Next-generation SDN based virtualized networks

Myriana Rifai

Ecole doctorale EDSTIC
Research Team: SIS-SigNet

PhD Thesis
to obtain the title of
Docteur en Informatique
de l'Université de Côte d'Azur

by
Myriana RIFAI

Next-Generation SDN Based Virtualized Networks

Directed by: **Guillaume URVOY-KELLER**
Co-supervised by: **Dino LOPEZ-PACHECO**

Defended on 25 September 2017
Jury Members:

Dino	**Lopez-Pacheco**	Maître de Conférences, Université de Côte d'Azur, France	Co-supervisor
Guillaume	**Urvoy-Keller**	Professor, Université de Côte d'Azur, France	Thesis Director
Laurent	**Mathy**	Professor, Université de Liège, Belgium	Reporter
Mathieu	**Bouet**	Research Team Leader, Thales Communications & Security, France	Examinator
Pietro	**Michiardi**	Professor, Eurecom, France	Reporter
Thierry	**Turletti**	Researcher, INRIA Sophia Antipolis, France	Examinator
Yin	**Zhang**	Staff Software Engineer, Facebook, USA	Examinator

Abstract

Software Defined Networking (SDN) was created to provide network programmability and ease complex configuration. Though SDN enhances network performance, it still faces multiple limitations. In this thesis, we build solutions that form a first step towards creating next-generation SDN based networks.

In the first part, we present MINNIE to scale the number of rules of SDN switches far beyond the few thousand rules commonly available in Ternary Content Addressable Memory (TCAM), which permits to handle typical data center traffic at very fine grain. To do so MINNIE dynamically compresses the routing rules installed in the TCAM, increasing the number of rules that can be installed.

In the second part, we tackle the degraded performance of short flows and present our coarse grained scheduling prototype that leverages Software Defined Networking (SDN) switch statistics to decrease their end-to-end delay. Then, we aim at decreasing the 50ms failure protection interval which is not adapted anymore to current broadband speeds and can lead to degraded Quality of Experience (QoE). Our solution, PRoPHYS, leverages the switch statistics in hybrid networks to anticipate link failures drastically decreasing the number of packets lost.

Finally, we tackle the greening problem where often energy efficiency comes at the cost of performance degradation. We present SENAtoR, our solution that leverages SDN nodes in hybrid networks to turn off network devices without hindering the network performance. Finally, we present SEaMLESS that converts idle virtual machine (VM) into virtual network function (VNF) to enable the administrator to further consolidate the data center by turning off more physical server and reuse resources (e.g. RAM) that are otherwise monopolized.

Keywords: SDN, TCAM, Scalability, Hybrid Networks, Performance, Scheduling, Resilience, Energy Efficiency, Service Availability

Résumé

Les réseaux logiciels "*Software Defined Networking (SDN)*" permettent la programmation du réseau et facilitent sa configuration. Bien que SDN puisse améliorer les performances, il reste confronté à de multiples défis. Dans cette thèse, nous avons développé des solutions qui constituent un premier pas vers les réseaux SDN de prochaine génération.

D'abord, nous présentons MINNIE qui permet la scalabilité des commutateurs SDN, qui ne supportent que quelques milliers de règles dans leur coûteuse mémoire TCAM. MINNIE comprime dynamiquement les règles de routage installées dans le TCAM, augmentant ainsi le nombre de règles pouvant être installées.

Ensuite, nous abordons le problème de la dégradation de performance des flux courts avec un prototype d'ordonnancement qui exploite les statistiques des commutateurs pour diminuer leur délai de bout-en-bout. Puis, nous visons à diminuer l'intervalle de protection de 50ms qui n'est plus adapté aux applications modernes et réduit leur qualité d'expérience. Notre solution PRoPHYS s'appuie sur les statistiques des commutateurs dans les réseaux hybrides pour découvrir les pannes de liens plus vite que les solutions existantes.

Enfin, nous abordons le problème de l'efficacité énergétique qui souvent mène à une dégradation de performance. Nous présentons SENAtoR, qui exploite les nœuds SDN en réseaux hybrides pour éteindre les noeuds réseau sans entraver la performance. Également, nous présentons SEaMLESS qui convertit le service fourni par une machine virtuelle inactive en une fonction de réseau virtuelles pour permettre à l'administrateur d'utiliser les ressources bloquées tout en maintenant la disponibilité du service.

Mots clés: SDN, TCAM, Scalabilité, Réseaux Hybrides, Performance, Ordonnancement, Résilience, Efficacité Énergétique, Disponibilité des Services

Acknowledgments

First, I want to thank my supervisors Prof. Guillaume Urvoy-Keller and Mr. Dino Pacheco Lopez for believing in me and for all the help, guidance, support and encouragement that they have offered me during my stay at I3S lab. Working with my professors was a real pleasure, as they were constantly supporting me and they have managed to provide a friendly environment which allowed me to prosper and develop my skills, as well as bypass my flaws.

I also sincerely thank all my friends and colleagues in the SIS team for their support and the nourishing friendly environment that they have provided in I3S lab. It was very nice working with you.

Moreover, I thank the members of the COATI team Joanna Moulierac, Frederic Giroire, Nicolas Huin and Christelle Caillouet and Signet team Engineer Quentin Jacquemart for their cooperation, help and support during the projects that we conducted together.

And, I deeply thank prof. Yusheng Ji for hosting me in her lab at the National Institute of Informatics and for guiding me and providing me with everything I needed during my internship in Tokyo, Japan.

I also thank my parents, grandparents and brother for their help during all the previous years. I would not have been here if it was not for their financial help, their encouragement to develop, prosper, and follow my dreams.

Finally, I dedicate this thesis to my fiancée Majdi Kharroubi and thank him for all his support and encouragement during my PhD years. In particular, I want to thank him for the inspiration that he provided me which allowed me to solve the complex problems that I was faced with during my PhD years, and for his patience when I had to work all day long non-stop.

Résumé Étendu

Les réseaux, tels que les réseaux des centres de données (en anglais *Data Center "DC"*) et les fournisseurs d'accès Internet (en anglais *Internet Service Provider "ISP"*), sont généralement composés de multiples périphériques réseau hétérogènes comme les commutateurs, les routeurs, les pare-feux, etc. Dans les centres de données, les périphériques réseau sont généralement situés dans la même zone géographique et fourni multiples services. D'autre part, les périphériques réseau dans les réseaux ISP, également appelés réseaux backbone, sont dispersés géographiquement autour d'un pays ou d'un continent et génèrent une énorme quantité de trafic. Dans les réseaux traditionnels, ces périphériques réseau doivent être configurés individuellement par les administrateurs réseau utilisant l'interface de ligne de commande (en anglais *Command Line Interface "CLI"*) pour fournir la configuration requise pour appliquer la politique définie. Ce processus est très difficile à gérer, puisque les politiques de réseau sont de plus en plus complexes et parfois restreints par le fournisseur de périphérique réseau [BAM09]. De plus, la complexité de la configuration, l'augmentation de la variabilité du trafic et de la dynamique du réseau a poussé vers la nécessité d'avoir des réseaux programmables qui permettront aux administrateurs de programmer les périphériques pour leur permettre de prendre des mesures automatiquement sur certains événements [NMN+14]. Afin de résoudre tous les problèmes mentionnés auparavant, les réseaux définis par les logiciels (en anglais appelé *Software Defined Network "SDN"*) ont été créés.

SDN est une nouvelle architecture de réseau prometteuse et hautement flexible qui surmonte les limites des mécanismes d'acheminement des réseaux basés sur le protocole Internet (abrégé en IP) existants. En effet, les réseaux SDN peuvent non seulement utiliser l'adresse de destination IP pour prendre des décisions d'acheminement, comme les réseaux IP traditionnelles, mais peuvent également utiliser plusieurs autres informations de l'en-tête de paquets niveau couche de liaison de données, réseau et transport [MAB+08]. Les technologies SDN font également une séparation nette entre le plan de commande et le plan de données où les deux plans sont physiquement séparés. Le plan de données est matérialisé par les commutateurs, et le plan de commande est déplacé vers une entité centralisée, le contrôleur (Figure 0.2). Dans les réseaux SDN, les commutateurs SDN sont considérés comme des périphériques qui ne suivent que les politiques de transfert dictées par une entité programmable externe: le contrôleur, qui implémente le plan de contrôle (Figure 0.1). Ainsi, il n'est plus nécessaire d'avoir plusieurs périphériques réseau en boîte noire avec des fonctions de réseau spécifiques telles que les commutateurs, les routeurs, les pare-feux, etc. De plus, il n'est plus nécessaire d'exécuter une commande pour configurer des périphériques réseau

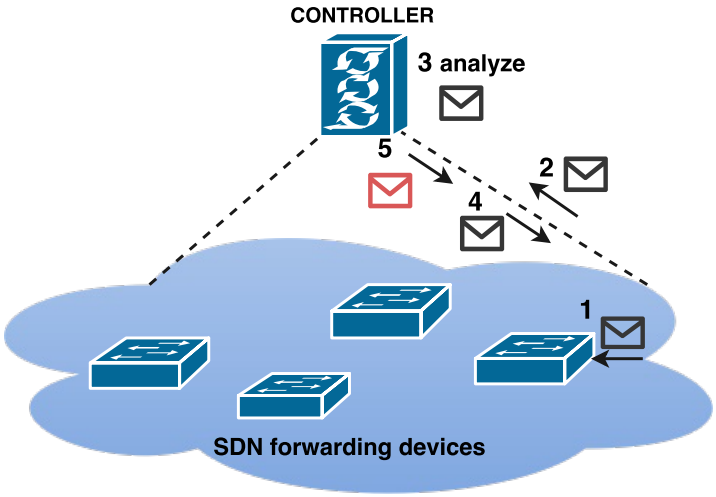


Figure 0.1: Traitement des paquets dans un réseau SDN.

