3 DIMINET's Logic Core . . . 88
9.3.1 Resources Sharding Characteristics . . . 88
9.3.1.1 L3 Routing Resource . . . 89
9.3.1.2 L2 Extension Resource . . . 90
9.3.2 Data Model . . . 92
9.4 Communication Interfaces . . . 95
9.5 Data Plane Traffic Exchange . . . 96
9.6 Summary . . . 99

10 Evaluation of DIMINET **102**
10.1 DIMINET with OpenStack: PoC Validation . . . 102
10.1.1 L3 Routing Resources . . . 102
10.1.2 L2 Extension Resources . . . 104
10.2 Grid'5000: Testbed and Setup . . . 105
10.3 Evaluation of Inter-Site Resources Deployment . . . 106
10.3.1 Layer 3 Routing Resource . . . 107
10.3.2 Layer 2 Extension Resource . . . 109
10.4 Evaluation of Resiliency . . . 112
10.5 Evaluation of DIMINET Scalability . . . 113
10.6 Summary . . . 117

IV Conclusions & Perspectives **119**

Conclusions and Perspectives **121**

List of publications **135**

V Appendices **137**

Bibliography **152**

List of acronyms

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ARP	Address Resolution Protocol
AS	Autonomous System
AWS	Amazon Web Services
BGP	Border Gateway Protocol
BGPVPN	Border Gateway Protocol based Virtual Private Network
CNF	Cloud-native Network Function
CIDR	Classless Inter Domain Routing
CIS	Container Infrastructure Service
CISM	Container Infrastructure Service Manager
CNI	Container Network Interface
CORD	Central Office Re-architected as a Datacenter
CRD	Custom Resource Definition
CRDT	Conflic-free Replicated Data Type
CRI	Container Runtime Interface
CRUD	Create Read Update Delete
D-SDN	Decentralized SDN
DB	DataBase
DC	Datacenter
DBaaS	Data Base as a Service
DCI	Distributed Cloud Infrastructure
DISCO	Distributed SDN Control Plane
DHCP	Dynamic Host IP Protocol
DHT	Distributed Hash Table
DIMINET	Distributed Module for Inter-site Networking

DNS	Domain Name System
eBPF	extended Berkeley Packet Filter
Elasticon	Elastic Controller
ER	Entity Relationship
ETSI	European Telecommunications Standard Institute
EVPN	Ethernet Virtual Private Network
GBP	Group Based Policy
GCP	Google Cloud Platform
GUI	Graphical User Interface
HA	High Availability
IaaS	Infrastructure as a Service
IBGP	Internal Border Gateway Protocol
ICMP	Internet Control Message Protocol
IPAM	IP Address Management
IoT	Internet of Things
IP	Internet Protocol
IPVPN	Internet Protocol Virtual Private Network
IT	Information Technology
K8S	Kubernetes
LISP	Local ID Separation Protocol
L2	Layer 2
L2GW	Layer 2 Gateway
L3	Layer 3
MAC	Media Access Control
MANO	Management and Orchestration
MC	Main Controller
MEC	Mobile Edge Computing
ML2	Modular Layer 2
MP-BGP	Multi Protocol Border Gateway Protocol
MPLS	Multi Protocol Label Switching

NAT Network Address Translation

NETCONF Network Configuration Protocol

NetVirt Network Virtualization Services

NFV Network Function Virtualization

NIB Network Information Base

NIC Network Interface Card

ODL OpenDayLight

ONOS Open Networking Operating System

OS Operating System

OSI Open System Interconnection

OVS Open Virtual Switch

OVSDB Open Virtual Switch DataBase

P2P Peer-to-Peer

PaaS Platform as a Service

PoC Proof-of-Concept

PoP Point of Presence

QoS Quality of Service

RAN Radio Access Network

RDBMS Relational Database Managing System

REST Representational State Transfer

RMS Resource Management System

RR Route Reflector

SaaS Software as a Service

SAL Service Adaptation Layer

SC Secondary Controller

SDN Software Defined Network

SNMP Simple Network Management Protocol

SONA Simplified Overlay Network Architecture

SFC Service Function Chaining

SPF Shortest Path First

SPOF Single Point Of Failure

SQL Structured Query Language

TC Linux Traffic Control

TCP Transmission Control Protocol

Telco Telecommunication Operator

TRL Technological Readiness Level

URL Uniform Resource Locator

UUID Universally Unique Identifier

vCPU virtual CPU

VIM Virtual Infrastructure Manager

VLAN Virtual Local Area Network

VM Virtual Machine

VN Virtual Network

VNF Virtualized Network Function

vNIC virtual Network Interface Card

VPN Virtual Private Network

VPNaaS Virtual Private Network as a Service

vRAM virtual RAM

vRouter virtual Router

VTN Virtual Tenant Network

VXLAN Virtual Extensible Local Area Network

WAN Wide Area Network

WIM WAN Infrastructure Manager

WSGI Web Server Gateway Interface

XMPP Extensible Messaging and Presence Protocol

List of Figures

1 Evolution de l'infrastructure cloud vers le DCI. x
2 Architecture de DIMINET. xv
3 Ressource de routage L3 : Stratégie de partage de l'information. xvi
4 Ressource d'extension L2 : Stratégie de partage de l'information. xvii

2.1 Cloud Services types: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), & Software as a Service (SaaS) 10
2.2 OpenStack landscape 12
2.3 OpenStack overview. 13
2.4 Kubernetes cluster components. 15
2.5 SDN general architecture 17
2.6 Neutron deployment components. 19

3.1 Evolution of a cloud infrastructure towards a DCI. 22
3.2 DCI management techniques. 23
3.3 Network Information Challenge 2: Layer 2 extension request 28
3.4 Network Information Challenge 3: Operate in isolated mode. 29
3.5 Technical Challenge 1: automatized interfaces 30

4.1 SDN topologies 36

5.1 DISCO SDN controller architecture 45
5.2 D-SDN SDN controller architecture 47
5.3 Elasticon SDN controller architecture 48
5.4 FlowBroker SDN controller architecture 50
5.5 HyperFlow SDN controller architecture 51
5.6 Kandoo SDN controller architecture 53
5.7 Orion SDN controller architecture 55

6.1 DragonFlow SDN controller architecture 59
6.2 OpenDayLight Federation NetVirt SDN controller architecture 61
6.3 Onix SDN controller architecture 63
6.4 ONOS SDN controller architecture 65

6.5 Tungsten SDN controller architecture 67

7.1 P2P external proxy-agents architecture 70
7.2 OpenStack Tricircle architecture 72
7.3 Kubernetes Federation architecture 74
7.4 Istio Multi-Cluster Service Mesh architecture 76

9.1 DIMINET overview. 87
9.2 DIMINET internal design. 88
9.3 DIMINET L3 Routing Resource. 90
9.4 DIMINET L2 Extension Resource. 92
9.5 DIMINET data model. 93
9.6 Neutron-to-Neutron Interconnection Plug-in. 97
9.7 Neutron BPG-VPN Plug-in. 99

10.1 DIMINET & OpenStack L3 traffic capture. 103
10.2 DIMINET & OpenStack L2 traffic capture. 104
10.3 DIMINET testbed setup 106
10.4 L3 routing resource creation time. 107
10.5 L3 routing resources creation time comparison (seconds). 108
10.6 L2 extensions resource creation time 110
10.7 L2 extension resources creation time comparison (seconds). 111
10.8 DIMINET Resiliency test 112
10.9 DIMINET scalability tests. 114
10.10 Scalability time comparison tests. 115
10.11 Cassandra DataBase (DB) DHT example. 126
10.12 Cilium Multi-Cluster architecture 128
10.13 General data model based on DIMINET. 131

A.1 DIMINET L3 routing Resource creation sequence diagram. 139
A.2 DIMINET L2 extension Resource creation sequence diagram. 140

C.1 DIMINET GUIs. 149

D.1 BPG Route Reflectors. 151
D.2 DIMINET & OpenStack with BPG Route Reflectors. 152

List of Tables

1 Résume des solutions analysés. xiv

3.1 DCI Challenges summary 26

5.1 Classification of surveyed network-oriented SDN solutions. 56
5.2 Challenges summary of network-oriented solutions. 57

6.1 Classification of surveyed cloud-oriented solutions. 68
6.2 Challenges summary of cloud-oriented solutions. 68

7.1 Classification of surveyed solutions. 78
7.2 Challenges summary of other solutions. 78

8.1 Classification of surveyed solutions. 80
8.2 Summary of the analyzed solutions. 81

9.1 DIMINET CRUD Operations 95

INTRODUCTION

The cloud computing paradigm has redefined the way Information Technology (IT) resources are managed and delivered. By proposing a series of on-demand services such as PaaS, SaaS, and IaaS, users can avoid the, sometimes, prohibitive prices of *on premises* hardware and software. Traditionally, a few large data centers are deployed to deliver these services to users located in a large geographical zone. While this model is well-fitted to provide traditional cloud services, it cannot guarantee the operational requirements of new services such as Internet of Things (IoT), Mobile Edge Computing (MEC), Radio Access Network (RAN), Network Function Virtualization (NFV) (*i.e.,* delay constraints of the order of a millisecond, geo-distributed fault tolerance, or the respect of legal requirements for data). In consequence, cloud infrastructures should be adapted to take into account these new services.

Since the backbone of Telecommunication Operators (Telcos) is already composed of a series of regional and local Points of Presences (PoPs) used to interconnect access networks, they can be extended with additional servers to expand the cloud even closer to the user. The proposals to manage such a Distributed Cloud Infrastructure (DCI) are based either on a centralized approach or a federation of independent Virtual Infrastructure Managers (VIMs). 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 the PoPs. The latter consists of deploying one VIM on each DCI site and federate them through a brokering approach to give the illusion of a single coherent system promoted by ETSI NFV Management and Orchestration (MANO) framework [1]. This federated approach could better fit the requirements and constraints of a DCI since each site is independent and can continue to operate locally. However, the downside relates to the fact that resource management systems do 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 to manage a single deployment.

This limitation has motivated the work presented in this manuscript. More precisely, we address the management of DCIs in a collaborative way among independent instances of resource managers.

To allow a seamless utilization of such architecture, a key point in this context is the

networking connectivity among independent sites. Indeed, virtual network resources are critical to allow other cloud resources to exchange traffic among them and to provide their services to users. Since networking operations affect these resources (*i.e.,* side effects), they need to be carefully managed in DCIs. Technologies such as Software Defined Network (SDN), which propose a decoupling among control and data plane [2], can be leveraged to provide such networking operations among VIMs [3].

Research Questions and Contributions

Following the aforementioned scope, this thesis focuses on the distributed management of DCIs with a particular focus on virtual network connectivity. The principal research questions are the following:

- **How to manage a distributed cloud infrastructure composed of several independents instances/sites?**. A simple, scalable, and efficient management solution is needed to administrate and use such a DCI.
- **How to provide connectivity for inter-site virtual networking resources?**. The same virtual networking abstractions already provided at the data center should be proposed at the DCI level.

This thesis aims to explore and propose distributed approaches inspired by SDN solutions to manage DCIs connectivity while assuring the scalability, resiliency, locality awareness, and usability of the entire system.

The **first contribution** of this thesis is the identification of the challenges present in DCI scenarios when trying to distribute services management, and more precisely the connectivity one. The analysis of the challenges allows us to find the major fields where further research and propositions can be done. This work led to a publication on the national *Conférence d'informatique en Parallélisme, Architecture et Système 2019*.

Our **second contribution** is an extensive review of literature on distributed SDN controllers. This review focuses on principle design aspects as well as the analysis of whether the proposition can coop with the DCI challenges or not. The review finishes by highlighting the benefits of some of the surveyed SDN controllers and how they can contribute to a collaborative model for multi-site management. This contribution has been published at the *IEEE Communications Surveys & Tutorials journal 2021*.

The **third contribution** concerns the design and implementation of a Distributed Module for Inter-site Networking (DIMINET) leveraging a fully distributed architecture. By leveraging the principles of the retained SDN controllers, the proposed architectural model can provide on-demand inter-site networking resources among independent sites of

a DCI. DIMINET's proposition has been published at the *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*. Moreover, the Proof-of-Concept (PoC) of this contribution has been presented at the *2020 Open Infrastructure Summit*.

Thesis Organization

This thesis is composed of 11 chapters gathered in four parts. The first part consists of two chapters, it defines the background and context of this doctoral work and introduces the *first contribution*. The second part is a state-of-the-art analysis composed of five chapters detailing the *second contribution*. The third part is composed of two chapters and introduces our *third contribution*. The fourth part is composed of one chapter concluding this thesis and providing some insights about future research directions, specially by providing a first step towards a generalization of our works. We underline that each part starts with a brief preamble providing an outline of the chapters. The same is done in each chapter, starting with a brief preamble, which gives an overview of the content, and ending with a summary, which highlights the major elements to keep in mind.

Part I

Background

These two chapters allow the reader to understand the context of this doctoral work.
— *Chapter 2 covers the context of cloud infrastructures.*
— *Chapter 3 explains the evolution of cloud infrastructures towards DCI.*

Chapter 2

Background on Cloud Infrastructures

This chapter introduces the background of this doctoral work. It starts by describing the characteristics of the cloud computing paradigm, with a special focus on network virtualization. It also introduces the concepts of SDN giving insight on its functionality and how it can be used for cloud computing networking management.

2.1 Cloud computing

Cloud computing [4, 5] can be easily accepted to be one of the most revolutionary IT paradigm shift in the last 15 years. By proposing an on-demand access model to virtual resources and services located at data center facilities, users pay for what they consume, and they no longer need to buy all the necessary hardware and software. While it is possible to find several cloud computing definitions provided in the literature, the most used definition is given by the U.S. National Institute of Standard and Technology [6]:

Definition of cloud computing by the NIST

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

The cloud computing model hides all the provisioning process of resources and services to the user, facilitating its use, and allowing the emergence of new applications and services.

2.1.1 Cloud Computing Characteristics

Because of its nature, cloud computing targets a large market with several use cases. Generally speaking, the characteristics of the cloud computing model, also gathered by the NIST, are the following:

— **On-demand self-service**: At any particular moment, a user can provision resources using automated management systems without the need for human interaction with the cloud provider.
— **Broad network access**: The resources are delivered to the users over the Internet and can be accessed by several heterogeneous means.
— **Resource pooling**: Resources are pooled together to serve multiple users at the same time (*a.k.a.* multi-tenancy) [7]). These resources can be dynamically assigned and reassigned accordingly to users' demands. As a result of the pooled resources, users are unaware of the location of the exact resources. However, users may specify the location using high-level abstractions (*e.g.,* zones, regions, countries).
— **Rapid elasticity**: Resources can be provisioned and released dynamically according to the user demand, scaling up and down at any needed time.
— **Measured service**: Resources utilization by users can be measured, controlled, and reported to provide transparency for users about their cloud consumption.

2.1.2 Cloud Deployment Types

Depending on the owner and the targeted users, clouds can be classified generally into four categories [8]: public clouds, private clouds, hybrid clouds, and community clouds.

Public Cloud

Public or external clouds are the most common form of cloud computing, in which cloud providers offer services to the general public in a pay-as-you-go manner. Some key benefits of public clouds are on-demand scalability of resources, data availability, uninterrupted services, and cost reduction. Top cloud providers such as Amazon, Google, and Microsoft provide a globally deployed infrastructure to provide cloud services to users over the entire world [9, 10, 11].

Besides the benefits of public clouds, we still find some issues related to data security as well as concerning the data governance. Since users do not have a fine data granularity control, they are unaware of where exactly data is stored or how it is backed up.

Private Cloud

Private clouds are maintained by an organization to offer cloud services exclusively to its members. They can be managed by the organization using on-premises equipment or by a third-party. In the former, the private cloud is hosted within the organization infrastructure (*a.k.a. on-site private cloud*), meaning that the organization has full control over the hardware and software components. In the latter, the private cloud is outsourced to a hosting company (*a.k.a. outsourced private cloud*), which takes care of the management of the infrastructure. This kind of cloud is well suited for organizations wanting to maintain control over their data location such as governments or large private groups.

Community Cloud

Community clouds can be seen as private clouds maintained by two or more organizations having similar concerns in terms of mission, security, requirements, or policy. As in private clouds, they can be deployed using on-premise equipment in the organizations' infrastructure, or by using a third-party provider. When compared to private clouds, community clouds can be considered to be cheaper in terms of the cost since multiple participants share the cloud.

Hybrid Cloud

A hybrid cloud is a composition of two or more cloud infrastructures (public, private, or community) that remain independent but are bounded together through standardized or proprietary technology that enables data and application portability. Hybrid clouds can be useful for organizations looking to maintain critical data in their private cloud while consenting less critical information to be stored in a public cloud.

2.1.3 Cloud Service Types

Cloud computing allows a broad range of services to be proposed to users. Services can be essentially classified into three big categories [12] as represented in Figure 2.1.

Software as a Service (SaaS) : Services proposing already packaged and ready to use software. Most of these are web-based applications usually accessible by using web browsers. Some examples are Dropbox, Office 365, Google Mail, or Facebook.

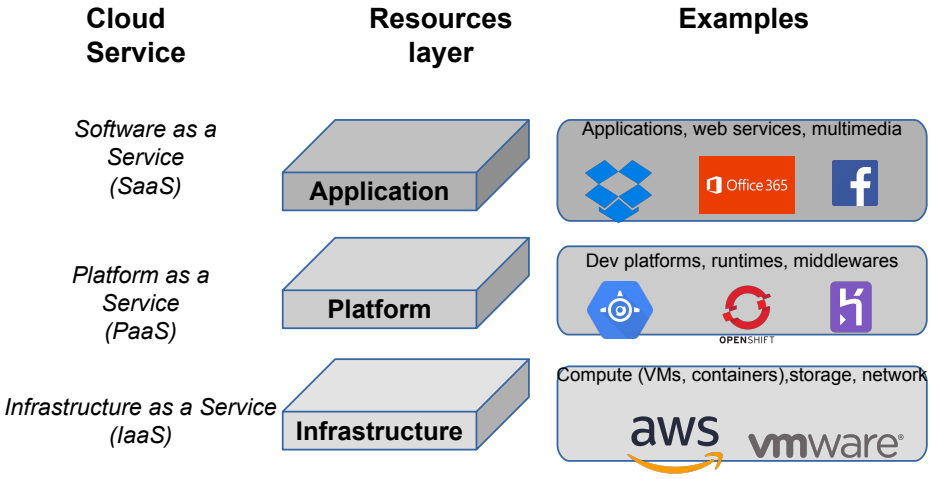


Figure 2.1 – Cloud Services types: IaaS, PaaS, & SaaS

Platform as a Service (PaaS) : Services proposing high-level software frameworks that users can use to build and deploy applications using the tools and programming languages supported by the framework. Some examples are Google App Engine, OpenShift, or Heroku which are PaaS systems for developing and managing web applications.

Infrastructure as a Service (IaaS) : Services proposing infrastructure resources such as compute capacity, disk storage, or network connectivity for the user to deploy her software on the provisioned resources. Some examples are Amazon Web Services (AWS), or Google Cloud Platform (GCP). Because IaaS provides the infrastructure needed for other services, PaaS and SaaS can be instantiated atop of IaaS, making it a key feature to furnish cloud services.

To correctly propose IaaS services, cloud providers usually leverage virtualization technologies [13]. Virtualization abstracts physical resources such as processors, memory, disk, or networks, into logical (*a.k.a.* virtual) resources which are then proposed to the users. At the IaaS layer, the most important types of virtualization are the following [14]:

— *Server virtualization*: Virtualization technique allowing one physical server to appear as multiple servers. Historically, server virtualization has been done using hypervisors, and more recently using lighter container-based technologies [15]. In hypervisor-based virtualization (*e.g.,* using KVM or VMWare), the physical server is composed of multiple Virtual Machines (VMs), each one running its own Operating System (OS) and possibly having dedicated resources on its own (*e.g.,* memory or disk) that are presented in the form of virtual appliances such as virtual

CPU (vCPU) or virtual RAM (vRAM). In container-based virtualization, the physical host kernel is shared among processes (*a.k.a.* containers) running on dedicated resources such as CPU, memory, or disk.

— *Storage virtualization*: Virtualization technique allowing a physical storage unit to be abstracted into several logical storage units such as disk drives or file systems.

— *Network virtualization*: Virtualization technique allowing a physical network to be shared among isolated logical networks. Similarly to server virtualization, Network Interface Cards (NICs) are also presented in the way of a virtual resource virtual Network Interface Card (vNIC). Being the communication medium for server and storage capabilities, network virtualization is critical to offer today's cloud services.

2.1.4 Managing cloud virtualized infrastructures

Virtualized infrastructures are today essential for all cloud services. However, their use implies adding an extra complexity from a management point of view.

To minimize this complexity from the user's perspective and also to centralize the control, management, and monitoring tasks, it is necessary to rely upon robust platforms gathering these operations. Such tasks can be provided by a VIM, an entity responsible for controlling and managing the compute, storage, and network resources according with the European Telecommunications Standard Institute (ETSI) standards [16]. VIMs allow to easily orchestrate the allocation and release of virtual resources, and the association of virtual with physical resources, including resources' optimization. Some examples of VIMs are OpenStack [17], OpenNebula [18], and Eucalyptus [19].

In more recent times with the advent of container-based virtualisation, the ETSI has proposed a new service that provides a runtime environment for one or more container technologies *a.k.a.* Container Infrastructure Service (CIS) [20]. Some examples of CIS are Kubernetes Kubernetes (K8S) [21], and Apache Mesos [22].

In the following, we introduce two concrete examples of these Resource Management System (RMS): OpenStack as VIM and Kubernetes as CIS.

2.1.4.1 OpenStack

OpenStack is a VIM for IaaS platforms considered the de facto open-source platform for operating cloud infrastructures. It allows controlling large pools of computing, storage, and networking resources throughout a deployment, normally a data center. OpenStack is composed of several services allowing the users to customize their deployments depending on the use case. All services are managed and provisioned through Representational

State Transfer (REST) Application Programming Interfaces (APIs) with common authentication mechanisms and some shared services. Figure 2.2 shows how the OpenStack landscape is composed of standard IaaS functionalities and additional components to provide orchestration, fault management, or lifecycle management.

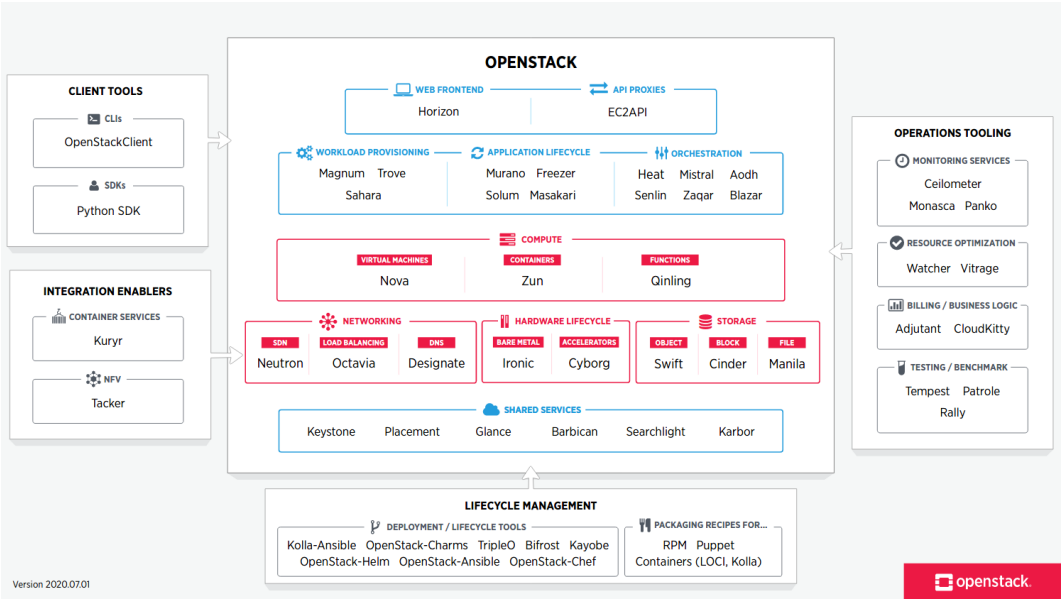


Figure 2.2 – OpenStack landscape ([17])

Among the main functionalities and corresponding to the main virtualization types presented in 2.1.3 for IaaS, OpenStack proposes Nova [23], Cinder [24], and Neutron [25] as main services to manage respectively server, storage, and network virtualization. Besides these three services, it is noteworthy to mention Keystone [26], the OpenStack identity service used to provide authentication and service discovery. Figure 2.3 shows an overall architecture of OpenStack with these four services which are also explained in the following.

Nova

Nova is in charge of provisioning on-demand compute instances (*a.k.a.* virtual servers or VMs). While its main functionality is to provision VMs, Nova can also do the provisioning of bare-metal servers, and limited support for containers. It is one of the original services provided in the first release of the OpenStack platform, and its functionality could be compared with Amazon EC2 service. Internally, Nova is composed of several components as shown in Figure 2.3(a) and as explained as follows:

— *Nova API*: Component that receives HTTP requests concerning the computing

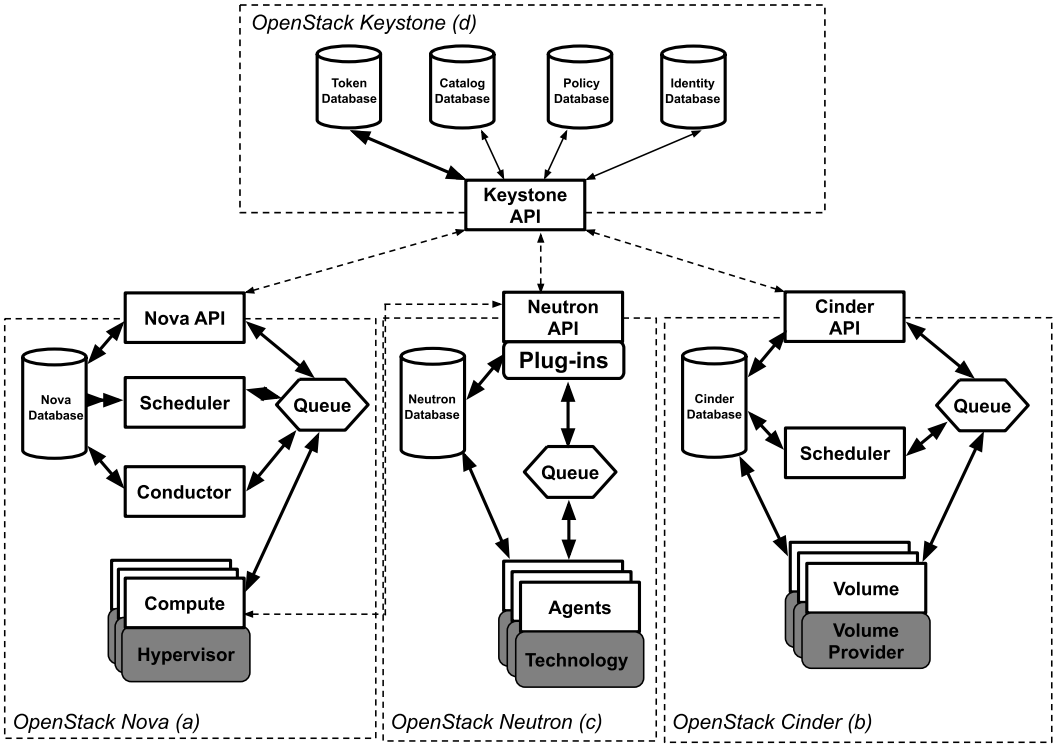


Figure 2.3 – OpenStack overview.

resources.

— *Nova Compute*: Component that manages communication with the hypervisor technologies allowing the creation of VMs. It is placed in the same machine as the hypervisor which is called *host*.
— *Nova Scheduler*: Component that decides which host should be used to instantiate a new VM request.
— *Nova Conductor*: Nova Conductor handles requests that need coordination such as build or resize a VM.
— *Database*: Relational DB used to store Nova objects.
— *Messaging queue*: Used to route information between the internal components using Remote Procedure Calls (RPC).

Cinder

Cinder is in charge of providing volumes (*a.k.a.* block storage) that Compute instances can consume. It offers volumes as basic resources for block storage. Cinder is similar to Amazon Elastic Block Storage (EBS) service. Figure 2.3(b) shows Cinder internal

composition:

- *Cinder API*: Component that receives HTTP requests concerning the storage resources.
- *Cinder Volume*: Manage block storage devices.
- *Cinder Scheduler*: Component that decides which Cinder Volume should be used to instantiate a new storage request.
- *Database*: Relational DB used to store Cinder objects.
- *Messaging queue*: Used to route information between the internal components using Remote Procedure Calls (RPC).

Neutron

Neutron provides on-demand, scalable, and technology-agnostic networking resources, it is generally used with Nova to provide VMs with networking capabilities. Neutron is a modular and pluggable platform allowing multiple networking technologies to coexist inside the same OpenStack deployment. The reference Neutron architecture as shown in Figure 2.3(c) is composed of the following elements:

- *Neutron 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.
- *Network plug-ins*: Plug-ins process the networking requests accepted by Neutron API.
- *Network Agents*: Agents implement the actual networking functionality closely associated with specific technologies and the corresponding Plug-ins.
- *Database*: Relational DB to store Neutron objects.
- *Messaging queue*: Used to route information between the internal components.

Keystone

Keystone provides API client authentication, service discovery, and multi-tenant authorization by implementing the OpenStack Identity API. Keystone is organized as a set of internal services exposed by an endpoint as shown in Figure 2.3(d):

- Keystone API: Keystone API exposes the different Keystone resources.
- Token service: Validates and manages tokens used for authenticating requests once a user's credentials have already been verified.
- Catalog service: Provides an endpoint registry used for endpoint discovery.
- Policy service: Define the access policies for resources of other OpenStack services.

— Identity service: Provides authorization credential validation and data about users.

2.1.4.2 Kubernetes

Although OpenStack remains the de facto open-source 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 [27]. By promising low-overhead virtualization and improved performance, containers have become the new center of interest of DevOps [28], and consequently, a couple of new frameworks in charge of managing the lifecycle of container-based applications have been developed [29]. Released by Google in 2016, Kubernetes 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 complexity of the distributed infrastructure.

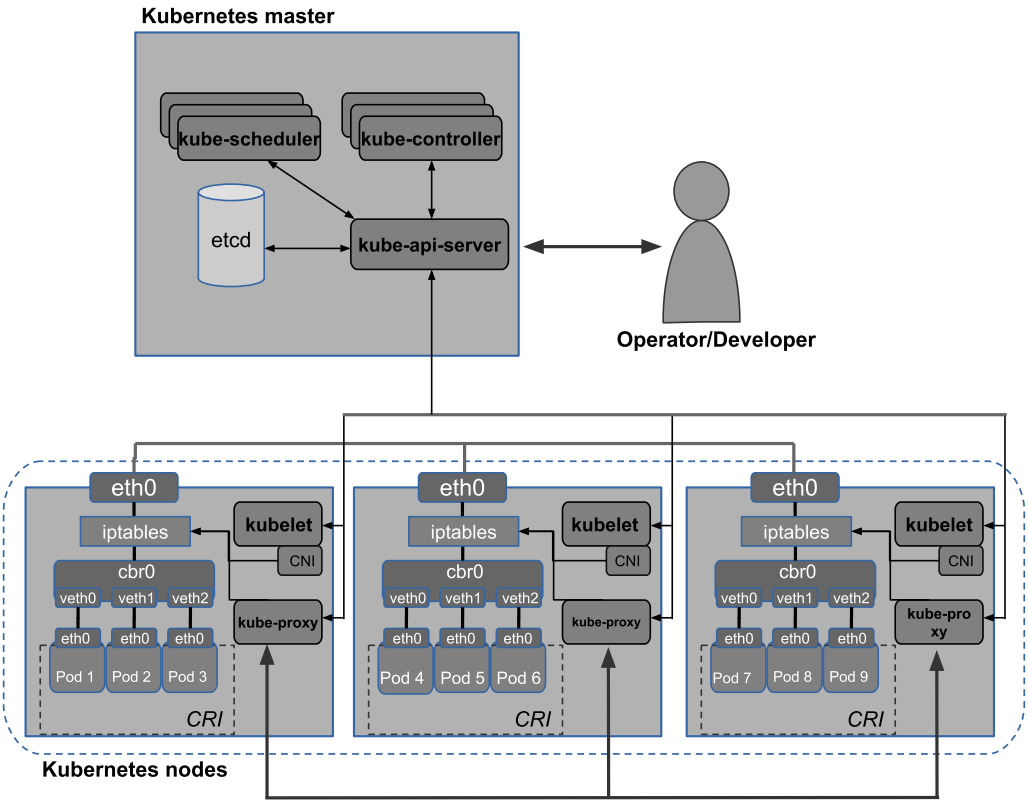


Figure 2.4 – Kubernetes cluster components.

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 2.4 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, and finally, a container runtime such as Docker [30], Linux Containers [31], or any other implementation of Kubernetes Container Runtime Interface (CRI) [32] 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 expected state of the cluster using control loops, and etcd, a key-value store used as Kubernetes backend.

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.).

In addition to basic operation on Pods (creation, deployment, etc.), K8S proposes several abstractions (objects or resources in the K8S terminology) to hide the distribution of the architecture. 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 group of Pods with a Domain Name System (DNS) name and a virtual Internet Protocol (IP). Finally, a *Namespace* enables DevOps to isolate Pods within a cluster. Additional objects have been built on top of these abstractions to offer supplementary functionalities. For instance, ReplicaSet enables DevOps to define a specific number of replicas for a Pod and let the K8S controller to maintain this number. Thanks to the modularity of K8S, more objects can be exposed by the API (for an up-to-date list please refer to [33]). This philosophy of using predefined abstractions is a major change concerning the OpenStack solution where DevOps should deal with a large number of infrastructure details.

2.2 Software-Defined Networking

In the last decade, the management of virtualized cloud infrastructures has become a hot topic in research. Indeed, every kind of virtualization has been studied to simplify control and management. Among the virtualization types, network virtualization can benefit from one such emerging concept, the SDN paradigm. The Software-Defined Networking paradigm offers the opportunity to program the control of the network and abstract the underlying infrastructure for applications and network services [34]. 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 pieces of equipments 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 [35] and even earlier with the concept of programmable and active networks [36, 37], the work achieved in 2008 around OpenFlow [2] is considered as the first appearance of Software Defined Networks in modern literature [38]. 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 APIs called southbound interfaces. In addition to OpenFlow, the most popular southbound APIs are Cisco's OpFlex ones [39].

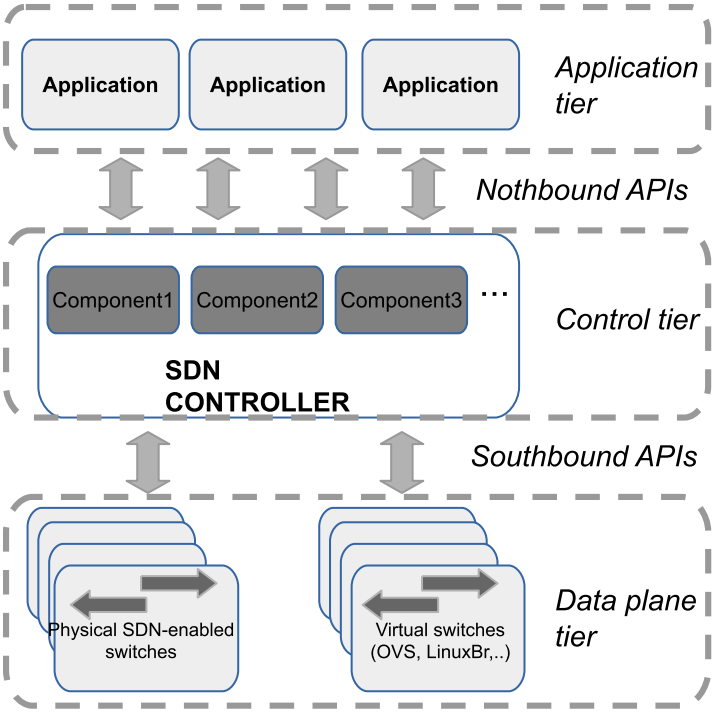


Figure 2.5 – SDN general architecture

Controllers also expose a northbound API, which allows 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 2.5. 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 [40].

2.2.1 SDN-based Cloud Networking

The SDN paradigm has been successfully 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 [41, 42, 43], including the well-known Google B4 controller [44], and commonly referred nowadays as Software Defined WAN [45]. In the cloud computing area, SDN has also been applied in an approach commonly referred to as SDN-based cloud computing [46]. To illustrate how the network resources are managed in a cloud computing infrastructure using SDN technologies, in the following, we complete our previous presentation of OpenStack and Kubernetes by discussing how they abstract the network.

2.2.1.1 OpenStack Neutron: Networking as a Service

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 Layer 2 (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*, *floating IP*, among others [47, 48, 49]. A more complete list of OpenStack networking objects can be found in [50].

As aforementioned in 2.1.4.1, Neutron uses a series of Plug-ins to manage its abstractions. The Core objects are managed by a Core Plug-in (*i.e.,* IP address management and L2 connectivity for a Sub-Network), being the Modular Layer 2 (ML2) Plug-in the default choice, while the extension resources are managed each one by a Service Plug-in (*i.e.,* Routing among Sub-Networks). Depending on the Plug-ins used, the deployment will also have a series of Agents receiving messages and instructions from Plug-ins to do the actual implementation of the networking capabilities. Figure 2.6 shows the relation among all the different components of Neutron.

Agents configure the mechanisms allowing the networking constructions to communicate among them (*e.g.,* a tunnel between two OpenvSwitches to communicate VMs belonging to the same Subnetwork) or with the Internet (*i.e.,* Network Address Transla-

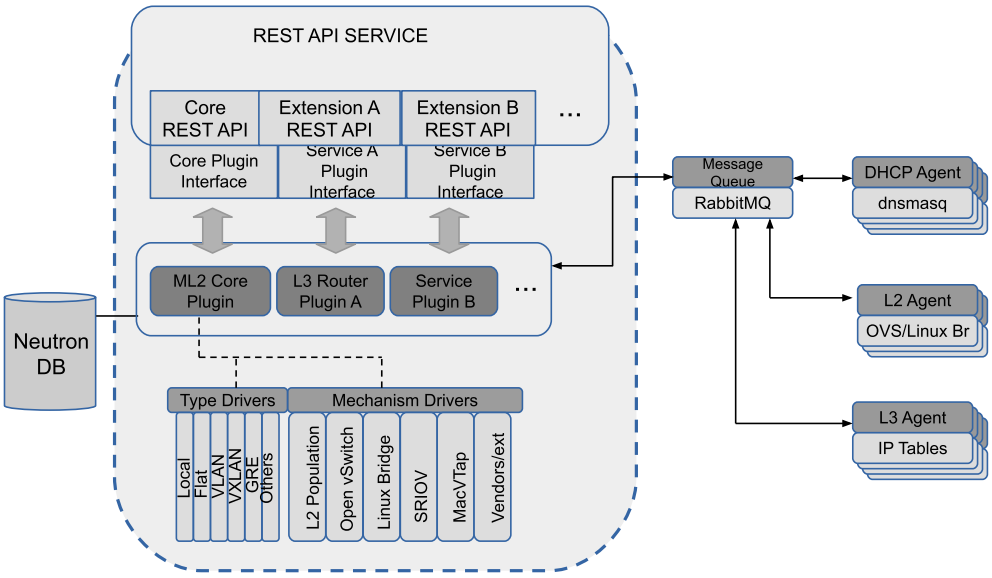


Figure 2.6 – Neutron deployment components.

tion (NAT)) capabilities normally implemented by a Router to route traffic to the outside). The most common agents are L2 (Layer 2 functions), Dynamic Host IP Protocol (DHCP), and L3 (Routing functions) agents.

2.2.1.2 Kubernetes Networking

The way Kubernetes manages network resources differs from the OpenStack solution. K8S does not propose means to virtualize and control multiple network segments but rather exposes services that relieve DevOps from the burden of managing network low-level aspects (*e.g.,* IP assignments, traffic forwarding, load balancing, etc.).

From the network point of view, K8S has four types of communications: (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 to leverage modularity, K8S supports its networking functionality through the Container Network Interface (CNI) [51], 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. For the IP and routes configurations, a second plug-in, called the IP Address Management (IPAM) plug-in is used. Several CNI as well as IPAM plug-in implementations have

been proposed to abstract the network within a K8S cluster [52, 53, 54, 55, 56, 57, 58, 59]. 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, the implementation of 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 [60]). 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-Balancer* or *Ingress controller* objects).

2.3 Summary

The cloud computing paradigm has changed the way online services and resources are accessed and provided. Thanks to the different deployment and service types, cloud computing can be used in a large range of scenarios, providing both, service providers and users, with a flexible and efficient tool. In the case of IaaS, the use of several types of virtualization technologies has provided a cheap and easy-to-use way to access virtualized resources. As stated in Section 2.2.1, cloud networking virtualization is enhanced by SDN technologies, allowing centralized control of the virtual networking objects with high-level abstractions. The next chapter describes the evolution of cloud computing infrastructures into distributed cloud computing infrastructures.

Chapter 3

Distributed Cloud Infrastructures: From the data center to the edge

This chapter presents the challenges addressed in this thesis: DCIs and associated networking management expectations. Moreover, it discusses the networking management challenges in DCIs.

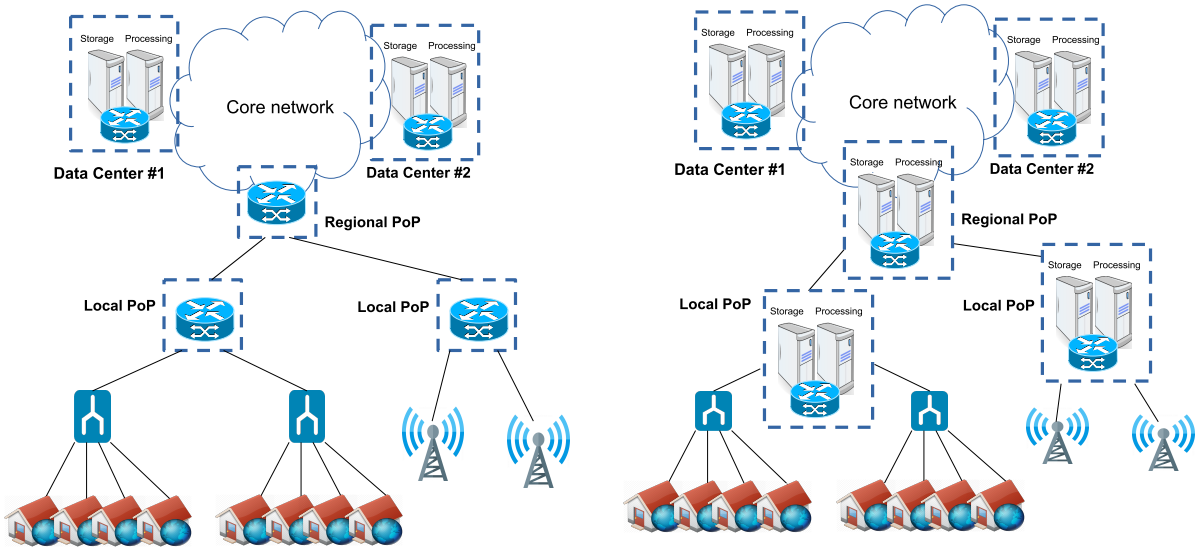
3.1 Distributed Cloud Infrastructures

As we asserted in Chapter 2, cloud computing has enabled the appearance of different kinds of applications and services for a considerable range of use cases. However, in the last years, new trends such as the IoT, the MEC, the cloud RAN, or the NFV present new and heavy operational requirements (*i.e.,* delay constraints of the order of a millisecond to ten milliseconds, geo-distributed fault tolerance, or the respect of legal requirements for data management), for which the traditional cloud infrastructure has not been designed [61].

As a consequence, VIMs or CISs such as OpenStack or Kubernetes need to be deployed closer to the end users. In this way, the traditional cloud infrastructure composed of a few large data centers will evolve to a massively DCI. This DCI will be formed not only by the data centers but also by a high number of locations across the Telco's network to provide cloud services. While DCIs can be used for providing all cloud services, we focus on IaaS functionalities in this manuscript. In the following, the term *resource managers* will be employed in a generic way to talk indistinctly of software stacks providing resource management such as VIMs (OpenStack) or CISs (Kubernetes).

Within the Telco's infrastructure, such a DCI can use the regional and local PoP. While traditionally used to interconnect different access networks, PoPs can be extended with additional computing and storage capacity to be used as mini data centers for delivering IaaS capabilities. Figure 3.1 shows the evolution of traditional cloud infrastructures

towards DCIs composed by several locations proposing IaaS capacities.



(a) Traditional cloud infrastructure

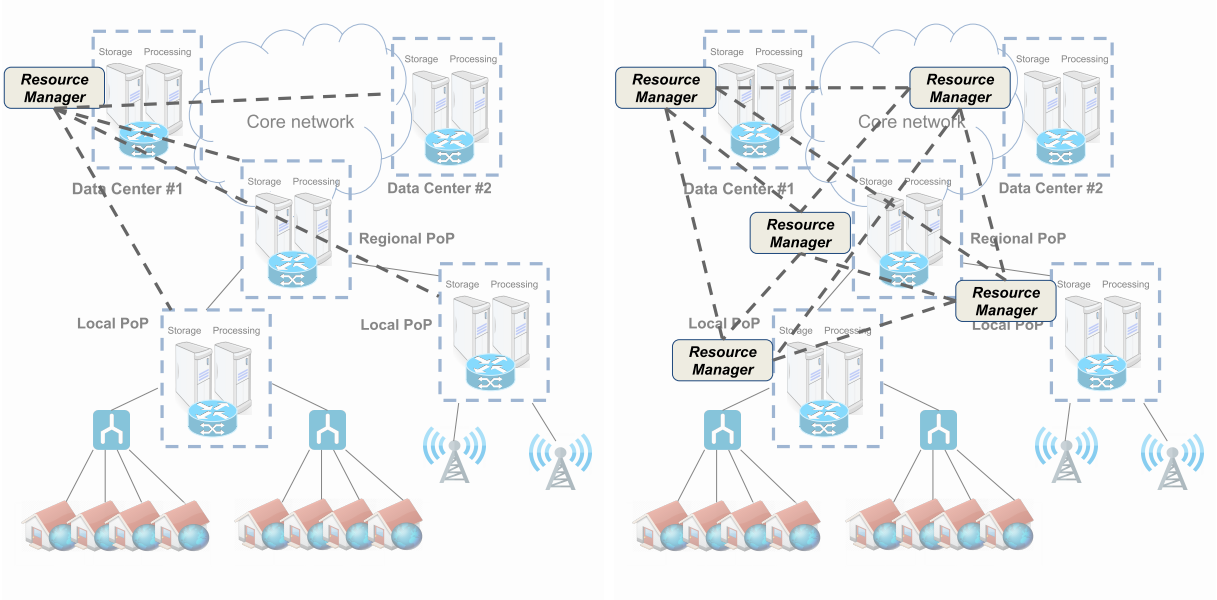
(b) Distributed cloud infrastructure

Figure 3.1 – Evolution of a cloud infrastructure towards a DCI.

From a management perspective, resources management can leverage two techniques in DCIs as depicted in Figure 3.2 [62]. The first one consists of administering the DCI following a single "centralized" data center manner, with WAN between the resource manager at the data center and the different PoPs. The second approach consists of deploying one instance of the resource manager at each DCI site and federating them providing the illusion of a single unified system as promoted by ETSI NFV MANO framework [1].

While a centralized approach could be easily extended to the entire DCI, it still presents the well-known problems of having a bottleneck and a Single Point Of Failure (SPOF) at the data center's resource manager. This approach reveals a significant risk in network disconnection: Since distant sites do not have complete autonomy, they will be unavailable to process users' requests in case of network disconnection. Therefore, isolated sites may be inoperative, and the worst-case scenario will envision the entire failure of the system to process requests if the resource manager's location is unreachable.

On the other hand, the distributed approach may exceed the centralized one since each site is independent and can continue to operate locally with its resource manager. Moreover, the distributed nature of the architecture can alleviate problems such as bottleneck and SPOF. Because resource managers such as OpenStack or Kubernetes have not been conceived natively to peer with other instances, several projects have been investigating



(a) Centralized DCI management

(b) Distributed DCI management

Figure 3.2 – DCI management techniques.

how it might be possible to deliver inter-site management [63, 64, 65]. These projects aim at exposing multiple instances of the software stack as a single entity. Unfortunately, these solutions have been designed around centralized architectures and face notable limitations (*e.g.,* scalability, network partitions, etc.). In other words, resource managers have not been designed to peer with other instances to establish inter-site collaboration but rather in a pretty stand-alone way to manage a single deployment. However, as defended in this thesis, it is possible to extend resource managers with additional code pieces to offer the same functionality but over multiple instances. In the following, we analyze the characteristics of such a distributed management for DCIs.

3.1.1 DCI Distributed Management Characteristics

From a general management point of view, several studies have investigated how DCI management can be delivered either through a bottom-up or a top-down approach [66]. A bottom-up collaboration aims at revising low-level resource managers' mechanisms to make them collaborative using, for instance, a shared DB between all the resource managers [61, 66, 67]. A top-down design implements the collaboration by interacting only with the resource managers' API using a set of components acting as proxies for the resource managers [68, 69, 63].

Unfortunately, there are some major limitations concerning both approaches. In the former, the use of shared DBs helps developers implement such inter-site operations. But in addition to being intrusive and complex, reusing the code to use another DB is not satisfying to deal with network partitioning. In the latter, while the high-level utilization of APIs can lessen the burden for DevOps, some inter-site operations cannot accurately work without revising existing resource managers code base [70]. Therefore, to overcome the limitations of current solutions, we consider that a DCI resource manager should be capable of guaranteeing the following properties:

Scalability Since new sites can be added to the architecture at any time, DCIs should not be restricted by design to a certain amount of resource managers.

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 Resource managers should have autonomy for local domain management. It implies that locally created data should remain local as much as possible and only shared with other instances if needed, thus avoiding the maintenance of a global knowledge.

Abstraction and automation The configuration and instantiation of inter-site resources 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.

3.2 Why Revising Software Stacks Is Needed For DCIs

Building such kind of DCI resource manager from scratch would be too expensive technically speaking, and does not take into account frameworks already used in production environments. For this reason, open-source initiatives such as OpenStack with a massive community behind it may be used to be extended and to allow collaboration among its instances.

Among the required features that resource managers should offer in a DCI, the capacity to manage and interconnect virtual networking constructions belonging to several independent sites is critical since other services' resources, such as the compute or storage

ones, communicates atop of the virtualized network. From an IaaS networking management point of view, resource managers leveraging a distributed management should be capable of providing the same resources already proposed when using a single cloud infrastructure. We consider, in particular, the following inter-site networking resources [71]:

— *Routing function*: being able to route traffic between a Virtual Network (VN) A on site 1 and a VN B on site 2.

— *Layer 2 network extension*: being able to have a Layer 2 VN that spans several sites. It is the ability to plug into the same VN, VMs instantiated at locations belonging to different resource managers.

— *Traffic filtering, policy, and QoS*: being able to enforce traffic policies and Quality of Service (QoS) rules for traffic between several sites.

— *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 should be able to deploy a service chain spanning several sites, having service VMs placed at different resource managers.

3.3 Networking Management Challenges in DCIs

As discussed in Section 3.1, the control of the network elements of a DCI infrastructure should be performed in a distributed fashion (*i.e.,* with multiple resource managers that collaborate to deliver network capabilities across several sites) by leveraging as much as possible existing resource managers. In the following, we analyze OpenStack's multi-site challenges, and we also provide this analysis for Kubernetes.

Multi-instance OpenStack challenges

In an OpenStack-based DCI (*i.e.,* one OpenStack instance by PoP), inter-site networking resources management will rely on the networking module, Neutron. Unfortunately, the software architecture of Neutron does not allow collaborations between multiple instances. For instance, the Neutron DB belongs to a single Neutron entity. As a consequence, the resources created at one site can only be managed by that Neutron. A distant Neutron cannot have knowledge nor access the objects present in a DB of another instance. Because information of resources is not shared among Neutrons, the notion of a virtual network spanning different resource managers does not exist today in the Neutron DB. Further, operations like identifiers assignation, IP and Media Access Control (MAC) addresses generation and allocation, DHCP services, and security group management are

also handled internally at each Neutron instance. Such kind of constraints makes it impossible to manage Neutron resources in a distributed way.

Multi-instance Kubernetes Challenges

Like OpenStack, Kubernetes has been designed in a stand-alone manner: it exposes a single cluster's resources. Hence, being able to execute container-based applications across multiple sites raises different questions in the Kubernetes ecosystem [72]. From the network viewpoint, a K8S Multi-Cluster architecture should deliver means to provide the communications detailed in Section 2.2.1.2. More precisely, since Container-to-Container communication is limited at Pod's scope, solutions for the other three communication cases are required.

Independent from the software stack, decentralizing networking management brings forth new challenges and questions. We can divide them into two categories: the ones related to the organization of network information and the ones related to the technical implementation. Table 3.1 summarizes these challenges. The "keywords" column is used to introduce the name of the challenges used in the rest of the document. For each of the challenges, we also expose examples in the context of IaaS networking resources.

Challenge	Key words	Summary
Network information's challenges		
Sharing networking information	**information granularity**	Propose good information sharding strategies
Identifying how to communicate with remote sites	**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
Technical challenges		
Standard automatized and distributed interfaces	**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 3.1 – DCI Challenges summary

3.3.1 Network Information's Challenges

Giving the illusion that multiple resource managers behave like a unique one, 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 considering several dimensions as discussed in the following.

Sharing networking information: (information granularity)

The first dimension to consider is the organization of the information related to the network elements. For example, the provisioning of an IP network between two resource managers will require sharing information related to the IPs allocated on each site. A first approach may consist of sharing information between the two resource managers each time an IP is assigned to one resource. This will prevent possible conflicts, but with an additional 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 sites at the network's creation time. Thus, each site has a subset of the IPs and can allocate them without communicating with other controllers. This prevents facing IP conflicts even in the case of network partitioning issues without exchanging information each time a new IP is allocated to a particular resource.

Therefore, understanding the different data structures manipulated will enable the definition of different information sharding strategies between multiple resource managers and identify each of their pros and cons. 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 public fixed IPs may not need to be shared with remote sites. Consequently, depending on the inter-site service, the shared objects' information's granularity 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 coordination mechanisms to provide some data consistency.

Identifying how to communicate with remote sites: (information scope)

The second dimension to consider is related to the scope of a request. Networking information should stay as local as possible to privilege data locality. For instance, network information like MAC/IP addresses of ports and identifiers of a network related to one site does not need to be shared with the other sites that compose the DCI. Similarly,

information associated with a Layer 2 network shared between two resource managers as depicted in Figure 3.3 does not need to be shared with the third resource manager. The extension of this Layer 2 network could be done later, only when it will be relevant to extend this network to *resource manager 3*.

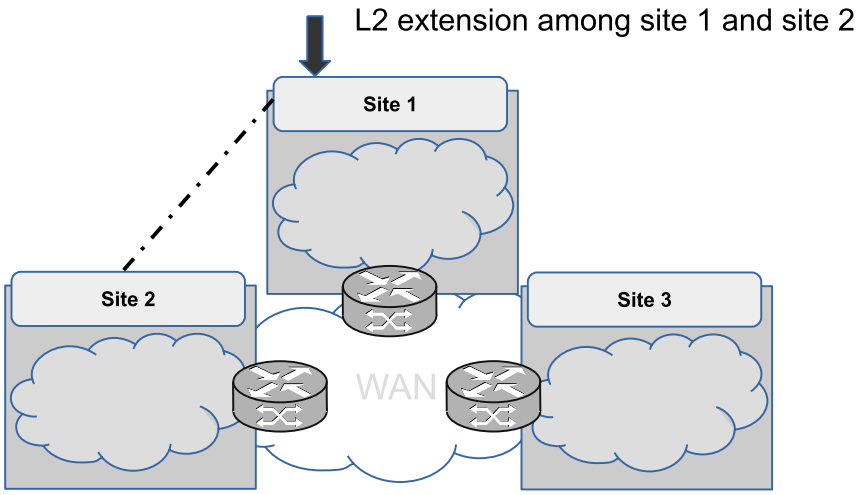


Figure 3.3 – Network Information Challenge 2: Layer 2 extension request

Considering each request's scope is critical since sharing information across all resource managers should be avoided due to heavy synchronization and communication needs. In other words, contacting only the relevant sites for a request will mitigate the network communication overhead, enhancing the limitations regarding scalability and network disconnections.

The information-sharing protocol needs to be fast and reliable to avoid performance penalties that could affect multi-site networking resources' deployment.

Facing network disconnections (information availability)

Each resource manager should propose networking resources management even in the case of network partitioning issues.

Two situations must be considered: (i) the inter-site network resource (for instance, a Layer 2 network) deployed before the network disconnection and (ii) the provisioning of a new inter-site network resource. In the first case, the isolation of a site (for instance *resource manager 2* in Figure 3.4) should not impact the inter-site network elements. Hence, *Resource manager 2* should still be able to assign IPs to VMs using the "local" part of the inter-site Layer 2 network. Meanwhile, *Resource manager 1* and *Resource*

manager 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 resource manager cannot reach other sites due to the network partitioning issue, it is impossible to get information mandatory to finalize the provisioning process. The first way to address such an issue is merely revoking 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 tools to recover from a network disconnection.

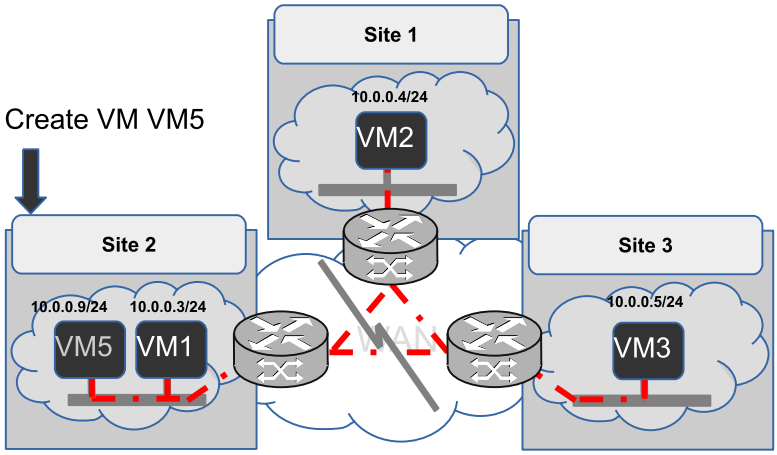


Figure 3.4 – Network Information Challenge 3: Operate in isolated mode.

Depending on how the resource has been created during the partitioning, the re-synchronization procedure's complexity may vary.

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

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

3.3.2 Technical challenges regarding inter-site networking services

Once the challenges related to network information are analysed, it is necessary to consider the dimension of the mechanisms allowing a deployment of distributed networking services. Technical challenges are related to the technical issues presented when trying to implement DCI networking services.

Standard automatized and distributed interfaces (automatized interfaces)

A first challenge is related to the definition of the vertical and horizontal interfaces to allow the provisioning of inter-site resources from the end-users viewpoint and to make the communication/collaboration between the different resource managers possible as depicted in Figure 3.5.

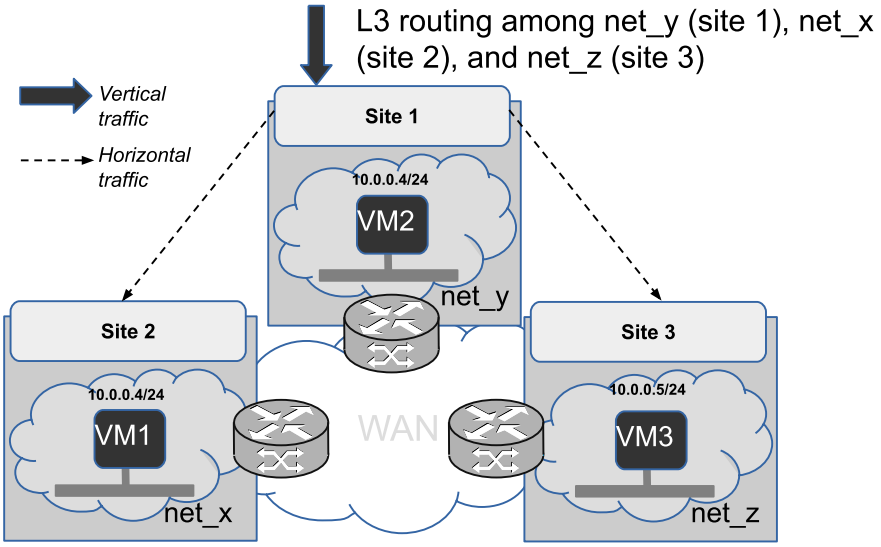


Figure 3.5 – Technical Challenge 1: automatized interfaces

It means that the interface which faces the user (user-side or North-side) and the interface which meets other networking services (resource manager-side or East-West-side) have to be smoothly bridged among them. This integration needs to be done to provide the necessary user abstraction and the automation of the resource manager communication process. Consequently, this necessitates the specification and development of well-defined North- and East-West-bound interfaces presenting to the user a list of abstractions of multi-site networking operations. Thus, designing efficient interfaces for both North-bound and East-West-bound communication is another problem to address in the case of inter-

site connectivity management tools.

Support and adaptation of networking technologies (networking technologies)

In addition to the initial networking information exchanges among resource managers to provide inter-site connectivity (MAC/IP addresses, network identifiers, etc.), identifying the implementation mechanism will be needed. Although there are many existing networking protocols to implement, they will need adaptation in the DCI case. Because the networking mechanisms' configuration needs to be known by all the participant sites 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 essential because the user should not be aware of how these networking constructions are configured at the low-level implementation. Since a DCI could scale up to hundreds of sites, manual networking stitch techniques like [47][48] will be 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 [73], OpenvSwitch switches[74] or Linux bridges[75]

— At the underlay level, which implies the ability to talk or communicate with some physical equipment like the Edge site gateway. Since not all physical equipment is OpenFlow-enabled, the possibility of using other protocols may be an advantage when 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 3.3.1, the mechanisms used for the implementation need to reconfigure themselves to re-establish the inter-site traffic forwarding.

3.4 Summary

The popularity and characteristics of cloud computing have allowed the emergence of new trends for which it was not designed. The advent of IoT, MEC, or NFV poses new operational constraints that can be assured by deploying IaaS functionalities closer to the user in Telco's PoPs. By doing this, cloud computing is evolving towards a DCI that can be managed using a distributed approach. Unfortunately, current software stacks have been conceived following a stand-alone manner to operate a single location and do not propose native collaboration tools.

Among the expected features of resource managers for DCIs, the ability to manage and interconnect virtual networking constructions belonging to independent resource managers is critical to provide inter-site cloud services.

This chapter has introduced networking management challenges in DCIs, assembling them into two main categories: network information and technological implementation. Addressing the information granularity, information scope, and information availability challenges will allow to define a management strategy leveraging distributed principles. And complementary, addressing the automatized interfaces, and networking technologies challenges will allow a correct implementation of such management strategy.

Part II

State Of The Art

This second part covers the study of propositions leveraging decentralized management models that may be of interest in the context of DCIs.

- *Chapter 4 presents the definitions and properties needed to understand the studied solutions.*
- *Chapter 5 discusses and surveys network-oriented SDN controllers.*
- *Chapter 6 presents and surveys cloud-oriented SDN controllers.*
- *Chapter 7 presents and surveys propositions that provide decentralization of the resource manager (OpenStack and Kubernetes).*
- *Chapter 8 presents the insights gained from the different studies.*

Chapter 4

Multi-instance solutions for DCI architectures: Properties

SDN approaches could be used to gain insights for decentralizing resource managers networking functionalities as suggested in Section 2.2.1. This chapter presents to the reader the principal properties of SDN propositions that we discuss in our state-of-the-art review. These properties are used to detail the different solutions of the following chapters.

4.1 SDN Solutions for DCIs

Similar to the use of SDN paradigm in cloud computing, DCI management may also benefit from it. The study and analysis of decentralized SDN principles may provide an insight into how to address the DCI networking management challenges we presented in Section 3.3 while taking into account the different characteristics of DCIs. In Chapters 5, 6, and 7, we propose an extensive literature review on distributed SDN controllers to identify and highlight their behavior when used in a DCI context. Besides SDN propositions, we discuss some influential projects that we consider of interest to manage DCIs.

4.2 Architecture

The first point that distinguishes one SDN solution from another is the way controllers are interconnected with each other [76, 77, 78, 79, 80]. Figure 4.1 presents the possible connection topologies identified. In the following, we discuss 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 problem of centralized architectures (SPOF, bottlenecks, network partitioning, etc.).

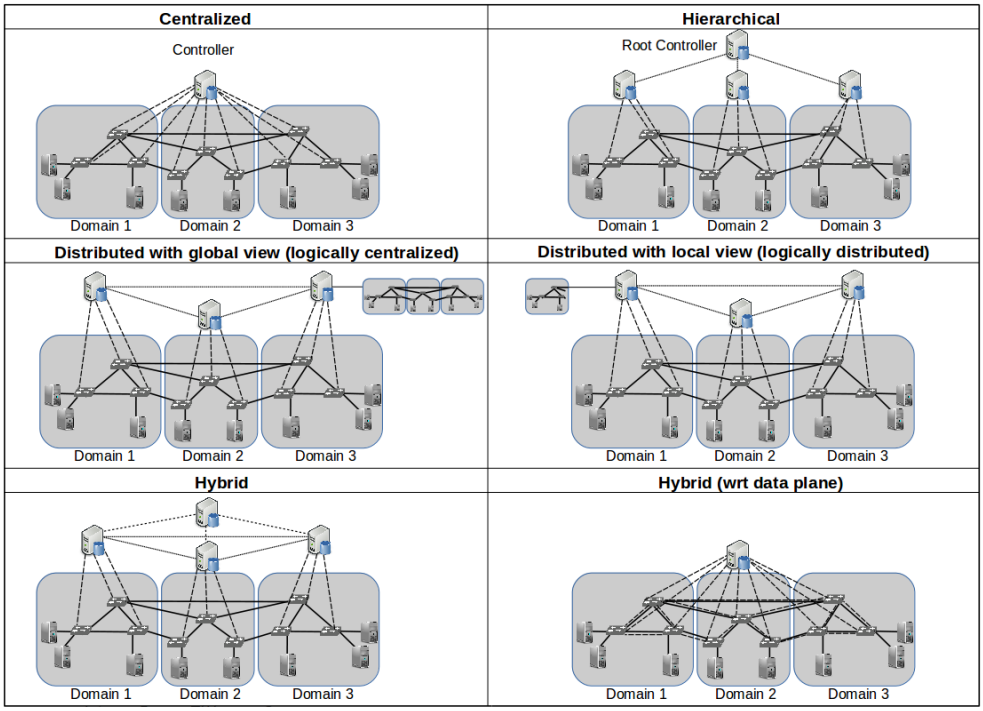


Figure 4.1 – SDN topologies

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 events. While local controllers only have their 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 architecture, 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 to all other controllers' modifications. It 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 [34] as several controllers share global network information to present themselves as one single controller.

Fully distributed: Architecture similar to the previous one but without commu-

nicating 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 to exchange technical information to establish, for example, 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 root 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.

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.

4.3 Leader-Based Operations

When implementing DCI network management, 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 [78] and thus do not require leadership mechanisms. On the opposite, leader-based operations (*i.e.,* 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 [81]: 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 [82] or Alvin [83]), and leader-based consensus (*i.e.,* such as PAXOS [84] or RAFT [85]).

Leader-based consensus protocols such as the ones above 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 [86]. Consensus typically involves multiple instances agreeing on values. Moreover, consensus can be reached when most instances are available; as an illustration, a cluster of five 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 average operation time of the protocol; in the worst case, they render RAFT unable to reach consensus by failing to elect a consistent leader [87].

To avoid the limitation imposed by a single leader election, leaderless consensus protocols allow multiple nodes to operate as a leader at-a-time [88]. 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 process 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 conflicting operations between them. As a consequence of this possibility, there is a higher probability of different nodes viewing different dependencies and proposing to be the operation-leader. In such cases, the nodes will try to collect dependencies following their own order while being in conflict with the order of other nodes. The time needed to order conflicting operations will grow letting the system in a potential incoherent state, thus, failing in delivering fast decisions.

4.4 Database Management System

As broadly discussed in Section 3.3.1, storing and sharing the state of the DCI network service would be a key challenge. Studied SDN solutions rely either on relational Structured Query Language (SQL) or NoSQL DBs.

SQL Databases

SQL DBs are based on the relational data model and are also known as Relational Database Managing System (RDBMS). In most cases, they use Structured Query Language for designing and manipulating data and are regularly deployed in a centralized node [89]. Relational DBs can generally be vertically scalable, which means that their performance could increase using more CPU or RAM. Some SQL DBs such as MySQL Cluster [90] or VoltDB [91], a NewSQL DB, try to scale horizontally, generally sharding

data over multiple DB servers (*a.k.a.* "shared nothing" architecture [92]), but they still present performance problems when scaling in infrastructures composed by a high number of nodes [93].

NoSQL Databases

NoSQL DB is a general term that gathers several kinds of DBs that do not use the relational model. NoSQL DBs can be gathered in four main types [94]: document-based (*i.e.,* MongoDB [95]), key-value pairs (*i.e.,* Redis [96]), graph DBs (*i.e.,* Neo4j [97]) or wide-column stores (*i.e.,* Apache Cassandra [98]). This kind of DBs can by nature scale horizontally as the unstructured data scheme allows information sharding in different sites as the opposite of Relational DBs. Such a model permits different entities to access data simultaneously in a geographically distributed way [99, 100].

More generally, the DB management system would be an essential element of DCI networking management. It could be used to share information between controllers and, thus, eliminating the need for a complex communication protocol to provide coherence at the application level.

4.5 SDN Interoperability and Maturity

SDN controllers should be capable to provide their capabilities in heterogeneous and dynamic network environments. In this section, we discuss the capacities of configuring different kinds of equipment and the supported networking protocols to accomplish controllers' tasks.

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).

— *An underlay network*: is a physical infrastructure deployed in one or several geographical sites. It comprises a series of active equipment like switches or routers connected among them using Ethernet switching, Virtual Local Area Networks (VLANs), routing functions, among other protocols. Due to the heterogeneity of equipment and protocols, the Underlay network becomes complex and hard to manage, affecting the different requirements that must be addressed like scalability, robustness, and high bandwidth.

— *An overlay network*: is a 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 encapsulation protocols like Virtual Extensible Local Area Network (VXLAN) because of its scalability (VXLAN provides up to 16 million identifiers whereas VLAN is limited to 4096 identifiers).

Some SDN controllers may deal with only one level. The richer the operations offered by controllers, the more difficult it would be to distribute the DCI networking management.

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 (*e.g.,* automation or orchestration tools). Southbound interfaces are used to allow the SDN controller to communicate with the network's physical/virtual equipment. OpenFlow [101] 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 Network Configuration Protocol (NETCONF), Local ID Separation Protocol (LISP), Extensible Messaging and Presence Protocol (XMPP), Simple Network Management Protocol (SNMP), Open Virtual Switch DataBase (OVSDB), Border Gateway Protocol (BGP), among others [34]. For example, BGP allows different Autonomous System (AS) to exchange routing and reachability information between their routers.

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

Technological Readiness Level

The Technological Readiness Level (TRL) is an indicator of a particular technology's maturity level. Due to the mechanisms' complexity, it is crucial to consider the TRL of technology to mitigate as much as possible development efforts. This measurement provides a common understanding of technology status and establishes the inspected SDN solutions' state, as not all solutions have the same maturity degree. To this end, the TRL proposed by the European Commission presented in [102] has been used to classify

the different solutions we studied.

Additional Considerations: OpenStack Compatibility

Because of the industrial context of this doctoral work, we also consider to take the example of an OpenStack-based system to explain how resource managers could use SDN solutions in a multi-site deployment. Therefore, the capability to integrate with Neutron is considered as an illustrative example. Indeed, some SDN controllers may be able to integrate with Neutron to implement networking management or add additional functionalities, consuming the Neutron core API or its extensions. Therefore, having a driver to pass network information to the controller.

4.6 Related Works in SDN

Since SDN has been a hot topic for the last few years, several studies about SDN characteristics, such as its decentralization, 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.

In [80] the authors present a general survey on SDN technologies introducing a taxonomy based on two classifications: physical classification and logical classification. For every one of the categories, multiple subcategories are presented and explained. Moreover, they placed the surveyed SDN solutions accordingly to their architectural analysis. The work finished by exposing a list of open questions such as scalability, reliability, consistency, interoperability, and other challenges such as statistics collection and monitoring.

In [77], and [76], the authors focus on the scalability criteria. More precisely, the work done by Karakus et al. [77] 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 point of view describes the relationship 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 the number of flows established per second and the flow setup latency.

In [76], 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 these two articles compare several controller architectures, 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 [79], and [78] are probably the most interesting ones concerning DCI objectives. In their article [79], Blial et al. give an overview of SDN architectures composed of multiple controllers. The study emphasizes the distribution methods and the communication systems used by several solutions to design and implement SDN solutions to manage traditional networks. Similarly, the survey [78] discusses some design choices of distributed SDN control planes. It delivers a captivating 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 issues mentioned above. While these two studies provide meaningful information for our analysis, they do not address the cloud computing viewpoint and the DCI challenges.

In the field of SDN applied especially to cloud computing, the works of Azodolmolky et al. [46, 103] provide information about the benefits and potential contributions of SDN technologies applied for the management of cloud computing networking. While these works represent a good 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 to provide networking federation among a few data centers. More recently, Son et al. [104] 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-data center 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). Datacenter 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. We can cite the question related to the extension of cloud computing concepts to the network's edge (i.e the DCI we envisioned).

4.7 Summary

The design principles of SDN controllers may vary the way they provide their networking functionalities. In this chapter, we presented the definitions of properties necessary to

understand the analysis we discuss in the next chapters.

Although it would be valuable, reviewing all SDN controller solutions that have been proposed [105, 106] is beyond the scope of our objective of delivering an appropriate solution for inter-site network connectivity management. We limited our study to the major state-of-the-art solutions and selected the best candidates that may fit DCIs. For the sake of clarity, we present the solutions we studied into three categories:

— Network-oriented SDN (Chapter 5) solutions conceived to provide network programmability to traditional or virtualized network backbones. The controllers gathered in this category do not provide SDN capabilities for cloud computing networking environments.

— Cloud-oriented SDN (Chapter 6) solutions proposing an SDN way to manage the networking services of cloud computing infrastructures (as explained in Section 2.2.1). 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.

— Other propositions (Chapter 7) are considered to analyze how the DCI management can be decentralized within resource managers.