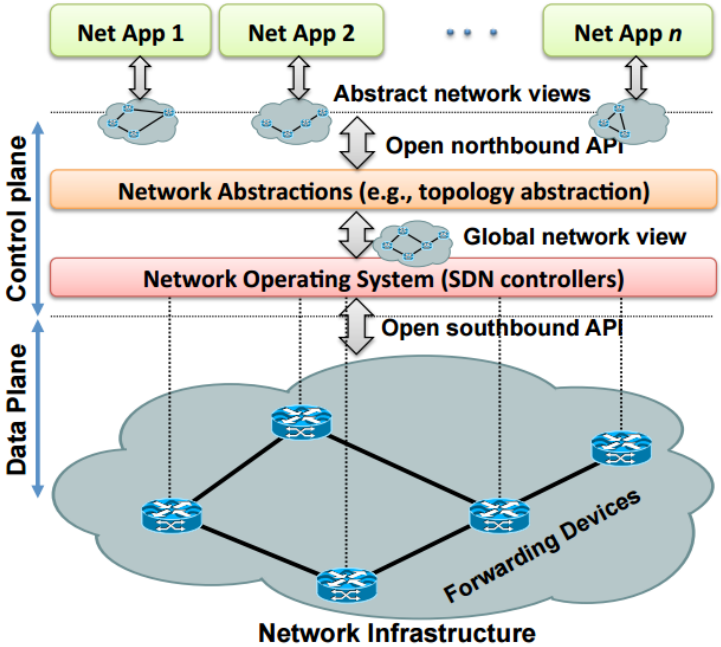


Figure 0.2: Structure de réseau basé sur SDN [KREV+15]

et de répéter ces opérations pour chaque appareil concerné. Dans les réseaux SDN, toute modification nécessaire dans les politiques d'acheminement est codée dans le contrôleur, et ce dernier propagera les règles d'acheminement à tous les équipements SDN [HPO13].

La création de SDN qui permet la centralisation du plan de contrôle, la vue globale du réseau et la programmabilité du réseau offre la possibilité de créer de nouveaux services qui permettent d'améliorer les performances des réseaux traditionnels et de contourner leurs limites et restrictions. Cependant, SDN est une nouvelle architecture de réseau évolutive qui en est encore à ses balbutiements. Pour permettre sa maturité et le déploiement complet des réseaux SDN, plusieurs sujets de recherche, y compris la conception, la scalabilité et la résilience des commutateurs et des contrôleurs, sont encore en cours d'étude [KREV+15].

Dans cette thèse, nous abordons d'abord le problème de scalabilité des commutateurs SDN. Le principal problème qui se pose ici, c'est que les commutateurs SDN utilisent la mémoire ternaire adressable par contenu (en anglais *Ternary Content-Addressable Memory "TCAM"*) pour enregistrer les règles de transfert et fournir les meilleures performances. Cependant, cette mémoire est à la fois coûteuse et consomme trop d'énergie. Pour résoudre ce problème, nous avons créé une solution appelée MINNIE. MINNIE est une solution construite en tant que module du contrôleur Beacon qui maximise l'utilisation de l'espace TCAM. Lorsque la limite TCAM est atteinte sur un nœud SDN, MINNIE comprime automatiquement du côté du contrôleur la table d'envoi du périphérique SDN concerné, en fonction des adresses IP source ou destination. Ensuite, MINNIE transmet la nouvelle table de routage compressée au périphérique SDN pour remplacer l'ancienne table de routage qui a utilisé l'espace SDN complet. À l'aide d'expérimentations, nous prouvons que MINNIE fourni un taux de compression entre 71% et 97% quelle que soit la topologie DC (Figure 0.3) et n'a pas d'impact négatif sur les performances du réseau. Nous avons également prouvé numériquement, que la durée de compression de MINNIE est de l'ordre de quelques millisecondes. MINNIE peut compressé des millions des règles et la plupart des topologies de réseau peuvent installer toutes leurs règles avec un espace TCAM de 1000 règles.

Ensuite, dans la deuxième partie de cette thèse, nous utilisons des dispositifs SDN pour améliorer la performance des flux. Dans cette partie, nous tirons parti de la centralité du contrôleur et de la programmation du réseau, et créons d'abord un nouveau prototype d'ordonnancement qui peut détecter dynamiquement les flux longs dans le réseau et les deprioritiser sans modifier les hôtes finaux ou les périphériques réseau. Ensuite, nous créons une deuxième solution nommée PRoPHYS: *Providing Resilient Path in HYbrid Sdn* qui vise à fournir un chemin résilient dans un réseau SDN hybride. PRoPHYS insère des périphériques SDN dans les réseaux traditionnelles pour créer des réseaux hybrides afin d'améliorer la résilience des flux en détectant les défaillances de liens qui sont connectées à des nœuds SDN ou des routeurs traditionnels avant qu'ils ne soient déclarés comme tels par le périphérique voisin. Cette méthodologie permet ainsi de diminuer le nombre de paquets et connexions perdus lors du reroutage du flux avant que les pannes de liaison ne soient déclarées.

L'ordonnanceur demande les statistiques de renvoi des dispositifs SDN pour détecter les

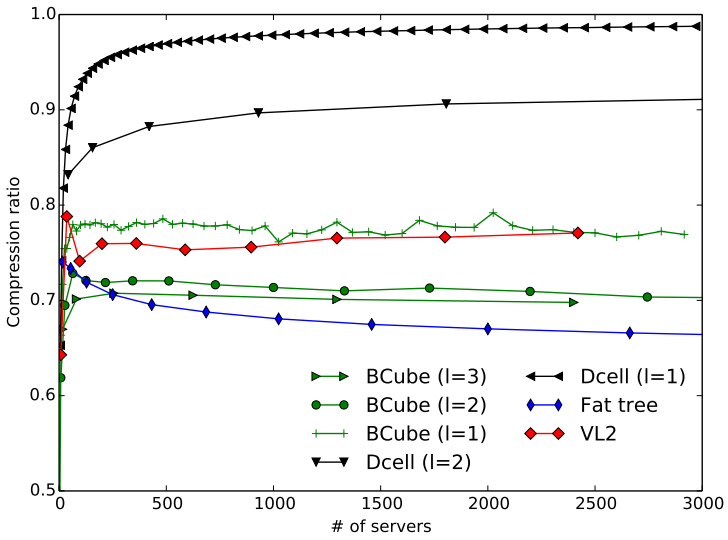
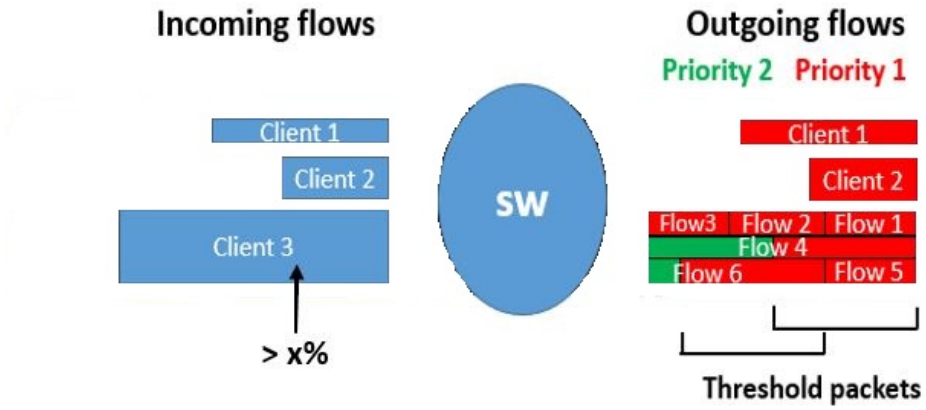


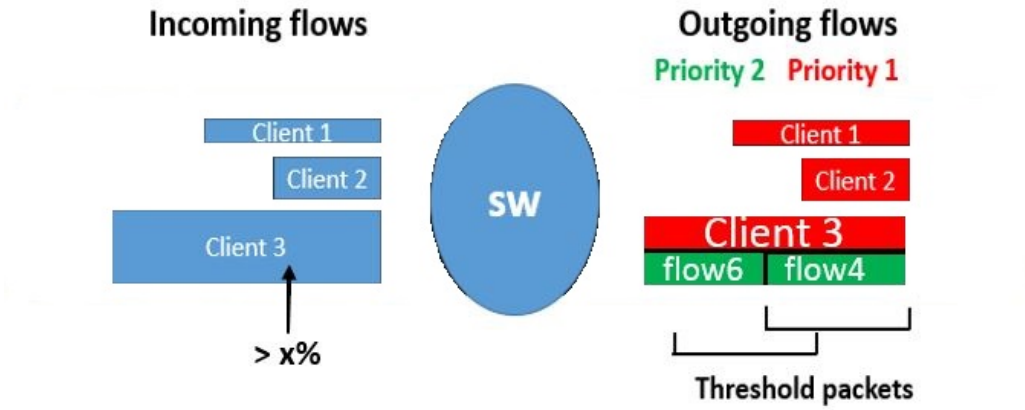
Figure 0.3: Taux de compression de MINNIE.

grands flux. Notre prototype comporte deux ordonnanceurs: (i) à état et (ii) sans état. L'ordonnanceur à état détecte des gros flux en extrayant les statistiques de chaque flux des périphériques SDN. D'autre part, l'ordonnanceur sans état surveille d'abord l'utilisation de la bande passante du client. Ensuite, lorsque l'utilisation du trafic client est supérieure à son seuil de bande passante prédéfini, l'ordonnanceur zoome dans le trafic du client et utilise l'ordonnanceur à état pour détecter les grands flux (Figure 0.4). Les grands flux sont détectés en surveillant le nombre de paquets transmis de chaque flux. Tout flux qui a transmis plus que le seuil de paquets défini par l'administrateur (par exemple 100 paquets) est considéré comme un flux grand. Après avoir détecté les gros flux, ce prototype - dans les deux variantes - change la priorité des grands flux (passe de la file d'attente de priorité la plus élevée à la file d'attente de priorité la plus basse) afin de permettre aux flux courts de se terminer rapidement. Les résultats des tests ont montré que cette solution était efficace sur les petites topologies linéaires, où tous les flux courts ont réussi à se terminer rapidement. Cependant, ces ordonnanceurs ont offerts de moins bonnes performances sur les topologies de l'arbre et des VL2 (Figure 0.5). Pour cette raison, nous avons abandonné cette piste d'étude et nous avons capitalisé sur le savoir acquis dans cette étude pour l'étude suivante qui porte sur la résilience des réseaux SDN hybrides.