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

Chapter 5

Network-oriented SDN controllers

This chapter discusses SDN solutions designed to provide network programmability to traditional or virtualized network backbones. The seven network-oriented SDN solutions we present are Distributed SDN Control Plane (DISCO) [107], Decentralized SDN (D-SDN) [108], Elastic Controller (Elasticon) [109], FlowBroker [110], HyperFlow [111], Kandoo [112], and Orion [113]. We analyse these solutions by considering the DCI challenges identified in Chapter 3. We use the key words ***Information granularity***, ***Information scope***, ***Information availability***, ***Automatized interfaces***, *and* ***Networking technologies*** *introduced at Table 3.1 to reference each one of the challenges. A synthesis of the characteristics analysis and the DCI challenges analysis are proposed at the end of the chapter and gathered at Table 5.1 and Table 5.2 respectively.*

5.1 DISCO

DISCO [107] relies on the segregation of the infrastructure into distinct groups of elements where each controller is in charge of one group using OpenFlow as a control plane protocol. Each controller has an *intra-domain* (or intra-group) and a *inter-domain* (or inter-group). The intra-domain part provides local operations like managing virtual switches. The inter-domain part manages communication with other DISCO controllers to make reservations, topology state modifications, or monitoring tasks. This architecture is illustrated in Figure 5.1. For the communication between controllers, DISCO relies on an Advanced Message Queuing Protocol (AMQP) message-oriented communication bus [114] where every controller has at the same time an AMQP server and a client. The central component of every controller is the DB where all intra- and inter-domain information is stored. We underline that there is no specific information on how the DB 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 SDN domain only and establishes inter-domain commu-

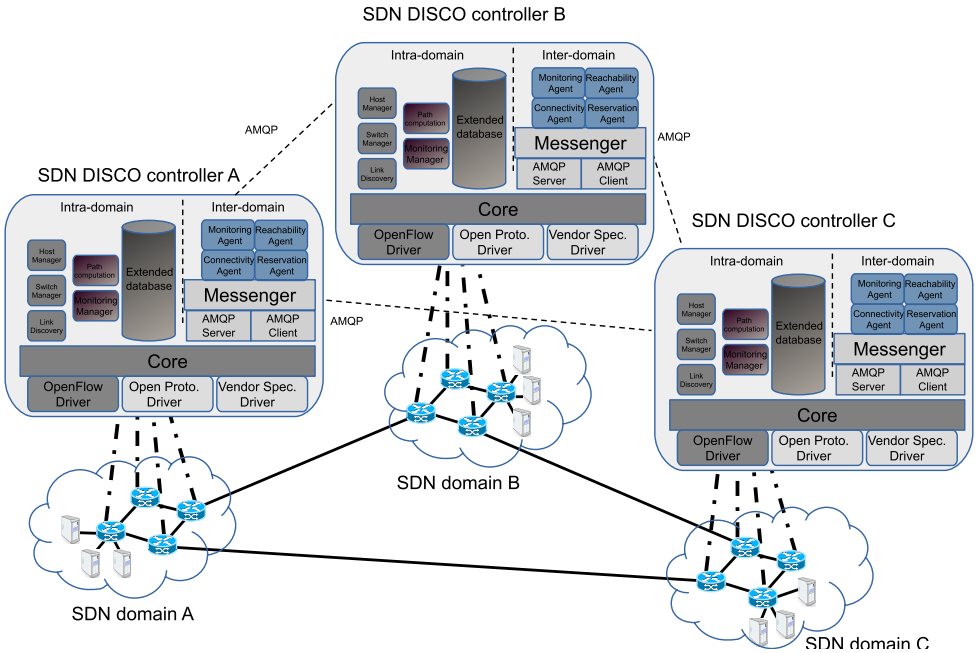


Figure 5.1 – DISCO SDN controller architecture (based on [107] © 2014 IEEE)

nication with other controllers to provide end-to-end connectivity only if needed. DISCO controllers do not act as a centralized entity and instead work as independent entities peering among them. It has 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 DCI 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 only to 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 traffic for-

warding when the connectivity is reestablished. As aforementioned, we underline that DISCO is conflict-less due to its implementation and the information that is manipulated. It makes the recovery process relatively 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.

5.2 D-SDN

D-SDN [108] distributes the SDN control into a hierarchy of controllers as shown in Figure 5.2; *i.e.,* Main Controllers (MCs) and Secondary Controller (SCs), using OpenFlow as control plane protocol. Similar to DISCO, SDN devices are organized into 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 subset of SDN devices). We underline that the current proposition does not give sufficient details regarding how states are stored within MC and SC. The proposal 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 scenarios. The main idea is to have replicas of SCs 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 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.

Addressing the DCI Challenges

— **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, there is no information that specifies how the MC deals with such a disconnection. Besides, the controller does not provide any recovery method as D-SDN does not consider network partitioning issues.

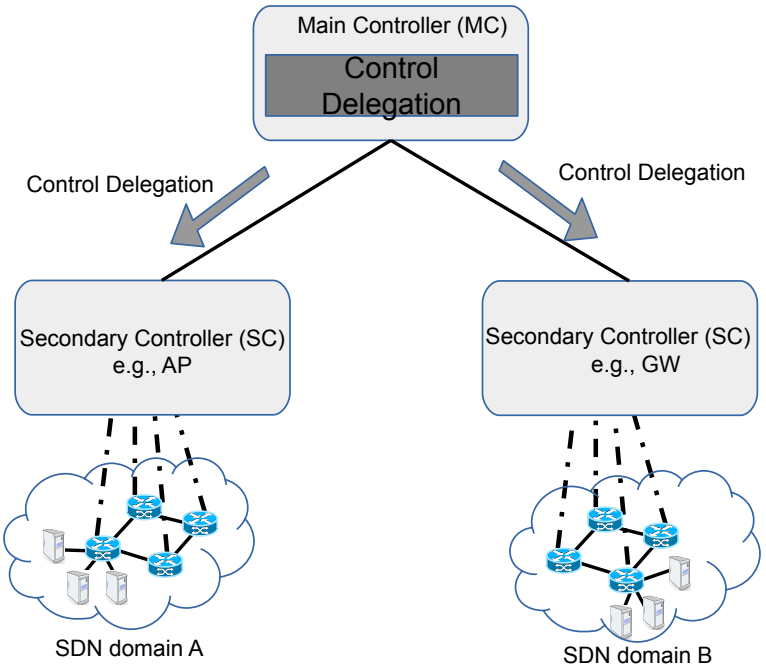


Figure 5.2 – D-SDN SDN controller architecture (based on [108] © 2014 IEEE)

— **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 resources deployment.

5.3 Elasticon

Elasticon [109] is an SDN controller composed of a pool of controllers as shown in Figure 5.3. The collection 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 collection varies according to a load window that evolves. 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 the controllers perform actions. To do so, each controller maintains a Transmission Control Protocol (TCP)

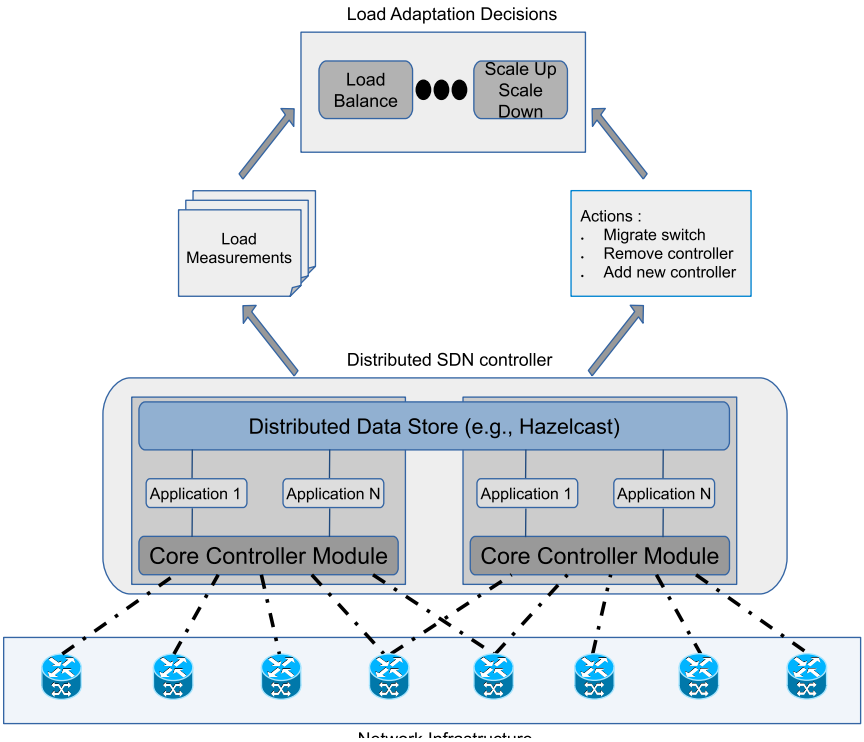


Figure 5.3 – Elasticon SDN controller architecture (based on [109])

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 [115], which can be accessed by all controllers. The use of a shared back-end by the pool gives a physically distributed but logically centralized design. As stated in Elasticon's work, the solution has been implemented as a prototype. Thus, a TRL of 3 has been assigned to it.

Addressing the DCI Challenges

— **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 DB, controllers will be able to add information to the DB and fetch the others' knowledge to establish inter-site connectivity. However, the consistency of the data store might be another issue to deal with.

— **Information scope**: Undefined - it is linked to the DB 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 DB systems such as CockroachDB [116] only favor data-locality across several geo-distributed sites partially.

— **Information availability**: Undefined - similarly to the last challenge, it is linked to the way the DB services deal 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 communicate 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.

5.4 Flowbroker

FlowBroker [110] is a two-layers architecture using OpenFlow as 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 mindful 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 architecture is presented in Figure 5.4.

FlowBroker presents a hierarchical design 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 domain, communication with the broker agents is mandatory for inter-domain forwarding. Because of this reason, FlowBroker presents leader-based coordination, where brokers act as leaders. However, we underline that there is no 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.

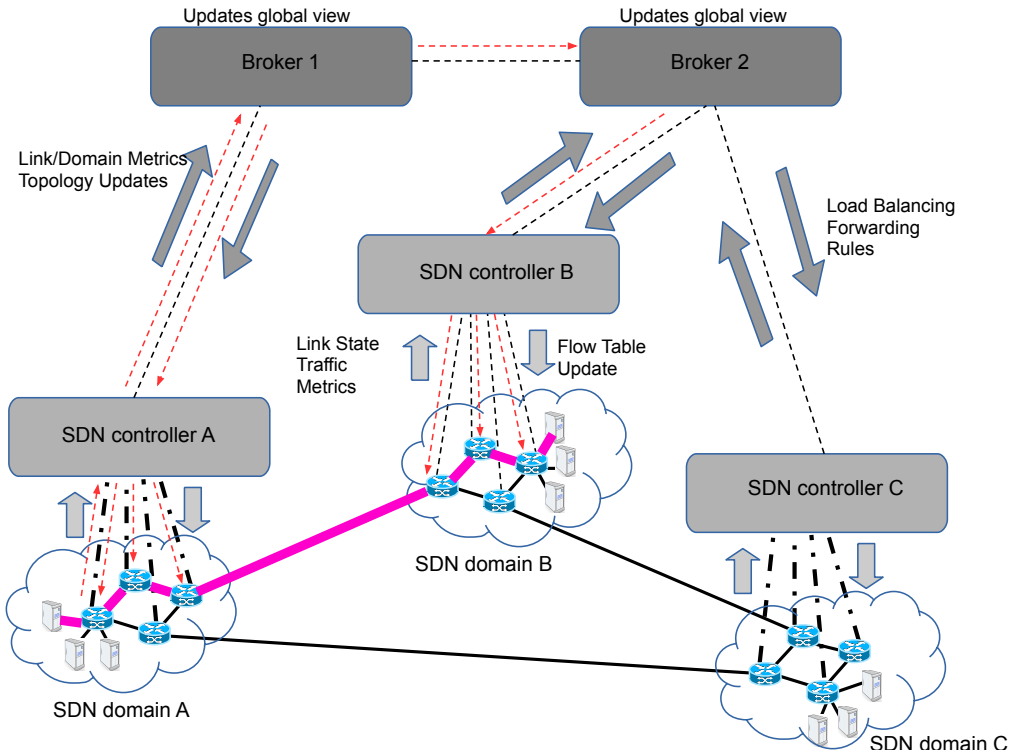


Figure 5.4 – FlowBroker SDN controller architecture (based on [110])

Addressing the DCI Challenges

— **Information granularity**: Not addressed - the segregation into semi-autonomous controllers enables the efficient sharding of the information per site. However, the brokers' global view information does not enable the validation of this challenge.

— **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-site creation, the result of each operation is forwarded to each broker 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. They only need to contact brokers to request 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.

5.5 HyperFlow

Hyperflow [111] is an SDN NOX-based[117] multi-controllers 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 [118]. Each controller subscribes to three channels: data channel, control channel, and its 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.

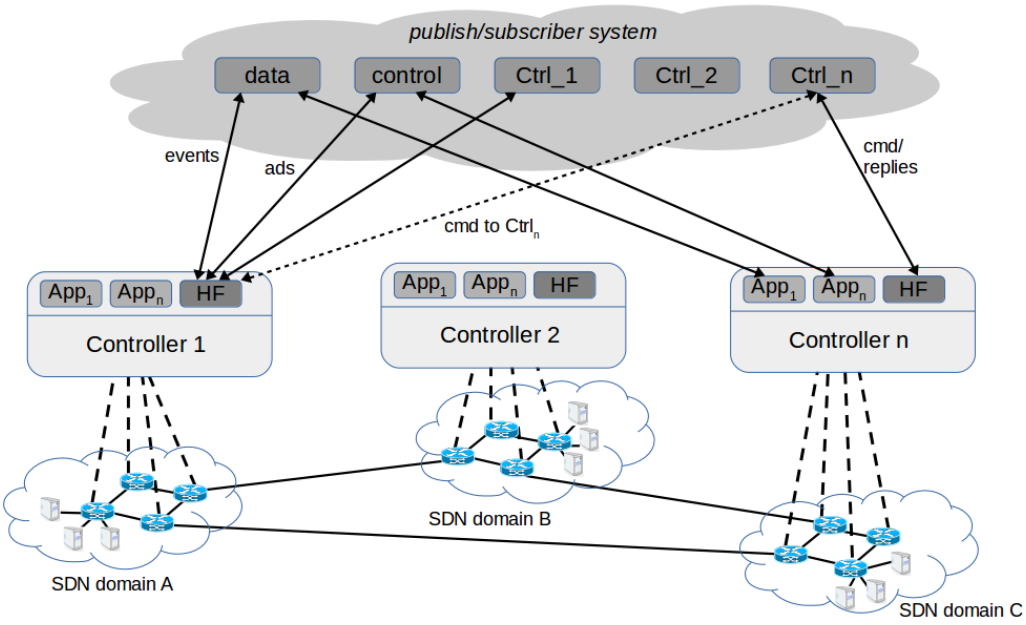


Figure 5.5 – HyperFlow SDN controller architecture (based on [111])

As all participant controllers share global networking information, the controller topology presents a physically distributed but logically centralized design. Every controller

manages its 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 concerning 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 leader-less. As this proposition has been implemented as a proof-of-concept, a TRL of 3 has been assigned to HyperFlow.

Addressing the DCI Challenges

— **Information granularity**: Addressed - thanks to WheelFS, it is possible to deploy one controller per site. Each one uses WheelsFS to share networking information 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 traffic 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, all controllers will share the information.
— **Information availability**: Partially addressed - in case of 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 among them). Moreover, WheelFS provides a recovery method to deal with network partitioning issues. Such a feature should enable controllers to requests new inter-forwarding paths after disconnections without requiring to implement 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.

5.6 Kandoo

Kandoo [112] is a muti-controller SDN solution built around a hierarchy of controllers using OpenFlow as control plane protocol. At the low level, local-domain controllers are in charge of managing a set of SDN-enabled 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. This architecture is presented in Figure 5.6. 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, there is no sufficient information to understand how this channel works and how the inter-domain flows are set up.

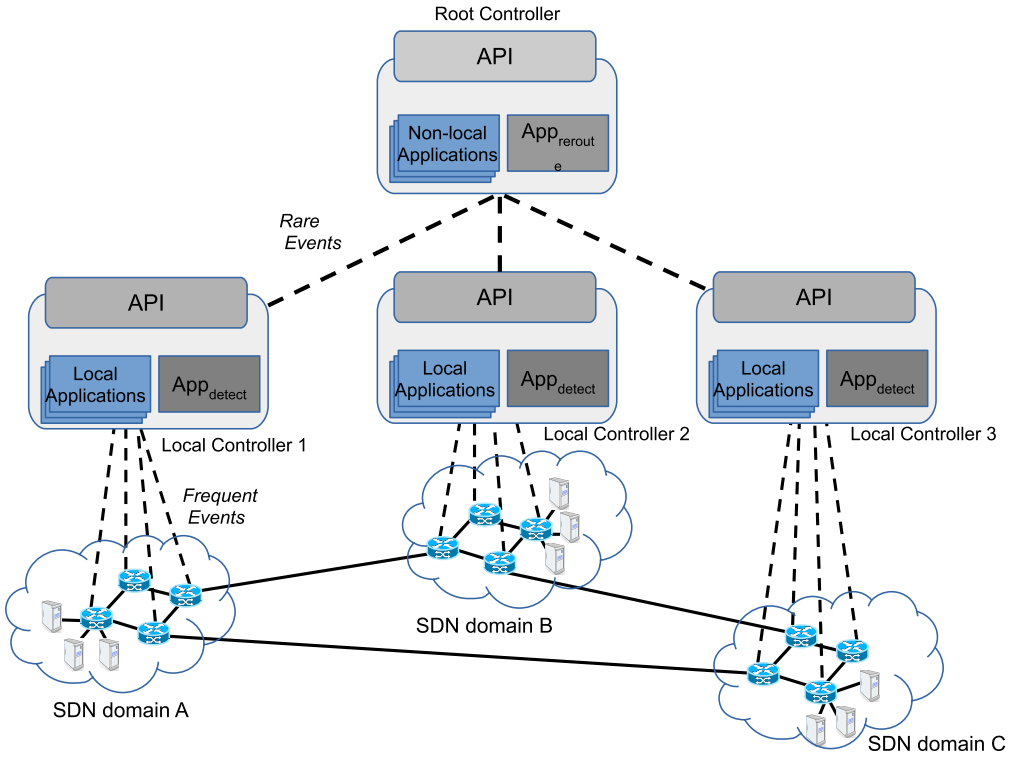


Figure 5.6 – Kandoo SDN controller architecture (based on [112])

By its two-level hierarchical design, Kandoo presents 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 DCI 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.

5.7 ORION

Orion [113] is presented as a hybrid SDN proposition using OpenFlow as 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 sub-set 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. This architecture is presented in Figure 5.7.

Synchronization of network states between domain controllers is done using a scalable NoSQL DB. 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. Moreover, all domain controllers are arranged in a Peer-to-Peer (P2P) way, using a NoSQL DB to share information. Unfortunately, there are no details regarding

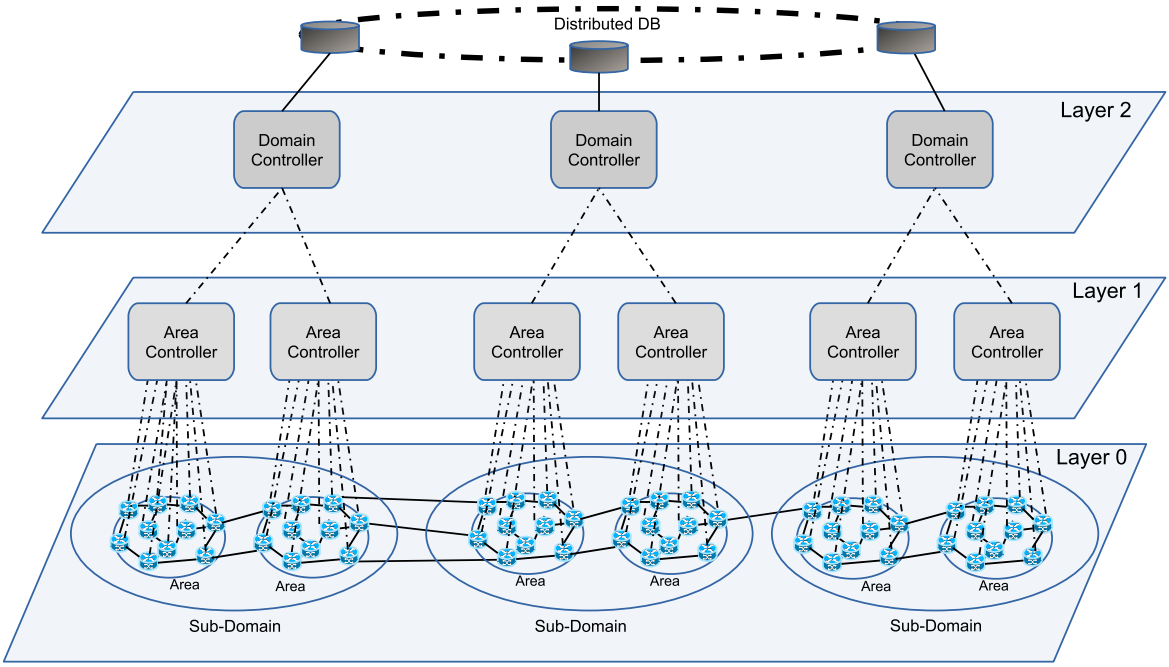


Figure 5.7 – Orion SDN controller architecture (based on [113] © 2014 IEEE)

the NoSQL DB or coordination protocol among the domain controllers. Hence, it is not clear whether Orion uses 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 DCI Challenges

— **Information granularity**: Not addressed - although the infrastructure is divided into domains (each domain controller maintains its 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 traffic forwarding, and second, we do not know how information is shared between domain controllers (*i.e.,* it is related to the DB 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

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			
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)	✗

Table 5.1 – Classification of surveyed network-oriented SDN solutions.

disconnection, the area controller only needs to reconnect to its domain controller when required 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 used to share information among domain controllers is unclear.

— **Automatized interfaces**: Not addressed - because 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.

5.8 Summary

This chapter introduced seven SDN solutions conceived to operate traditional or virtualized network backbones. We have detailed each proposition using the design principles described in Chapter 4 as summarized in Table 5.1, and we have done an analysis to indicate if the solutions are capable or not to fulfill the DCI networking management challenges introduced in Chapter 3 completely. Most of the solutions rely on physically distributed but logically centralized (*i.e.,* Elasticon, Hyperflow) and hierarchical (*i.e.,* D-SDN, FlowBroker, Kandoo) architectures designs. Only two solutions present other architecture designs, such as the fully distributed (*i.e.,* DISCO) or the hybrid ones (*i.e.,* ORION). Using a distributed DB is the most common approach to store information and communicate with distant instances. However, it is questionable whether this approach can fully address the DCI management challenges in the case of network disconnections. Among the propositions, only DISCO entirely relies on a P2P communication exchange employing its east-west interface. Overall, none of the analyzed solutions can completely

Proposals	Organization of network information			Inter-site networking resources implementation	
	Information granularity	Information scope	Information availability	Automatized interfaces	Networking technologies
Network-oriented solutions					
DISCO	✓	✓	✓	✓	✗
D-SDN	✓	~	?	✗	✗
ElastiCon	~	?	?	~	✗
FlowBroker	✗	✗	✓	✗	✗
HyperFlow	✓	~	~	~	✗
Kandoo	✗	✗	✓	✗	✗
Orion	✗	✗	?	✗	✗

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

Table 5.2 – Challenges summary of network-oriented solutions.

coop with all of the DCI networking management challenges as summarized in Table 5.2.

The next chapter introduces the second part of this state-of-the-art overview, presenting a detailed analysis of the cloud-oriented SDN solutions.

Chapter 6

Cloud-oriented SDN controllers

In this chapter we continue our state-of-the-art review by focusing on solutions designed to provide SDN capabilities to cloud computing networking management. The five solutions we present are DragonFlow [119], OpenDayLight (ODL) [120], Onix [121], Open Networking Operating System (ONOS) [122], and Tungsten [123]. We analyse these solutions by considering the DCI challenges identified in Chapter 3. We use the key words ***Information granularity****,* ***Information scope****,* ***Information availability****,* ***Automatized interfaces****, and* ***Networking technologies*** *introduced at Table 3.1 to reference each one of the challenges. Similar to the previous chapter, a synthesis of the characteristics analysis and the DCI challenges analysis are given at the end of the chapter and gathered at Table 6.1 and Table 6.2 respectively.*

6.1 DragonFlow

DragonFlow [119] 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 2.2.1.1). From the software architecture, DragonFlow relies on a centralized server (*i.e.,* the Neutron server) and local controllers deployed on each compute node of the infrastructure as illustrated in Figure 6.1. 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 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 [124], RAMCloud [125], Cassandra [126], and using ETCD [127] as default back-end).

Local controllers periodically fetch all network state information 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 a distributed but logically centralized controller (see Section 4.2) at first sight. However, there is a root controller (*i.e.,* the Neutron

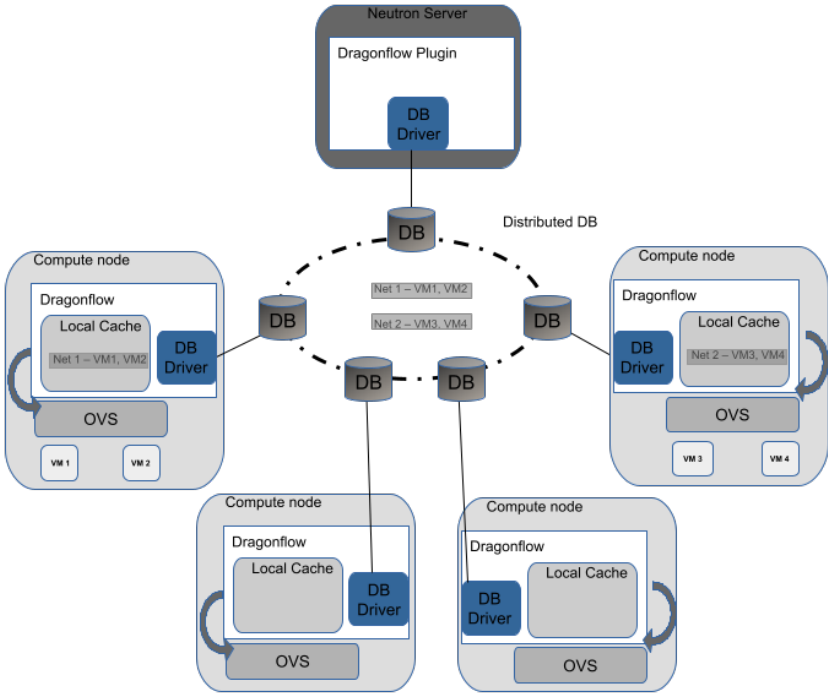


Figure 6.1 – DragonFlow SDN controller architecture (based on [119])

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 TRL 6. Initially supported by Huawei, the project is relatively inactive right now, however.

Addressing the DCI 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-ins. Each one will add information to the database, and all local controllers will be capable of fetching the necessary information to provide inter-site resources.

— **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 - similar to the last challenge and 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. For instance, currently, DragonFlow does not support BGP dynamic routing [128].

6.2 OpenDayLight

OpenDayLight (ODL) [120] is a modular SDN platform supporting a wide range of protocols such as OpenFlow, OVSDB, NETCONF, BGP, among others. Originally, ODL has been developed as a centralized controller to merge legacy networks with SDN in datacenters, but its modularity allows users to build their SDN controller to fit specific needs [129]. The internal controller architecture comprises three layers: The southbound interface, which enables communication with network devices. The Service Adaptation Layer (SAL) adapts the southbound plug-ins' 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 introduces features aiming to enable different controller instances to peer among them like the SDNi [130] or the more recent Federation [131] projects. ODL Federation service facilitates the exchange of state information between multiple ODL 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 view of the system). Moreover, it has leader-less coordination because there is flat on-demand communication between controllers, and no leader is needed for these exchanges.

Concerning the OpenStack compatibility, the modularity of the controller allows multi-

ple 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 (VTN), Group Based Policy (GBP), and OVSDB-based Network Virtualization Services (NetVirt) [132]. Figure 6.2 shows the architecture of ODL when using the Federation and NetVirt projects.

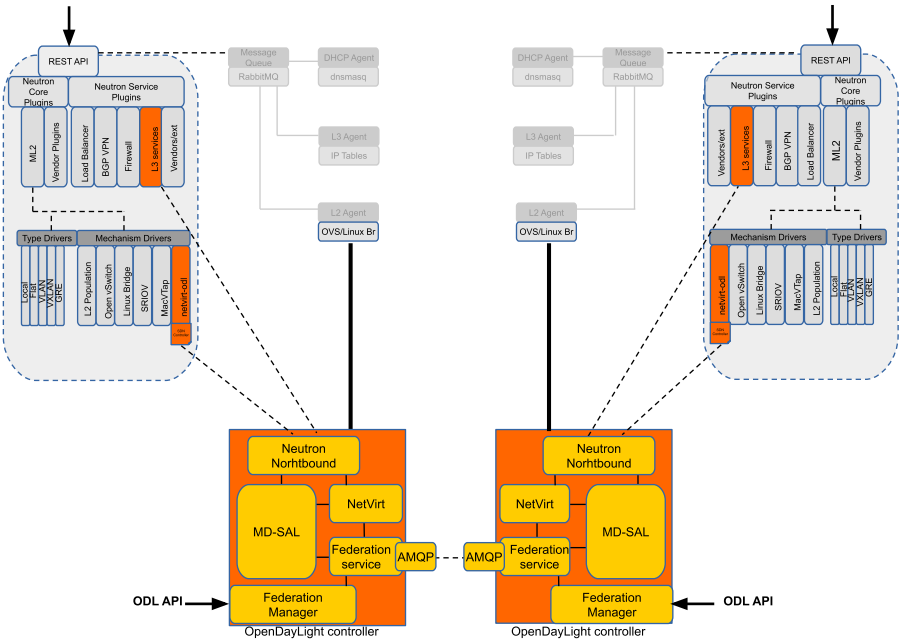


Figure 6.2 – OpenDayLight Federation NetVirt SDN controller architecture (based on [120] © 2014 IEEE)

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 resource between two OpenStack instances, it realizes the interconnection at the ODL level (*i.e.,* creating shadow elements, etc.). Moreover, it performs the matching with the OpenStack Neutron resources on the different sites. Although this enables the interconnection of 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 not 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 [133].

Addressing the DCI 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) with the mechanisms to do information sharding.
— **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 requested by the user.
— **Information availability**: Partially addressed - in case of network disconnection, ODL instances can satisfy local networking services (including the resource manager ones). At the same time, the non-disconnected controllers can continue to provide the inter-site resources. Since inter-site resources are proposed outside the knowledge of the resource manager networking module, ODL assumes that there are no conflicts between networking objects when establishing the resource. 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 essential 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 service.

6.3 Onix

Onix [121] 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) as presented in Figure 6.3. 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, the instances should use a replicated transactional SQL database. Otherwise, it is possible to use any NoSQL system.

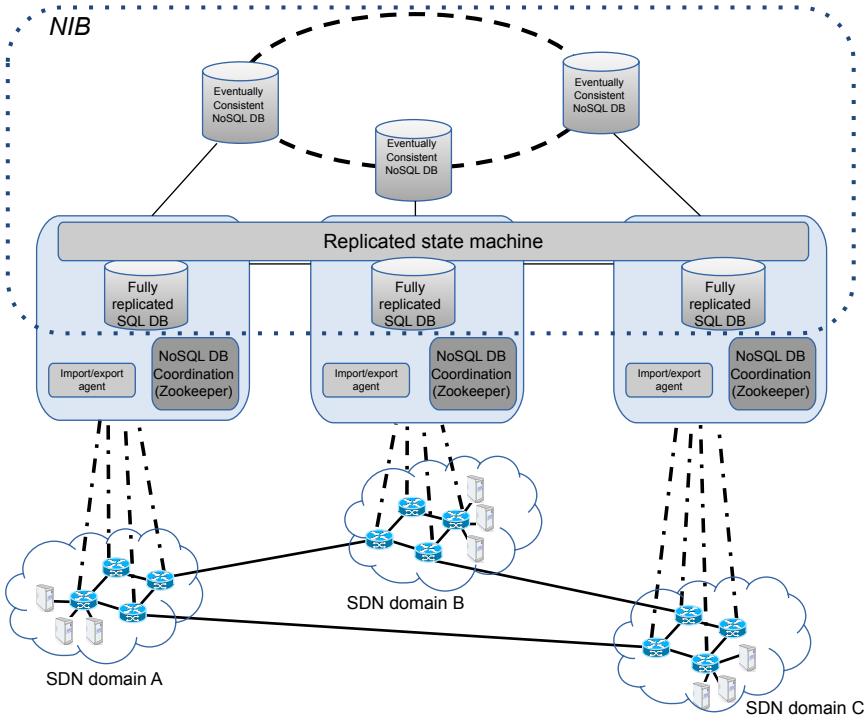


Figure 6.3 – Onix SDN controller architecture (based on [121])

Regarding coordination aspects, the system leverages ZooKeeper [134] 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.

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 a commercial product, we assigned a TRL of 7.

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

Addressing the DCI 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, all instances should synchronize the information. In the case of a NoSQL system, it depends on how the DB shards the information across different instances.
— **Information availability**: Undefined - established inter-site resources 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 DBs 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 NIB's use as an abstraction entity for network elements.

6.4 ONOS

ONOS [122] is a modular and distributed SDN framework consisting of several network applications build on top of Apache Karaf OSGi container [135]. It supports the use of multiple control plane protocols like OpenFlow, NETCONF, among others. ONOS has been created for overlay and underlay networks of service providers. Network states' information is stored using the Atomix database [136], a NoSQL framework developed for ONOS, which is also used for coordination tasks among controllers. Figure 6.4 shows such a configuration of multiple ONOS 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 [85]). Considering that ONOS is one of the most popular SDN open-source controllers and is used by several key actors in telecommunications [137], 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: Simplified Overlay Network Architecture (SONA) [138],

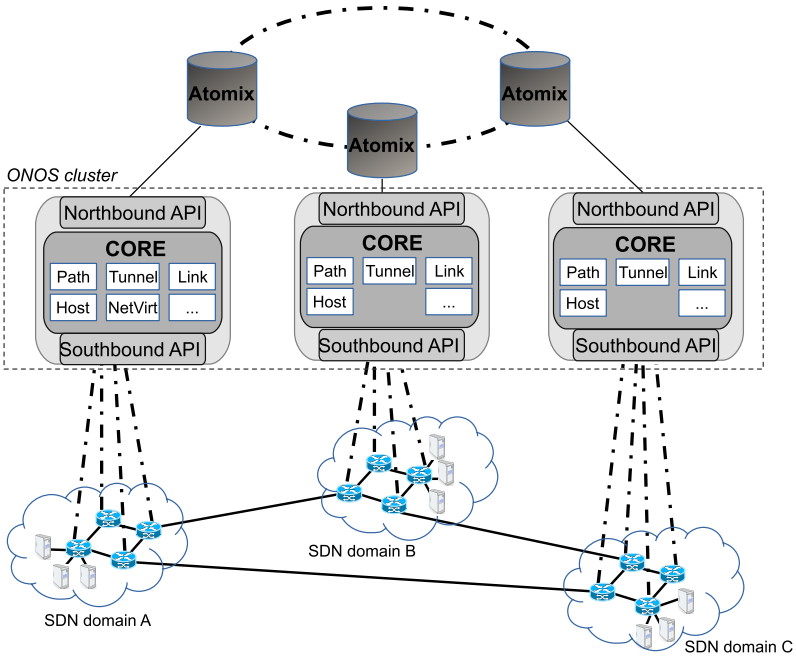


Figure 6.4 – ONOS SDN controller architecture (based on [122])

VTN , and Central Office Re-architected as a Datacenter (CORD) VTN [139]. Each application has been designed with different targets [138, 139]. SONA provides an ML2 driver and an L3 service plug-in implementation. VTN provides service function chaining capabilities. CORD VTN extends VTN with its interfaces for switching and routing configuration [140].

Addressing the DCI 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.

6.5 Tungsten

Tungsten Fabric (previously known as Juniper's Open-Contrail) [123] 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, as shown in Figure 6.5:

— Configuration nodes are responsible for the management layer. They provide a REST API [141] that can be used to configure the system or extract operational status. Multiple nodes of this type can be deployed for High Availability (HA) purposes. Note that configuration states are stored in Cassandra, a NoSQL database.

— 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 Internal Border Gateway Protocol (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 virtual Router (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 as 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 viewed as the leaders). Given 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 DCI 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 [142]. However,

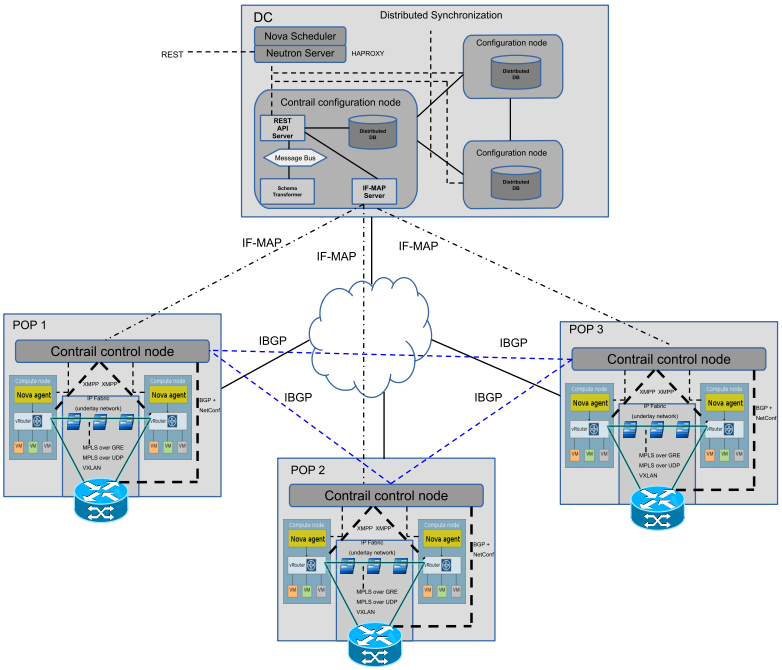


Figure 6.5 – Tungsten SDN controller architecture (based on [123] © 2018 Tungsten Fabric)

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 can configure a different kind of network equipment.