En effet après avoir exploité la centralité du contrôleur pour améliorer la performance du réseau, nous l'avons exploité pour améliorer la résilience du réseau. Pour fournir de la résilience, notre solution PRoPHYS, dispose de deux méthodes pour améliorer la résilience du réseau en diminuant le temps de détection de la défaillance de la liaison dans les réseaux hybrides SDN. La première méthodologie estime le dysfonctionnement d'une liaison en détectant les divergences entre les statistiques des ports transmis et reçus du même ensemble de flux sur les nœuds SDN. Cette méthodologie construit d'abord une matrice de ports communicants. On surveille ensuite les statistiques transmises et reçues de cet ensemble de ports au niveau du contrôleur. Une fois



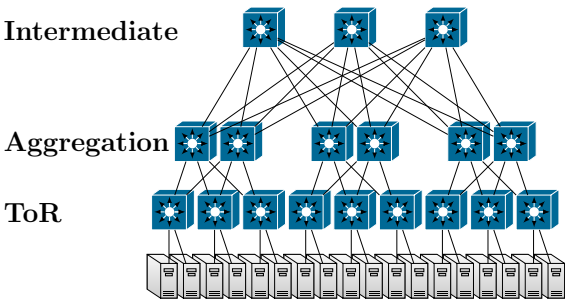
(a) Étape 1



(b) Étape 2

Figure 0.4: Ordonnanceur à état.

Figure 0.5: Réseau VL2.



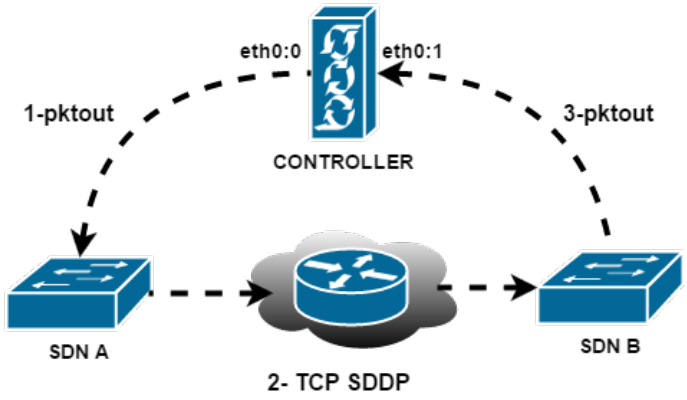


Figure 0.6: Transmission d'un paquet sonde de suivi du chemin issu du contrôleur.

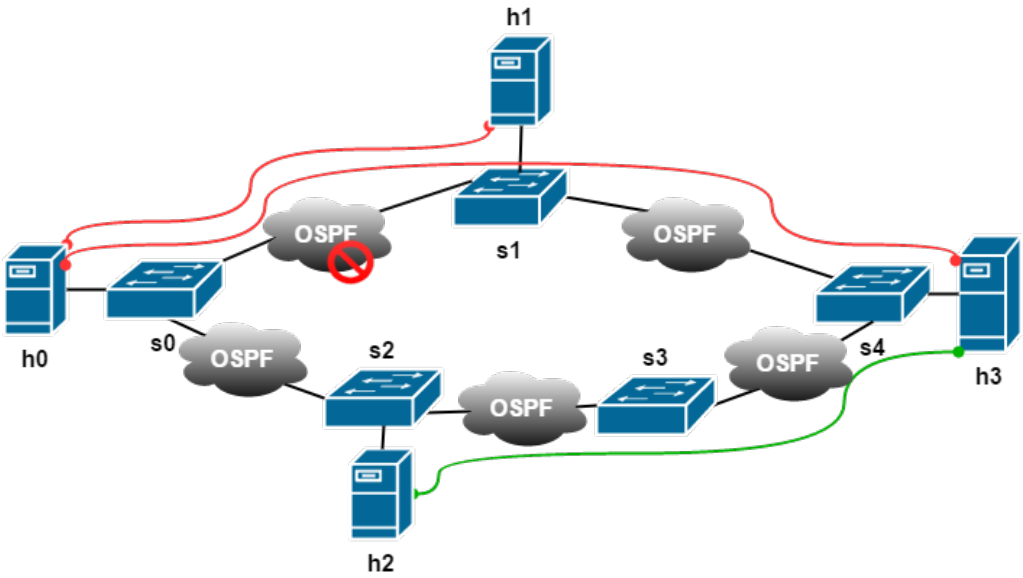


Figure 0.7: Réseau de testde PRoPHYS.

que cette méthode détecte que les statistiques transmise sont inférieures aux statistiques reçues sur un segment de réseau, cela suppose l'existence d'une panne de lien ou de nœud dans ce segment de réseau. Cette méthodologie a permis de réduire l'intervalle de détection de défaillance de liaison/nœud réseau de 50%. La deuxième méthodologie utilisée par PRoPHYS dépend de la transmission d'un paquet sonde de suivi du chemin issu du contrôleur (Figure 0.6), au lieu des commutateurs, ce qui diminue la surcharge des processeurs généraux (non dédiés au transfert de paquets entre interface) dans les commutateurs. Cette méthodologie est plus rapide que les méthodes traditionnelles de détection de panne telles que la détection de transfert bidirectionnel (en anglais *Bidirectional Forwarding Detection "BFD"*) car elle détecte une défaillance de lien ou de segment une fois que le paquet n'est pas reçu après un délai d'attente moyen dynamique calculé en fonction du délai réel entre les nœuds au lieu d'un délai d'attente fixe. Nos simulations utisant la topologie dans Figure 0.7 prouvent que cette méthodologie a également permis de réduire de 50% le nombre de pertes de paquets par rapport aux méthodes classiques de détection des pannes. Les deux méthodologies (envoi de paquets sondes et suivi des statistiques) ont permis de maintenir les connexions vivantes malgré les pannes introduises.

La dernière partie de cette thèse est consacrée à résoudre le problème de l'efficacité énergétique des réseaux actuels, où l'efficacité énergétique vient habituellement au détriment de la performance du réseau. La première solution que nous proposons est appelée SENAtoR (en anglais *Smooth ENergy Aware Routing*). SENAtoR utilise les nœuds SDN pour désactiver les périphériques réseau sans perdre des paquets lors de la désactivation des liens, des pannes de liens ou lorsque des pics de trafic se produisent. La deuxième solution que nous proposons dans cette section s'appelle SEaMLESS. SEaMLESS transforme les services des machines virtuelles VMs inactifs en fonctions virtuelles des réseaux (en anglais *Virtual Network Functions "VNFs"*) dans le but de maintenir les services disponibles tout le temps, permettre une meilleure consolidation de serveur (c'est à dire augmenter le nombre de machines virtuelles hébergées par serveur physique), et la libération de mémoire utilisée par la machine virtuelle– puisque la mémoire est la ressource rare dans les centres de données et les nuages d'entreprise (private cloud).

SENAtoR est une solution éconergétique pour les réseaux hybrides de backbone développés conjointement avec l'équipe COATI. Comme pour les algorithmes de routage compatibles avec l'énergie, SENAtoR éteint/allume les périphériques SDN en fonction du trafic. SENAtoR à permet en plus de préserver les performances du réseau et d'éviter les pertes de paquets lorsque des pannes soudaines ou des pics de trafic se produisent (Figure **??**). Tout d'abord, pour éviter les pertes de paquets lors de l'arrêt des périphériques réseau, SENAtoR demande au contrôleur d'arrêter d'envoyer des paquets OSPF hello du commutateur SDN à ses périphériques OSPF voisins, une fois que le commutateur SDN doit être éteint ou mis en mode veille. Après, une durée supérieure à la période de détection et de convergence des pannes en OSPF, le contrôleur met le commutateur SDN correspondant en mode veille qui permet d'économiser de l'énergie sans perdre les règles d'acheminement précédemment installées dans la mémoire vive (TCAM en l'occurrence). Deuxièmement, SENAtoR surveille le trafic réseau pour détecter les éventuelles défaillances de la liaison ou les pointes de trafic soudaines lorsque les nœuds SDN sont désactivés. Si des défail-

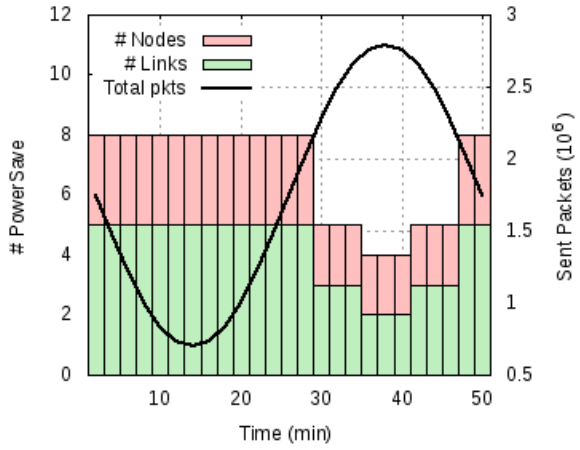


Figure 0.8: Numéro des liens/nœuds éteint en fonction du numéro des paquets envoyé pour `atlanta`.

lances de liaison/nœud sont découvertes ou si des pics de trafic apparaissent, SENAtoR sort de leurs état de veille tous nœuds SDN précédemment désactivés pour empêcher la déconnexion du réseau ou la perte des paquets de données. Troisièmement, SENAtoR utilise des tunnels pour renvoyer le trafic des voisins des nœuds SDN vers la destination correcte pour empêcher les boucles de routage en raison des différences dans la table de routage des nœuds OSPF (en anglais *Open Shortest Path First*) et SDN.