6.6 Summary

This chapter introduced five SDN solutions conceived to provide management for cloud computing IaaS networking. As in the previous chapter, we have provided an analysis for

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			
Solutions																
DragonFlow		✓		✓		✓		DB-in	DB-in	DB-in	NoSQL DB/ others		✓	OpenFlow	Demonstrated (TRL 6)	
ODL (Fed)	✓		✓				✓	AMQP	-	-	In-memory	✓	✓	OpenFlow, BGP & others	Proven system (TRL 9)	✓
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)	✓
Tungsten	✓			✓		✓		BGP	IFMAP	DB-in	NoSQL DB	✓	✓	XMPP, BGP & others	Proven system (TRL 9)	✓

Table 6.1 – Classification of surveyed cloud-oriented solutions.

Proposals	Organization of network information			Inter-site networking resources implementation	
	Information granularity	Information scope	Information availability	Automatized interfaces	Networking technologies
Cloud-oriented solutions					
DragonFlow	~	?	?	~	~
ODL (Fed)	✓	✓	~	✓	✓
Onix	~	?	?	~	~
ONOS	~	?	?	~	✓
Tungsten	✗	✗	?	✗	✓

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

Table 6.2 – Challenges summary of cloud-oriented solutions.

each proposition detailing their design principles. We have assessed to indicate if the solutions cannot fulfill the DCI networking management challenges.

Similar to the SDN solutions we presented in Chapter 5, most of the solutions rely on physically distributed but logically centralized (*i.e.,* ONIX, ONOS) and hierarchical (*i.e.,* DragonFlow, Tungsten) architectures designs. ODL in Federation mode is the only solution proposing a fully distributed architecture leveraging a collaborative model among the controller instances. Most of the solutions leverages the database, both as a storage and communication tool, to synchronize and share information states across the controller's instances. Table 6.1 gives a synthesis of the characteristics analysis. Similar to the results of Chapter 5, none of the analyzed solutions can completely coop DCI networking management challenges as summarized in Table 6.2.

The next chapter introduces the third part of this state-of-the-art overview, presenting some projects we consider of interest in analyzing networking management decentralization in DCIs.

Chapter 7

Other decentralized propositions

In this chapter we finalize our state-of-the-art review by presenting solutions that provide decentralization of the resource managers. While not being purely SDN solutions, we consider them of great relevance to analyze resource managers' functionalities in DCIs. Similar to the SDN propositions, we analyze whether or not these solutions can cater to the DCI networking management challenges. The four solutions we present are OpenStack P2P external proxy-agents [68], Tricircle [64], Kubernetes Federation [143], and Kubernetes Istio Multi-Cluster Service Mesh [144]. We analyse these solutions by considering the DCI challenges identified in Chapter 3. We use the key words ***Information granularity***, ***Information scope***, ***Information availability***, ***Automatized interfaces***, *and* ***Networking technologies*** *introduced at Table 3.1 to reference each one of the challenges. Similar to the previous chapter, a synthesis of the characteristics analysis and the DCI challenges analysis are given at the end of the chapter and gathered at Table 7.1 and Table 7.2 respectively.*

7.1 OpenStack P2P External Proxy-Agents

The OpenStack P2P external proxy-agents proposes to abstract several instances of OpenStack as a single instance exposed to the user. The idea is to associate an agent (*a.k.a.* proxy-agent) with each instance of the resource manager whereby the user's request is received and processed. This proposal is not conceived to target the inter-site networking connectivity, but at the place, to propose a broker-alike approach to massively scale-out cloud infrastructures.

Every one of the agents implements three key functionalities. First, agents implement the OpenStack services APIs to act as proxies for the users' requests. Second, agents implement an overlay management function used to maintain several overlay networks. These networks allow agents to discover each other and identify the source agent of a request. Finally, agents implement a state management function responsible for keeping information about users and cloud resources.

From the user's perspective, an agent is an entry point for the abstract P2P system,

and resource managers behind the agent are transparent. Each time the user requests one of the agents, it may choose to forward the request to its associated resource manager instance or may decide to deliver it to another agent (*i.e.*, by using a series of filters) via an agent-to-agent interface as depicted in Figure 7.1.

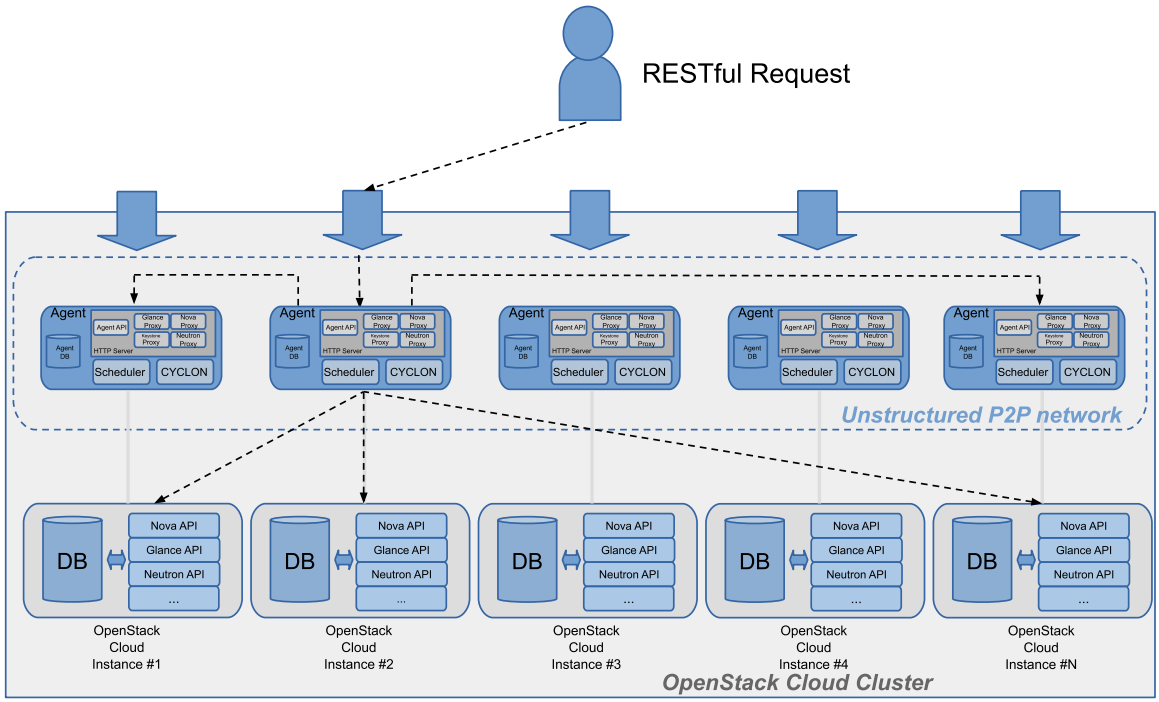


Figure 7.1 – P2P external proxy-agents architecture (based on [68] © 2017 IEEE)

Each agent maintains a list of the provisioned resources in a local database. This list is build using the identifier created by the resource manager to indicate in which instance the resource exists. While not implemented in the proof-of-concept, the model allows the introduction of resources federation to interconnect heterogeneous resources across independent cloud infrastructures. To maintain the overlay network among agents, each agent keeps possession of a list of resource managers based on the CYCLON group membership protocol [145]. The protocol allows maintaining a list of neighbor agents that is exchanged periodically. Consequently, each agent is aware of only a small, continuously changing set of other agents. Because the agents only process or forward users' requests to create resources in several resource managers, no coordination is needed. As a consequence, it presents leader-less coordination.

Considering that peer-to-peer communication is used to communicate among the differ-

ent agents, P2P proxy-agents presents a fully distributed architecture. Since this proposal has been implemented as a PoC, we have assigned a TRL of 3.

Addressing the DCI Challenges

— **Information granularity**: Partially addressed - Similar to Kubefed (Section 7.3), the segregation of information into each resource manager is proposed. However, since the agents act as a system proxy, there is not management sharding strategies proposed.
— **Information scope**: Partially addressed - while agents communicate only with other agents for a request, the user does not provide the scope, but it rather depends on the overlay management network. As a consequence, resources may be treated by remote geographical agents.
— **Information availability**: Addressed - in case of network disconnection, each resource manager and its agent are entirely independent of the others.
— **Automatized interfaces**: Addressed - the agents can communicate among them using the agent-to-agent interface. We remember that this interface can only be used to contact agents belonging to the overlay management network created with CYCLON when a request is made.
— **Networking technologies**: Not addressed - Similar to Kubefed 7.3, the proxy-agents use a broker-like approach. In consequence, no networking technology is used.

7.2 OpenStack Tricircle

Tricircle is an OpenStack project that aims to provide network automation across Neutrons in multi-region OpenStack deployments using a Central Neutron Service, several Local Neutron Services, and a series of modified Neutron plug-ins. Tricircle can be used to add more OpenStack instances into the Cloud for capacity expansion, to deploy geo-site distributed applications for higher reliability and availability, for security considerations, or to deploy NFV in sites closer to the user [64].

As we explained in Section 2.2.1.1, in a default multi OpenStack setup, each Region has its OpenStack deployment, including its API endpoints, networks and compute resources, message queues, and databases. Therefore, two independent Neutron deployments are neither aware of the Agents belonging to the other one nor can use the message bus of the remote Region.

To tackle down this problem, Tricircle presents a two-side solution as represented in

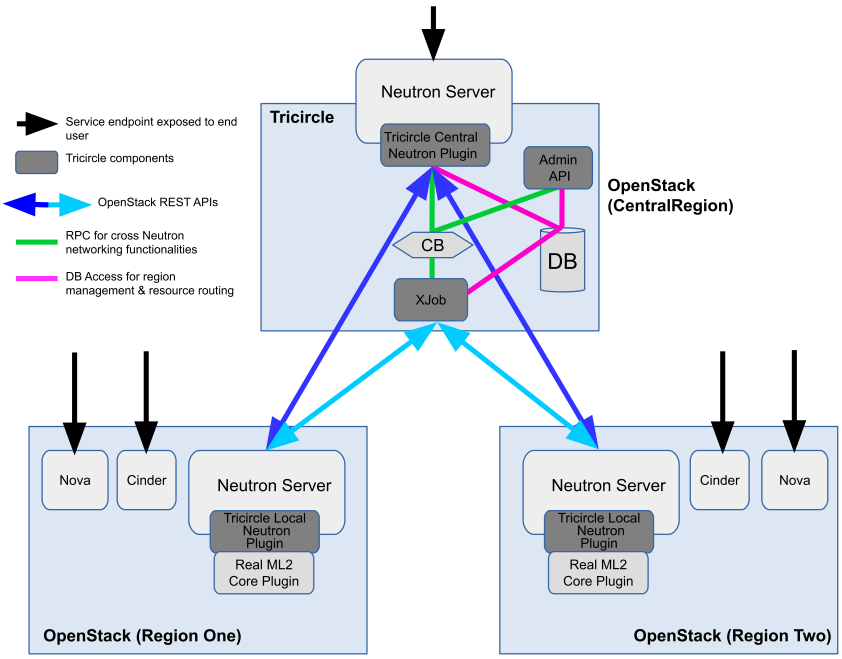


Figure 7.2 – OpenStack Tricircle architecture (based on [64])

Figure 7.2; a Tricircle service located at Central Neutron which is composed of an Admin API, a Tricircle Central Neutron plug-in, a database, a service called XJob for receiving and processing asynchronous requests, and a message bus. The second side is a Tricircle Local Neutron Plugin that inherits from the ML2 plug-in and is located at every Local Neutron.

Tricircle's proposed architecture presents a hierarchical structure, being the Central Neutron the natural leader. The principal drawback of these kinds of solutions is the fragility exposed by the root or central element controlling the entire architecture. In this case, the Central Neutron can experiment troubles that may turn the whole infrastructure inaccessible. Thus, presenting the well-known problem of being a SPOF or bottleneck. When a Local Neutron requests the Central Neutron to get a resource, information is firstly stored in the Central Neutron's database. Suppose a network disconnection affects the link between that Local Neutron and the Central Neutron. In that case, the former will not contact the data keeper and answer all requests with a message informing that the requested resource does not exist even if this resource has been already deployed in another Local Neutron.

While only using the Central Neutron API's endpoint to manage and control network constructions, the user can effectively deploy multi-site resources at the cost of not using the multiple Local Neutron endpoints present in the entire deployment. As the Tricircle

Local plug-in is closely coupled with the Tricircle Central plug-in, the user cannot perform local actions without affecting the data coherence managed at the central level. Additionally, Local Neutrons are not aware of each other but only about the presence of Central Neutron, turning the multi-region deployment into a single giant structure depending on Central Neutron.

Huawei conceived Tricircle to propose networking automation for OpenStack. Unfortunately, the project is no longer maintained and has not been deeply adopted by other actors. As a consequence, we assigned a TRL of 6.

Addressing the DCI Challenges

- **Information granularity**: Not addressed - the segregation of information into each instance of the resource manager is proposed. However, the Central Neutron gathers global information since it is the only entry point to the system.
- **Information scope**: Partially addressed - while Tricircle proposes to only contact the relevant Neutrons for a request, this is only done by the Central Neutron.
- **Information availability**: Partially addressed - Because Tricircle has been designed following a hierarchical approach, network disconnections between Central and Local Neutrons can render inoperative these last.
- **Automatized interfaces**: Not addressed - because Local Neutrons do not have an east-west interface to communicate among them, but only receive requests from the Central Neutron.
- **Networking technologies**: Addressed - Since it is an in-Neutron proposal, Tricircle incorporates several networking technologies to provide inter-site connectivity.

7.3 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 to deploying container-based applications across multi-sites through a standard API. In other words, it does not leverage nor integrate SDN technologies but instead 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 Definition (CRDs), an object provided by the vanilla Kubernetes that

allows DevOps to define their 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.

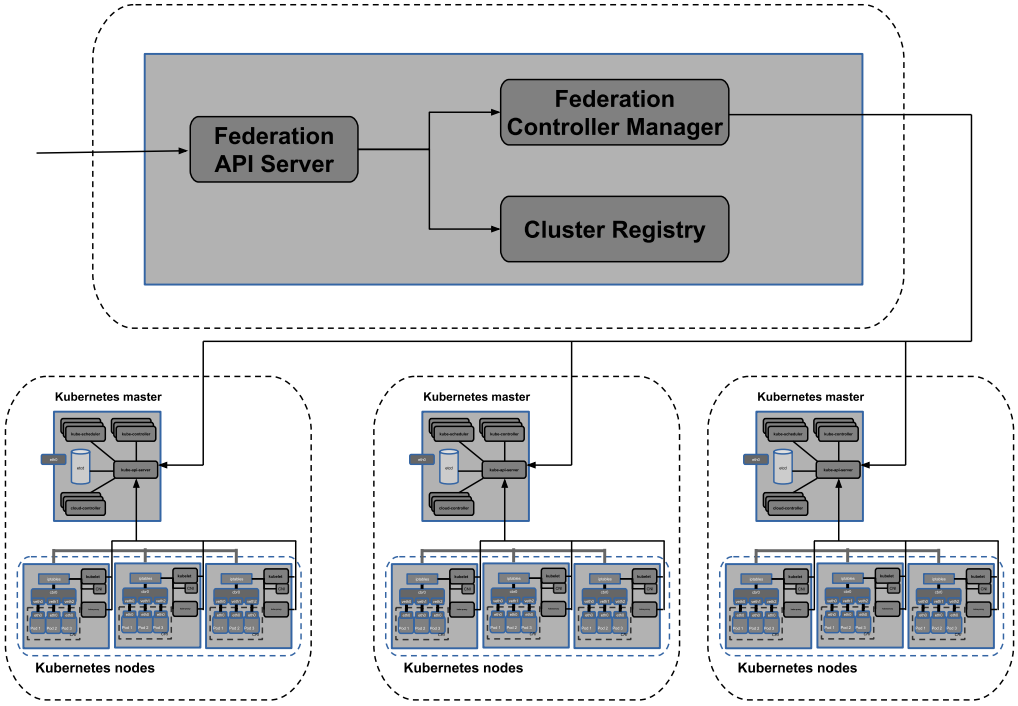


Figure 7.3 – Kubernetes Federation architecture (based on [143])

When a *FederatedService* is created, Kubefed creates matching K8s Services in the selected member clusters. To propagate the DNS records of each cluster, Kubefed gathers all the locally generated DNS records in the host cluster and then pushes the registries to each one of the concerning clusters. It 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 offered. More generally, Kubefed has some shortcomings 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.

As stated above, Kubefed has a hierarchical design being the host cluster the root

entity of the architecture. It has leader-based coordination, with the host cluster being the natural leader in the hierarchy. As Kubefed is a recent proposal within the Kubernetes community and is currently in alpha state, we assigned a TRL of 5.

Addressing the DCI 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 on federated resources.
- **Information scope**: Partially addressed - while Kubefed only contacts the relevant sites when deploying a federated resource, it is only executed by the host cluster.
- **Information availability**: Addressed - in case of network disconnection, each cluster is entirely 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.

7.4 Kubernetes Istio Multi-Cluster Service Mesh

Istio is an open-source implementation of a service mesh that provides traffic management, security, and monitoring. 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 functionality, a service mesh introduces a set of network proxies used to route requests among services. A central authority will then control over the proxies to route traffic at the application layer (L7 of the Open System Interconnection (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]). Istio also proposes an Ingress Gateway object that acts as a load balancer at the mesh's edge receiving incoming or outgoing traffic.

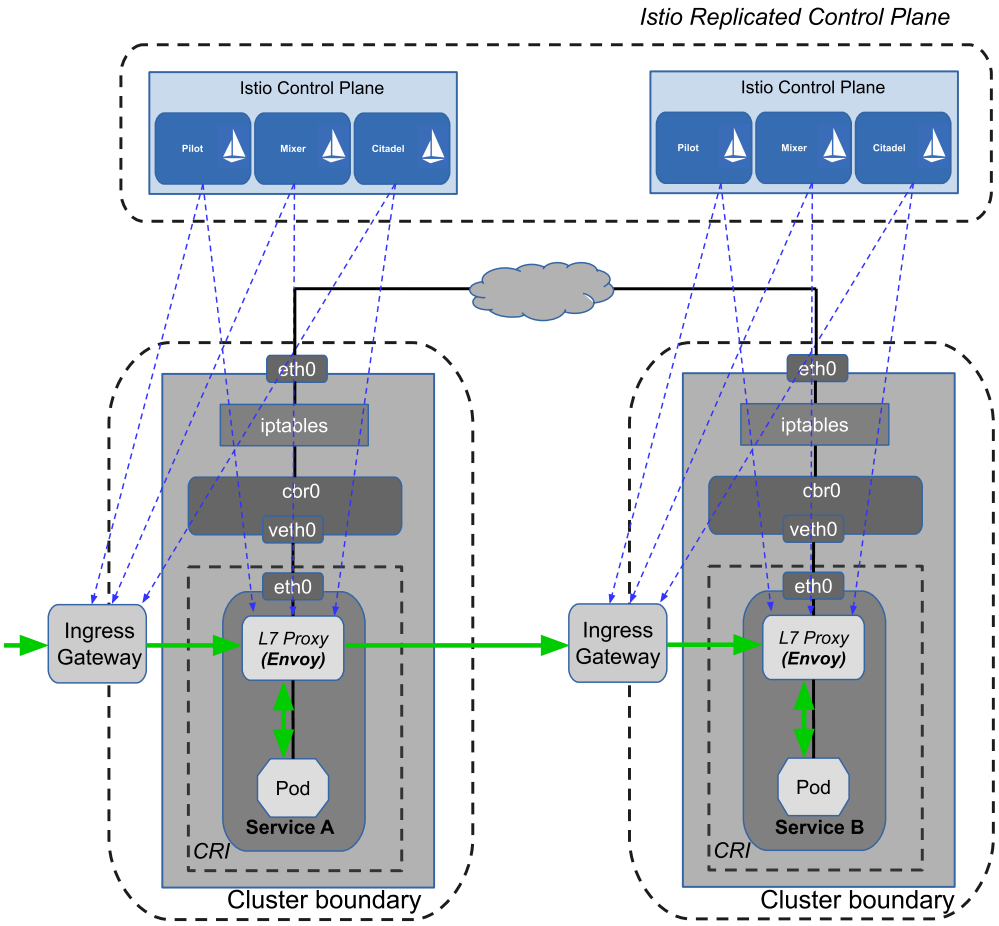


Figure 7.4 – Istio Multi-Cluster Service Mesh architecture (based on [144])

The concept of service meshes may be extended to take account of multiple clusters. The idea is then to have a logical service mesh composed of several clusters as proposed in Istio Multi-cluster [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 proxies to containers at each Pod in a DCI context and all the ServiceEntry rules treatment and processing requests on remote gateways could considerably add latency and potentially add a performance overhead [150, 151].

Regarding the three communication types, since Istio is a service mesh-oriented solu-

tion, it can only provide cross-cluster Pod-to-Service and External-to-Service communications using the replication of ServiceEntries. In Pod-to-Service communication, Services with private virtual IPs are reachable through the Istio Ingress gateways.

Istio Multi-cluster can be considered to have a fully distributed design but without automatized communication among the clusters. As a consequence, it has leader-less coordination. As we stated above, it is the user who has to define the replicated elements by hand. Since several industry actors use Istio, a TRL of 9 has been assigned to it.

Addressing the DCI 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 to deliver a Multi-Cluster service mesh.
— **Networking technologies**: Not addressed - although the network routing logic is implemented at the Envoy proxy and the Istio Ingress Gateway, Istio is independent of the low-level network technology used by the cluster.

7.5 Summary

This chapter introduced four proposals conceived to decentralize resource managers' functionalities to scale out in DCIs. Like the previous chapters, we have analyzed whether they can or not answer the different DCI challenges.

As some of the proposed SDN solutions of chapters 5 and 6, Kubefed and Tricircle leverage a hierarchical design with the corresponding limitations in terms of the SPOF and bottleneck. On the other hand, the P2P proxy-agents proposes a fully distributed architecture with collaborative agents. Unfortunately, it cannot provide networking con-

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			
Other solutions																
P2P agents		✓					✓	A2A interface	-	-	SQL DB		✓	-	PoC (TRL 3)	✓
Tricircle				✓		✓		-	API	-	SQL DB	✓	✓	-	Demonstrated (TRL 6)	✓
Kubefed				✓		✓		-	API	-	NoSQL DB		✓	-	Technology validated (TRL 5)	✗
Istio Multi		✓					✓	API	-	-	NoSQL DB		✓	-	Proven System (TRL 9)	✗

Table 7.1 – Classification of surveyed solutions.

Proposals	Organization of network information			Inter-site networking resources implementation	
	Information granularity	Information scope	Information availability	Automatized interfaces	Networking technologies
Other solutions					
OpenStack P2P external proxy-agents	~	~	✓	✓	✗
OpenStack Tricircle	~	?	?	~	~
Kubernetes Federation	✗	~	✓	✗	✗
Kubrnetes Istio Multi-Cluster Service Mesh	✓	✓	~	✗	✗

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

Table 7.2 – Challenges summary of other solutions.

nectivity due to its proxy API approach where no configuration can be done at the low level networking mechanisms. Unlike these three projects, Istio Multi-Cluster relies on a federated design where the user has to do the interconnection by hand, and no automation is proposed to communicate the Istio control planes. Table 7.1 summarizes the characteristics analysis and Table 7.2 gathers the analysis of whether the propositions are able to address the DCI challenges.

The next chapter summarizes the lessons learned from Chapters 5, 6, and 7.

Chapter 8

Multi-instance learned lessons and perspectives

In this chapter, we present a summary of the state-of-the-art analysis, detailing the most important design principles that may be used to provide DCI networking management.

8.1 Lessons Learned on Multi-instance Cloud Controllers

As stated in Chapter 2, SDN-based cloud networking could be extended to propose distributed networking management for DCIs. To analyze the implications of this statement deeply and accumulate insights about the distributed management, we have described sixteen solutions leveraging distributed principles in the previous chapters. Twelve of these proposals are SDN controllers, and the other four are in-resource managers solutions.

For each proposal, we present a qualitative analysis and summarize their characteristics in multiple tables. We consolidate these tables at Table 8.1 and Table 8.2 respectively.

Solutions such as FlowBroker, D-SDN, Tungsten, Kandoo, Kubefed, and Tricircle use a hierarchy of controllers/instances to gather networking states and maintain a global view of the infrastructure. To avoid the root controller's SPOF issue (see Section 4.2), 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 nodes (*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 DB, enabling controllers to maintain and share global networking information easily. While it is one more step to fulfill the system's requirements, these systems' efficiency depends on the DB system's capabilities. Even if dedicated systems have been designed for some of them (*e.g.,* ONOS), they do not

Table 8.1 – Classification of surveyed 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)	✓
ODL (Fed)	✓		✓				✓	AMQP	-	-	In-memory	✓	✓	OpenFlow, BGP & others	Proven system (TRL 9)	✓
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)	✓
Tungsten	✓			✓		✓		BGP	IFMAP	DB-in	NoSQL DB	✓	✓	XMPP, BGP & others	Proven system (TRL 9)	✓
Other solutions																
Kubefed				✓		✓		-	API	-	NoSQL DB		✓	-	Technology validated (TRL 5)	✗
Istio Multi		✓					✓	API	-	-	NoSQL DB		✓	-	Proven System (TRL 9)	✗
P2P agents		✓					✓	A2A interface	-	-	SQL DB		✓	-	PoC (TRL 3)	✓
Tricircle				✓		✓		-	API	-	SQL DB	✓	✓	-	System prototype (TRL 7)	✓

Proposals	Organization of network information			Inter-site networking resources 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	~	?	?	~	~
ODL (Fed)	✓	✓	~	✓	✓
Onix	~	?	?	~	~
ONOS	~	?	?	~	✓
Tungsten	✗	✗	?	✗	✓
Other solutions					
Kubernetes Federation	✗	~	✓	✗	✗
Kubrnetes Istio Multi-Cluster Service Mesh	✓	✓	~	✗	✗
OpenStack P2P external proxy-agents	~	~	✓	✓	✗
OpenStack Tricircle	~	?	?	~	~

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

Table 8.2 – Summary of the analyzed solutions.

cope with the requirements we defined in terms of data locality awareness or network partitioning issues.

The remaining systems, *i.e.,* DISCO, ODL, Istio Multi-Cluster, and P2P proxy-agents, propose a fully distributed architecture (*i.e.,* without the need for a global view). Istio Multi-Cluster falls short due to its lack of horizontal automation. As we explained in Section 7.4, the user must declare by hand the inter-site resources at each one of the deployments. The OpenStack P2P proxy-agents solution also falls short because it does not target the distributed networking management case, but rather, it is a massively distributed proxy for scaling.

In the case of the DISCO and ODL controllers, 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 the case 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 resource manager (*i.e.,* it delivers mainly domain-forwarding operations, which includes only conflict-less exchanges). Offering the resource manager's expected functions (such as dynamic IP assignment) is prone to conflict and might be harder to implement in such an architecture. We discussed this point for ODL, which has many 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 management but at a level that does not enable it to solve conflicts. Leveraging the DISCO or ODL architecture and investigating how to avoid conflicts is a prominent insight identified thanks to this analysis.

As we outlined, the East-West interface proposed by DISCO and ODL provides some references to design an efficient horizontal interface for inter-resource managers 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 resource managers.

If we consider the model proposed by DISCO and ODL, the use of independent and local DBs implies managing consistency at the application level (*i.e.,* between the different controllers). It 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.

8.2 Summary

In this chapter, the analysis of retained lessons from multi-instances propositions has been presented to gain insights about distributed networking management for DCIs.

Proposals leveraging hierarchical approaches still present the SPOF and bottleneck issues, which cannot entirely address the DCI networking challenges. Solutions presenting physically distributed but logically centralized designs usually depend on distributed DBs to exchange information among the instances. This approach's principal drawback relies precisely on the used DB system, which may not guarantee the locality awareness property. Even more, it may present flaws in the case of network partitioning. Solutions relying on a fully distributed architecture respect the principle of locality awareness thanks to the architecture's nature. Moreover, independence among instances let the architecture to be resilient enough against networking partitioning. The downside of this approach relates to how the sharding strategies are conceived to avoid conflicts among the resource managers and how collaboration is done to minimize the management information exchange.

Therefore, the following part of this work introduces our proposal to provide DCI networking management leveraging the retained distributed principles.

Part III

DCI networking: Going the distributed way

This third part introduces DIMINET, our proposal to provide inter-site networking connectivity for DCIs.

— *Chapter 9 discusses essential design choices, and introduces our DIMINET implementation for the OpenStack ecosystem. On account of the industrial nature of this thesis, Orange has privileged OpenStack as the main technological choice for the developments in our work because of its maturity and its use within the enterprise as VIM.*

— *Chapter 10 introduces DIMINET functionalities validation and large-scale tests done using the test-bed Grid'5000.*

Chapter 9

Distributing connectivity management with DIMINET

This chapter introduces the fundamentals of DIMINET, a distributed architecture that aims to provide networking management for DCIs that leverages a distributed module for inter-site networking resources management. In Orange's interest to explore the OpenStack ecosystem, we have implemented DIMINET as a module deployed besides Neutron. For this reason, we inform the reader that this chapter includes both, theorical and technical information about DIMINET. While DIMINET has been developed for OpenStack, its abstractions can be used for other cloud services as we present in the conclusions. The chapter is composed as follows. Firstly, an overview of DIMINET's architecture is presented. After that, we detail DIMINET internals, explaining the chosen sharding strategies for inter-site networking resources. The chapter then details DIMINET data model, along with the communication interfaces. For the sake of clarity, we also introduce Neutron Interconnections and BGP-based VPN Service Plug-ins, the Neutron technologies used to provide data plane connectivity among DCI sites. We have preferred to include some of DIMINET technical details as appendices. Appendix A presents the sharding strategies implementation, Appendix B includes the list of DIMINET API operations, and Appendix C presents the installation guide of DIMINET with OpenStack.

9.1 Leveraging Retained SDN Principles

We saw in the previous chapter that distributed SDN principles adopted by controllers such as DISCO and ODL are both well appropriated for the DCI context. Notably, they address most of the DCI networking challenges and guarantee fundamental properties such as locality awareness and resiliency against network partitions. The logical step is then to leverage one of them to implement the inter-site networking connectivity management for DCIs within a resource manager. Because of the industrial nature of this doctoral work, one requirement is to use OpenStack as the resource manager for the inter-site connectivity management development efforts. Therefore, the compatibility with OpenStack of these

two solutions becomes a fundamental criteria.

Unfortunately, the DISCO code is not publicly available, and as we described in Section 5.1, DISCO's evaluations were performed on a PoC and the controller does not provide compatibility with OpenStack. On the other hand, ODL is one of the most mature SDN controllers in the industry with a large community supporting it, and with already-built Neutron connectivity capabilities. However, as we explained in Section 6.2, Neutron instances are unconscious about the information exchanged between controllers due to the low-level interaction with ODL that does not allow the controllers to implement coordination mechanisms to prevent coherence errors.

For these reasons, we decided to leverage these insights from SDN controllers by proposing a distributed module for inter-site networking management called DIMINET. DIMINET extends retained SDN principles of DISCO and ODL (*i.e.,* fully distributed architecture with East-West communication). On each site composing a DCI, a module is deployed at the local site's networking management service. This module can communicate with remote modules, on-demand, to provide virtual networking constructions spanning several resource managers.

9.2 DIMINET's Architecture Overview

This section gives an overview of DIMINET's architecture. As shown in Figure 9.1, DIMINET is fully decentralized: each DIMINET instance is deployed besides a local resource manager networking service. This architecture guarantees DCI characteristics, as explained as follows.

Scalability: New DIMINET instances representing remote sites can smoothly join the deployment without affecting other instances' normal behavior.

Resiliency: Because of the fully distributed architecture, DIMINET does not present the centralized architecture limitations. This means that in the case of network partitions, as every DIMINET instance and its respective resource manager are independent of the others, they will continue to provide, at least, their cloud services locally.

Locality awareness: Because of its East-West communication between instances that happens only on demand, DIMINET does not build a global knowledge but instead relies on the collaboration among instances to share the necessary inter-site resource-related information.

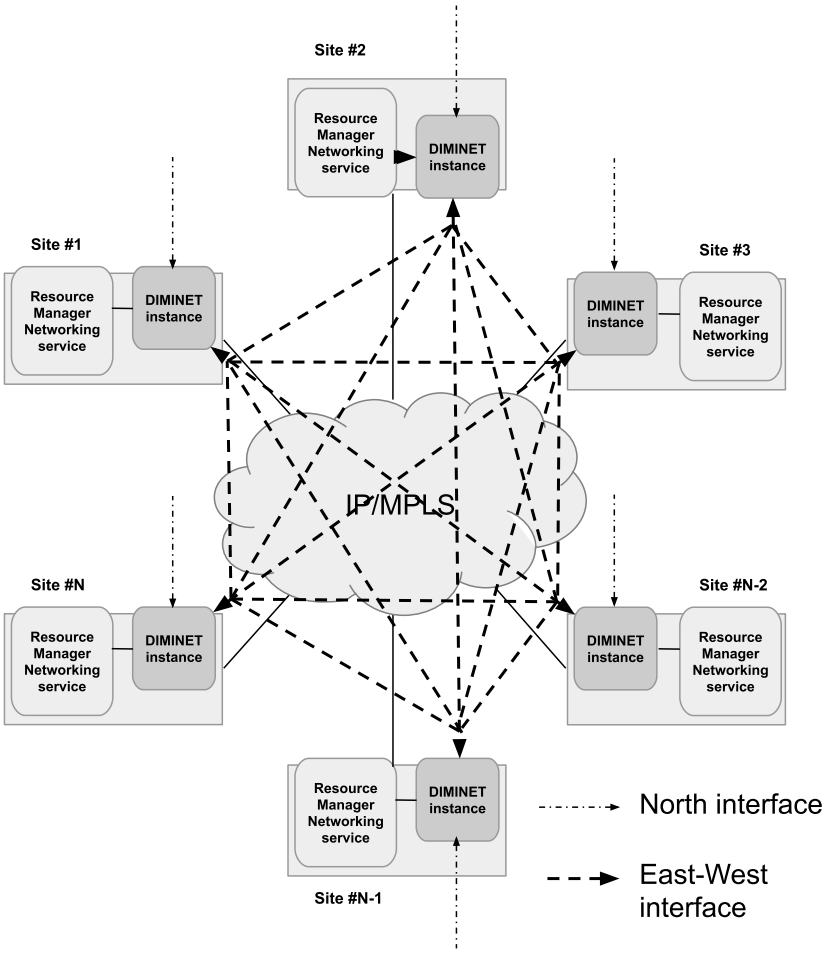


Figure 9.1 – DIMINET overview.

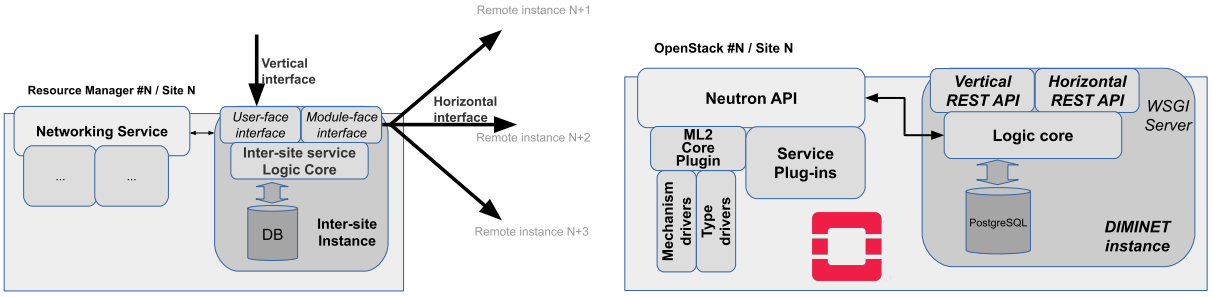
Abstraction and automation: Thanks to its communication interfaces, DIMINET realizes the creation and configuration of inter-site resources automatically without further actions needed from the user besides the initial resource creation request.

Figure 9.2(a) depicts more in detail the internal architecture of a DIMINET instance. Each instance comprises the Logic Core which implements the necessary strategies to manage and deploy inter-site resources (explained in details at Section 9.3), and the communication interfaces, one for end-users and one for collaboration with remote DIMINET modules (explained in details at Section 9.4).

We implemented a first PoC of DIMINET as a module deployed beside the networking service of OpenStack, Neutron, as depicted in Figure 9.2(b). This approach enabled us to keep the collaboration code outside the Neutron one.

Each DIMINET module runs as a Web Server Gateway Interface (WSGI) server listening to users' requests on port 7575. We have used Python's Flask framework to build up this implementation since it provides a similar code structure as the one proposed by OpenStack code. Following OpenStack's general approach, we decided to implement the North and East-West interfaces as REST API interfaces to provide a server-level list of operations to the users and the other modules. The Logic Core runs over the WSGI server and uses a PostgreSQL DB to store the inter-site resources information.

Considering that DIMINET modules are arranged in a P2P manner, the join and leave methods are essential to allow instances finding neighbors on the system. For practicality, since we use OpenStack as resource manager we did not implement these methods in our implementation. Instead, we rely on the Keystone service (users authentication, service discovery, and authorization as explained in Section 2.1.4.1) to find out distant DIMINET instances knowing that they are deployed in the same IP address as Neutron.



(a) DIMINET's internal architecture

(b) DIMINET implementation besides OpenStack Neutron.

Figure 9.2 – DIMINET internal design.

9.3 DIMINET's Logic Core

The core of DIMINET is the *Logic Core*, which is in charge of the actual management and coordination of inter-site resources including, when required, communication with other DIMINET instances and with the resource manager's networking service.

9.3.1 Resources Sharding Characteristics

To effectively address both the information granularity and the availability challenges detailed in Section 3.3.1, the information sharding strategy for each resource is defined in the Logic Core. In the following, we present our sharding strategies for L3 routing (*i.e.*,

being able to route traffic between a VN A on site 1 and a VN B on site 2) and L2 extension (*i.e.,* being able to have a Layer 2 VN that spans several sites) resources (See Section 3.2).

9.3.1.1 L3 Routing Resource

The inter-site Layer 3 routing resource is provided for traffic to be routed among different subnetworks. By design, subnetworks should not overlap. That is, the range of addresses in one subnetwork should be unique compared to all other subnetworks. If two subnetworks overlap, when a *router* needs to send a packet to an IP address inside that range of overlapped addresses, the router may forward the packet to the wrong subnetwork. In this context, the subnetworks CIDRs must not overlap.

Let be $\{SN_1, SN_2, SN_3, ..., SN_{n-1}, SN_n\}$ a set of independent subnetworks deployed on n resource managers sites which are requested to have an L3 routing resource among them. The condition $\bigcap_{i=0}^{n} SN(CIDR)_i = \emptyset$ (the sets of subnetworks CIDRs have to be disjoint sets) should be verified to accept the L3 routing request.

This verification should be done by the first instance that receives the L3 routing resource creation request. Once the user provides the resources to interconnect in a Layer 3 routing resource and the sites where they belong, the first instance should proceed to query the network information from every site listed in the user's request to ensure that the IP ranges are not overlapping among them. Once this condition is verified, the process of information exchange is launched among the listed sites to allow the low-level mechanism to do the virtualized traffic forwarding.

For example, suppose the user requests the DIMINET instance of resource manager 1 to instantiate a Layer 3 routing service among two networks A and B, belonging to resource managers 2 and 3 respectively. In that case, this DIMINET instance will contact the remote site to find the subnetwork CIDR related to the remote network, and of course, it does the same search locally. Consider the IPv4 CIDRs 10.1.2.0/23 and 10.1.4.0/23 for network A and B, respectively, as depicted in Figure 9.3. The DIMINET instance will do the overlapping verification with the ranges [10.1.2.0-10.1.3.255] for 10.1.2.0/23 and [10.1.4.0-10.1.5.255] for 10.1.4.0/23.

Thus, $10.1.2.0/23 \bigcap 10.1.4.255/23 = \phi$ (the two subnetworks do not overlap). Since the condifiton is verified, DIMINET instance 1 will send a resource creation request to instance 2 with the information of the two resources and the type of resource (*i.e.,* L3 routing in our example). Then, instance 1 proceeds to advertise its routing information with instance 2.

When CIDRs overlap, DIMINET does not accept the request and notifies the user

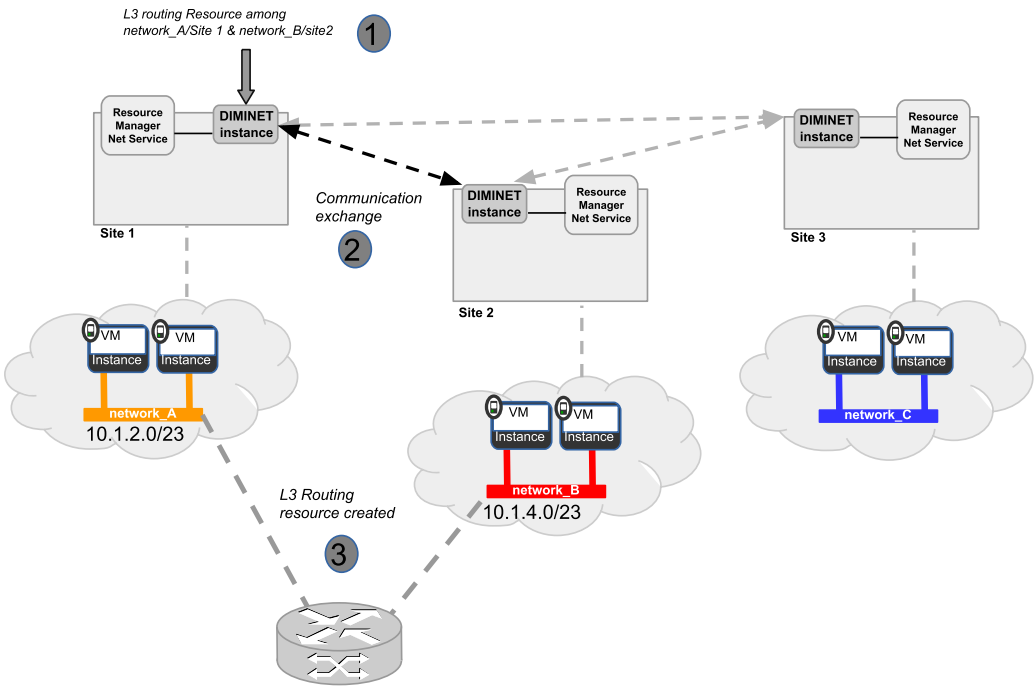


Figure 9.3 – DIMINET L3 Routing Resource.
(1) Resource creation request. (2) Information exchange among DIMINET instances. (3) L3 Routing deployed.

that the resource cannot be provided due to overlapping subnetworks CIDRs.

9.3.1.2 L2 Extension Resource

The inter-site Layer 2 extension resource gives the possibility to plug into the same virtual network, VMs belonging to different sites. To belong to the same virtual network, hosts must have the same subnetwork prefix (CIDR) and do not have duplicate MAC or IP addresses. Since every network exists as an independent network in each site, they can each have their DHCP service for IP assignment. Thus, MAC and IP assignments have to be coordinated among the requested sites for the L2 connectivity to be correctly provided.

In the proposed solution, two operations need to be considered over VNs: the join and the extension. The join operation refers to combining multiple independent L2 resources to create a single L2 resource. It implies that every independent L2 resource could have already deployed VMs on it. Suppose the join operation is to be applied between two resources. In that case, it will be potentially necessary for each resource manager to change the IP addresses already allocated and thus, interrupting the services provided by

those VMs, which is not desirable in operational environments.

On the other hand, the extension operation refers to expanding one of the L2 resource into the other sites to create a single L2 resource. This implies that these remote resources will be freshly created to make the initial request. Since this last operation does not impact every segment's behavior, we preferred to use it in our design.

For this reason, we have decided to propose the following approach:

— The instance receiving the initial L2 extension resource creation request assumes the role of master for that particular resource.

— This master instance does a logical split of the local resource's range of IP addresses within the same CIDR between all the sites specified by the user among which an L2 extension resource is to be created.

In this sense, the master instance will be in charge of providing the IP allocation pools to the other instances composing the L2 extension resource, and thus, to do the L2 extension. To avoid spending all the IP addresses from the first resource request, the master instance delivers mid-sizes allocation pools to the other participants. If, in any case, one of these instances needs more IP addresses or a new DIMINET instance joins to compose the inter-site resource, the master will provide a new allocation pool. If the total range of IP addresses are spent, the master will not accept new sites to join the inter-site L2 extension resource.

With this approach, we will avoid the communication overhead of sharing the information between the concerned resource managers each time an IP address is allocated to each resource manager. At the same time, we will avoid only doing a CIDR allocation pool static division at the resource creation time. This approach will allow the instances to maintain a segment logic division while providing a more dynamic sharding strategy.

With our approach, if the user requests the DIMINET instance of resource manager 1 a Layer 2 extension resource to a site 2 as depicted in Figure 9.4, this DIMINET instance should contact the instance of site 2 to verify that a subnetwork with the same characteristics (*e.g.,* CIDR) can be created. If so, the instance of site 2 should create the corresponding subnetwork, and the instance of site 1 will take the role of master of that specific L2 inter-site resource. This implies that this instance will decide how to do the CIDR IP allocation pool among the participant sites for the request and to manage further requests concerning the resource's modification.

This information of the master instance and the allocated IP range will be sent through the East-West interface to remote instances sharing this L2 inter-site resource. When receiving the L2 creation request, remote instances will store the resource information in the local database (Further details in the next paragraph). They will then use the L2 exten-

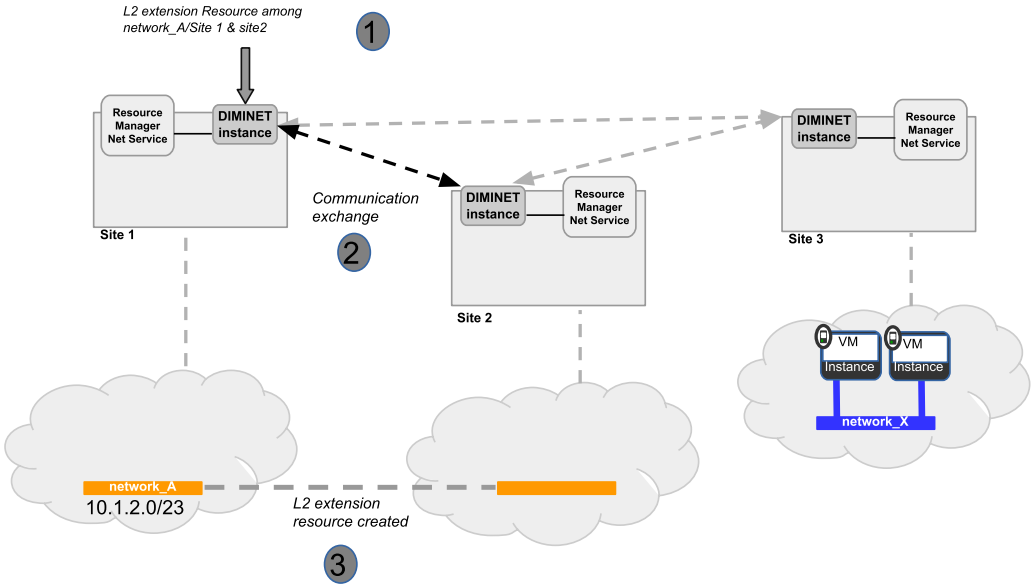


Figure 9.4 – DIMINET L2 Extension Resource.
(1) Resource creation request. (2) Information exchange among DIMINET instances. (3) L2 extension deployed.

sion resource-related information dictated by the master instance to do the appropriate changes in local networking constructions (*i.e.,* change the local IP allocation pool). These changes are also done in the master site to provide the logical division. Once this is done, the instances proceed to exchange the necessary routing and data plane information to allow VMs traffic to be forwarded among them.

The Logic Core has been implemented as a Python program running over the WSGI server as explained in Section 9.2. Moreover, Appendix A explains how each one of the sharding strategies (*i.e.,* for the L3 routing and L2 extension resources) is implemented, detailing the step by step processing of an inter-site resource creation request.

9.3.2 Data Model

The Logic Core stores inter-site resource information in a local database at each instance. To relate the same inter-site resource stored in different locations, the Logic Core generates a globally unique identifier that will identify the same resource either in Site 1 or Site N of the sites composing the resource. This global identifier will be created at the DIMINET instance that receives the initial user vertical request and transmitted to remote sites inside a East-West creation request. In this way, all sites will be capable of referencing the same inter-site resource.

We emphasize that there is a Master in charge of maintaining the consistency of the related information for each inter-site *Resource*. The Master is defined as the DIMINET module receiving the initial *Resource* creation request in our current model. The use of a per-*Resource* Master enables DIMINET to deal with network partitions for inter-site *Resources* straightforwardly: When an end-user request cannot be satisfied due to network issues (*i.e.*, either a remote site cannot be reached or reciprocally when a remote site cannot interact with the Master), the request is revoked, and the user is notified of the impossibility to serve the request.

Figure 9.5 shows the schema of the objects used by the Logic Core to represent an inter-site resource using an Entity Relationship (ER) diagram with crow's foot notation [152].

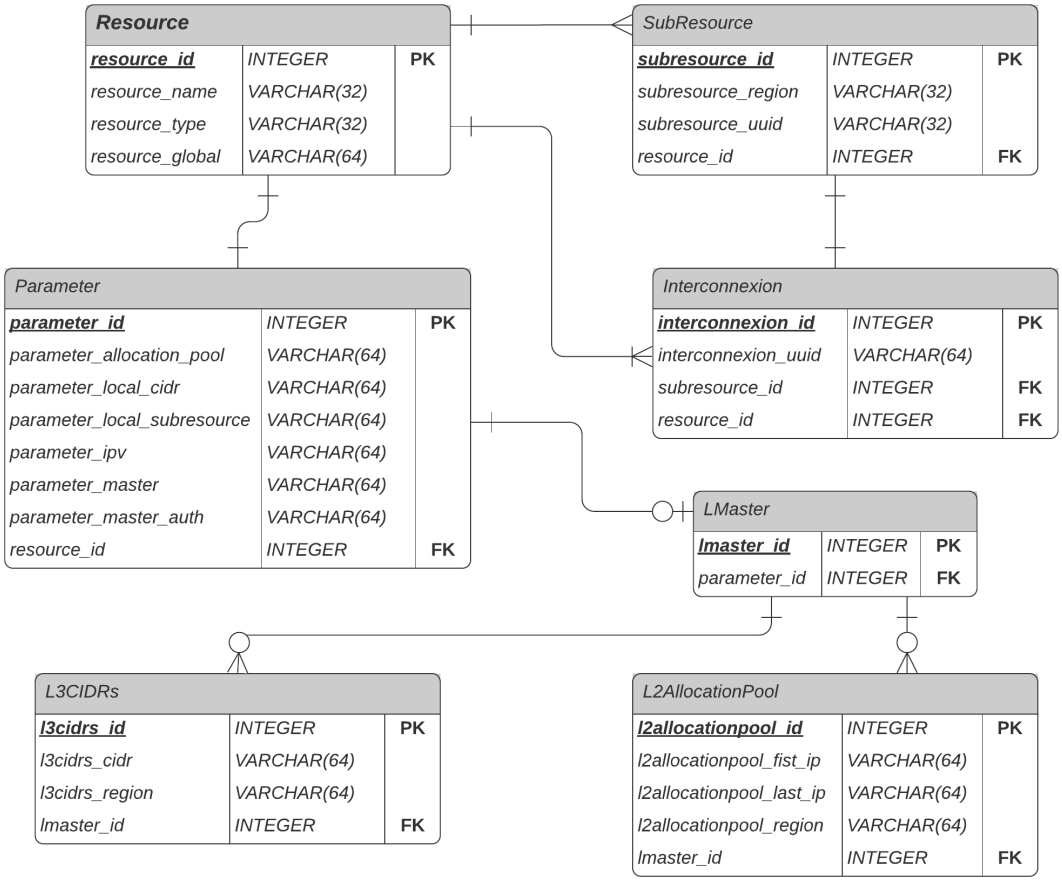


Figure 9.5 – DIMINET data model.

Resource: The main object of DIMINET. It represents an inter-site networking

resource. In our context, an inter-site *Resource* is a logical resource that exists at several locations simultaneously and needs to be coordinated not to affect the correct functionality of the networking constructions (See Section 3.3.1). A *Resource* is related to a unique *Parameter* instance, a list of *SubResources*, and a list of local *Interconnections*.

Parameter: As we already mentioned, since every proposed inter-site resource has its own needs, it is necessary to store different information per *Resource* type. The *Parameter* class is used to store *Resource*-related details to support the Logic Core's main functionalities. If, for instance, the *Resource* is of L2 extension type, it will store the IP allocation pool assigned by the master instance.

SubResource: A *SubResource* represents a virtual networking object belonging to a site's resource manager (*i.e.,* a network Universally Unique Identifier (UUID)). The *Resource* class holds a list of *SubResources* (the local one and a series of remote ones).

Interconnection: An *Interconnection* represents the mechanism enabling the interconnection for *SubResources* to contact or be contacted by remote resource managers to forward/route virtualized traffic. Unlike *SubResources* objects, *Interconnections* are only stored locally. The Interconnection holds all the necessary technical parameters to effectively implement the inter-site *Resource*. The parameters stored will depend on the technical implementation to establish the inter-site *Resource*.

LMaster: To maintain the consistency among modules while trying to diminish the burden of adding consensus protocols, DIMINET relies upon a per *Resource*-Master mapping that allows reducing the risk of SPOF and bottlenecks. The class *LMaster* is in charge of storing consistency-related information about the networking constructions. This class is instantiated only at a *Resource* master module (*i.e.,* the DIMINET module receiving the initial *Resource* creation request).

L2AllocationPool: An *L2AllocationPool* object stores information of IP allocation pools provided by the Master module for the L2 network extension Resource. This class is only instantiated at a *Resource* master module.

L3Cidr: An *L3Cidr* object stores information of the *SubResources* CIDRs composing an L3 routing *Resource*. This class is only instantiated at a *Resource* master module.

9.4 Communication Interfaces

In order to process end-users requests and allow communication among resource managers, DIMINET relies on two distinct interfaces inspired from the DISCO and ODL SDN controllers. The North interface and the East-West interface as depicted in Figure 9.1. These two interfaces interact with the Logic Core to automatize the inter-site resource provisioning.

The interfaces propose Create Read Update Delete (CRUD) operations on inter-site resources. The CRUD actions of both interfaces and their explanation are summarized in Table 9.1. Appendix B provides the entire list of DIMINET REST API operations.

North Interface			
Operation	*Prefix*	*Parameters*	*Description*
POST	/	Name Type SubResources	Create a new Resource
GET	/		Retrieve local information of all Resources
GET	/*global_id*		Retrieve local information of one Resource with identifier *global_id*
PUT	/*global_id*	Name SubResources	Modify a Resource with identifier *global_id*
DELETE	/*global_id*		Delete a Resource with identifier *global_id*
East-West Interface			
Operation	*Prefix*	*Parameters*	*Description*
POST	/	Global_id Name Type SubResources Parameters -Allocation pool -Local CIDR -IPv -Master -Master auth	East-West request to create a Resource
GET	/		East-West request to retrieve the list of Resources
GET	/	Resource_CIDR Resource_type Global_ID Verification_Type	East-West request to retrieve a single Resource with identifier *global_id*
UPDATE	/*global_id*	Name SubResources Type Post create refresh	East-West request to modify a Resource with identifier *global_id*
DELETE	/*global_id*		East-West request to delete a Resource with identifier *global_id*

Table 9.1 – DIMINET CRUD Operations

North Interface

The north or vertical interface allows the user to request the establishment of inter-site networking resources among several sites. This interface exposes an service-level list of operations to enable the user to execute CRUD actions on inter-site resources.

East-West Interface

Once the DIMINET instance receives an inter-site resource provisioning request from the user using the North interface, it initiates a communication with the requested remote DIMINET instances using the East-West interface.

The exchanged information should be both the logical information to do the distributed management of the networking constructions and the necessary low-level information required for the implementation of the data plane.

As stated above, the exchange on the East-West interface occurs only among DIMINET instances involved in each inter-site resource (*i.e.,* no broadcast-like communication are required in this interface). In other words, contacting only the relevant sites for a request will mitigate the network communication overhead and the limitations regarding scalability and network disconnections.

9.5 Data Plane Traffic Exchange

In addition to the logical information on *Resources*, DIMINET instances need to exchange also the information related to the reachability of virtualized traffic connectivity mechanisms to provide ultimately data plane connectivity.

At this point, two possibilities can be considered. The first one is to exchange this information over the East-West interface and to proceed with the data plane interconnection. By doing this, DIMINET instances will need to have the capacity to control the low-level technological mechanisms to implement the forwarding logic (*i.e.,* configure virtual switches, add flows, create tunnels). The second possibility is to rely on already existing technologies proposed by the local resource managers (*e.g.,* in the OpenStack case, Neutron's service Plug-ins). In this case, DIMINET instances will need to interact with the resource manager to implement these inter-site connectivity mechanisms.

As we proposed DIMINET to be deployed besides Neutron, we do not implement the information exchange for the virtualized traffic connectivity over the East-West interface. Instead, we rely on the Interconnection Service Plug-in [153] developed by Orange to contribute to the OpenStack community.

The Interconnection Plug-in allows creating an "interconnection" resource that references a local resource having the semantic informing that connectivity is desired with a remote "interconnection" resource that references a remote resource as depicted in Figure 9.6. This Plug-in assumes that coherence is taken care of by the user (*e.g.,* the user chooses IP addresses consistently across the two sites). By itself, it does not provide any kind of sharding logic, but rather a technical mechanism to interconnect two resources belonging each one to a different OpenStack deployment. The process depicted in the Figure 9.6 is composed of four main steps:

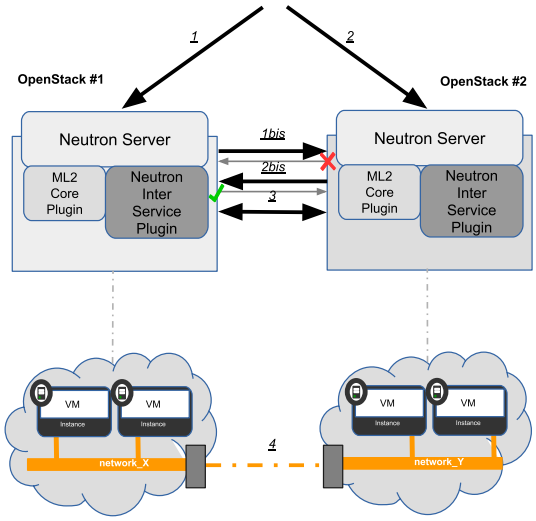


Figure 9.6 – Neutron-to-Neutron Interconnection Plug-in.

1. The user requests Neutron 1 to create an interconnection resource among network A in local VIM1 and a remote resource network B in VIM2.

1bis. Neutron 1 checks that a symmetric interconnection exists on the remote Neutron 2 and will not proceed further until this becomes true.

2. The user request to Neutron 2 the creation of an interconnection resource among network B in local VIM2 and a remote resource network A in VIM1. This object is the symmetric interconnection resource.

2bis. Neutron 2 checks that a symmetric interconnection exists on the remote Neutron 1. Since this resource has been defined in step 1, it allocates the required network identifiers (*i.e.,* allowing the mechanism to be reached).

3. Both Neutron retrieve the network identifiers of the mechanisms used for the interconnection.

4. The interconnections is configured and in active state.

The process described needs to be done for each pair of Interconnection objects. If the user wants to connect N OpenStack sites, it will be necessary to do N API calls per site by hand. In total, the user will be doing $N * N$ API requests. DIMINET automatizes this process thanks to the communication among modules: By doing an API *Resource* creation request to a DIMINET instance, this module will communicate with all the requested sites to create the Interconnections objects, reducing the total of API calls that the user has to do from $N * N$ to a single one.

The Interconnection Plug-in remains agnostic to the network technique that will be finally used to realize the connectivity and data plane traffic exchange. It allows the user to specify at each OpenStack configuration the possible techniques to use when establishing and creating Interconnections. In its initial proposition, the Neutron Interconnection Plug-in leverages the use of Border Gateway Protocol based Virtual Private Networks (BGPVPNs) [48] at both sides to create an overlay network connecting the two local segments.

The BGPVPN Service Plug-in itself uses the well-known networking protocol BGP for the establishment of Internet Protocol Virtual Private Network (IPVPN)/Ethernet Virtual Private Network (EVPN) [154, 155]. In BGP-based Virtual Private Networks (VPNs), a set of identifiers called *Route Targets* are associated with a VPN. Similarly to the publish/subscribe pattern, BGP-peers use an export and import list to let know the interest of receiving updates about announced routes. Each site have a set of independent identifiers called *Route Target*, a Multi Protocol Border Gateway Protocol (MP-BGP) extended community attribute that is attached to the network constructions and exchanged employing a BGP implementation.

This exchange can be done at the underlay layer, using physical equipment, or at the overlay level, using BGP software implementations such as GoBGP. A *Route Target export* identifier is used for advertising the local routes of the VPN to the other BGP-peers. On the other hand, a *Route Target import* identifier is used to import remote routes to the VPN. For instance, Figure 9.7 shows net_A from VIM 1 and net_B from VIM 2 that compose an Interconnection resource A and that belong to the same BGPVPN. Each site will have the following information to exchange their BGP routes: site A will have *route-target-export 64512:4000* and *route-target-import 64512:6000*, while site B will have *route-target-export 64512:6000* and *route-target-import 64512:4000*.

When this information is exchanged among the BGP-peers, each one will share and receive the routes associated with the respective *route-target* identifiers as mentioned above. For instance, the BGP-peer of VIM 1 will share the routes associated with net_A

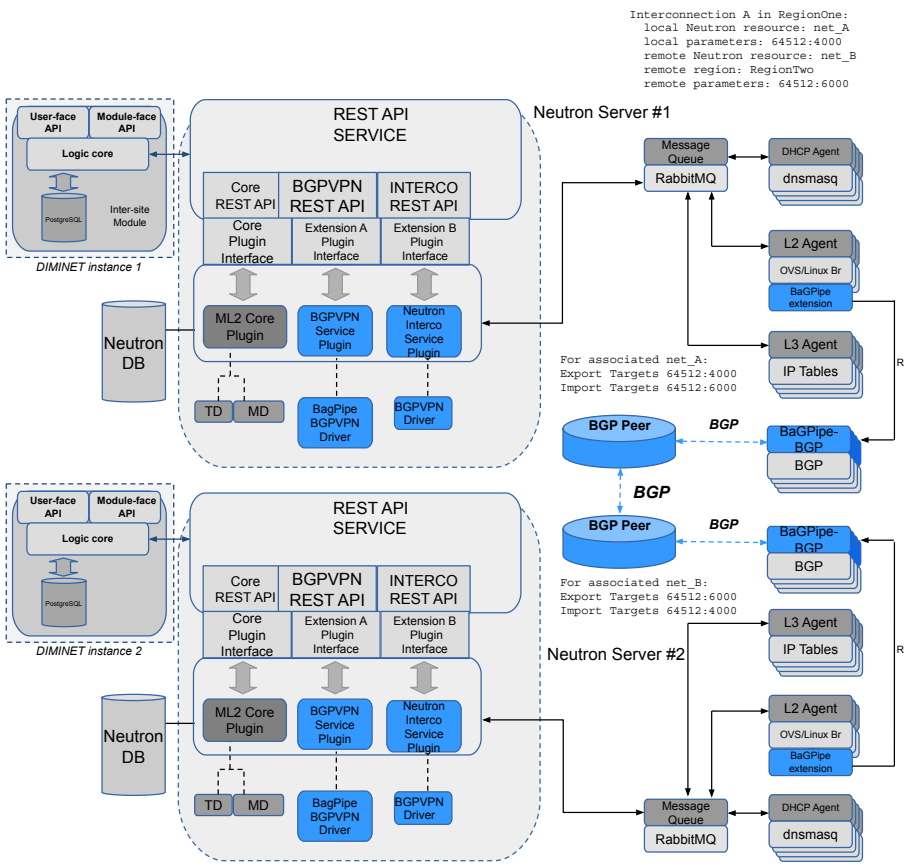


Figure 9.7 – Neutron BPG-VPN Plug-in.

providing a BGP UPDATE message with some information such as the route's MAC and IP addresses, *route target*, tunnel endpoint, and encapsulation type.

These routes are then shared at each site with *Bagpipe BGP*, a light implementation of BGP used to establish BGP sessions. A bagpipe extension is deployed on compute nodes and will receive the routes from *Bagpipe BGP* to populate the data plane accordingly. Data plane traffic is exchanged using an overlay encapsulation, with VXLAN at the typical choice for a virtual switch to virtual switch communication.

9.6 Summary

In this chapter, we have presented a distributed architectural design to provide inter-site networking constructions management leveraging fully distributed modules deployed at each resource manager of the DCI. DIMINET takes inspiration from distributed SDN principles of DISCO and ODL controllers, such as the use of an East-West interface

for on-demand communication between modules to provide virtual networking constructions spanning several resource managers. We have presented a fully automatized implementation of DIMINET to provide inter-site networking resources management in an OpenStack-based DCI. DIMINET modules run as WSGI servers deployed besides Neutron instances to propose integrated networking management of networking connectivity. By relying upon a collaboration of independent modules only on-demand, DIMINET addresses the limitations of hierarchical and logically centralized architectures in terms of availability in the case of network partitions.

We also proposed per-*Resource* type sharding strategies (i.e. depending on whether the *Resource* is an L3 routing or an L2 extension) to provide distributed management and coherence at the application level (Appendix A presents the sharding strategies implementation). Consequently, each DIMINET module has its own local database, and all the communications with the user and other modules are done using the North and East-West interfaces, implemented as REST API interfaces like other OpenStack services (Appendix B provides the complete list of DIMINET REST API operations). To avoid the overhead of adding consensus protocols to manage the coherence, we decided to declare Master of the *Resource* the first DIMINET module receiving the user's creation request. DIMINET's model still uses the notion of a static per *Resource* Master. However, every DIMINET module can act as Master when contacted to create a *Resource*, reducing the SPOF problem. A future work in this field could involve the possibility to change the Master role among modules to avoid the static Master. Finally, Neutron Interconnections and BGPVPN Plug-ins are leveraged to provide the data plane connectivity among sites since this mechanism is already fully supported by OpenStack. Appendix C proposes a guide to install DIMINET along with OpenStack.

DIMINET's development effort involved more than 3000 lines of Python code without counting the Graphical User Interface (GUI) HTML and javascript code. Its initial commit dated from 1 July 2019 and has been in continuous modifications since then. DIMINET proposal has been presented at the *2019 Journées Cloud*, and more recently at the *2020 Open Infrastructure Summit* where it has been shared with the OpenStack community.

Beyond the technological solution proposed for Orange, we believe DIMINET design is abstract enough to understand the challenges related to cloud services distributed management. Indeed, the distributed nature of DIMINET allows guaranteeing the different DCI properties (highlighted in Section 3.1.1) and at the same time addressing the DCI networking information's challenges (see Section 3.3):

Information granularity: DIMINET proposes to divide the DCI architecture into several independent domains, each one managed by a DIMINET instance with its

local information. Moreover, DIMINET implements sharding strategies to provide a distributed management of inter-site networking constructions, providing these mechanisms depending on the network construction characteristics.

Information scope: To avoid heavy synchronization needs when maintaining global knowledge, DIMINET only contacts the relevant neighbors on-demand to provide inter-site *Resources* when requested by the user.

Information availability: Because DIMINET instances are intended to be deployed besides independent resource managers, in case of network disconnections, these resource managers will provide local services without any problem. Non-disconnected sites will still be able to use their DIMINET instances to provide inter-site *Resources*. Furthermore, since DIMINET implements sharding strategies for inter-site *Resources* on creation when connectivity is reestablished, there will be neither conflicts among DIMINET instances nor between resource managers.

Automatized interfaces: DIMINET proposes well-defined and fully-integrated REST API interfaces to communicate with users and other modules. The Logic Core takes charge of using these interfaces to completely fulfill a user's request transparently. In the case of the OpenStack-based implementation, the way DIMINET's *Resources* operations are exposed allows an easy understanding of the *Resources* since they are similar to the resource definitions proposed at the Neutron API (see Section 2.2.1.1).

Networking technologies: While DIMINET by itself does not implement the low-level networking mechanisms to forward inter-site traffic, it is fully coupled to Neutron, which implements several networking technologies. While BGPVPN Neutron technologies are privileged to exchange data plane information among sites, it could be possible to add support for other Neutron technologies.

The next chapter presents large-scale tests we did to assess the proposed architecture and the DIMINET OpenStack implementation.

Chapter 10

Evaluation of DIMINET

This chapter presents a series of tests of DIMINET. First, we present a test of functionalities in a laboratory environment. Then, in order to assess the proposed implementation of DIMINET, we present large-scale experiments we carried out using the french testbed Grid'5000[1] *to emulate a DCI deployment.*

10.1 DIMINET with OpenStack: PoC Validation

This section presents a test of DIMINET's implementation in a development environment. The purpose of this test is to verify that DIMINET modules correctly instantiate an inter-site *Resource*, and then that the data plane is set up to allow VMs traffic exchange.

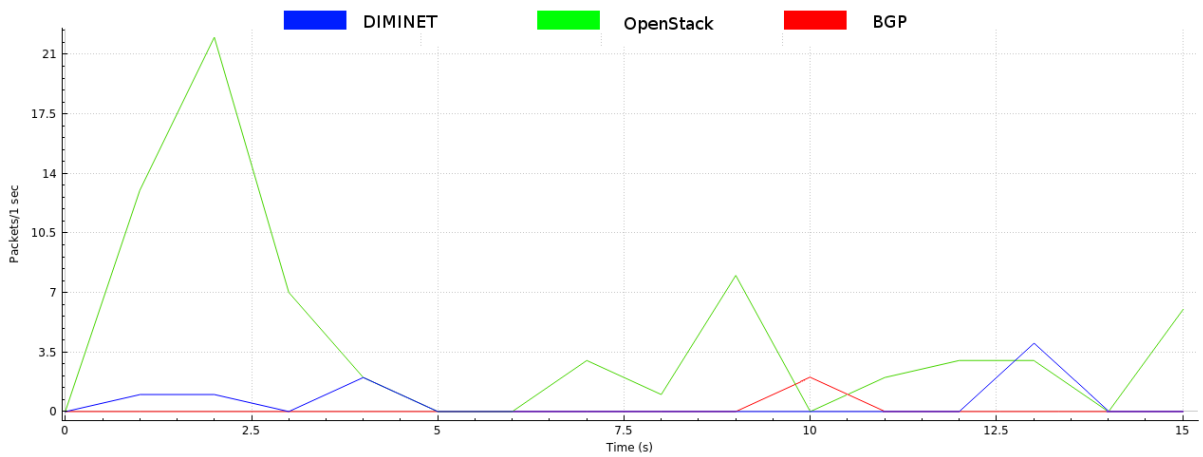
For the test, we use two Ubuntu 18.05 VMs each one with an installation of OpenStack Stein; in the following, we refer to them as OpenStack A and OpenStack B. On both deployments, a DIMINET module has been deployed beside Neutron (See Appendix C), and an instance of GoBGP is used to connect the Bagpipe-BGP peers. For the tests detailled in the following sections, we used the DIMINET's GUI interface (See Appendix C).

10.1.1 L3 Routing Resources

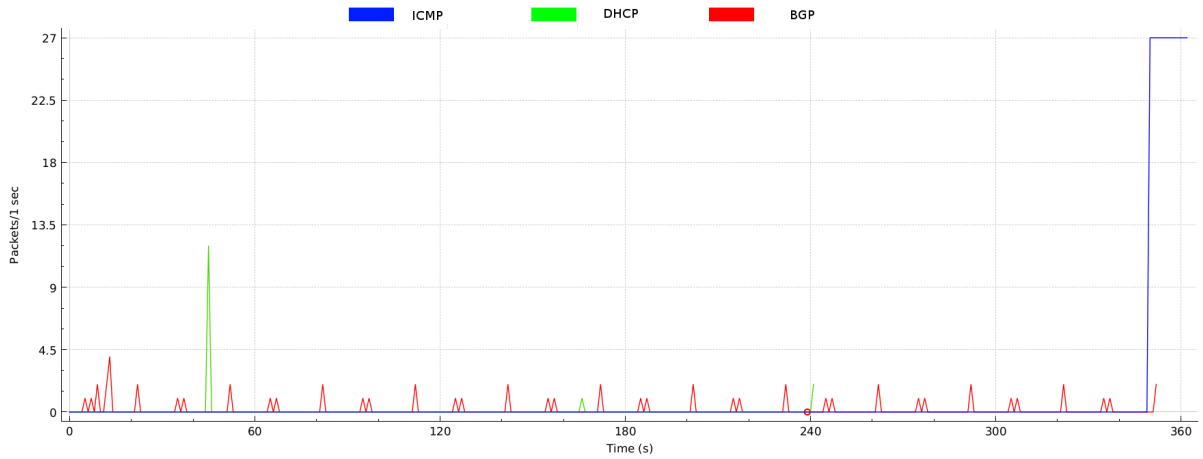
In OpenStack A we use a network net_A with CIDR 20.0.0.0/24 and in OpenStack B we use a network net_B with CIDR 30.0.0.0/24. Figure 10.1 shows the traffic captured at site A with Wireshark to provide a graphical representation of the different kinds of messages (*i.e.,* DIMINET, OpenStack, and BGP) that are exchanged.

Figure 10.1(a) shows the traffic at the moment the DIMINET L3 routing *Resource* is created at the second one with subsequent traffic to communicate with OpenStack (*i.e.,* Keystone and Neutron) and the East-West requests (See Appendix A.1.1). The BGP traffic corresponds to the *Keep Alive* messages exchanged among BGP peers. The user receives a *Resource* successful creation message at the GUI at around second four. Since

1. Experiments were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several Universities as well as other organizations.



(a) L3 routing: Logical information exchange phase



(b) L3 routing: Data plane forwarding phase.

Figure 10.1 – DIMINET & OpenStack L3 traffic capture.

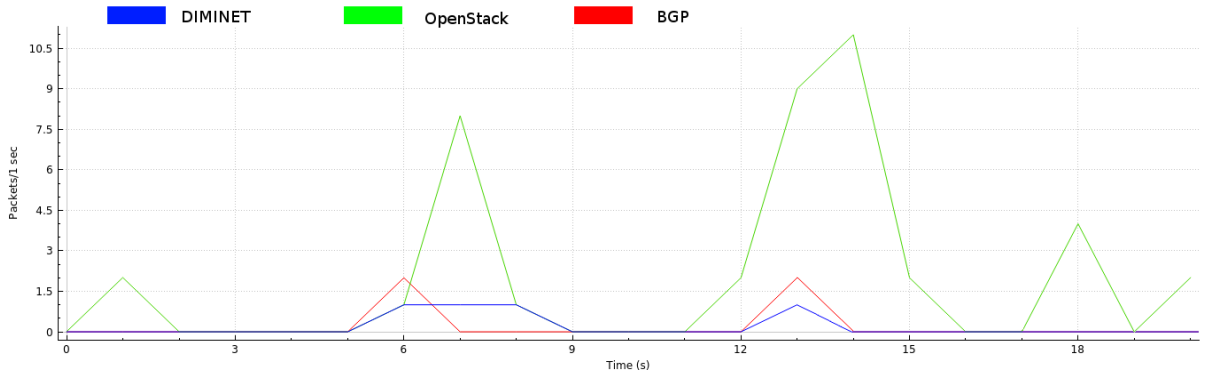
we used DIMINET's GUI to create the *Resources*, the DIMINET traffic of around second thirteen is because of the GUI update to refresh the list of *Resources*.