SEaMLESS est une solution créée par l'équipe Signet à laquelle j'ai participé qui résout le problème des VMs inactives dans les datacenters actuels et les nuages d'entreprise. Le problème consiste principalement à libérer toutes les ressources utilisées (par exemple, la mémoire et l'énergie) tout en maintenant la disponibilité du service fourni. Afin de libérer la mémoire utilisée par ces machines virtuelles et de permettre une consolidation optimisée des serveurs, SEaMLESS migre le processus passerelle des VMs inactives vers un fonction de réseau virtuel (VNF) qui permet d'éteindre la machine virtuelle VM tout en maintenant ses services accessibles (Figure 0.9). Lorsque les utilisateurs essaient de se connecter à ce service, la VNF établit la connexion en premier, puis, en cas d'une tentative d'accès aux données du VM, le contrôleur de SEaMLESS redémarre la VM inactive et la VNF fait migrer la session vers la VM pour qu'elle soit traitée. Cela permet ainsi une disponibilité de 100% des services VM dans les data centers et les nuages d'entreprise tout en optimisant l'utilisation de la mémoire et des ressources énergétiques. Nos résultats montrent que la suspension de la VM inactive, même sans la reconsolidation du serveur, permet d'économiser entre 5 % et 10 % d'énergie. De plus, des dizaines de VNF peuvent être déployés dans 1 Go de mémoire vive (appelé en anglais *Random Access Memory "RAM"*) au lieu de 1 ou 2 machines virtuelles selon les pratiques habituelles de dimensionnement dans les serveurs virtualisés.

À la fin de cette thèse, nous analysons les limites de chaque solution et leurs extensions possibles. Nous indiquons principalement la nécessité de tester toutes les solutions fournies dans un véritable scénario de réseau à grande échelle pour tester l'efficacité de chaque solution. De

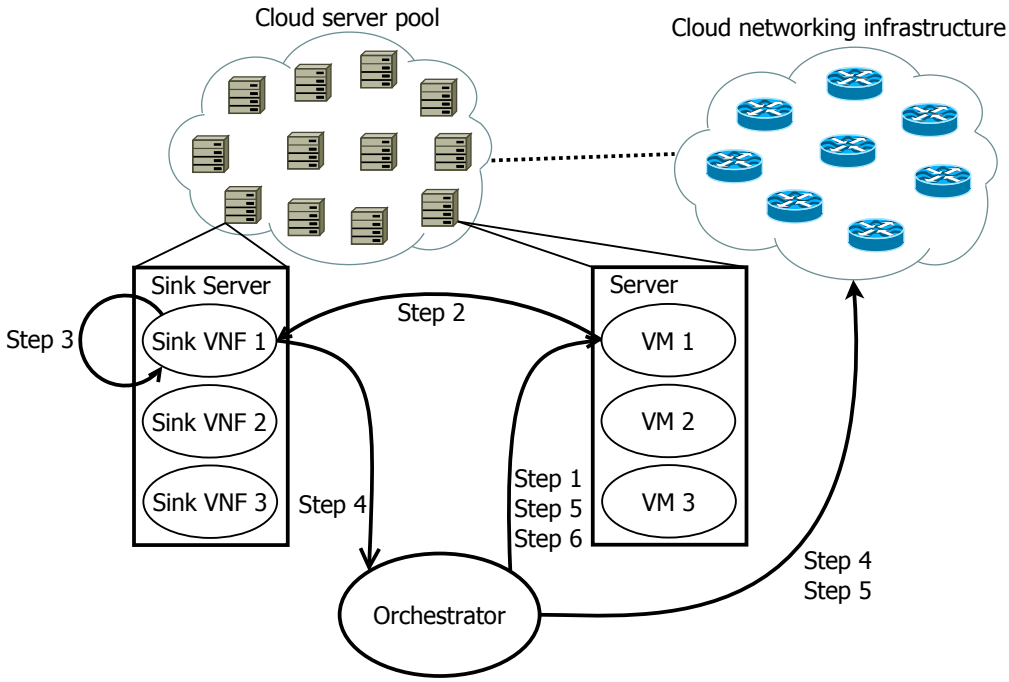


Figure 0.9: Procédure de migration d'une machine virtuelles a un Sink Server

plus, nous conseillons de développer une extension du schéma de routage couche 3 du SDN aux périphériques existants pour permettre la même vue réseau sur SDN et les périphériques existants dans des réseaux hybrides.

Enfin, nos études ont montré que SDN peut être utilisé pour améliorer les performances actuelles des ISP, du centre de données et du cloud et de l'efficacité énergétique. SDN pourrait aller plus loin avec l'analyse de réseau temps réel et la modification dynamique à l'aide de techniques d'intelligence artificielle.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

Networks such as Data Centers (DC) networks and Internet Service Providers (ISP) networks are usually composed of multiple heterogeneous network devices such as switches, routers, firewalls etc. In DC, the network devices are generally located in the same geographic area and host multiple services. On the other hand, the network devices in ISP networks, also called backbone networks, are geographically dispersed around a country or a continent and host huge amount of traffic. In legacy networks, these network devices should be configured individually by network administrators using Command Line Interface (CLI) to provide the required configuration to apply the defined policy which is very **hard to manage**, as network policies are getting **more complex and are sometimes restrained** by the network device provider [BAM09]. In addition to configuration complexity, the increase in traffic variability and network dynamics pushed towards the need for **programmable networks** that will allow network administrators to program network devices to allow them to take actions on certain events [NMN+14]. **Active networks (AN)** were proposed as a first attempt to program the network. AN allow network devices to do complex computations based on the received packet's contents, and then based on the computation result one might change the packet's contents [TSS+97]. The Software Defined Networking (SDN) inherits from the AN, where SDN networks also perform complex computation on the received packet's content. However, the computation is done on a centralized entity called the controller and the context has changed where, nowadays, flexibility is not an option anymore as exemplified by the wide industrial support (HP, AWS, etc.) behind SDN.

SDN is a new highly flexible promising network architecture, which overcomes the limits of forwarding mechanisms of legacy IP networks. Indeed, SDN-based networks (or SDN networks, as we will call it in the remaining of this document) might not only use the IP destination address to make forwarding decisions, like legacy IP networks do, but can also use several other information from the MAC, Network and Transport header of packets [The12]. SDN technologies make also a clear separation between the control plane and the data plane where the two planes are physically separated where the forwarding devices only have the data plane and the control plane is moved to a centralized entity, the controller. In SDN, the forwarding devices (or switches) are considered as dummy devices that only follow the forwarding policies dictated by an external programmable entity: the controller, which implements the control plane (Figure 1.1). Thus, there is no more need to have multiple black-box network devices with specific network functions such as switches, routers, firewalls, etc. Moreover, there is no need anymore to run a CLI to configure network devices, and repeat such operations for every concerned device. In SDN networks, any needed change in the forwarding policies is coded only once in the controller, and the latter will propagate the forwarding rules to all the SDN equipments [HPO13].

1.1 Software Defined Networking

As explained before, an SDN network is composed of four basic components: (i) the controller that implements the control plane, (ii) the forwarding devices/switches which feature the data plane, (ii) the southbound interface, e.g. OpenFlow [The12], which is the communication channel between the controller and the forwarding devices and (iv) the northbound interface which allows the controller to communicate with the user-defined networking applications. In this section, we will first describe in general how SDN networks deal with user's data traffic, and then we will explain in brief SDN network's main components.

1.1.1 Mode of Action

Pure SDN networks are composed of SDN switches that are connected to the controller (Figure 1.1). When a packet arrives at an SDN switch, see Figure 1.1 step 1, the switch will first check its list of pre-installed forwarding rules (e.g. Rule: incoming packets from port 1 source A to destination B should be sent at port port 2). If the packet header information do not match any previously installed rule, i.e. *packet miss*, the switch will then forward the packet to the controller using the default SDN rule (step 2). Upon reception of the data packet on the controller, the network applications in the controller will analyze the packet headers and decide the list of *action*s to be taken when matching packets arrive at the SDN switches (step 3) e.g. modify the packet header information, forward to port B, flood, drop, etc. Afterwards, the controller will transmit this packet back to the switch and implement the actions directly to it (step 4). In addition to that, the controller will transmit a *flow_mod* event to the switch which contains the list of forwarding rules that need to be installed in the switch (step 5) so that next upcoming matching packets can

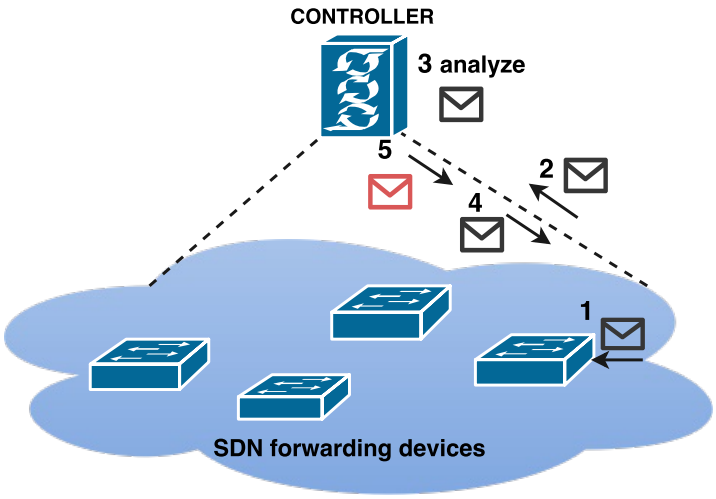


Figure 1.1: SDN data packet treatment process.

be treated directly on the the switch without the need to forward them to the controller.

1.1.2 Main Components

1.1.2.1 SDN Forwarding devices

As explained earlier, SDN forwarding devices also called SDN switches do not have any built intelligence that allows them to analyze the packets. When SDN devices are integrated in an SDN network, they first notify the controller of their existence, their basic configuration and state of their components (ports, directly connected links, tunnels etc). These switches then rely on the controller to give them a set of rules to know how to treat incoming packets. These rules, also called forwarding rules, are then saved in the switch physical memory. SDN switches mainly use the TCAM to store the flow. TCAM allows rapid matching of packets as it is able to search all of its entries in parallel [ZHLL06]. However, TCAM memory is both expensive and power hungry [COS], hence most physical switches provide small TCAM memory supporting around a couple of thousands to no more than 25 thousands flows [SCF+12a]. When the TCAM is full, the SDN rules are then placed in the software memory. However, installing rules in software (that is classical RAM) degrades the performance as packet matching will then require to use the Central Processing Unit (CPU) of the switch which will increase the delay, a.k.a the slow path.