In Figure 10.1(b) we proceed to create a VM using CirrOS at each site (*i.e.*, net_A and net_B) composing the L3 *Resource*. The first pike of BGP traffic corresponds to the routes exchanged among BGP peers to announce the routes to reach the VMs. Then, it is possible to observe the virtual DHCP traffic allocating an IP to the VM of site A. This process is also done at site B for the second VM. Finally, when the two VMs are fully active, we execute a ping (Internet Control Message Protocol (ICMP)) request among the two VMs, represented with the great pike of ICMP traffic at around 350 seconds. Since both VMs are CirrOS machines with minimal capabilities (ephemeral hard disk and

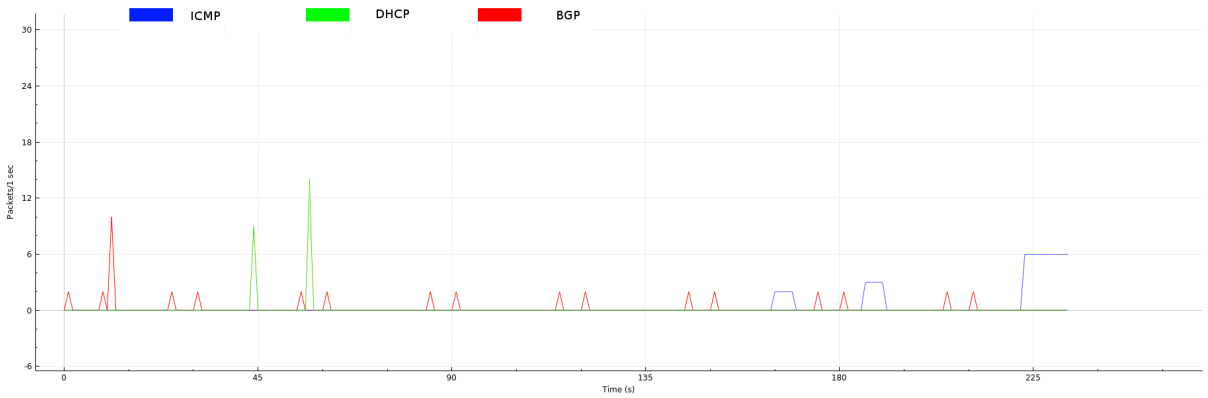
32MB RAM memory) they take a large amount of time to be booted up, VMs booted at production environments require a lower booting time.

10.1.2 L2 Extension Resources

In OpenStack A we use a network net_A with CIDR 20.0.0.0/24 to be spanned to OpenStack B. Similar to the previous experiment, Figure 10.2 shows the traffic captured at site A with Wireshark to provide a graphical representation of the different kinds of messages (*e.g.,* DIMINET, OpenStack, and BGP) that are exchanged.



(a) L2 extension: Logical information exchange phase.



(b) L2 extension: Data plane forwarding phase.

Figure 10.2 – DIMINET & OpenStack L2 traffic capture.

Figure 10.2(a) shows the traffic in the moment the DIMINET L2 extension *Resource* is created at second six with subsequent traffic to communicate with OpenStack (*i.e.,* Keystone and Neutron) and the East-West requests. Finally, at the second eight, the DIMINET Master module sends the East-West verification and creates and updates requests. We can see the augmentation in the OpenStack traffic because of Neutrons' pro-

cedure to update the state of the Interconnection objects. Similar to the L3 *Resources,* the BGP traffic corresponds to the *Keep Alive* messages exchanged among BGP peers. At this point, the user receives a *Resource* successful creation message. As in the previous test, the DIMINET traffic of around second thirteen corresponds to the GUI refresh.

In Figure 10.2(b) we proceed to create a VM using CirrOS at each site belonging to the *Resource*. The first pike of BGP traffic corresponds to the routes exchanged among BGP peers to announce the routes to reach the VMs. Then, it is possible to observe the virtual DHCP traffic for the allocation of an IP address to the VM of site A. This process is also done for the VM at site B. While the L2 extension *Resource* provides connectivity at the virtual link layer, for simplicity we execute a ping (ICMP which comprises an Address Resolution Protocol (ARP) communication) request among the two VMs. This ICMP request is started once the two VMs are fully active at around the second 225. In this test, the VMs took less time to boot up compared to the test of the previous section.

The results of the test for both type of *Resource*, L3 routing and L2 extension, show that once DIMINET modules A and B complete the exchange of logical information (*i.e.,* the inter-site *Resource* is created at each local DB), and the proper data plane interconnection mechanisms are set up, communication is effectively established among the VMs located at different sites.

10.2 Grid'5000: Testbed and Setup

To better assess DIMINET capabilities, we conducted several experiments carried out using the Grid'5000 testbed. In this section, we detail the testbed setup to emulate a DCI. Figure 10.3 shows the experimental platform: each gray box which represents a DCI site, corresponds to a physical machine of Grid'5000 with a DIMINET instance and an OpenStack deployed atop of it. Grid'5000 provides the physical connectivity for the machines, so our sites can communicate among them at the IP level. We have used Linux Traffic Control (TC) to emulate the WAN links of a DCI among the sites. Each of the OpenStack deployed uses the Stein release with the following networking services: ML2 Open Virtual Switch (OVS) driver, Neutron Interconnection Plug-in, networking BGPVPN Plug-in, and networking-bagpipe driver.

Since the Interconnection Service Plug-in also relies in the BGPVPN Service Plug-in (See Section 9.5), it is necessary to either deploy a BGP peering overlay on top of the IP WAN connectivity or have a BGP peering with WAN IP/Multi Protocol Label Switching (MPLS) BGPVPN underlay routing instances. Because Grid'5000 does not allow the user to interact with the physical routers (underlay BGP), we deployed the first scenario

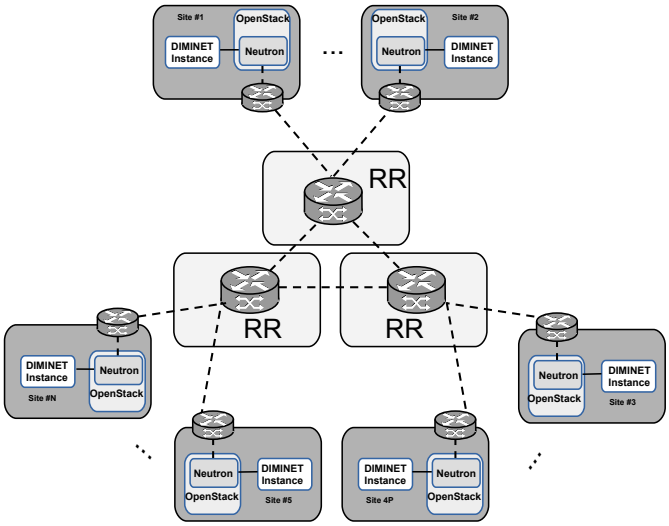


Figure 10.3 – DIMINET testbed setup

using GoBGP to provide the functionality of the BGP instances in each site. These BGP instances are deployed on the same Grid'5000 machines used for the OpenStack and DIMINET deployments. Moreover, we deployed some Route Reflector (RR) instances in independent physical machines represented with white boxes (See Appendix D.1 for more information about the RR).

For the deployment setup, we have used a modified version of Juice [156], a tool originally conceived to test the performance of several DBs with OpenStack using EnOSlib [157]. EnOSlib is a library to build experimental frameworks on multiple platforms including Grid'5000. Our modified version of Juice entirely automatizes the deployment of the OpenStack, DIMINET, and BGP instances (routers and RRs) using a series of Ansible roles. Because of the continuous evolution of all the technologies used (*i.e.,* new releases, changes in code, fixed bugs, deprecated packages, etc.), Juice has been modified constantly to maintain up-to-date the deployment.

10.3 Evaluation of Inter-Site Resources Deployment

Since DIMINET is intended to be used in a DCI environment, it is necessary to verify if the time needed to instantiate an inter-site *Resource* stays constant when several sites compose the *Resource*. This section presents the results of time measures taken for L3 routing and L2 extension *Resources* creation. Since the data plane interconnection performance depends on the number of instances (VMs) booted at every site, we do not

measure this time. Instead, we rely on former works on BGP performance proving the benefits and disadvantages of BGPVPN routes exchanges [158, 159]. For this test, we have deployed 21 sites in total, each RR is connected to 7 sites, and the BGP sessions are pre-configured among the RR and their BGP instance clients at each site that can be seen as a PoP gateway.

10.3.1 Layer 3 Routing Resource

For every experiment, a random instance has been chosen to receive the user request and start the inter-site Layer 3 routing *Resource* creation. We have varied the quantity of sites composing a *Resource* from 2 up to 21 in total (the size of the DCI for this experiment) Figure 10.4 shows a graphical representation of the L3 routing *Resources* mean creation time. We remind the reader that the steps implied in the implementation of the Logic Core functionalities for L3 routing *Resources* are explained in Appendix A.1.1.

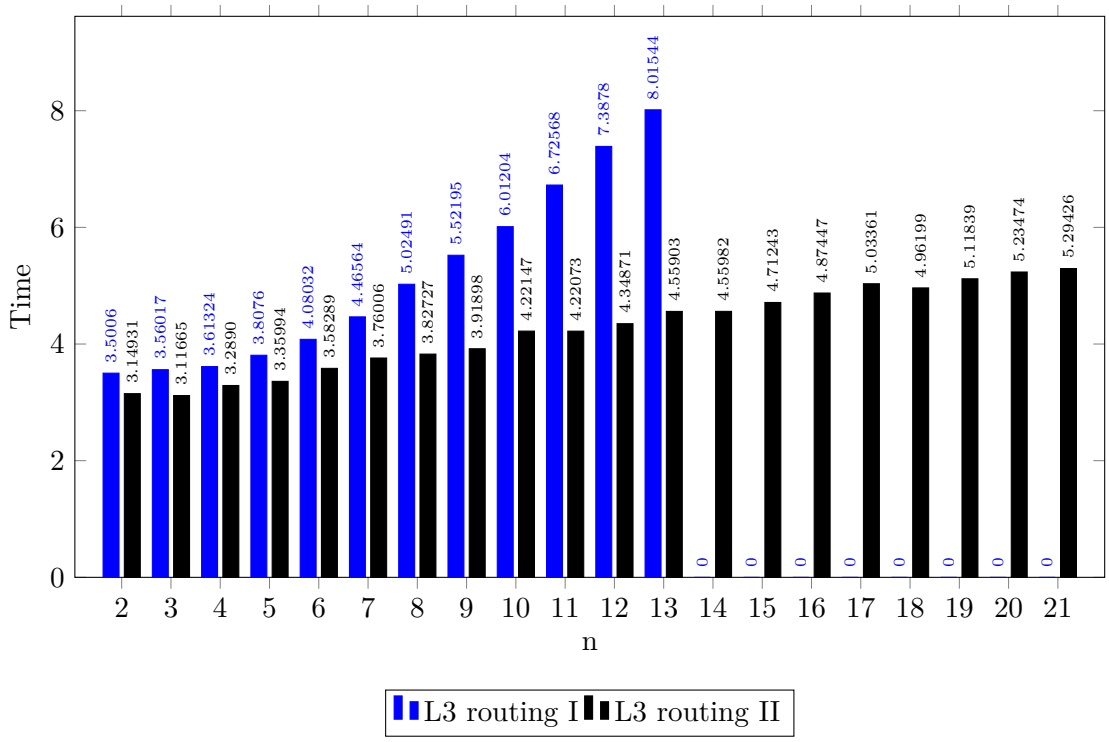


Figure 10.4 – L3 routing resource creation time.

The blue bars (*i.e.,* L3 routing I) represent the results of a first DIMINET implementation that executed all the East-West and Neutron API calls sequentially. Consequently, L3 *Resources* 's mean creation time augmented exponentially (the figure only shows the results of this implementation up to 13 sites because of the augmentation). To dimin-

ish the time expended when doing the API calls sequentially, we modified DIMINET's Logic Core to execute all East-West and Neutron API calls in parallel as represented by the black bars (*i.e.,* L3 routing II). Since DIMINET architecture is fully distributed and each resource manager is also independently from others, the interactions can be executed in parallel because remote information can be provided without dependency among the requests.

While we expected to find a constant pattern because of the East-West and Neutron API requests done in parallel, there is an augmentation of around 121.472 milliseconds per new site composing the *Resource*. To better understand the reason for this time augmentation, in the following, we propose to compare the results of two random L3 routing *Resource* creation experiments; the first one for 3 sites, and the second one for 18 sites. For this analysis, we use the steps numbering provided in Appendix A.1.1.

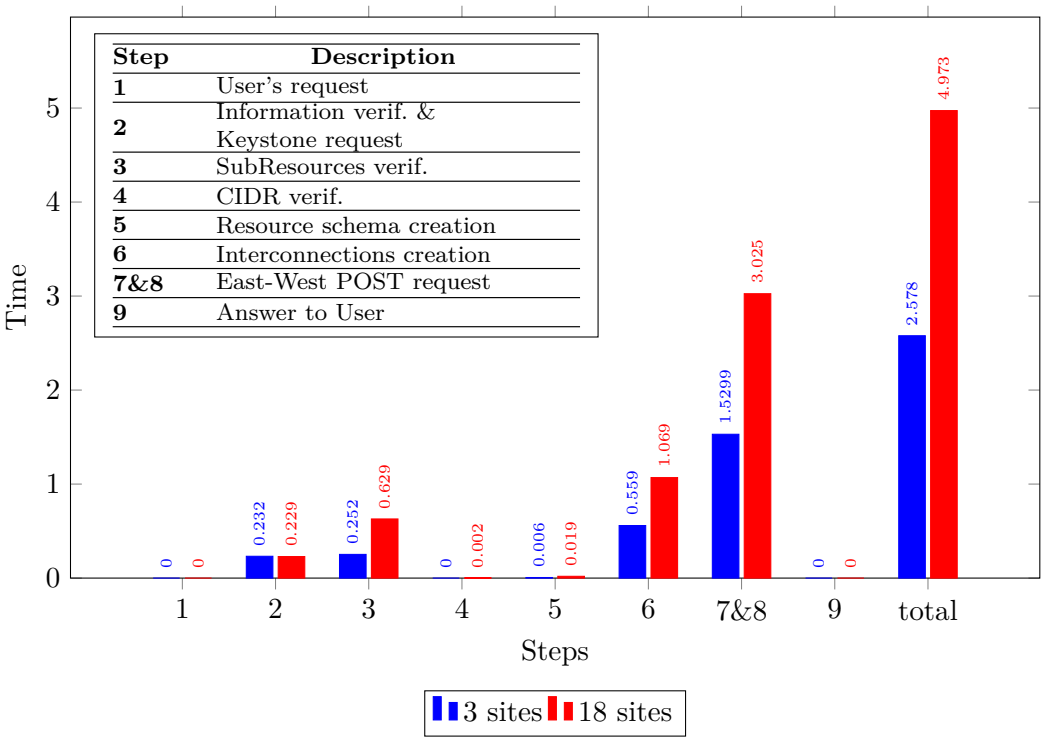


Figure 10.5 – L3 routing resources creation time comparison (seconds).

Figure 10.5 presents the time comparison among the 3 and 18 sites experiments. Step 3 (*SubResources* verification) is the first one to show a slight difference between the two processes, passing from 0.252 seconds to 0.629 seconds, a 2.49 factor difference. The following difference is found at Step 6 (Neutron Interconnections creation), which changes from 0.559 seconds to 1.069 seconds. Finally, in Steps 7&8 (East-West creation request),

the difference doubles from 1.529 seconds to 3.025 seconds.

These three steps have two common elements: the use of threads and the communication with Neutron API (the East-West communication implies that remote DIMINET modules will locally communicate with their Neutron). For the former, due to the manner the implementation applies threads (*i.e.,* using Python's threading and concurrent.futures packages), they are executed with a time difference closely to 1.19 milliseconds, which represents a total time difference of around 29 milliseconds between the first and last threads in the 18 sites experiment.

For the latter, and to deeply understand the time augmentation when communicating with Neutron, we analyse the local Neutron's logging information. We found that Interconnections creation randomly presents unexpected errors due to mishandling of the *Route Target* identifiers necessary to create the BGPVPN connections when processing concurrent API calls. Indeed, when these requests arrive, Neutron tries to allocate a *Route Target* identifier to each Interconnection following an incremental loop. In some cases, while the process is presumed to be thread-safe, the same *Route Target* is selected for different Interconnections. When the process tries to write the *Route Target* identifier in Neutron DB, an error is presented because the value is already stored in the DB. This error impacts Steps 6, 7 & 8 as Interconnections are created at these steps.

While the *Route Target* error has been well identified and reported to the OpenStack community (including the Orange Labs experts), this code is not a priority and the error remains not addressed. As a consequence, we were not able to conduct new experiments. It is noteworthy to mention that while this error impacts the steps demanding to request Neutron API for Interconnections creation (*i.e.,* Steps 6 and 7&8), it does not necessarily affect the main functionality of DIMINET to provide the logical management for inter-site *Resources*. Due to the nature of our implementation, such kinds of nested requests where DIMINET depends on the execution of a third platform, in this case, OpenStack, are a limiting point that cannot be controlled. Further works will be needed in order to fix the *Route Target* assignment in Neutron and measure if a significant difference in these steps is done to achieve L3 routing *Resources* constant creation time.

10.3.2 Layer 2 Extension Resource

Similar to the L3 routing *Resources*, for every experiment, a random instance has been chosen to receive the user request and start the inter-site Layer 2 extension *Resource* creation. The quantity of sites composing a *Resource* has also been varied from 2 up to 21 (*i.e.,* the total size of our DCI). Figure 10.6 shows a graphical representation of the L2 extension *Resources* mean creation time. We kindly remind the reader that the steps

implied in the implementation of the Logic Core functionalities for L2 extension *Resources* are explained in Appendix A.1.2.

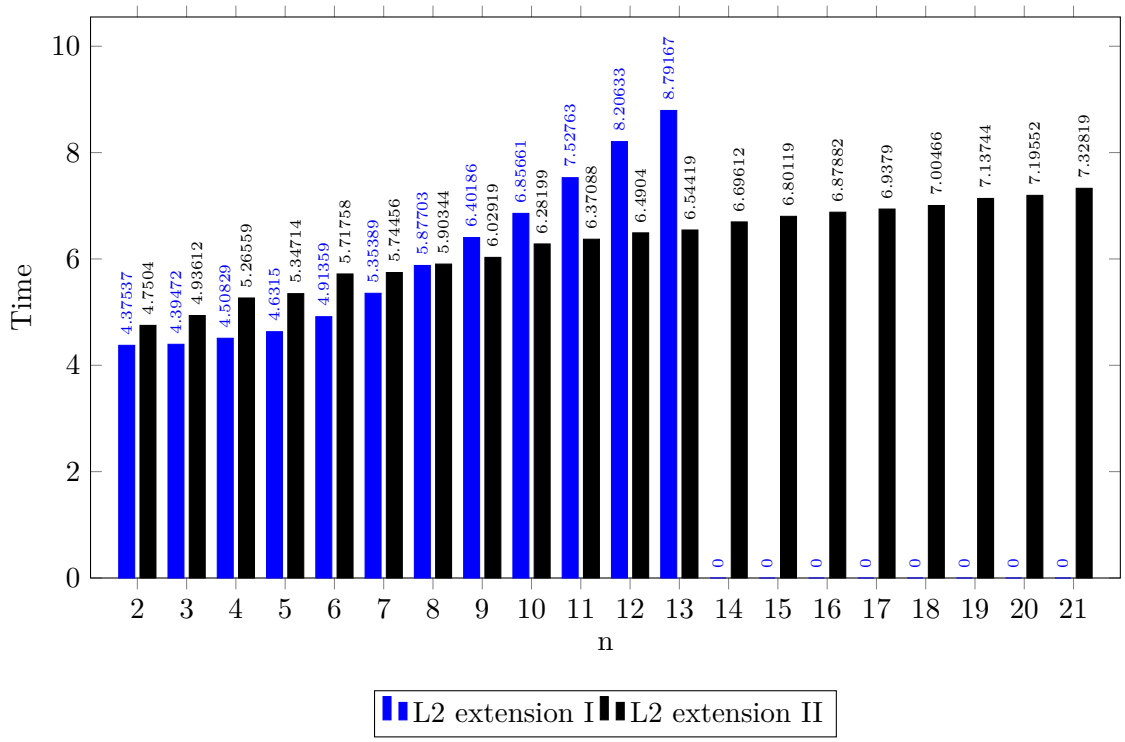


Figure 10.6 – L2 extensions resource creation time

Following the same color guides that the last section, the blue bars (*i.e.,* L2 extension I) represent the results of the first DIMINET implementation that executed all the East-West and Neutron API calls sequentially as explained in Section 10.3.1. The black bars (*i.e.,* L2 extension II) represent the results of the DIMINET modification to run East-West and Neutron API calls in parallel. Similar to the results of the experiments for L3 routing *Resources*, the L2 extension *Resources* experiments do not follow a constant creation time as we expected, but rather there is a slight augmentation of around 109.5 milliseconds per new site composing the *Resource*. As we did in the last section to better understand the time augmentation, in the following we propose to compare the results of two random L2 extension *Resources* experiments; the first one for 3 sites, and the second one for 18 sites. Since the sharding strategy differs between L3 routing and L2 extension *Resources* (See Section 9.3), this analysis uses the steps numbering provided in Appendix A.1.2.

Figure 10.7 presents the time comparison among the 3 and 18 sites experiments. Similar to the results for L3 routing *Resources*, the L2 extension type also presents a difference among some of the steps involving parallel calls to remote modules and the local Neutron

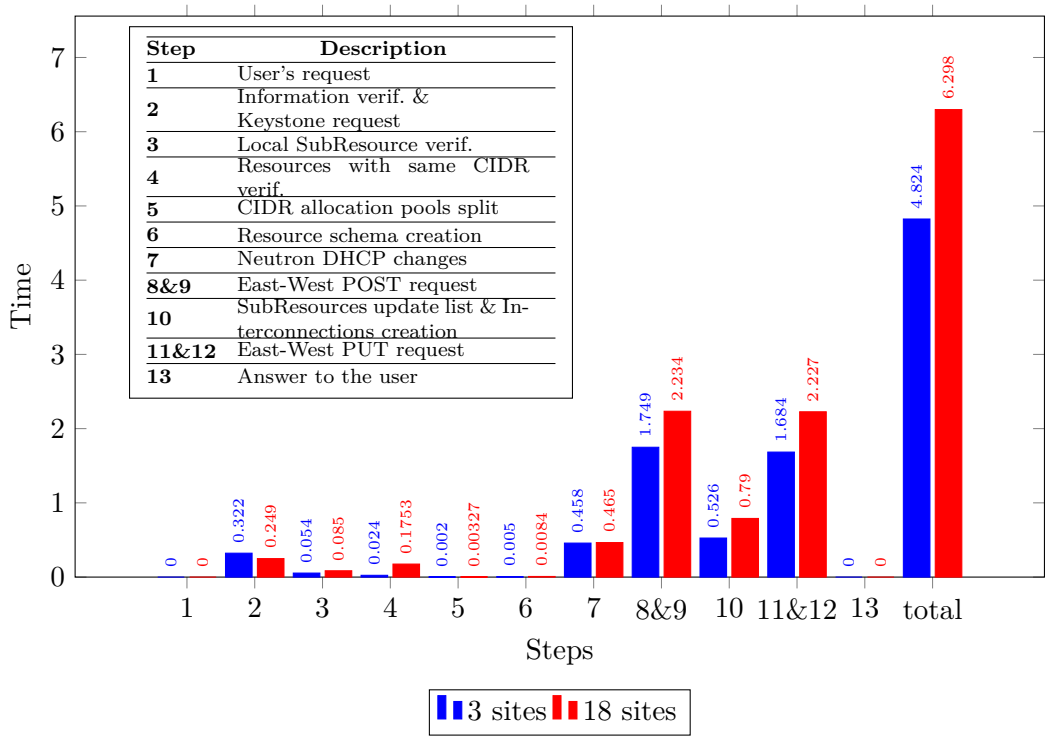


Figure 10.7 – L2 extension resources creation time comparison (seconds).

API. Step 8&9 (East-West POST request) is the first one showing a slight augmentation, passing from 1.749 seconds to 2.234 seconds. Since L2 extensions require to create networks at remote Neutrons that are referenced as *SubResources*, we see clearly that the Neutron API takes more time to answer the totality of network and subnetworks creations and DHCP updates nested in Step 8&9. Contrary to the L3 routing *Resources*, Neutron Interconnections' creation does not augment in the same manner at Step 10. For this particular experiment, the error explained for BGPVPN *Route Targets* allocation did not occur at the DIMINET module's local Neutron. Then, in Steps 11&12 (East-West PUT request), the time passes from 1.684 seconds to 2.227 seconds. We presume that remote modules encountered the *Route Target* allocation error when requesting the Interconnections creation at their local Neutron API.

Although the *Route Target* allocation error may interfere with the correct forwarding of data plane traffic, in cases where it is not present, the communication is perfectly established among VMs. By fixing the faulty code for *Route Targets* assignment, nested requests for Interconnections creation should not augment DIMINET *Resources* creation time in the same manner as in the actual implementation. However, Python's thread implementation and Neutron's code lack of optimization will continue to penalty the time

needed for steps involving parallel Neutron API calls. Even with the problems explained above, we believe that this test allowed to do a first verification of DIMINET's implementation for OpenStack, showing promising results in terms of time consumed to create inter-site *Resources*.

10.4 Evaluation of Resiliency

The purpose of this test is to show the improved resiliency of a distributed architecture against networking partitioning issues. To explain this, we have deployed an L2 extension *Resource* with CIDR IPv4 11.0.0.0/24 depicted in Figure 10.8 (A) among sites A and B. Once the *Resource* has been deployed, two VMs have also been deployed on each site.

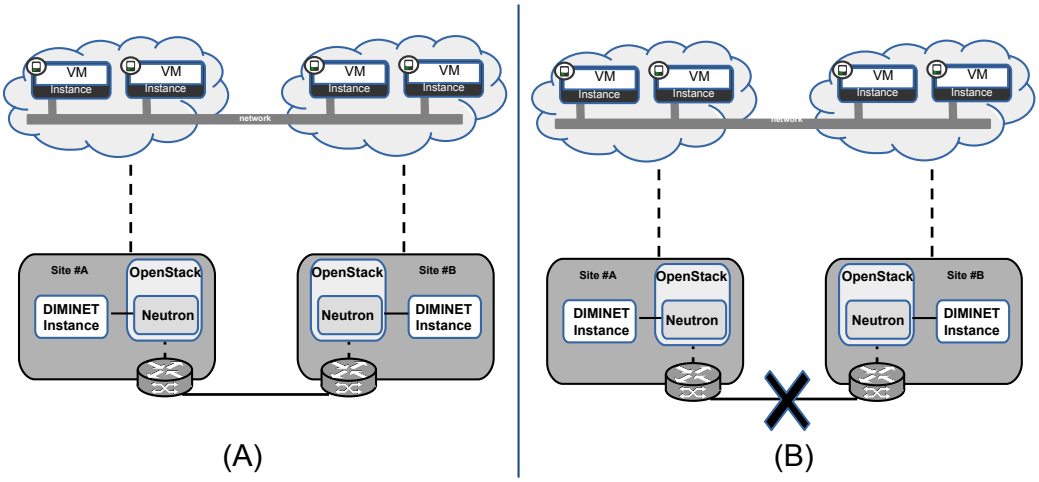


Figure 10.8 – DIMINET Resiliency test: (A) Initial deployed service. (B) Inter-site service in presence of networking partitioning.

Firstly, we have checked that the traffic was being carried at the intra-site level, this is, between the VMs deployed in the same site. We also checked that the traffic was being carried at the inter-site level. At this point, thanks to the different technologies used (*i.e.*, BGPVPN routes exchanges and VXLAN tunnels among sites establishment), traffic was correctly forwarded in both cases.

Secondly, we emulated a network disconnection using TC to introduce a network fault in the link between the sites, as shown in Figure 10.8 (B). We decided to impact the network by allowing the BGP routes exchanges. As a result, intra-site traffic continues to

be forwarded while inter-site traffic is only forwarded a little more until the local BGP router finds its distant BGP peer is no longer reachable. At that point, the local BGP router decides to withdraw the remote routes from its local deployment, impacting the inter-site data plane traffic forwarding.

Because of the independence between the deployments and the logical division done by DIMINET instance, we effectively arrived to instantiate new VMs during the network failure. This corresponds to the behavior we expected since the OpenStack deployments are completely independent among them.

Finally, when connectivity is reestablished, inter-site traffic takes some time to be forwarded again between sites. This delay is because the BGP peers wait the configured *Keep Alive* time to query the distant peer about its availability to reestablish the BGP peering among them, thus, impacting the time needed to reestablish the traffic.

10.5 Evaluation of DIMINET Scalability

Since one of the announced characteristics of DIMINET is its scalability *w.r.t.* traditional cloud infrastructures, we conducted additional experiments to use up to 59 Grid'5000 sites and measure the mean *Resource* creation time. While this quantity is far from representing a Telco's DCI, it allows us to analyse if the different tasks inside DIMINET's Logic Core can continue to perform well when adding more sites to a request. Since we are aware that the BGPVPN *Route Target* assignment random error can affect the total time of a *Resource* creation, we have modified the Neutron Interconnection *Route Target* allocation so this process continues to provide Interconnection objects but without the allocation of *Route Targets* to them.

Similar to Section 10.3, for every experiment, a random instance has been chosen to receive the user request and start the inter-site *Resource* creation. This process has been done for both types of *Resources*, L3 routing and L2 extensions. Figure 10.9 shows a graphical representation of L3 routing and L2 extension *Resources* mean creation time. As expected and similar to the results of Section 10.3, there is a slight augmentation of time per new site composing the *Resource*. It is noteworthy no mention that the mean creation time has diminished in both cases and this can be explained because the Neutron Interconnections Plug-in does not allocate *Route Targets*, thus, the allocation error should not happen. However, there is still a time augmentation when comparing a few sites against the total DCI size. Since we have disabled the *Route Target* allocation process, it is possible to analyze in which steps DIMINET adds time to the creation request processing. As we did in previous sections, in the following, we propose to compare the

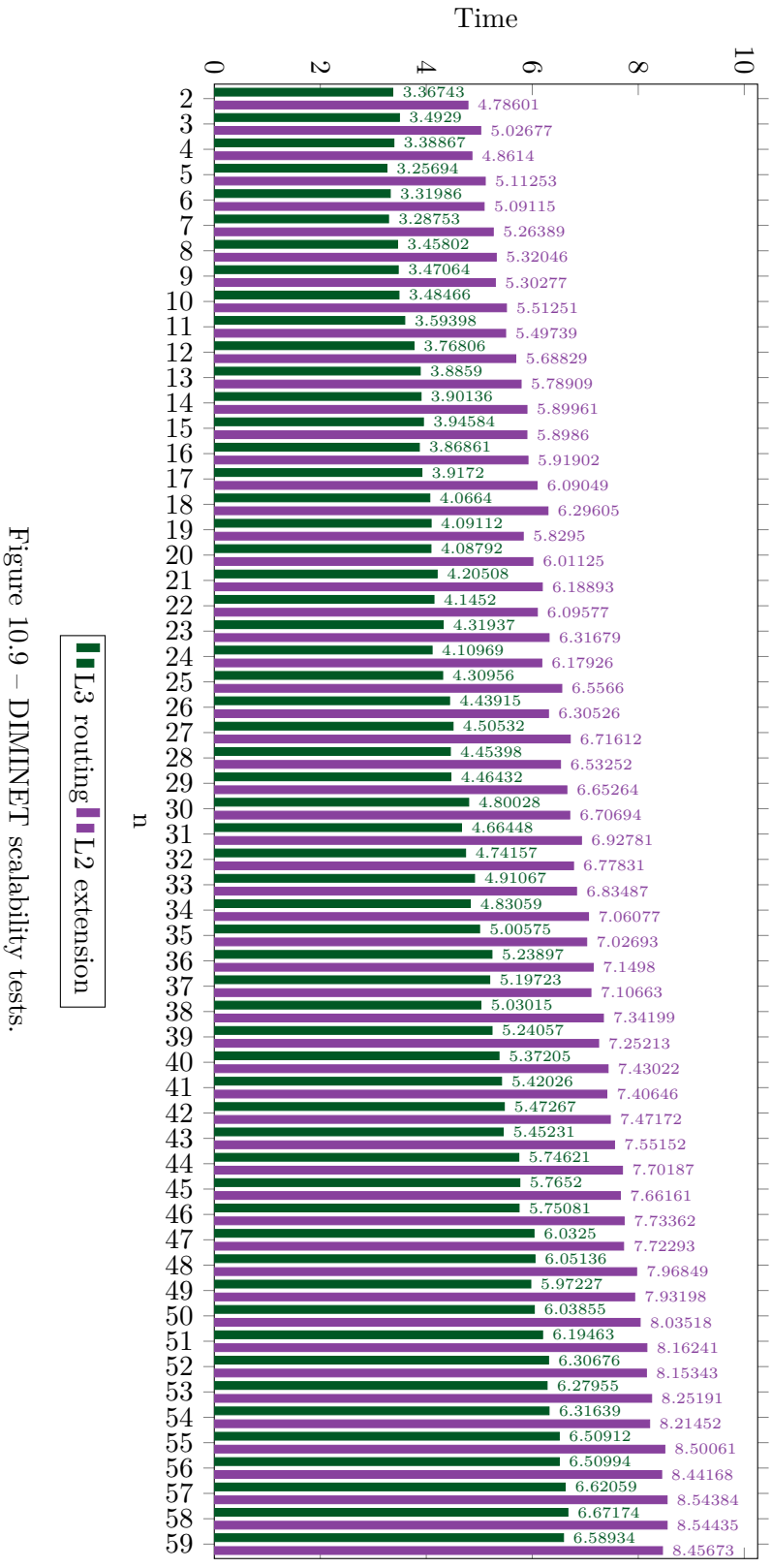
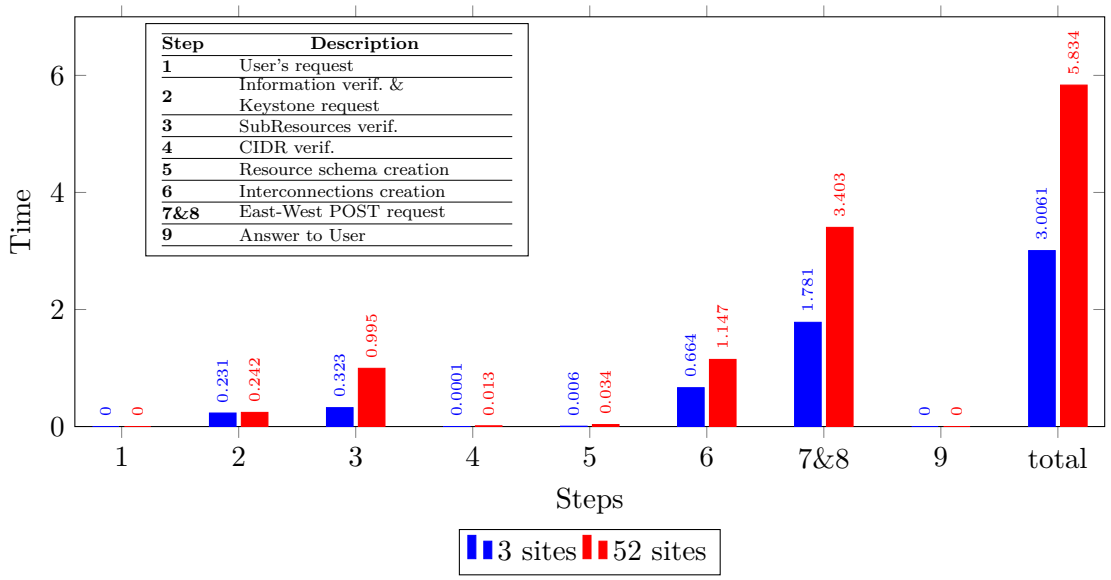
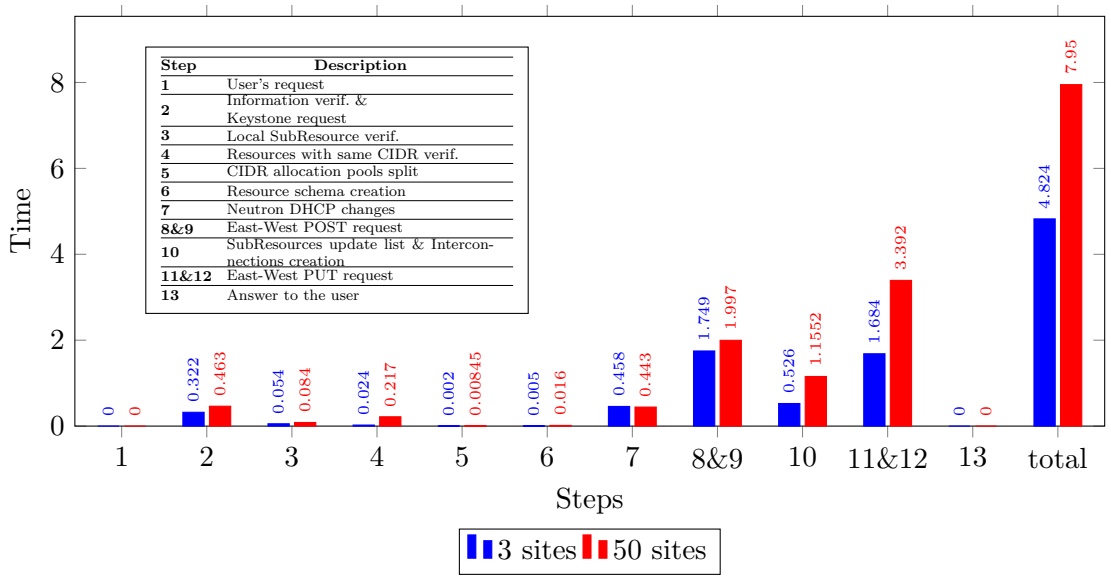


Figure 10.9 – DIMINET scalability tests.

results of two randoms *Resource* creation experiments, for both kinds of *Resources*, L3 routing and L2 extension.



(a) L3 routing resources creation time scalability comparison.



(b) L2 extension resources creation time scalability comparison.

Figure 10.10 – Scalability time comparison tests.

Figure 10.10 gathers a summary of the different steps for both L3 routing (3 sites against 52 sites) and L2 extension (3 sites against 50 sites) *Resources*. As expected, in both cases, the steps requiring to execute parallel tasks present some significant time differences.

Figure 10.10(a) depicts the time comparison among the 3 and 52 sites experiments for the L3 routing *Resource*. The first difference is found at Step 3 (*SubResources* verification). When analyzing the details at DIMINET's code, we found that this step implies an authentication request against the local Keystone to found the endpoint of distant Neutrons. The problem lies in the implementation: instead of doing a single authentication before the threads, each thread executes a Keystone request, potentially saturating the Keystone endpoint and impacting all the other threads' execution. The second difference is found at Step 6 (Interconnection creation). As we stated above, we did not use the *Route Target* allocation to avoid the unexpected error (See Section 10.3.1). However, there is still a time augmentation between the two experiments that cannot be simply explained with the time difference when executing Python threads. We believe that this difference is caused by our Neutron instance (*i.e.,* a test DevStack version running on a VM), which cannot optimally handle the parallel requests to create the Interconnection objects. Finally, at Steps 7&8 (East-West POST request), the time consumed depends on the contacted remote DIMINET instances. Since they also contact their local Neutron to create Interconnections objects, the time augmentation may be due to Neutron.

Figure 10.10(b) depicts the time comparison among the 3 and 50 sites experiments for the L2 extension *Resource*. Since the steps of the sharding strategy are different from the L3 routing *Resource*, the time augmentation may have different origins as we already analyzed in Section 10.3.2. Similar to previous results, Step 8&9 (East-West POST request) is the first one showing a difference among the experiments. This augmentation is due to remote Neutrons API answering the create network and subnetwork requests and doing the DHCP modification. Step 10 (*SubResources* update list & Interconnections creation) is the following presenting a time difference. In this step, the Interconnection objects are created by Neutron API which, as stated above, cannot handle well several parallel requests. Finally, Step 11&12 (East-West PUT request), also presents a time augmentation due to the need of creating Interconnections objects at remote sites.

The analysis of the experiment' results for both L3 routing and L2 extension *Resources* has revealed some pieces of code in DIMINET's implementation that need to be refactored but also some problems at the OpenStack code. In general, DIMINET's performance (*i.e.,* East-West and Neutron API calls) could be optimized by refactoring some pieces of code to have a more constant creation time and allow its use in a real DCI without the time augmentation presented in this experiment. Besides these implementation problems, DIMINET is still capable to provide the logic to manage inter-site *Resources* in a distributed fashion.

10.6 Summary

This chapter has presented a first test of DIMINET functionality with OpenStack in a lab environment. Also, we have presented and detailed several test results of DIMINET running in an OpenStack-based DCI using Grid'5000 testbed. While unfortunately, the chosen Neutron technology to implement the data plane exchange and forwarding (BGPVPN Service Plug-in) present some errors, both types of DIMINET *Resources* can be effectively provided. Tests showed an augmentation of time consumed to execute creation requests, mostly when needing to contact Neutron API.

The analysis of time consumption has allowed us to understand different problems related to DIMINET's implementation but also related to the way Neutron API treats requests. While these problems are mostly technical and do not affect the logic behind the principles of DIMINET, they, unfortunately, impact its utilization in an OpenStack-based DCI. In consequence, while our large-scale experiments showed the possibility to instantiate on-demand inter-site *Resources*, we were not able to show the communication at the data plane level among the different sites composing a *Resource*. As stated in Section 9.5, we do not implement the exchange information about the virtualized traffic connectivity mechanisms over the East-West interface but instead, we preferred to rely on Neutrons Service Plug-ins. While adding extra complexity to DIMINET, a possible future work could be to implement this exchange over the East-West interface in order to avoid the dependence with Neutron's Interconnection and BGPVPN technologies.

An optimization of DIMINET's code should be done in future works in order to test and compare the results with the ones presented in this chapter. Besides, more development is needed in Neutron to fix the BGPVPN *Route Target* allocation error, and, in general, to optimize Neutron's code to better answer parallel requests. Moreover, due to lack of time, we did not execute additional experiments to continue the validation of DIMINET's implementation for OpenStack. Experiments measuring DIMINET's behavior to modify inter-site *Resources* on-demand adding or deleting new sites were the next ones to be executed to better analyse all DIMINET capabilities.

Part IV

Conclusions & Perspectives

This fourth part concludes this doctoral work and gives four major perspectives for future works.

- *Perspective 1 discusses how DIMINET's WRITE operation could be optimized.*
- *Perspective 2 analyses the possible use of DBs as communication tool instead of the proposed East-West interface.*
- *Perspective 3 presents Cilium, a project inside the Kubernetes ecosystem with similar characteristics to DIMINET.*
- *Perspective 4 provides a fist generalization of DIMINET data model to contribute to further research for other kinds of inter-service collaboration.*

Conclusions and Perspectives

Conclusions

The Cloud computing paradigm has completely shifted the way IT services are proposed and accessed. This model has allowed the apparition of new use cases, each one requiring guaranteeing different constraints. With operational needs such as low delay and resiliency against networking failures, Telcos such as Orange have found an interest in deploying resource managers such as OpenStack closer to the end-users in PoPs of their backbone. Different actors are interested in answering how to manage such a DCI while assuring scalability, resiliency, locality awareness, and abstractions. While the simplest way is to use a centralized approach to control and manage the DCI, this method presents several problems and is not well suited for guarantee all the DCI characteristics.

In this doctoral work, we have studied SDN-based approaches to provide distributed management for DCIs focusing mostly on the virtualized networking aspect.

Chapter 2 described the basics of this thesis. It gives an overview of cloud services and introduces the SDN paradigm, mainly focusing on the SDN-based cloud networking approach by giving an overview of OpenStack and Kubernetes, the most popular VIM and CIS resource managers, and their networking aspects.

Chapter 3 described this work's problem statement, introducing the evolution of cloud infrastructures towards DCIs. We have explained the DCI characteristics, and we have focused on the challenges that need to be addressed in order to provide distributed networking management in DCIs. These challenges were divided into two categories, those related to network information management and those associated with the technological implementation of such a management.

Chapter 4 detailed the different properties used to classify and analyze the reviewed solutions of our state-of-the-art study on decentralized SDN solutions. We explained key concepts such as the kind of architectures, the leader-based strategy, the internal communication protocols, the database management system, interoperability, and the maturity level.

Chapter 5 presented seven network-oriented SDN controllers. DISCO, D-SDN, Elasticon, FlowBroker, HyperFlow, Kandoo, and Orion are presented and classified following

Chapter 4's properties. Then, we analyzed how each of these solutions may address the DCI challenges.

Chapter 6 presented five cloud-oriented SDN controllers. DragonFlow, ODL, Onix, ONOS, and Tungsten are analyzed and classified *w.r.t.* Chapter 4's properties. We also analyzed whether each solution is capable or not of addressing the DCI challenges.

Chapter 7 presented four solutions that, while not being purely SDN solutions, we consider essential to analyze. Kubernetes Federation, Kubernetes Istio Multi-Cluster, OpenStack P2P external proxy-agents, and Tricircle were analyzed and studied to classify them accordingly to Chapter 4' properties. Then we explored how each solution may address the DCI challenges. We underlined that this study is a scientific contribution by itself. Such a state-of-the-art study was not available before this work.

Chapter 8 presented the lessons retained from the state-of-the-art review. Solutions such as DISCO and ODL, proposing a fully distributed architecture, presented interesting insights on how to provide a distributed management. Although both solutions cannot fully address the DCI challenges, they provided key ideas such as using an East-West interface to communicate among controllers while having a local DB at each controller. We have also presented several open questions such as the standardization of East-West SDN interfaces, the use of new DB engines, the implementation of new data plane technologies, the performance analysis of the solutions, and the security in SDN technologies.

Chapter 9 introduced our proposal of a distributed architecture capable of providing inter-site networking connectivity management based on DIMINET, a distributed module capable of establishing communication using an SDN-inspired architecture where independent modules communicate among them by using a horizontal interface and local DBs. By proposing a per-resource sharding strategy, we have shown the possibility to provide coherent resource management with minimal traffic exchange among the different sites composing a DCI. The chapter also introduced DIMINET's implementation for OpenStack Neutron. We also presented the details inherent to the Neutron-to-Neutron Interconnection Plug-in, the Neutron's networking mechanism chose to provide the data plane connectivity for the virtual instances. By relying upon the BGPVPN technology, the Interconnection Plug-in allows to effectively share routes among BGP peers and provide data forwarding for VMs belonging to independent sites.

Chapter 10 introduced an assessment of the distributed architecture by presenting

a first demonstration of DIMINET in a small OpenStack-based scenario. The first results showed that DIMINET effectively allows the creation of L2 extension and L3 routing resources. Moreover, we showed that traffic is forwarded among the independent sites by using the Neutron networking mechanisms. Then, to proceed with further analysis of our module, we presented the tests executed using the Grid'5000 testbed to better asses DIMINET capabilities. The evaluation of inter-site resources deployment showed a slight augmentation in time spent when DIMINET executes its parallel requests due to some errors in the implementation for OpenStack. The resiliency test showed the expected result of using a distributed architecture; if a network partition is presented, each site continues to operate locally. Finally, we presented the results of scalability tests where more Grid'5000 machines are used to emulate the DCI. While the time consumption increased in the parallel requests, DIMINET remained capable of instantiating the inter-site resources. Unfortunately, the Neutron-to-Neutron Interconnection Plug-in presented an error that does not allow the correct exchange of BGP routes among sites. Consequently, we were not able to show a fully deployed and operational inter-site resource in our Grid'5000 tests emulating a DCI.

Perspectives

The studies, analysis, and developments carried out in this doctoral work tend to assess the possibility of providing a distributed DCI network management by relying upon SDN principles. We proposed DIMINET a distributed module relying upon two key elements: a Logic Core implementing sharding strategies for providing resource management coherence and on-demand communication exchanges among modules leveraging an East-West interface from SDN controllers. The distributed approach presents itself as a good choice to provide distributed DCI management as demonstrated in this thesis.

Because of the industrial nature of this thesis, we have proposed a first implementation of DIMINET for the OpenStack ecosystem, more in detail, for the networking service Neutron. While DIMINET tests allowed us to assess the architecture and analyze the creation of inter-site resources, some technical errors in our implementation, and inside Neutron's code avoided a full assessment of DIMINET for the OpenStack ecosystem. While being purely a technical issue, further works are needed in order to optimize DIMINET's code when doing parallel requests using the East-West interface. Besides, it will be necessary to fix and debug some code in Neutron allowing a correct use of the Interconnection Service Plug-in to provide the data plane connectivity for DIMINET. When these software mod-

ifications are done, DIMINET will be completely integrated with OpenStack and further tests as the one presented in this thesis should be done. Ultimately, DIMINET could be used in Orange network to provide connectivity management for the PoPs composing the DCI.

Although it would have been valuable to continue our research around decentralized DCI management using DIMINET's architecture, there are still several subjects that may be further studied and analyzed to contribute even more to the research of DCI management. In the following, we present four major perspectives for future works.

Perspective 1: Improving DIMINET WRITE operations

Since DIMINET tries to diminish the quantity of exchanged messages among modules and given that we use a per-*Resource* Master mapping (*i.e.,* avoiding the complexity and overhead of using a consensus leader election), WRITE/creation requests are served once for all. Therefore, unless an UPDATE/modification request is made to the *Master* module, no further synchronization is done among the agents. As we have already mentioned, the *Master* module will then act as the entry point to modify that specific *Resource*. Moreover, information related to the management of the *Resource* will be stored at the *Master* module database. This way to ensure coherence presents some similarities with the well-studied cache coherence protocols in shared-memory multiprocessor systems, especially with directory-based coherence mechanisms. In directory-based protocols, a directory keeps track of the status of all blocks of the cache. Among the stored cache information, the directory includes which state the block is and which nodes share that particular block. In this protocol, two kind of actions can be taken when a line is written, the protocol either invalidates all copies of the line in others caches (*a.k.a. invalidation-on-write* or *write-invalidation*), or it updates remote caches (*a.k.a. update-on-write* or *write-update*) with the new value given to the line [160].

Similarly to cache functionalities, DIMINET local databases save a subset of a *Resource* information locally. The *Resource* Master database acts similarly as the memory unit for that particular *Resource*. Moreover, the Master module is used to track the sites composing a *Resource* as in directory-based protocols. Another similarity is the use of a *write-update* mechanism by the Master module to ensure that all sites composing the *Resource* are up to date with the latest information of the *Resource* to provide the inter-site networking connectivity. In the context of inter-site resources, a *write-invalidation* mechanism does not make much sense because sites will not be able to provide the desired state of the *Resource* requested with the WRITE operation. As a consequence, modules will only do an invalidation of the local data, hence rendering the inter-site resource useless.

Besides these similarities, there are some differences in the way DIMINET treats WRITE requests. Contrary to directory-based systems, DIMINET only permits further modifications to the Resource by using the Master module. While the update acknowledges are proposed in DIMINET, they are only done between the modules and the Master. A general model could then greatly benefit from the *write-on-update* mechanism of cache-coherence protocols by allowing every module to propose modifications to the *Resource*: Instead of only using the Master module to execute updates on *Resources*, every module composing the *Resource* should be able to perform WRITE operations on the *Resource*, hence, gaining more dynamism for the entire system.

In the proposed model, the Master module still acts as the memory unit validating the writing operations. Whether the writing module proposition is accepted or not by the Master module will depend on the mechanism established to verify if the inter-site *Resource* can be provided. In case that the Master module considers that the request cannot be furnished, it should answer to the writing module announcing the impossibility to advance further in the *Resource* provisioning. If multiple UPDATES are done for the same *Resource*, the Master module should implement a fence for the *Resource* being modified. Consequently, all writing modules proposing the modification will receive an answer informing that the request cannot be served.

By introducing this change inspired in cache-coherence protocols, all modules belonging to a Resource could modify the inter-site *Resource*, improving the potential bottleneck problem of using a per-*Resource* Master mapping.

Perspective 2: Using DBs as communication tool

Concerning, DIMINET's communication among modules, in DIMINET's proposition and in its generalization, we have leveraged the use of an East-West interface among modules to provide coordination among modules while maintaining local DBs. Nevertheless, another approach could consist of leveraging distributed DBs. As we highlighted in Section 8.1, solutions such as Elasticon, HyperFlow, Orion, DragonFlow, Onix, and ONOS, use a distributed DB 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 DB, it does not respect data locality principles. Even more, it is not resilient enough in case of network partitions.

To illustrate this point, one can consider the Cassandra DB [126]. Cassandra is based on a Distributed Hash Table (DHT) principles to uniformly distribute states across multiple locations based on the computed hash. It means that one specific controller's states are not necessarily stored in the same geographical area as the controller (*e.g.,* a controller

in site A will have states stored in remote locations B, C, and so forth as depicted in Figure 10.11). 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.

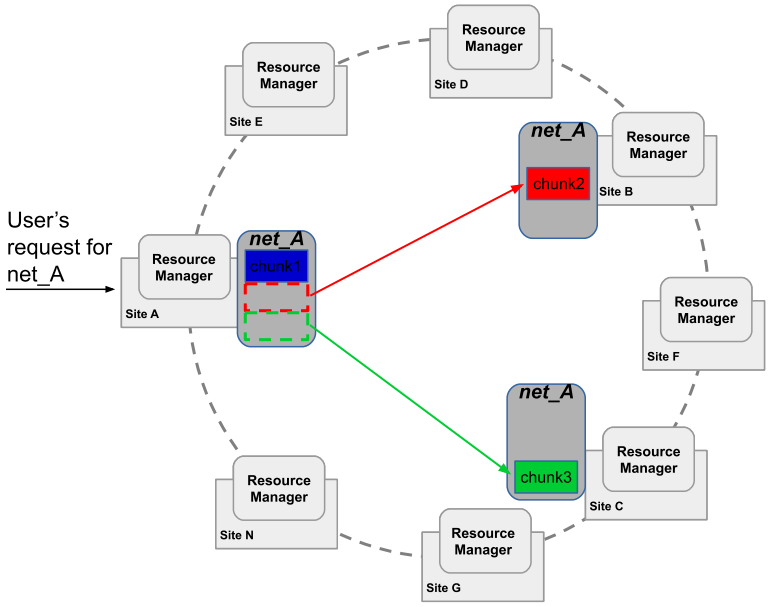


Figure 10.11 – Cassandra DB DHT example.

Likewise, a distributed architecture that leverages Cassandra will not be resilient to network isolation. It will be impossible for the local site 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 [161]. NewSQL engines generally try to leverage characteristics such as different memory storage, modes of partitioning, and concurrency control to provide the properties mentioned above.

Regarding performance, DB engines have used disk-oriented storage. However, today's 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 [162]. Moreover, almost all systems used to scale out splitting a DB into subsets, called partitions or shards.

Overall, NewSQL engines can be gathered in three main classes [163]: new architec-

tures, 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 [92]. 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 [116], Google Spanner [165] and its related solutions such as Google F1 [166], VoltDB [91], or HyPer [167].

— *Transparent sharding middleware*: This group gathers engines that split a DB into multiple shards that are stored across a cluster of single node instances. In this sharding, each node runs the same DB management system. Each one has only a portion of the overall DB, and data does not mean being accessed and updated independently by separate applications. 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 [168] or MariaDB MaxScale [169].

— *Data Base 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 [170] or ClearDB [171]. 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. To do this operation, Aurora relies upon a log-structured storage manager capable of improving I/O parallelism [172].

To measure these new engines' value in the DCI context, a DIMINET-like application needs to be built on top of the selected solution. It will let to analyze each one's pros and cons and verify if they can satisfy the DCI requirements highlighted in Section 3.1.

On the other hand, solutions such AntidoteDB [173] or Riak [174] that have been designed on top of Conflic-free Replicated Data Type (CRDT) [175] could be leveraged to address the 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. It means that each replica will be locally accessible and ready to use in case of network partitions. The richness of the CRDT structure rends possible to resolve the inconsistency between multiple replicas eventually. However, CRDTs comes with two significant limitations. First, it requires replicating 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 CRDT. If future research might

find solutions for these two problems, CRDT may represent a step forward to provide a distributed solution while respecting the DCI properties.

Perspective 3: Cilium at Kubernetes: Decentralising K8S

As we have explained in several chapters, DIMINET has been implemented for the OpenStack environment. In recent times, Cilium [55, 176], a proposition with similar characteristics (*i.e.,* distributed architecture, East-West communication, local DBs) has emerged in the K8S environment. 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 to IP Tables, which delivers better performance [177, 178]. 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 set up 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.

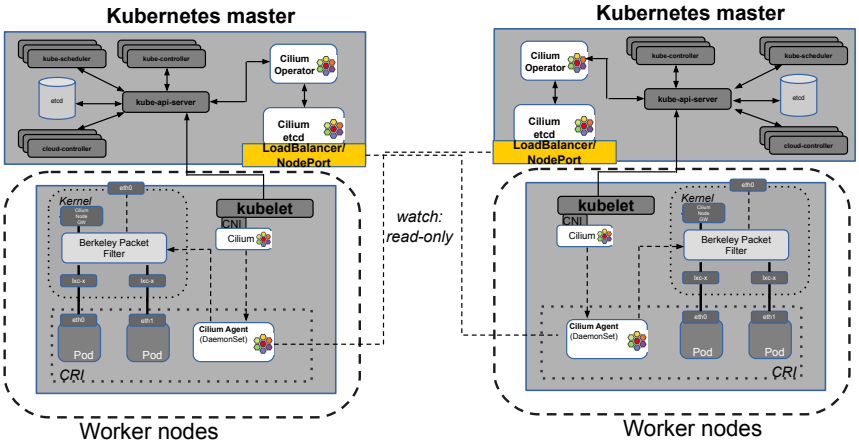


Figure 10.12 – Cilium Multi-Cluster architecture

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. Figure 10.12 depicts a deployment using Cilium ClusterMesh. 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 required types of communications for the K8S environment (See Section 3.3). However, the scalability of such a model is questionable as ClusterMesh initiates a tunnel (*e.g.,* VXLAN or Geneve) between each worker nodes pair.

Because Cilium ClusterMesh requires connecting 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. It is noteworthy to mention that the user still needs to deploy applications and Services in each cluster to provide a Multi-Cluster Service.

Cilium proposes to expose the local database to remote clusters to exchange information. The way this information is consumed by remote clusters could inspire more propositions leveraging the exposition of their databases as East-West interface among clusters. While being developed in parallel from DIMINET, Cilium uses the distributed DB approach we did not use at DIMINET. Still, on-demand communication is not considered at Cilium. In order to assess and compare the two propositions, a DIMINET implementation for the K8S environment should be done to analyse the differences between the East-West interfaces of both propositions.

Perspective 4: Towards a generalization of DIMINET for inter-service collaboration

As we have explained all along this doctoral work, our model to provide management for a DCI is based on a distributed architecture where independent and autonomous resource managers (Virtual Infrastructure Managers (VIMs) or Container Infrastructure Services (CISs)) are deployed at every site of the DCI. Unfortunately, most cloud services and applications were neither designed following distributed approaches (*e.g.,* e-commerce, web-services, stream processing, etc.) nor thinking about the edge, as we explained in Section 3.3.1 for OpenStack and Kubernetes. Moreover, modifications to such software stacks are tedious while not impossible [66, 179].

In the context of OpenStack, another **cross** collaboration needed could be found at Nova (See Section 2.1.4.1). Let's consider the capacity to manage several VMs through different independent OpenStacks coherently: VMs may evolve independently from others, but all the CRUD operations done to a VM at one Nova should impact all the other VMs too. In consequence, a strategy to manage such kind of VM will be needed. While most of the proposed DIMINET' ideas have been conceived to allow distributed networking

resources management, as in the case of OpenStack Neutron, it has sense to consider that these ideas may contribute to building a more general framework allowing other different types of services to collaborate among them.

To provide cross-service collaboration for the virtualized networking resources management, DIMINET strongly relies upon two key characteristics: the communication exchanges that have to be done among instances, and the data model intended to store information of inter-site *Resources* locally while providing coherence. Since an inter-site *Resource* is composed by several independent local *SubResources*, DIMINET provides the necessary logic to bound and interconnect them, assuring the coherence of the whole *Resource*. Thus, each site will manage the states of the *Resource* locally while following the established logic. A presumption in this case is that no external modification rather than DIMINET' ones can be done to *SubResources* or *Interconnections*.

First of all, the logical notion of an inter-site *Resource* composed by a series of local *SubResources* should remain the basic concept for a more general data model. Since the communication is expected to be managed in the same way as DIMINET (*i.e.,* using the North and East-West interfaces), each database will be deployed per module. While the *Resource* type may vary in function of the desired kind of resource, it does not affect the relation among *Resource* and *SubResources*. Moreover, the *Parameter* class could be expanded to add more properties related to the inter-site *Resource*. For instance, DIMINET stores the CIDR of the local *SubResource* as a property in the *Parameter* class. Each *Resource* is managed by a *Master* modules which instantiates the *LMaster* class. Moreover, the *LMaster sharding strategies* are used to specify the management strategy of *Resources*.

With respect to DIMINET data model, the changes we introduce as presented in Figure 10.13 for a general inter-site **cross** collaboration data model are the following:

Resource: Main object which represents an inter-site resource. It is a logical resource that exists at several locations simultaneously and needs to be coordinated to not affect the correct functionality of the construction (*i.e.,* **cross** collaboration). The notion of *Resource* will depend on the cloud service to be distributed. A *Resource* is composed of some *Parameters*, a list of *SubResources*, and a list of local *DataImplementation* objects.

Parameter: Depending on the inter-site *Resource* to distribute, the *Parameter* class could be expanded to add more properties related to the inter-site *Resource*. For instance, DIMINET stores the CIDR of the local *SubResource* as a property in the *Parameter* class.

SubResource: A *SubResource* represents a virtual object belonging to a site (*e.g.,* a

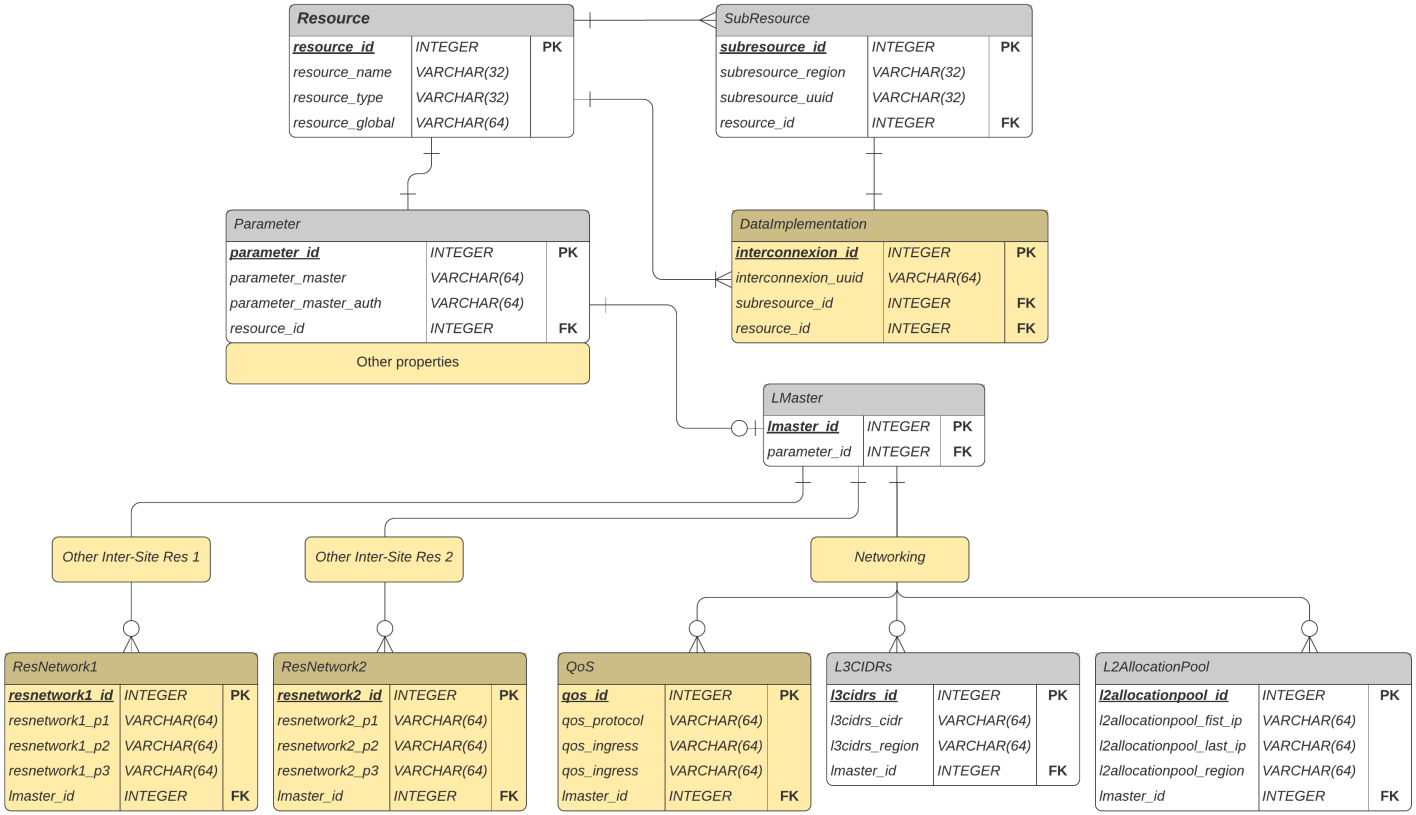


Figure 10.13 – General data model based on DIMINET.

UUID). The *Resource* class holds a list of *SubResources* (the local one and a series of remote ones). This list exists in every module instance composing a *Resource*.

DataImplementation: Initially, the *Interconnexion* class was defined to answer the Challenge 3.3.2. In the networking context, it makes sense to define an entity allowing the local *SubResource* to be reached from remote sites. In a more general context, while reachability information could still be useful (*e.g.,* A tunnel endpoint, a DNS name, ...), more information related to the technical implementation may be present. In consequence, the *Interconnexion* class has been redefined to become the *DataImplementation* class.

LMaster: The concept of a static per-*Resource* master was introduced as a way to maintain coherence among the different sites composing the *Resource*. In the case of DIMINET, depending on the Resource type (*i.e.,* L2 extension or L3 routing), different information is stored to assure this coherence. Hence, other inter-site *Resources* specific information allowing the Master to maintain coherence can be easily added. Furthermore, they can be gathered depending on the kind of Resource. For example, Figure 10.13 gathers L2 extension, L3 routing, and QoS as Networking.

LMaster sharding strategy: Object allowing to declare the strategy to manage a *Resource*. In the DIMINET's case, it represent the *L2AllocationPool* and *L3Cidr* classes. This class is only instantiated at a *Resource* master module.

While the East-West interface will allow communication with remote modules, its use will strongly depend on the sharding strategy conceived for other kinds of *Resources*. We remember that, for instance, DIMINET uses a different sequence flow for the L3 routing and L2 extension *Resources* as presented in Appendix A.

Generally speaking, and as stated above, this data model could be used for other cloud services needing collaboration. While we provide a first data model based on our experience with DIMINET, it will need adaptation depending on the cloud service's resources. It will be interesting to analyze how this general model could be applied in the K8S ecosystem to provide for example cross namespaces resources.

Summary

The works developed through this thesis have provided ideas and motivations for two doctoral thesis at the Inria's STACK around cloud services collaboration. Notably, the proposition Cheops [180] emerges as a building block to generally approach cloud services to the edge of the DCI, potentially including the contributions of DIMINET for

inter-service collaboration. We hope that these propositions and others conceived at these thesis could use the ideas and insights we have provided all along this doctoral work.

List of publications

International Journals

— D. Espinel, A. Lebre, L. Nussbaum, and A. Chari, "Decentralized SDN Control Plane for a Distributed Cloud-Edge Infrastructure: A Survey", IEEE Communications Surveys & Tutorials, 2021, doi.10.1109/COMST.2021.3050297.

International Conferences

— D. Espinel, A. Lebre, L. Nussbaum, and A. Chari, "Multi-site Connectivity for Edge Infrastructures DIMINET:DIstributed Module for Inter-site NETworking", 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, 2020, doi.10.1109/CCGrid49817.2020.00-81.

Workshops

— J. M. S. Vilchez and D. E. Sarmiento, "Fault Tolerance Comparison of ONOS and OpenDaylight SDN Controllers," 2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft), Montreal, QC, Canada, 2018, pp. 277-282, doi: 10.1109/NETSOFT.2018.8460099.

National Conferences

— D. Espinel, A. Lebre, L. Nussbaum, and A. Chari, "Distributing connectivity management in Cloud-Edge infrastructures: Challenges and approaches", Conférence d'informatique en Parallélisme, Architecture et Système, 2019.

Part V

Appendices

Appendix A

DIMINET Logic Core implementation

A.1 DIMINET Logic Core implementation

A.1.1 L3 Routing Resource Creation

As described in Section 9.3.1.1 for L3 routing Resources, the provisioning of this kind of Resource has been implemented as depicted in Figure A.1:

1. The user issues a request to DIMINET A for the creation of an inter-site Resource of type L3 routing among networks A, B, and C, of sites A, B, and C, respectively.
2. DIMINET instance A verifies if the user's request contains all the necessary information and authenticates against Keystone to find the information of remote deployments.
3. DIMINET instance A requests the remote sites B and C, and its local Neutron about the network-related information to find out the subnetworks' identifiers. In our implementation, this is done in parallel because remote network information can be provided without dependency among the requests.
4. Once DIMINET instance A finishes the query, it does the overlapping verification locally.
5. Since the verification is good, DIMINET instance A proceeds to create the *Resource* schema at its local DB and all its related dependencies: It instantiates a *Parameter* object which comprises a *LMaster* object because DIMINET A is the *Resource* Master. Since the *Resource* is of L3 routing type, several *L3Cidr* objects are created at *LMaster* to store the information related to *SubResources*' CIDRs. This process will allow the Master module to execute modifications on the *Resource* when demanded.
6. DIMINET instance A proceeds to request Neutron API A to create the data plane interconnection objects.

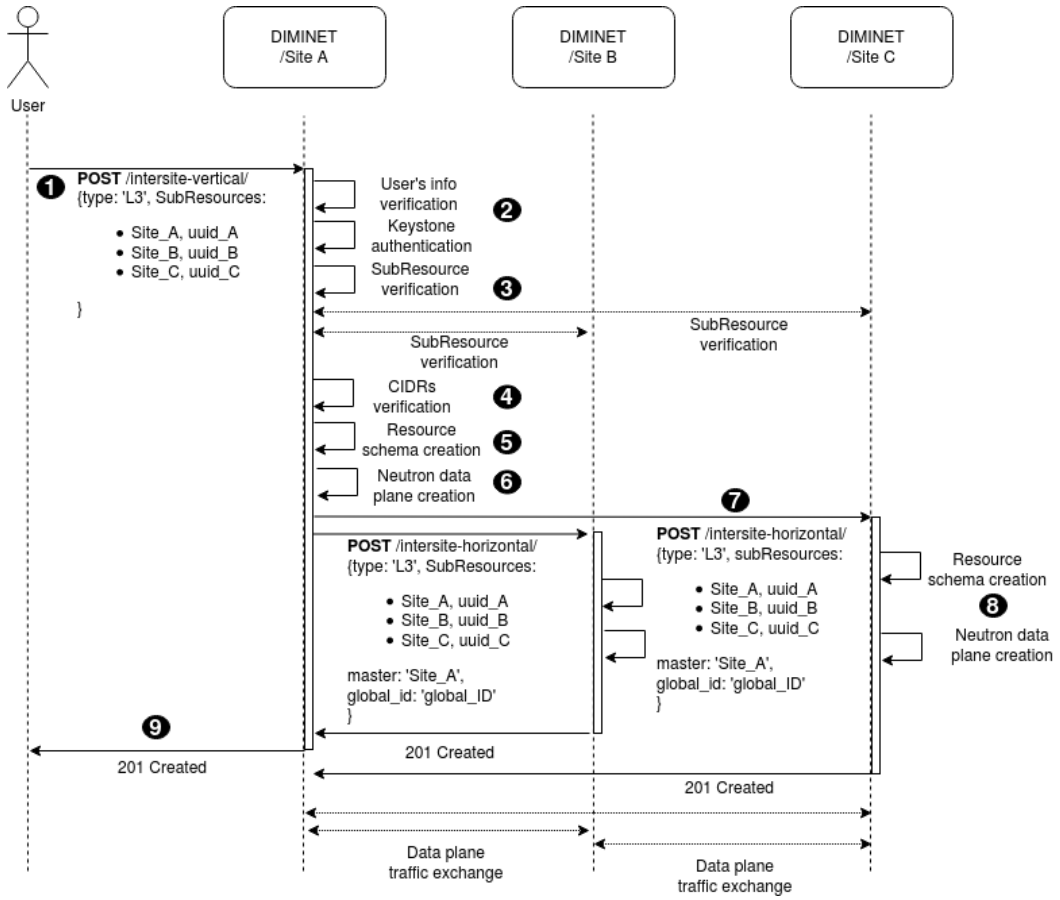


Figure A.1 – DIMINET L3 routing Resource creation sequence diagram.

7. Next, DIMINET instance A sends in parallel the create API request to remote DIMINET instances of sites B and C. The East-West requests comprise information such as the global_id (generated at the Logic Core), the *SubResources* list, and the announcement that DIMINET instance A is the *Resource* Master. The instance waits for remote answers to continue.
8. Remote DIMINETs create the *Resource* schema along with the Neutron data plane interconnection objects.
9. Once all the remote instances answered the horizontal request, DIMINET instance A proceeds to answer the original user request with a creation acknowledgement.