Two types of SDN switches exist: (i) hardware switches such as Pica8 [PIC], HP 5412zl [hp5] etc and (ii) software switches such as OpenvSwitch (OvS) [PPK+15]. Only hardware switches install the forwarding rules (flows) in the TCAM. Software switches such as OvS can however benefit from the memory cache to boost performance. The functionality of the forwarding device, depends on the forwarding rules installed, it can act for example as standard switch, router, firewall, load balancer all together.

1.1.2.2 Southbound Interface

The southbound interface allows the exchange of control messages between the controller and the SDN forwarding devices. This interface dictates the format of the control messages exchanged between the controller and the forwarding devices in the network protocol. Multiple southbound interfaces exist such as OpenFlow [The12], ForCES [DDW+10], and POF [Son13]. ForCES southbound interface requires the presence of the ForCES architectural framework in the legacy networking node, e.g. router. The ForCES architectural framework logically separates the control elements (i.e. operating system) from the forwarding elements (i.e. ASIC) inside the legacy node. In this architecture, the ForCES protocol allows to define logical functions on the control elements. These functions are then sent to the forwarding elements without the need to: (i) insert a centralized controller entity or (ii) change the legacy network architecture. Thus, ForCES requires the control plane to be managed by a third-party firmware and lacks centralization of the control plane. On the other hand, POF southbound interface allows– similarly to OpenFlow– the total physical separation of the control and the data plane which leads to a total change in the legacy network architecture to enable the use of controllers and forwarding devices. However, unlike OpenFlow, POF does not dissect the packet header on the switch to match incoming packets, it rather uses a generic flow instruction set (FIS) generic key that the switch uses to perform packet matching on the forwarding devices.

In our work, we used the OpenFlow SDN architecture which uses the OpenFlow protocol as the southbound interface as it is the most deployed SDN protocol southbound interface [MOS]. Multiple OpenFlow protocol versions exist especially (*v1.0, 1.3, 1.5*). During this thesis, we used the most stable releases of OpenFlow at the time (*v1.0 and v1.3*). In OpenFlow, an SDN forwarding rule- also called a flow entry- is composed of three parts:

1. Match fields: packet header values to match the incoming packets in addition to the ingress port- to match any value of a specific field a wildcard is used.

2. Actions: set of instructions to apply to the matching packet such as forward to port B, flood, drop, send to controller or modify packet headers.

3. Counters: used to collect statistics of packet and byte count match for each flow in addition to the idle and hard timers information.

All OpenFlow protocol versions use the same structure of SDN rules, with some action and matching field additions in each version. The basic flow rule in *v1.0* matched 12 packet header fields. This increased to 15 fields in *v1.3*. The usage of multiple field matching instead of destination based matching in the switches allows thus to unite multiple legacy network device functionalities in a single rule. However, the forwarding rule complexity comes at the price of increase in memory space used per rule.

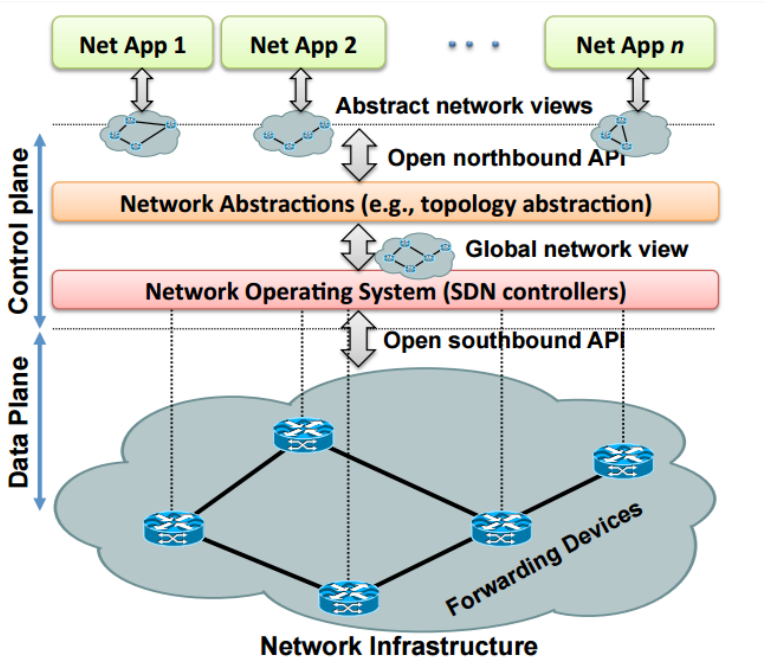


Figure 1.2: SDN network structure. [KREV+15]

1.1.2.3 Controller Server

In SDN, the controller server is responsible for managing the control plane of all the network. The controller server is composed of (Figure 1.2):

- Network Operating System (NOS)
- Northbound interface
- Network applications

The SDN controller runs on the NOS, where the NOS is the main software platform[1] that runs on any-purpose server and allows the access to the server resources, such as the network interface, and basic services such as input/output services. The NOS allows the applications on the controller server to communicate with the SDN network devices, and creates the global topology view of the network. The NOS is also able to monitor the state (e.g. connected or disconnected) of all the network forwarding elements regularly. The NOS, then, informs the network applications of the network states using the northbound interface such as the REST API. Then, the network applications manage and implement policies in the network devices using the northbound interface.

Based on the network configuration requirements and specific needs, the administrator can program new network applications (new network functionalities) in standard programming languages such as Java, C++ or Python. This gives the administrator full control over the network topology and allows the infrastructure to be reactive to network and traffic dynamics.

[1] the software platform is usually called the controller

Two types of SDN controllers exist:

- Centralized controller
- Distributed controller

The centralized controller provides a centralized global view of the network which allows online reactivity to spurious changes in network states and simplifies the development of sophisticated functions. These controllers are usually designed to handle small to medium-sized datacenters and enterprise networks flow throughput. However, the centralized controllers fail to handle large-networks throughput. Moreover, the centralization of the control plane in a single node reduces network resilience as the centralized controller represents a single point of failure in the network. Multiple centralized SDN controllers exist such as POX [POX15], NOX [Int12], Ryu [Ryu17], Beacon [Eri13] and Floodlight [Iza15]. In 2013, more than 30 different OpenFlow controller existed that were created by different vendors or research groups [SZZ+13]. These controllers use different programming languages and different runtime multi-threading techniques. The POX controller is mainly used for prototyping [SZZ+13]. The NOX controller is not supported anymore. As for the remaining most known controllers (*e.g.* Ryu, Beacon and Floodlight), the study in [SZZ+13] shows that Beacon has the best performance, i.e. it features the highest throughput and lowest latency. Thus, in this thesis, we used the Beacon controller. However, when the Beacon controller was not maintained anymore we used Floodlight v2.0 which is a fork of Beacon.

A distributed controller can be either a physically distributed set of controllers, or a centralized cluster of controller nodes. A distributed controller provides fault tolerance, but requires an additional overhead to maintain the network state consistent across all the controllers. Hence, when the network state changes there will always be an inconsistency period of time. Multiple distributed controllers exist, e.g. OpenDaylight [Opeb] and ONOS [Ono]. Both OpenDaylight and ONOS provide similar functionalities with similarities in performance. However, while ONOS focuses more on meeting the needs of service providers, OpenDayLight focuses on providing all of the detailed network functions that one needs to be able to integrate any functionality required. OpenDaylight is thus said to be the "Linux of networking " [Lin15].

1.1.3 Migration from Legacy to SDN Networks

Since migrating legacy networks to SDN networks comes at a high cost and management complexity, different scenarios may be envisioned [VVB14]. The most realistic one is using progressive migration, where the administrators replace legacy devices by SDN devices incrementally i.e. enabling hybrid networks. In hybrid networks, SDN and legacy devices coexist and interact with each other. Nowadays, SDN hybrid devices exist, such as HP5412zl, where a switch device can be configured to have some ports in SDN mode and others in legacy mode. Moreover, SDN hybrid devices can act either as legacy devices by communicating using legacy routing protocols or as pure SDN devices. Hence, the migration from legacy to pure SDN network is cost efficient.

Multiple big companies have started integrating SDN in their networks, e.g. Google has deployed software defined network to interconnect its datacenters [JKM+13]. Based on [JKM+13] the integration of SDN in their production network helped improve operational efficiency and reduce network cost.

Several types of hybrid topologies exist such as topology-based, service-based, class-based and integrated hybrid SDN topologies [KREV+15]. In topology-based hybrid SDN networks, the network is divided into different zones, and in each zone all the nodes are either legacy nodes or SDN nodes. In service-based hybrid SDN networks, some network services are provided by legacy devices ($e.g.$ forwarding) while other services (e.g. traffic shaping) is provided by SDN. In class-based hybrid SDN nodes, the traffic is separated into classes, where depending on the class of the traffic and the network administrator configuration, the traffic would be managed either by SDN or by legacy protocols. As for the last type, i.e. integrated hybrid SDN networks, SDN is responsible to provide all the required networking services and then it uses the legacy protocol as an interface to the forwarding base ($e.g.$ OSPF). Several controllers such as OpenDaylight [Opeb] and OpenContrail [Opea] integrate non-SDN technologies such as BGP, SNMP and NETCONF[2] [EBBS11] to enable hybrid networks.

In this thesis, in addition to using pure SDN networks, we also used a mix of topology based and integrated based hybrid SDN networks. We constructed hybrid SDN networks that: (i) contain a mixture of SDN and legacy nodes (topology-based) and (ii) use the legacy routing protocols to communicate with the legacy devices and with the forwarding information base (integrated hybrid SDN).