A.1.2 L2 Extension Resource Creation

As described in Section 9.3.1.2 for L2 extension *Resources*, the provisioning of this kind of *Resource* has been implemented as depicted in Figure A.2:

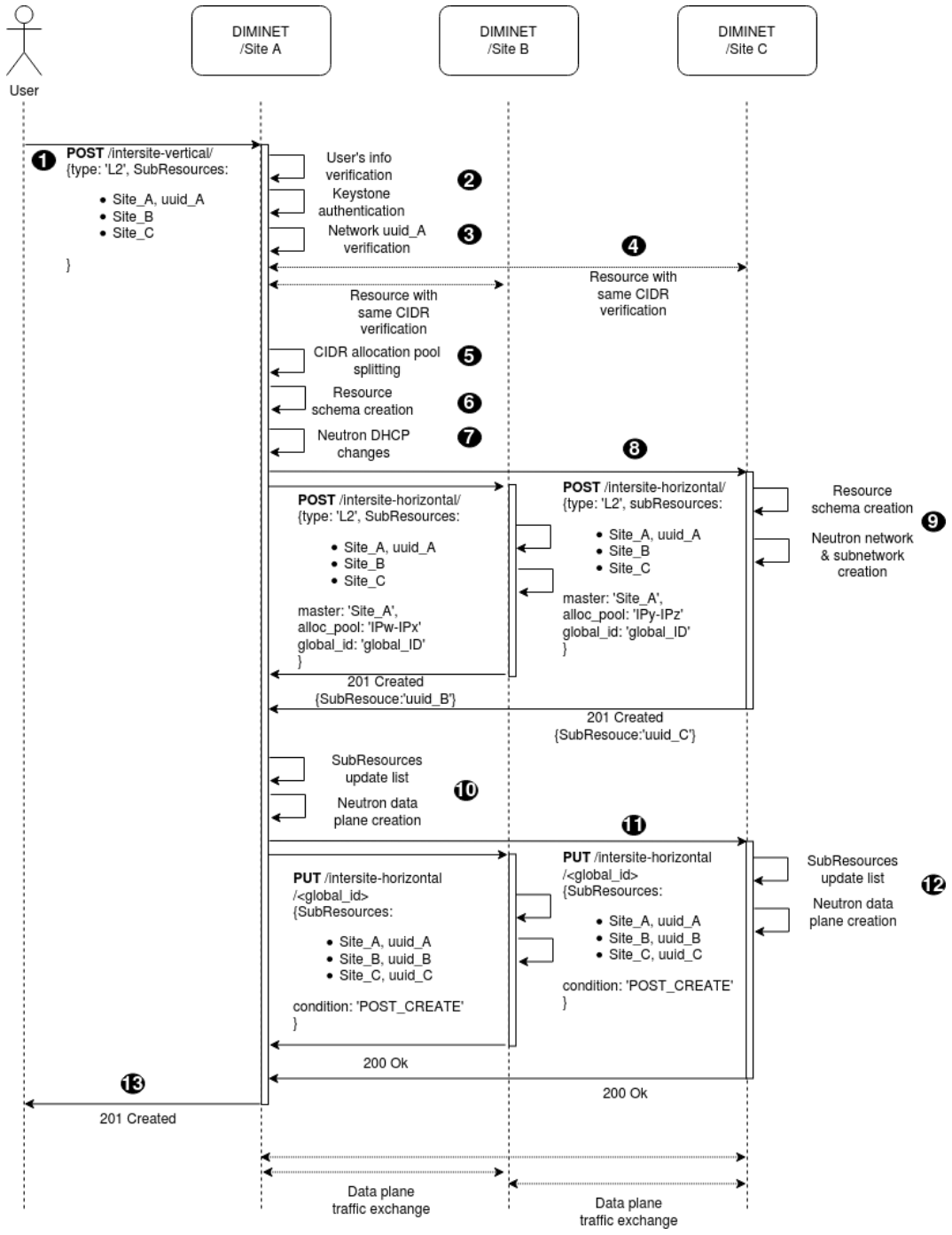


Figure A.2 – DIMINET L2 extension Resource creation sequence diagram.

1. The user request to DIMINET A the creation of an inter-site *Resource* of type L2 extension for network A of site A to be extended to sites B and C.

2. DIMINET instance A verifies the user's information and authenticates against Key-

stone to find the information of remote deployments.

3. DIMINET instance A requests its Neutron deployment about the information of network A to gather all specific information such as the subnetwork CIDR.
4. DIMINET instance A requests sites B and C about *Resources* of type L2 extension with the requested CIDR. This verification is done because distant sites may already have an L2 extension *Resource* deployed with the demanded CIDR. In such a case, it will be better to modify the *Resource* adding the site A rather than re-creating a similar *Resource*.
5. Since the verification is satisfactory, DIMINET instance A proceeds to calculate the IP allocation pools that will be assigned to sites B and C.
6. Similar as for L3 routing *Resources*, DIMINET instance A proceeds to create the *Resource* schema at the DB and all its related dependencies: It instantiates a *Parameter* object which comprises a *LMaster* object because DIMINET A is the *Resource* Master. Since the *Resource* is of L2 extension, several *L2AllocPools* objects are created at *LMaster* to store the information related to the allocation pools conceived to remote sites. This process will allow the Master module to execute modifications on the *Resource* when demanded.
7. Since the allocation pools affectation is for all the sites composing the *Resource*, DIMINET instance A does the necessary API calls to Neutron for changing the DHCP parameters of network A.
8. Next, the instance sends in parallel the horizontal create API request to remote DIMINET instances of sites B and C. The horizontal requests comprise information such as the global_id (generated at the Logic Core), the *SubResources* list, and the announcement that DIMINET instance A is the *Resource* Master. Moreover, the allocation pool defined in step five is sent in this request. The instance waits for remote answers to continue.
9. Remote DIMINETs create the *Resource* schema with the information provided by the Master. Since the *Resource* is of L2 extension, DIMINET modules proceed to request Neutron *API* the instantiation of a network and subnetwork with the characteristics provided by the Master. These resources will act as the extension of network A of site A. The answer to this request is accompanied with the UUID of networks created at sites B and C.
10. Once all the remote instances answer the horizontal request, DIMINET instance A proceeds to update the list of *SubResources* composing the *Resource*. With the

complete list of *SubResources*, the instance proceeds to call Neutron API to create the data plane interconnection objects.

11. DIMINET instance A sends in parallel horizontal modification API request to remote DIMINET instances of sites B and C in order to update their *SubResource* list.
12. Remote modules update the *SubResources* list and call Neutron API to request the creation of the data plane objects.
13. Once all the remote instances answered the horizontal request, DIMINET instance A proceeds to answer the original user request.

Appendix B

DIMINET REST API Operations

B.1 DIMINET Communication Interfaces: REST API operations

This section details the proposed operations of DIMINET to end-users and among DIMINET instances.

B.1.1 North Interface: User-to-Module Communication Exchanges

As stated in Section 9.4, DIMINET uses the vertical or North API interface to communicate with the user. Hence, the User-to-Module interactions occurs using this interface whose list of possible actions is the following:

- **GET** /intersite-vertical: Request to retrieve the local list of all inter-site *Resources*.
- **GET** /intersite-vertical/{global_id}: Request to retrieve an inter-site *Resource* using the global_id.
- **POST** /intersite-vertical: Request the creation of an inter-site *Resource*. For this request, the user needs to provide the following items:
  - *Name*: The name of the inter-site *Resource*.
  - *Type*: The type of networking *Resource* (L2 network extension or L3 routing).
  - *SubResources*: The list of *SubResources*/sites that will compose the inter-site *Resource*. Depending on the *Resource* type, this list may vary. In the case of an L3 routing *Resource*, the list is composed of the network constructions identifiers and their sites that the user wants to be connected by a logical router. In the case of an L2 extension *Resource*, the list is composed of a network construction identifier and a list of sites where it will be spanned.
- **PUT** /intersite-vertical/{global_id}: Request the modification of an inter-site *Resource* using the global_id. The modification is proposed to add or delete *SubResources*/sites from the *Resource*. This operation can only be addressed to the *Resource*'s Master module. Thus, the user can provide the following information:
  - *Name*: An updated name for the *Resource*.

— *SubResources*: The updated list of *SubResources*/sites. The module will split these list into the *SubResources* that will be maintained, those who will be added, and those who will be deleted.

- **DELETE** /intersite-vertical/{global_id}: Request to delete an inter-site *Resource* using the global_id. This operation can only be addressed to the *Resource*'s Master module.

B.1.2 East-West Interface: Module-to-Module Interactions

As stated in Section 9.4, DIMINET uses the horizontal or East-West API interface to communicate with other modules when needed. These requests are triggered within some of the intersite-vertical interactions such as the POST, PUT, or DELETE requests and are sent by a Master module. Module-to-Module interactions occur using this interface, whose list of possible actions is the following:

- **GET** /intersite-horizontal: Request to retrieve a list of distant inter-site *Resources* using filters to do verification. The filters that the Master module uses are the following:
  — *Resource_CIDR*: Filter used for L2 network extension *Resources*. It is used to request if the user already has inter-site *Resources* with *Resource_CIDR* in remote modules.
  — *Resource_type*: Filter used for the *Resource* type.
  — *Global_ID*: Filter used to verify if remote *SubResources* can be deleted in order to delete an inter-site *Resource* of type L2 extension.
  — *Verification_type*: Filter used to announce remote modules if the verification is needed for a *CREATE* or a *DELETE* action.
- **GET** /intersite-horizontal/{global_id}: Request to retrieve a distant inter-site *Resource* using the global_id.
- **POST** /intersite-horizontal: Request the creation of a distant inter-site *Resource*. For this request, the Master module provides the following data:
  — *Name*: The name of the inter-site *Resource*.
  — *Global_id*: An identifier generated by the Master module that will globally identify the *Resource*.
  — *Type*: The type of networking *Resource* accordingly with the user's request.
  — *SubResources*: The list of *SubResources*/sites that will compose the *Resource*.
  — *Parameters*: A list of *Parameters* defined by the Master module to maintain consistency. Among the *Parameters*, we found the followings:
    — *Allocation pool*: *Parameter* used for L2 network extensions. It contains the

information concerning the allocation pool granted by the Master module.
  - *Local CIDR*: *Parameter* used for L2 network extension affirming the CIDR of the *Resource*. In the case of the L3 routing type, it is empty but locally filled with the CIDR of the local *SubResource*.
  - *IPv*: *Parameter* used to announce the IP version of a Resource (only IPv4 has been implemented.)
  - *Master*: *Parameter* used to announce the identity of the Master module for a particular *Resource*. In our implementation, this information corresponds with the OpenStack Region's name, where the Master module is deployed.
  - *Master auth*: *Parameter* providing the authentication Uniform Resource Locator (URL) of the Master module. In our implementation, this information corresponds with the URL of the OpenStack Region where the Master module is deployed.
- **PUT** /intersite-horizontal/{global_id}: Request the modification of a distant intersite *Resource* using the global_id. For this request, the Master module provides the following data:
  - *Name*: An updated name for the *Resource*.
  - *SubResources*: The updated list of *SubResources*/sites.
  - *Post create refresh*: This condition is used to define if the PUT method's behavior will be the default one or the one designed for the L2 network extension *Resource*. Because of its nature, when provisioning an L2 network extension, the Master module needs to send an updated list of the created *SubResources* to remote modules. Therefore, this is done as an update of remote *Resources*. When this condition is false, the default behavior to update a *Resource* is applied.
  - *Type*: The type of networking *Resource*.
- **DELETE** /intersite-horizontal/{global_id}: Request the removal of a distant intersite *Resource* using the global_id.

Appendix C

DIMINET with OpenStack: User Guide

C.1 DIMINET with OpenStack: User Guide

DIMINET is a distributed/decentralized module for inter-site networking resources capable to interconnect independent networking resources in an automatized and transparent manner. Layer 2 network extensions and Layer 3 routing functions are two main implementation tasks. This first proof-of-concept of the proposed solution has been implemented besides the networking service of Openstack, Neutron.

While this project is independent of Neutron networking API service, it acts as an plugin deployed on the same networking node of Neutron, such add-on service will be very useful to manage and utilize independent geo-distributed networking resources for services like network slicing.

This is an incremental effort based on OpenStack Networking services and its existing APIs.

C.1.1 OpenStack requirements

To be able to use DIMINET, the OpenStack installation needs to have the *bgpvpn* and *interconnection* Plug-ins activated at Neutron. The *neutron.conf* file should looks like:

```
[DEFAULT]
service_plugins=bgpvpn,interconnection
```

If using Devstack, the *local.conf* file should have the next lines:

```
enable_plugin networking-bgpvpn
https://git.openstack.org/openstack/networking-bgpvpn.git
stable/stein
enable_plugin networking-bagpipe
https://git.openstack.org/openstack/networking-bagpipe.git
stable/stein
enable_plugin neutron-interconnection
https://daespinel/neutron-inter.git stable/stein

NETWORKING_BGPVPN_DRIVER="BGPVPN:
BaGPipe:networking_bgpvpn.neutron.services.
service_drivers.bagpipe.bagpipe_v2.
BaGPipeBGPVPNDriver:default"

enable_service b-bgp
BAGPIPE_DATAPLANE_DRIVER_IPVPN=ovs
BAGPIPE_DATAPLANE_DRIVER_EVPN=ovs
BAGPIPE_BGP_PEERS=<BGP-PEER-IP>

[[post-config|$NEUTRON_CONF]]
[neutron_interconnection]
region_name = <REGION_NAME>
router_driver = bgpvpn
network_l3_driver = bgpvpn
network_l2_driver = bgpvpn
bgpvpn_rtnn = <RTT_LABELS>
username = neutron
password = secret
project = service
check_state_interval = 5
```

C.1.2 Installing DIMINET

Git clone this repository with the following command:

```
git clone https://github.com/daespinel/intersite.git
cd intersite
```

Create a virtual environment.

```
virtualenv -p python3 venv
source venv/bin/activate
pip install -U pip
pip install -r requirements.txt
```

C.1.3 Configuring DIMINET

Diminet require you to configure a simple configuration located in config/services.conf file:

```
[DEFAULT]
#IP address of the Host where also the local Neutron
server is located
host = <HOST_IP>
#Region Name of the local region deployment in order to
authenticate to the local Keystone service
region_name = <REGION_NAME>
#User info to authenticate against Keystone
username = <USER_NAME>
password = <USER_PASSWORD>
project = <PROJECT_NAME>
#Keystone authentication URL
auth url = http://<AUTH_URL>/identity/v3
```

Finally, run the following script to build the DB:

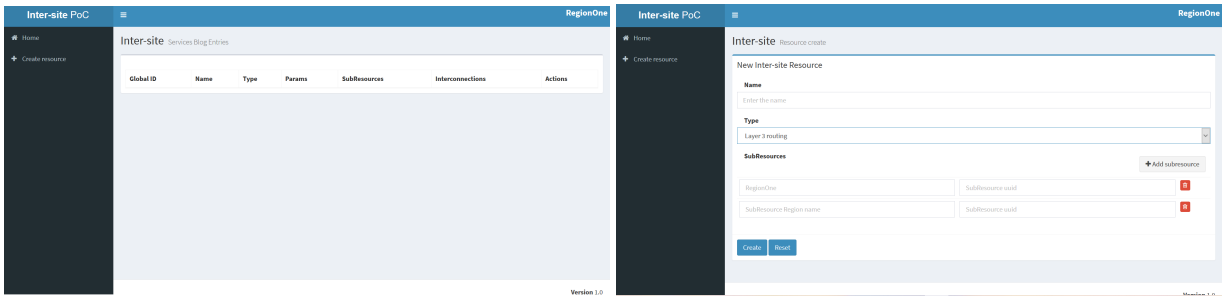
```
python build_database.py
```

C.1.4 Deploying Inter-site resources with DIMINET example

Once the previous steps have been done, the DIMINET server is executed with the following command:

```
python app.py
```

The user can access the <HOST_IP>:7575 url to access the GUI. As the server has been implemented using Flask and Swagger frameworks, it proposes a documented API in <HOST_IP>:7575/api/ui/ . Figure C.1 depicts DIMINET GUIs.



(a) DIMINET home GUI

(b) DIMINET create resource GUI

Figure C.1 – DIMINET GUIs.

DIMINET proposes two types of *Resources*, each one requiring to provide different information:

C.2 Layer 2 extension Resource

As it names indicates, this *Resource* provides an extension of a Layer 2 network (in Neutron, a network object with its subnetwork) to be extended to remote sites. The user needs to provide:

```
#Local region name with the Neutron network uuid that
will be extended:
# e.g., "RegionOne,3b8360e6-e29a-4063-a8bc-7bbd0785d08b"
(the network has a CIDR 10.0.0.0/24)
<LOCAL_REGION_NAME>,<NEUTRON_NETWORK_UUID>
# A list of remote sites where the network will be
extended
# e.g., "RegionTwo","RegionThree","RegionFour"
<REMOTE_REGION_ONE>, <REMOTE_REGION_TWO>,
<REMOTE_REGION_THREE>
```

C.3 Layer 3 routing resource

This *Resource* provides a logical router among several existing and independent network resources (*i.e.,* networks and their subnetworks) deployed in different sites.

```
 # Local region name with the Neutron network uuid
# e.g., "RegionOne,3b8360e6-e29a-4063-a8bc-7bbd0785d08b",
(the network has a CIDR 10.0.0.0/24)
"<LOCAL_REGION_NAME>,<NEUTRON_NETWORK_UUID>"
# A list of remote sites where the
network will be extended # e.g.,
"RegionTwo,c58089b1-c083-4532-9d7d-85d531097a62",
"RegionThree,3feae7ca-e66c-4006-aced-5f3a819c91f6",
(the network has a CIDR 10.0.1.0/24)
"RegionFour,5861e31f-074d-4f0b-a091-de569e5108fa",
(the network has a CIDR 10.0.2.0/24)
"<REMOTE_REGION_ONE>,<NEUTRON_NETWORK_UUID>",
"<REMOTE_REGION_TWO>,<NEUTRON_NETWORK_UUID>",
"<REMOTE_REGION_THREE>,<NEUTRON_NETWORK_UUID>"
```

Appendix D

BGP Scalability

D.1 BGP Scalability: The Route Reflector Method

In traditional BGP deployments, an AS with IBGP requires that all the IBGP peers connect to each other in a full mesh. Such configuration requires that each BGP peers maintain a session to every other peer. In large networks, maintaining a high number of sessions may degrade the BGP instance performance.

To address this scalability issue, a method known as *route reflection* can be used to alleviate the need for a full mesh connectivity among IBGP peers [181]. The route reflection is an operation where a BGP speaker advertises an IBGP learned route to another IBGP peer. The BGP speaker receives the name of Route Reflector (RR), and a advertised route is said to be a reflected route.

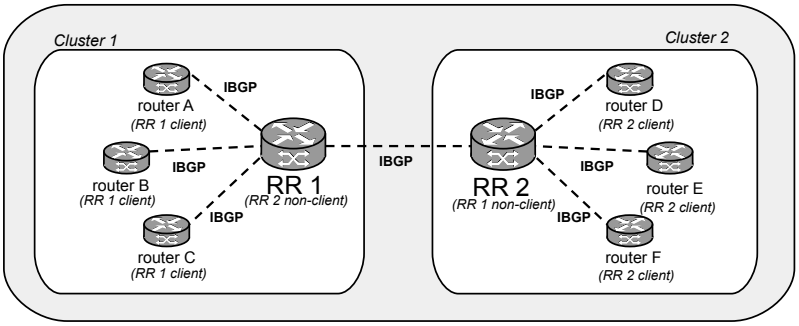


Figure D.1 – BPG Route Reflectors.

The peers of a RR are divided in two categories: client peers and non-client peers. The RR reflects routes between these groups, and may reflect routes among client peers. The set of client peers and the RR is called cluster. The non-client peer must be fully meshed but the client peers need not be fully meshed. An AS may have several RRs. In such case, each RR will be configured with other RRs as non-client peers (*i.e.,* the RRs will be fully meshed among them). The clients will be configured to maintain a IBGP session only with the RR in their cluster. Figure D.1 depicts an example of such configuration with

clusters 1 and 2 each one with a RR establishing a session among them and with several clients attached.

Figure D.2 presents a deployment of DIMINET and OpenStack including the use of a RR among several BGP peers each one connected to a Neutron deployment. One non-client session is present too to show the possibility of deploying more than one RR. As stated above, when using such configuration, there is no need to establish a full mesh among all the BGP peers but only to establish a session with the RR.

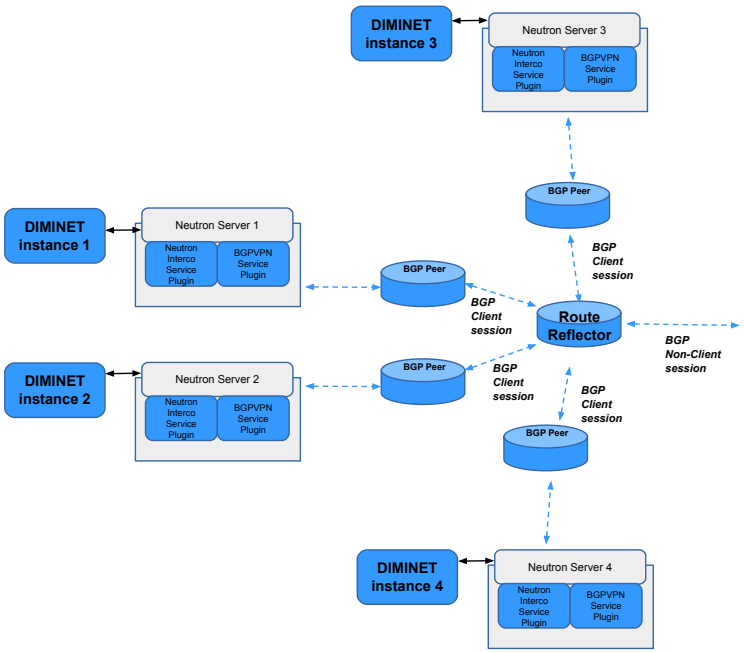


Figure D.2 – DIMINET & OpenStack with BPG Route Reflectors.

BIBLIOGRAPHY

[1] D. Sabella and A. Vaillant and P. Kuure and U. Rauschenbach and F. Giust, "Mobile-Edge Computing architecture: The role of MEC in the Internet of Things," *IEEE Consumer Electronics Magazine*, vol. 5, no. 4, pp. 84–91, Oct 2016.

[2] 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.

[3] ETSI, "Network Functions Virtualisation (NFV) Ecosystem, Report on SDN Usage in NFV Architectural Framework," https://www.etsi.org/deliver/etsi_gs/NFV-EVE/001_099/005/01.01.01_60/gs_NFV-EVE005v010101p.pdf, European Telecommunications Standards Institute, Tech. Rep., 2015.

[4] Zhang, Qi and Cheng, Lu and Boutaba, R., "Cloud Computing: State-of-the-art and Research Challenges," *Journal of Internet Services and Applications*, vol. 1, pp. 7–18, 05 2010.

[5] T. Dillon, C. Wu, and E. Chang, "Cloud Computing: Issues and Challenges," in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, 2010, pp. 27–33.

[6] Mell, Peter M. and Grance, Timothy, "SP 800-145. The NIST Definition of Cloud Computing," National Institute of Standards & Technology, Gaithersburg, MD, USA, Tech. Rep., 2011.

[7] H. AlJahdali, A. Albatli, P. Garraghan, P. Townend, L. Lau, and J. Xu, "Multi-tenancy in Cloud Computing," in *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, 2014, pp. 344–351.

[8] Simmon, Eric, "Evaluation of Cloud Computing Services Based on NIST SP 800-145," National Institute of Standards & Technology, Gaithersburg, MD, USA, Tech. Rep., 2018.

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

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

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

[12] C. N. Hoefer and G. Karagiannis, "Taxonomy of cloud computing services," in *2010 IEEE Globecom Workshops*, 2010, pp. 1345–1350.

[13] Y. Xing and Y. Zhan, "Virtualization and cloud computing," in *Future Wireless Networks and Information Systems*. Springer, 2012, pp. 305–312.

[14] Jamsa, Kris, *Cloud computing: SaaS, PaaS, IaaS, virtualization, business models, mobile, security and more*. Jones & Bartlett Publishers, 2012.

[15] A. Randal, “The ideal versus the real: Revisiting the history of virtual machines and containers,” *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3365199

[16] ETSI, “Network Functions Virtualisation (NFV); Management and Orchestration ,” https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf, European Telecommunications Standards Institute, Tech. Rep., 2014.

[17] The Open Infrastructure Foundation, “OpenStack,” https://www.openstack.org, 2020.

[18] OpenNebula Systems, “OpenNebula,” https://opennebula.io/, 2020.

[19] AppScale Systems, “Eucalyptus,” https://www.eucalyptus.cloud/, 2020.

[20] ETSI, “Network Functions Virtualisation (NFV) Release 3; Architecture; Report on the Enhancements of the NFV architecture towards "Cloud-native" and "PaaS" ,” https://www.etsi.org/deliver/etsi_gr/NFV-IFA/001_099/029/03.03.01_60/gr_NFV-IFA029v030301p.pdf, European Telecommunications Standards Institute, Tech. Rep., 2019.

[21] Linux Foundation, “Kubernetes,” https://kubernetes.io/docs/home/, 2020.

[22] The Apache Software Foundation, “Apache Mesos,” http://mesos.apache.org/, 2020.

[23] The Open Infrastructure Foundation, “OpenStack Nova Project,” https://docs.openstack.org/nova/latest/, 2020.

[24] ——, “OpenStack Cinder Project,” https://docs.openstack.org/cinder/latest/, 2020.

[25] ——, “Neutron - Openstack Networking Service,” https://docs.openstack.org/neutron/latest/, 2020.

[26] ——, “OpenStack Keystone Project,” https://docs.openstack.org/keystone/latest/, 2020.

[27] A. Randal, “The ideal versus the real: Revisiting the history of virtual machines and containers,” *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3365199

[28] S. Singh and N. Singh, “Containers & Docker: Emerging roles & future of Cloud technology,” in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 01 2016, pp. 804–807.

[29] 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*, ser. Middleware ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2988336.2988337

[30] Docker, “Docker,” https://www.docker.com/, 2020.

[31] Canonical, “Linux containers,” https://linuxcontainers.org/, 2020.

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

[33] ——, “Kubernetes API overview,” https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.18/, 2020.

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

[35] L. Yang, R. Dantu, T. Anderson, and R. Gopal, "Forwarding and Control Element Separation (ForCES) Framework," https://tools.ietf.org/html/rfc3746, RFC Editor, RFC 3746, April 2004.

[36] 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.

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

[38] 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.

[39] Cisco, "OpFlex: An Open Policy Protocol White Paper," Cisco, Tech. Rep., 2014.

[40] 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.

[41] 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*, 2013, pp. 1–7.

[42] 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*, 2015, pp. 179–188.

[43] 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.

[44] 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*, 2013, pp. 3–14.

[45] 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.

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

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

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

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

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

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

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

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

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

[55] Cilium, "Cilium," https://cilium.io/, 2019.

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

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

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

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

[60] 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.

[61] 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.

[62] OpenStack Foundation, "Edge Computing: Next Steps in Architecture, Design and Testing)," https://www.openstack.org/use-cases/edge-computing/edge-computing-next-steps-in-architecture-design-and-testing/, May 2020.

[63] OpenStack, "KingBird Project," https://wiki.openstack.org/wiki/Kingbird, 2018.

[64] ——, "Tricircle Project," https://wiki.openstack.org/wiki/Tricircle, 2018.

[65] ——, "OpenStack Trio2o," https://wiki.openstack.org/wiki/Trio2o, 2019.

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

[67] 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.

[68] 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.

[69] F. Brasileiro, G. Silva, F. Arajo, M. Nbrega, I. Silva, and G. Rocha, "Fogbow: A middleware for the federation of iaas clouds," *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2016.

[70] R.-A. Cherrueau, A. Lebre, D. Pertin, F. Wuhib, and J. M. Soares, "Edge Computing Resource Management System: a Critical Building Block!" *HotEdge*, 2018.

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

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

[73] OSRG, "GoBGP," https://osrg.github.io/gobgp/, 2018.

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

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

[76] 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.

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

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

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

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

[81] 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.

[82] Moraru, Iulian and Andersen, David and Kaminsky, Michael, "There is more consensus in egalitarian parliaments," in *SOSP 2013 - Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 11 2013, pp. 358–372.

[83] Turcu, Alex and Peluso, Sebastiano and Palmieri, Roberto and Ravindran, Binoy, "Be General and Don't Give Up Consistency in Geo-Replicated Transactional Systems," in *2014 International Conference on Principles of Distributed Systems (OPODIS)*, 12 2014, pp. 33–48.

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

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

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

[87] 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.

[88] Palmieri, Roberto, "Leaderless Consensus: The State of the Art," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 05 2016, pp. 1307–1310.

[89] 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.

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

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

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

[93] R. Cattell, "Scalable sql and nosql data stores," *SIGMOD Rec.*, vol. 39, no. 4, p. 12–27, May 2011. [Online]. Available: https://doi.org/10.1145/1978915.1978919

[94] Khasawneh, Tariq and Alsahlee, Mahmoud and Safia, Ali, "Sql, newsql, and nosql databases: A comparative survey," in *2020 11th International Conference on Information and Communication Systems (ICICS)*, 04 2020, pp. 013–021.

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

[96] M. Paksula, "Persisting objects in redis key-value database, white paper," 2010.

[97] J. Webber, "A programmatic introduction to Neo4j," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, 10 2012, pp. 217–218.

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

[99] A. Davoudian, L. Chen, and M. Liu, "A survey on nosql stores," *ACM Comput. Surv.*, vol. 51, no. 2, Apr. 2018. [Online]. Available: https://doi.org/10.1145/3158661

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

[101] Open Networking Foundation, "OpenFlow Switch Specifications," Open Networking Foundation, Tech. Rep., 2015.

[102] 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.

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

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

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

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

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

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

[109] A. Dixit, F. Hao, S. Mukherjee, Lakshman, and R. K. t, "Towards an Elastic Distributed SDN Controller," *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013.

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

[111] 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.

[112] S. Yeganeh and Y. Ganjali, "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications," *Proceedings of the first ACM SIGCOMM workshop on Hot topics in software defined networking*, 2012.

[113] 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.

[114] S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Computing*, vol. 10, no. 6, pp. 87–89, 2006.

[115] "Hazelcast Project," https://hazelcast.org/.

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

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

[118] 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.

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

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

[121] 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.

[122] 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," *Proceedings of the third ACM SIGCOMM workshop on Hot topics in software defined networking*, 2014.

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

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

[125] 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.

[126] Apache Software Foundation, “Apache Cassandra,” http://cassandra.apache.org/, 2016.

[127] Cloud Native Computing Foundation, “etcd,” http://etcd.io/, 2016.

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

[129] 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.

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

[131] ——, “OpenDaylight Federation Application,” https://wiki.opendaylight.org/view/Federation:Main, 2016.

[132] ——, “OpenDaylight NetVirt Application,” https://wiki.opendaylight.org/display/ODL/NetVirt, 2020.

[133] ——, “User stories OpenDaylight,” https://www.opendaylight.org/use-cases-and-users/user-stories, 2018.

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

[135] ——, “Apache Karaf,” https://karaf.apache.org/, 2010.

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

[137] ONOS, “ONOS - Community,” https://www.opennetworking.org/onos/, 2020.

[138] ——, “SONA architecture ONOS,” https://wiki.onosproject.org/display/ONOS/SONA+Architecture, 2018.

[139] ——, “CORD VTN ONOS,” https://wiki.onosproject.org/display/ONOS/CORD+VTN, 2018.

[140] ——, “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.

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

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

[143] SIG Multicluster, “Kubernetes Cluster Federation,” https://github.com/kubernetes-sigs/kubefed, 2020.

[144] Istio, “Multicluster Deployments ,” https://istio.io/v1.2/docs/concepts/multicluster-deployments/, 2020.

[145] S. Voulgaris, D. Gavidia, and M. V. Steen, “Cyclon: Inexpensive membership management for unstructured p2p overlays,” *Journal of Network and Systems Management*, vol. 13, pp. 197–217, 2005.

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

[147] G. Antichi and G. Rétvári, "Full-Stack SDN: The Next Big Challenge?" in *Proceedings of the Symposium on SDN Research*, ser. SOSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 48–54. [Online]. Available: https://doi.org/10.1145/3373360.3380834

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

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

[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.

[152] G. Everest, "Basic data structure models explained with a common example." 10 1976.

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

[154] E. Rosen and Y. Rekhter, "BGP/MPLS IP Virtual Private Networks (VPNs)," Internet Requests for Comments, RFC Editor, RFC 4364, February 2006. [Online]. Available: https://tools.ietf.org/html/rfc4364

[155] A. Sajassi, R. Aggarwal, N. Bitar, A. Isaac, J. Uttaro, J. Drake, and W. Henderickx, " BGP MPLS-Based Ethernet VPN," Internet Requests for Comments, RFC Editor, RFC 7432, February 2015. [Online]. Available: https://tools.ietf.org/html/rfc7432

[156] M. Delavergne, "Juice(modified version)," https://github.com/daespinel/juice, 2019.

[157] R.-A. Cherrueau and M. Simonin, "EnOSlib," https://github.com/BeyondTheClouds/enoslib, 2017.

[158] 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.

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

[160] Heinrich, Mark Andrew, "The Performance and Scalability of Distributed Shared-Memory Cache Coherence Protocols," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1999, aAI9924431.

[161] M. Aslett, "How will the database incumbents respond to NoSQL and NewSQL? ," https://www.cs.cmu.edu/~pavlo/courses/fall2013/static/papers/aslett-newsql.pdf, 451 Group, Tech. Rep., April 2011.

[162] S. Harizopoulos, D. Abadi, S. Madden, and M. Stonebraker, "Oltp through the looking glass, and what we found there," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 01 2008, pp. 981–992.

[163] A. Pavlo and M. Aslett, "What's really new with newsql?" *SIGMOD Rec.*, vol. 45, no. 2, p. 45–55, Sep. 2016. [Online]. Available: https://doi.org/10.1145/3003665.3003674

[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] 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.

[166] 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.

[167] 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*, 2011, pp. 195–206.

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

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

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

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

[172] 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*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1041–1052. [Online]. Available: https://doi.org/10.1145/3035918.3056101

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

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

[175] 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.

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

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

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

[179] G. Tato, M. Bertier, E. Rivière, and C. Tedeschi, "Sharelatex on the edge: Evaluation of the hybrid core/edge deployment of a microservices-based application," in *Proceedings of the 3rd Workshop on Middleware for Edge Clouds & Cloudlets*, ser. MECC'18. New York, NY, USA: Association for Computing Machinery, 2018, p. 8–15. [Online]. Available: https://doi.org/10.1145/3286685.3286687

[180] Marie Delavergne and Ronan-Alexandre Cherrueau and Adrien Lebre, "Geo-Distribute Cloud Application at the Edge," *International European Conference on Parallel and Distributed Computing*, 2021.

[181] E. Chen, T. J. Bates, and R. Chandra, "BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP)," RFC 4456, Apr. 2006. [Online]. Available: https://rfc-editor.org/rfc/rfc4456.txt

DOCTORAT BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Gestion distribuée d'un service de connectivité pour une infrastructures Cloud-Edge à partir des approches SDN

Mot clés : Infrastructure géo-distribué, réseau, automatisation, IaaS, Software Defined Network

Résumé : L'évolution du paradigme d'Informatique en nuage au cours de la dernière décennie a permis de démocratiser les services à la demande de manière significative (plus simple d'accès, économiquement attrayant, etc.). Cependant, le modèle actuel construit autour de quelques centres de données de très grande taille ne permettra pas de répondre aux besoins des nouveaux usages liés notamment à l'essor de l'Internet des Objets. Pour mieux répondre à ces nouvelles exigences (en termes de latence, volumétrie, etc.), les ressources de calculs et de stockages doivent être déployées à proximité de l'utilisateur. Dans le cas des opérateurs de télécommunications, les points de présence réseau qu'ils opèrent depuis toujours peuvent être étendus à moindre cout pour héberger ces ressources. La question devient alors : comment gérer une telle infrastructure nativement géo-distribuée (référencée dans le manuscrit sous l'acronyme DCI pour Distributed Cloud Infrastructure) afin d'offrir aux utilisateurs finaux les même services qui ont fait le succès du modèle actuel d'Informatique en nuage. Dans cette thèse réalisée dans un contexte industriel avec Orange Labs, nous étudions le problème de la gestion distribuée de la connectivité entre plusieurs sites d'une DCI et proposons d'y répondre en utilisant les principes des réseaux définis par logiciel (connus sous les termes "Software Defined Network"). De manière plus précise, nous rappelons les problèmes et les limitations concernant la gestion centralisée, et ensuite, examinons les défis pour aller vers un modèle distribué, notamment pour les services liés à la virtualisation réseaux. Nous fournissons une analyse des principaux contrôleurs SDN distribués en indiquant s'ils sont capables ou non de répondre aux défis des DCIs. Sur cette étude détaillé, qui est une première contribution en soi, nous proposons la solution DIMINET, un service en charge de fournir une connectivité à la demande entre plusieurs sites. DIMINET s'appuie sur une architecture distribuée où les instances collaborent entre elles à la demande et avec un échange de trafic minimal pour assurer la gestion de la connectivité. Les leçons aprises durant cette étude nous permettent de proposer les prémisses d'une généralisation afin de pouvoir "distribuer" d'une manière non intrusive n'importe quels services en charge de gérer une infrastructure géo-distribuée.

Title: Distributing connectivity management in Cloud-Edge infrastructures using SDN-based approaches

Keywords: Geo-distributed infrastructure, networking, automation, IaaS, Software-Defined Network

Abstract: The evolution of the cloud computing paradigm in the last decade has amplified the access of on-demand services (economical attractive, easy-to-use manner, etc.). However, the current model built upon a few large datacenters (DC) may not be suited to guarantee the needs of new use cases, notably the boom of the Internet of Things (IoT). To better respond to the new requirements (in terms of delay, traffic, etc.), compute and storage resources should be deployed closer to the end-user. In the case of telecommunication operators, the network Point of Presence (PoP), which they have always operated, can be inexpensively extended to host these resources. The question is then how to manage such a massively Distributed Cloud Infrastructure (DCI) to provide end-users the same services that made the current cloud computing model so successful. In this thesis realized in an industrial context with Orange Labs, we study the inter-site connectivity management issue in DCIs leveraging the Software-Defined Networking (SDN) principles. More in detail, we analyze the problems and limitations related to centralized management, and then, we investigate the challenges related to distributed connectivity management in DCIs. We provide an analysis of major SDN controllers indicating whether they are able or not to answer the DCI challenges in their respective contexts. Based on this detailed study, which is a first contribution on its own, we propose the DIMINET solution, a service in charge of providing on-demand connectivity for multiple sites. DIMINET leverages a logically and physically distributed architecture where instances collaborate on-demand and with minimal traffic exchange to provide inter-site connectivity management. The lessons learned during this study allows us to propose the premises of a generalization in order to be able to distribute in a non-intrusive manner any service in a DCI.