1.2 Contributions

The creation of SDN which allows the centralization of the control plane, global network view and network programmability, gives the opportunity to create new services that allow to enhance legacy network performance and bypass their limitations and restrictions. However, Software Defined Networking is a new evolving network architecture that is still in its infancy. To enable its maturity and full SDN networks deployment, multiple research topics including switch and controllers design, scalability, and resiliency are still under study [KREV+15].

At the beginning of this thesis, we tackled one of the basic problems that SDN devices have, that prevents SDN network's scalability while maintaining the same level of network performance, which is namely their TCAM limited size [RHC+15, RHC+17]. Then, we aimed at using SDN in order to enhance current network flow performance [RLPUK15, MR17] and decrease the network's energy consumption [HRG+17, DLP17]. In collaboration with the COATI team [COA] at INRIA and with my colleagues in SigNet team [SIG], we developed the following solutions:

- In collaboration with the COATI team, we addressed the problem of limited hardware mem-

[2]NETCONF is a protocol that allows to manage and configure network devices remotely.

ory (TCAM) used by the hardware SDN forwarding devices (Chapter 3). Their small size limits the number of SDN flow rules that can be installed in the hardware memory as SDN rules are complex and long. Thus, when the hardware memory is full, new rules will be installed in software which highly degrades the flow performance as we will see in Section 3.2. To allow network scalability while optimizing the usage of the TCAM memory, **we created a solution called MINNIE** that dynamically compresses the routing rules in an SDN node to maximize the number of flows that can use the TCAM memory.

- We leveraged the benefits of the centralized control plane at the SDN controller, and its capability to have a general view of the topology to **enhance the flow performance** in the network. We proposed two solutions:
  - **A prototype that provides dynamic scheduling** in the datacenter based on the flow and port statistics feedback at the controller. This approach is innovative as it tries to extend SDN to alter the data plane of the switch (here the scheduling policy). Unfortunately, this solution was effective only on small-size datacenters.
  - **A solution named PRoPHYS** that allows to provision network link failure or link disruption in hybrid ISP networks. This solution monitors the flow paths in the network, and leverages the received information to estimate whether a network failure has occurred. Then, after detecting a possible failure, PRoPHYS reroutes the traffic that uses the assumed down network segment to minimize the amount of traffic that could be lost in case of network failures.
- Then we focused on the energy problem in datacenters and SDN hybrid ISP networks.
  - To enhance energy efficiency in ISP networks **we created a solution called SENAtoR** (in collaboration with the COATI team) that inserts SDN nodes in legacy networks to avoid packet loss when network devices are turned off to save energy.
  - To tackle the energy problem in the datacenter in the SigNet team, we started working on the SEaMLESS project which helps decrease the energy consumption of enterprise cloud networks. SEaMLESS migrates an idle VM service (e.g. idle Web server) to a lightweight virtual network function (VNF) service. This enables to turn off idle virtual machines (VMs) while maintaining the connectivity to the network services provided by the virtual machine (VM).

1.3 Roadmap

In this chapter, we listed the added benefits of SDN over legacy networks, and defined the main concept and components of SDN networks and their mode of actions. We then introduced the concept of hybrid networks and their importance in the migration from legacy to SDN networks. Finally, we summarized in brief the main problems that we tackled during this thesis and our main

contributions. We provide below a brief outline of the rest of the manuscript.

Chapter 2 introduces the main challenging research topics of SDN networks. It then states in general the major contributions that the research community have already provided to enhance SDN networks.

Chapter 3 introduces our solution called MINNIE that was developed in collaboration with the COATI team. In this chapter, we first define the problem that MINNIE solves and then we explain the theoretical basis of this solution. Afterwards, we provide experimental testing results to validate the efficiency of this solution and its impact on the network traffic and network devices functionality. Then, we provide our numerical evaluation results that prove the scalability of this solution. At last, we sum up our findings along with a discussion of the possible extensions of this work, its adaptability to network traffic, and its impact on multiple domain such as security.

In chapter 4, we introduce our SDN solutions to manipulate the data plane of SDN switches to improve the resilience and performance of SDN networks. Both solutions leverage the centrality of the control plane at the controller, capability to maintain a general network topology view at the controller and capability to prompt the switches for statistics. In the first section, we describe our coarse grained scheduling solution for datacenter networks that aims at decreasing the flow completion time of small flows. We start by explaining the basic idea of our solution, then we provide some basic experimental results that show the efficiency and the limitation of our solution. Then, in the second part of this chapter, we extend our coarse grained scheduling solution and we develop a new solution called PRoPHYS for hybrid ISP networks that aims at enhancing flow resilience against unexpected failures in a hybrid network. We first describe the technique and the algorithm used to detect failures across non-SDN network segments in a hybrid network. Then, we provide our results that show the efficiency, performance and scalability of our solution. At the end of this chapter, we conclude with a summary of the results as well as the limitations and possible extensions of the solutions presented in this chapter.

In the last technical chapter of this thesis, Chapter 5, we tackle the energy efficiency problem in data center and ISP networks. We introduce our solution called SENAtoR that was developed in collaboration with the COATI team in INRIA and describe in brief the recently started project in SigNet team called SEaMLESS. For the case of SENAtoR, we describe the heuristic that computes the set of devices to turn off (nodes or links), and the basic technologies that this solution uses in order to keep near zero packet loss rate even when sudden traffic peaks occur. Then, we provide some testing results that prove the efficiency and practicality of our energy efficient solution SENAtoR which allows almost zero packet loss. Afterwards, in the second part of this chapter, we describe SEaMLESS where the SigNet team proposes to save energy by turning off idle VMs by keeping their service connectivity up and running by migrating the front end of the service from its hosting VM to a VNF. We describe first how does the migration mechanism works when migrating from VM to VNF and vice versa while preserving network state and connectivity. Then, we provide some basic testing results that show the efficiency of SEaMLESS and its impact on user's traffic and energy consumption.

We conclude this document in Chapter 6, where we sum up all of our contributions and describe in details the future work that can take place to enhance our work and extend its implementation use cases.

1.4 List of Publications

- **Journal**
  - Rifai, M., Huin, N., Caillouet, C., Giroire, F., Moulierac, J., Pacheco, D. L., & Urvoy-Keller, G. (2017). Minnie: An SDN world with few compressed forwarding rules. Computer Networks, 121, 185-207.
- **International Conferences**
  - N.Huin, M.Rifai, F.Giroire, D.Lopez Pacheco, G.Urvoy-Keller, J.Moulierac , "Bringing Energy Aware Routing closer to Reality with SDN Hybrid Networks", IEEE Globecom 2017.
  - M.Rifai, N.Huin, C.Caillouet, F.Giroire, D.Lopez, J.Moulierac ,G.Urvoy-Keller "Too many SDN rules? Compress them with Minnie", IEEE Globecom 2015.
- **National Conferences**
  - Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frédéric Giroire, Joanna Moulierac, et al.. MINNIE : enfin un monde SDN sans (trop de) règles. ALGOTEL 2016 - 18èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications, May 2016, Bayonne, France.
- **Poster**
  - D. Lopez Pacheco, Q. Jacquemart, M. Rifai, A. Segalini, M. Dione, G. Urvoy-Keller "SEaMLESS: a lightweight SErvice Migration cLoud architecture for Energy Saving capabilitieS", ACM SoCC 2017.
  - Myriana Rifai, Dino Lopez, Guillaume Urvoy-Keller,"Coarse-grained Scheduling with Software-Defined Networking Switches", ACM Sigcomm 2015.
- **Research Report**
  - Myriana Rifai, Dino Lopez, Quentin Jacquemart, Guillaume Urvoy-Keller "PRoPHYS: Providing Resilient Path in Hybrid Software Defined Networks".

Chapter 2

State of the Art

Contents

During the last years, Software Defined Networking has been developing rapidly and a lot of researchers have been working on migrating legacy networks to full SDN to benefit from network programmability, control plane centrality, and global network view that SDN technology features (Figure 2.1). However, the full migration to full SDN is blocked by the centrality of the controller which features a single point of failure, lack of standardization of SDN protocols and switch design, and the limited scalability and performance of SDN controllers and hardware switches [KREV+15]. However, even-though SDN is still under development, multiple research domains benefited already from its development where SDN is being deployed in current networks such as Facebook [dep15] and Google B4 network [JKM+13]. Legacy networks, traffic engineering, energy efficiency, network virtualization and network management domains have all leveraged the centrality of the control plane, programmability and the global network view of the whole topology at the SDN controller.

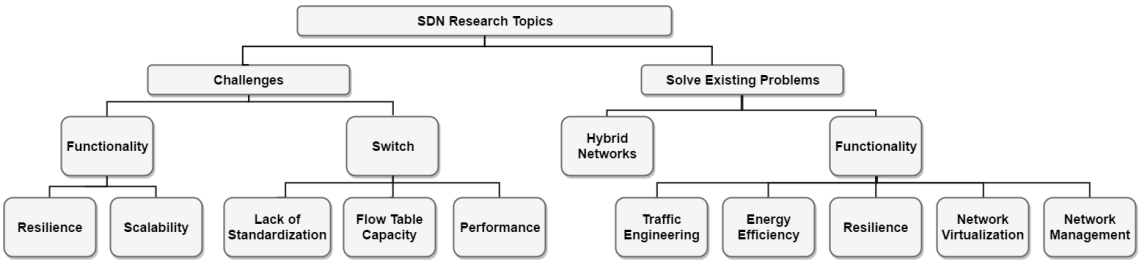


Figure 2.1: SDN main research topics discussed in this thesis.

In the first section of this chapter, we present a non-exhaustive list of the main solutions that address the main challenges of SDN. Then, in the second section, we provide a list of solutions that solve existing problems or limitations in current SDN or legacy networks and enhance the overall network performance and programmability.

2.1 Attempts to Overcome SDN Challenges

The basic idea behind Software Defined Networking is the physical separation of the control plane from the data plane and the centralization of the control plane at the controller. Though the centralization of the control plane at the controller provides a global view of the network, and provides administrators with the capability to dynamically control and program the network, it is accompanied by multiple challenges. First, the separation of the control plane from the data plane would require a communication channel between both layers to transmit control messages. This communication channel becomes the bottleneck when a large number of control messages are transmitted between the controller and the SDN switches. Second, the fact that the control plane is totally separated from the data plane decreases network resilience and fault tolerance. In SDN, the network undergoes the risk of loosing data when a network failure prevents the communication between the data plane and the control plane. Moreover, the centralization of the control plane might constitute a single point of failure.

In addition to the two main challenges stated above, recent studies proved that SDN switches (data plane) have pointed out other potential weaknesses of SDN. Due to the lack of standardization of SDN switch design and southbound interface, multiple SDN switches exist each featuring its own set of actions and communication protocol. Some switches use OpenFlow protocol (v1.0 or v1.3 till v1.5 etc), while others use POF. Hence, a new mechanism should be elaborated to either enable the communication between all the switches or standardize the switch design and southbound interface. Moreover, as we will see in this thesis, SDN devices have limited flow table capacity and limited network performance which prevents network scalability. In this section, we detail the challenges stated above and we discuss some works that were designed to address them.

2.1.1 Control Plane Scalability

As stated before, the separation of the control plane from the data plane hinders the scalability of the network as the communication channel between the controller and the data plane could become the bottleneck. The centralized controller could also be the bottleneck as all the SDN switches in the network will ask the centralized controller for decisions to analyze and route the traffic in the network, increasing the control traffic, the processing load on the controller and the flow installation and processing delays [TGG+12]. The data plane (SDN switch) could also become the bottleneck for the same reasons stated above. The lack of scalability in any of these three components (SDN switch, controller and the channel between them) will hinder the scalability of the whole network increasing the flow delay and possibly causing flow loss degrading the Quality of Service (QoS).

To enhance control plane scalability researchers attempted to enhance the performance of the controllers by: (i) building distributed controllers [Ono, Opeb, KCG+10, DHM+14, KCGJ14], (ii) dynamically deploying controllers based on network traffic demand (elastic control plane) [BRC+13, TG10] and (iii) enhancing the performance of a single controller [Eri13, NCC10, TGG+12] by using the new developments in parallelism and multi-threading. Distributed controllers such as ONOS and OpenDaylight [Ono, Opeb] distribute the control plane on a group of controllers which allows the controllers to manage bursts of network traffic. ElastiCon [DHM+14] is also a distributed controller with an additional elasticity feature that supports the addition and deletion of controllers based on traffic demands. However, the drawback of distributed controllers is their need to exchange information among each other to preserve a consistent view of the network topology which increases the control traffic, could increase the processing delay and can result in network inconsistencies. On the other hand, a centralized controller manages the whole network topology and will thus always have a global centralized view of the network. Thus in [BRC+13], the authors suggested to deploy centralized controllers dynamically in the network based on traffic demand to preserve scalability. Hyperflow [TG10], provides scalability while maintaining a centralized control plane. It features a logically centralized but physically distributed controller which allows to centralize decision procedures but distributes SDN switch events among the control nodes. Nonetheless, the centralization of the control plane could lead to a single point of failure.

In addition to enhancing the performance and scalability of the controller, some researchers enhanced the scalability of the control plane channel (channel between the controller and the SDN switches) [CMT+11, YRFW10, HSM12]. In [CMT+11], the authors propose DevoFlow which modifies the OpenFlow model by allowing only significant flows to contact the controller and uses wildcards aggressively in the flow table to decrease the number of control messages transfered between the controller and the switch. Difane [YRFW10], similarly to DevoFlow, tends to decrease the number of flows– that require the controller intervention– to be installed by using intermediate switches in which the controller stores some necessary rules to be used by the end switches. Though these solutions can decrease the load on the control channel, they can not be

deployed in networks where complex policies are implemented or flow rules are created due to network dynamics. In addition to decreasing the number of events arriving at the controller, some researchers studied the placement of the controllers in the network to decrease the traffic load on the links connecting the switches to the controllers [HSM12, JCPG14] to avoid traffic loss. Unfortunately, so far, these placement solutions can not decrease the control load on the links, maintain the minimum delay between the switch and the controller, and be resilient to link failures all together.

Finally, to enhance the scalability of SDN switches, some researchers aimed at decreasing the processing delay on SDN switches by migrating the counters from the hardware memory Application-Specific Integrated Circuit (ASIC) to the software memory such as Software-Defined Counters (SDC) [MC12], in addition to enhancing the SDN switch performance (see Section 2.1.5) and increasing the switch flow table size (see Section 2.1.4).

2.1.2 Resilience

The centralization of the control plane at the controller degrades the resilience of SDN networks. In such networks, the controller becomes a single point of failure. The loss of connectivity between the controller and the SDN devices would lead to the loss of network control, and thus data traffic could be lost. Multiple researchers tried to enhance the resiliency of the control plane ($i.e.$ the controller) [BBRF14] by trying to build distributed controllers such that if one of the controllers fails the remaining controllers can take over its nodes [Ono, Opeb]. But, as stated before these controllers need to constantly exchange network topology information and may result in inconsistent views of the network. In addition to building a fault tolerant controller, some researchers aimed at studying the placement of the controller taking into consideration the fault tolerance ratio and probability of link failure of the links connecting the data plane to the control plane [GB13, HHG$^+$13, HWG$^+$13, LGZ$^+$15]. However, as stated before, to our knowledge there is no placement algorithm that can provide scalability, resilience and performance altogether.

Others studied additional architectures or methodologies that enable the switch itself to overcome failures without disrupting the controller state machines [KZFR15, BBRF14]. In [KZFR15, BBRF14], the authors introduce a platform where the switches and the controller maintain a state machine (stored in a shared database or independently) to know whether the data traffic has been treated correctly. In case of failure, the switch can manage the data traffic using the state machine database information and then informs the controller once the connection is re-established. Notwithstanding, storing the state machine in a shared database would increase the controller analysis delay, which degrades the controller's performance.

Moreover, to maintain network resilience, the authors of [WWW$^+$14], suggested a simplistic solution that pre-installs backup routes in the switches. These backup routes are then used quickly as "fast-fail-over" actions where the switch monitors its port and upon the failure of one port, the switch uses the set of backup rules related to the port failure. However, these solutions require the

installation of additional forwarding rules on the switch decreasing the available memory space for main forwarding rules.

2.1.3 Multiple Switch Designs Interactivity

After the development of SDN architecture, many researchers and industrial companies (e.g. HP, Juniper, Brocode) developed SDN switches and southbound interfaces. However, the lack of standardization, lead to the creation of diverse SDN switches, each one having a unique flow table structure, set of actions to be applied to incoming traffic, and different versions of supported southbound interfaces, such as POF and OpenFlow v1.0, v1.3, etc. The diversity of SDN switches and southbound interface design increased the complexity of managing the communication between the controller and the SDN switch. To accommodate the heterogeneity of the data plane devices and the heterogeneity of the southbound interface multiple solutions such as NOSIX, tinyNBI, and libfluid [YWR14, CSS14, VRV14] arose.

Both libfluid [VRV14] and tinyNBI [CSS14] attempted to create an API capable of supporting the heterogeneity of OpenFlow. TinyNBI API supports the heterogeneity of all OpenFlow versions between 1.0 and 1.4 while libfluid API supports only versions 1.0 and 1.3. NOSIX [YWR14] on the other hand, was developed as an API that supports the heterogeneity of the SDN switch flow table features. NOSIX creates a single format of virtual flow table on all types of SDN switches, then it translates this virtual table to the actual switch forwarding table. Though NOSIX, tinyNBI, and libfluid were provided to allow the existence of multiple types of SDN protocols and SDN switch designs in the network, they still do not cover the full diversity of SDN devices and SDN protocols and they add processing delay to decrypt the messages and encode them in the correct protocol version or forwarding table design.

Finally in [HYG+15], the authors propose a new all programmable SDN dataplane switch, called ONetSwitch, that can cope with all SDN platforms while maintaining a high performance, flexibility, small size and low power consumption. ONetSwitch can be both reprogrammed and hardware restructured to cope with the different physical requirements and inner functionalities required so that it can cope with all current SDN protocols. Nonetheless, this switch is still a prototype and have not been fully implemented or tested in real networks.

2.1.4 Flow Table Capacity

Current SDN hardware switches use TCAM hardware memory to store their forwarding tables. Though this memory is both expensive, small, and power hungry [COS], it is used in current SDN network devices due to its very high lookup speed (around 133MHz [KB13b], $i.e.$ 133 million lookups per second). Thus, a lot of research has been conducted to maximize the number of SDN rules in the TCAM memory. Some researchers, among many others, aimed at decreasing the total number of rules such as [RSV87, ACJ+07, MLT10, MLT12] while others tried to decrease the size

of the forwarding entries $e.g.$ [HCW+15, ADRC14] (a more exhaustive list is detailed in Chapter 3). In [RSV87] the authors of the Espresso heuristic suggest to compress the wildcard rules. [MLT10] aimed at minimizing the number of rules necessary to apply a certain policy. In addition, some researchers aimed at maximizing the number of rules installed in the TCAM by decreasing the number of bits that describe an SDN rule ($i.e.$ compressing the rule) by inserting a small tag in the packet header [KB13a, BK14] or by inserting some data fields in the packets such as a shadow MAC [ADRC14] which uses label switching instead of packet switching. Nevertheless, these solutions require modifying the protocol stack, modifying the packet header which requires to pass by the central CPU a.k.a slow path or adding additional delay (see Chapter 3 for additional details).

To increase the flow table capacity, researchers did not only focus on optimizing the usage of the TCAM memory space, but they also searched for other memories and/or processors that can replace or be deployed in parallel with the TCAM memory to enhance the SDN switch performance. In brief, we give here a short list of technologies that are being studied for SDN: Static Random-Access Memory (SRAM), reduced latency Dynamic Random-Access Memory (DRAM), Graphics Processing Unit (GPU), Field Programmable Gate Array (FPGA), network processors among other specialized network processors [EFSM+11, NEC+08, MVL+13, LCMO09, RJK+12, PMK13]. Out of all of these technologies, the GPU processor appears to be very efficient and could become a direct competitor of the TCAM memory. Based on the study in [MVL+13], GPUs can process data at around 20Gbps speed with flow tables of 1 million match entries.

In addition to the software and hardware solutions, multiple solutions exist at the switch design level [LZH+14, KARW14, YXX+14]. For example, in [LZH+14, KARW14] the authors provide new design solutions that perform parallel lookup for fast response time and use cache-like Openflow switch arrangements to enable the scalability of the flow table. In [YXX+14] CAching in Buckets (CAB), the authors propose to change the switch design. In such case, the switch will use a geometric representation of the rule set and divide it into small logical structures called buckets and install the information in separate buckets. These solutions, however, require to change the whole switch design which is problematic for hardware switches.

2.1.5 Switch Performance

Multiple research studies like [HKGJ+15, RSKK16, BD14, SCF+12b, KPK14] were performed to study the performance of SDN switches. In [HKGJ+15], researchers studied the impact of separating the control plane from the data plane. In [BD14, SCF+12b], the authors show that while the TCAM memory of hardware SDN switches can provide 133MHz processing speed, SDN switches (both software and hardware) can perform a a maximum between 38 and 1000 flow rule modification per second. This low maximum flow modification throughput limits the number of flows that can be modified (installed, deleted or changed) on the switch. Hence, current SDN devices cannot deal with a huge burst of new flows in a second without degrading their

performance. In addition to that, in [HKGJ+15], the authors show that the inter-existence of flow modification events and data traffic increases the delay of both flow modification event and data plane traffic as they are both limited by the bus connecting the CPU and the ASIC hardware memory. Moreover, as explained in [RSKK16], the SDN switch performance decreases from 940 to 14Mbps when the rules are installed in software instead of hardware.

The deployments of commercial hardware OpenFlow-based switches [KSP+14] have pointed out that the embedded CPU is the performance limiting factor. Therefore, researchers suggested to: (i) add multiple powerful CPUs [MC12], or (ii) change/enhance the CPU currently in use [Int11, Dpd14, SWSB13]. Since the rise of the SDN technology, microchip companies have been studying new general-purpose technologies that have flexible SDN capabilities. In [Dpd14], the authors propose a CPU which includes a Data Plane Development Kit (DPDK) which allows high level programming of packet processing procedure in the network interface cards. Their CPU prototype in SDN, have proved so far its capability of providing high performance in SDN switches [PMK13]. In addition to DPDK integrated CPUs, FPGA and especially NETwork Field Programmable Gate Array (NETFPGA) have been proposed to augment switch performance [SWSB13]. Moreover, a recent study [ZJP14] shows that System-On-Chip (SoC) platforms, can be used in OpenFlow devices and are capable of providing up to 88 Gbps throughput for 1000 flow with dynamic updates.

Other researchers, however, suggested the redistribution of the actions between the controller and the SDN switch to decrease SDN switch CPU consumption [CMT+11] $i.e.$ to reconsider the structure of the southbound interface [Cha13, BGK+13a].

However, all of these solutions require to change the physical structure of the hardware switches or the southbound interface, which would not solve the performance problem of existing deployed SDN hardware devices.

2.2 Attempts to Enhance Network Performance using SDN

In the previous section, we investigated the main challenges that SDN devices face and we stated briefly some of the current solutions or prototypes that were proposed to enhance SDN network performance. In this section, we introduce some SDN architectures or methodologies that allow to propagate SDN functionalities from SDN devices to legacy network devices in hybrid networks. We also show how SDN can enhance current network performance by enhancing and providing new solutions for traffic engineering, energy efficiency, resilience and network virtualization.

2.2.1 SDN in hybrid networks

Multiple studies leveraged SDN nodes in legacy networks to enhance the performance of current networks [CXLC15, AKL13]. For example, in [CXLC15] the authors leverage SDN nodes to

provide 100% reachability under any single link failure by using SDN nodes to help them identify congestion and properly choose backup paths using tunneling. In [AKL13], the authors use the information collected by the SDN controller (traffic pattern, load, etc.) to choose the routes in such a way to enhance traffic engineering performance. The authors of [VVC+17] have used the SDN nodes to adapt the traffic when network forwarding anomalies are detected during rule setup to avoid policy violations.

Moreover, to allow full network programmability, some propositions try to propagate SDN functionalities from SDN to legacy nodes in hybrid networks, such as [LCS+14, JLX+15]. The authors of Panopticon [LCS+14], tried to propagate the enhanced functionality of the SDN nodes to the legacy nodes. They managed to decrease the failure detection interval to a minimum of 1s as they depend on the Spanning Tree Protocol (STP) to detect a failure. In Telekinesis [JLX+15], the authors leveraged the SDN controller to force legacy L2 switches to use the same forwarding tree as SDN nodes. In addition to these solutions that leverage SDN in hybrid networks, many are still to be explored to enable the full migration of SDN functionalities to the legacy devices (e.g. enable the controller to control layer 3 (i.e. routing) of the legacy routers).

2.2.2 Traffic Engineering and Energy Efficiency

The separation of the control plane from the data plane, and the localization of the control plane at a centralized entity i.e. the controller, permits the existence of a global network view on the controller. The SDN architecture, allowed the creation of new solutions that dynamically change the routing scheme based on the current load of the links and the global network traffic to provide an enhanced traffic engineering scheme e.g. [AFRR+10, CKY11, BAAZ11, SK14, SSD+14] or to decrease the energy consumption of the networks e.g. [HSM+10a, BBDL15, RRJ+15].

To enhance the Quality of Service, and manage the flows in the SDN networks, the authors of [AFRR+10] propose Hedera. Hedera is a traffic engineering methodology which leverages the controller to detect and estimate the demand of large flows at the edge switches in the network and assign them to paths that can manage their estimated load. Hedera, however, can only enhance the performance when multiple paths exist between the source and the destination as it does not enhance scheduling schemes within a link. Similarly, [CKY11] manages the traffic demand by detecting the large flows. Their solution, Mahout, detects the large flows by monitoring the end hosts socket buffer. On the other hand, MicroTE [BAAZ11] leverages the short term traffic information and the partial predictability of the traffic matrix to adapt traffic scheduling schemes. Nonetheless, Mahout and MicroTE require to insert modifications at the end host.

SDN also allows to envision new traffic engineering mechanisms for home broadband access networks, based on the MAC layer [SK14] or TCP layer, such as FlowQoS [SSD+14], to enhance the performance of certain applications or machines.

In [TAG16], the authors survey multiple energy efficient solutions that leverage SDN capabilities to provide a tradeoff between energy efficiency and QoS [HSM+10a, BBDL15, RRJ+15].

The authors of [HSM+10a] propose to route the traffic with the help of the OpenFlow controller such that they minimize the number of used components to decrease the energy consumption of the network. In [BBDL15], the authors propose a new routing methodology where the controller takes the topology resources, and traffic demands as input. Then, the controller calculates the energy consumption of each network entity based on the flow demand and optimize the choice between network performance and energy efficiency by using control policies defined in the Green Abstract Layer energy efficiency approach [BBD+13]. Similarly in [RRJ+15], the authors present GreenSDN which integrates three different methodologies that work on the chip, node and network level (Adaptive Link Rate [GC06], Synchronized Coalescing [MC11] and Sustainable Aware Network Management System [CAJ+12] respectively) to optimize the utilization of devices and energy efficiency in the network while maintaining an efficient traffic engineering. All the same, these solutions do not take into consideration the possible data loss that occurs when turning off network resources or when sudden traffic spikes occur when network devices are turned off.

2.2.3 Resilience

SDN networks can enhance the network resiliency and decrease the amount of traffic loss by leveraging the programmability and the full view of the network topology. For example, SlickFlow uses the SDN controller to insert the backup route information in the packet header to allow switches to reroute packets quickly through the backup route once they discover a failure across the main path [RMR13]. Similarly, INFLEX [ALCP14] alters the packet IP header Explicit Congestion Notification (ECN) field and insert a routing tag to identify the path used, this routing tag then changes when a failure is detected in the main path. Altering the packet header, however, degrades the performance as the data packets pass through the slow path (CPU).

In addition to modifying the packet header to enhance flow resiliency, some researchers depend on the usage of backup routing rules that are pre-installed in the switches such as [SGC+13, SO14], and then they define new methodologies to fast reroute between the main and the backup rules. In [SGC+13], the authors propose OSP which introduces backup rules in the SDN switches with different priorities, and tries to maintain the main and the backup rules in the switch by sending a renew packet which prevents the backup rules from expiring as long as the main rules exist. OSP also depends on the concept of auto-reject where a switch rejects to use a rule that will output the traffic to a failed link or port. To fast reroute between the main rule and the backup rule in [SO14] when link failures or congestion are detected, the authors monitor the flow expiry counters and then they modify the flow timers to expire immediately once the main path fails or is congested which allows to decrease the switch buffers, packet loss and end-to-end latency. On the other hand, installing backup routes in the SDN switch would decrease the TCAM remaining space which could hinders network flow scalability and performance.

2.2.4 Network Virtualization and Management

The introduction of SDN in datacenter, ISP and cloud networks enabled an improvement in network management where the network administrators can use SDN alongside Network Function Virtualization (NFV) to help virtualize all network services and functions [BBK15, BRL+14, Opea] and optimize the placement of network functions in the network [BRL+14, CRS+13]. Moreover, SDN helped the creation of virtual networks as it allowed the development of multiple solutions to solve the hanging problems of network and/or VM migration [WNS12, RLL12, LLJ15, KGCR12].

SDN allows the programmability of the network and the centralization of the control plane at the controller. This allows SDN to virtualize and program all network functions in the controller [BBK15]. It also allows the creation of any user defined network function which permits the creation of fine-grained network functions [BRL+14]. For instance, HyperFlex [BBK15] is an SDN hypervisor that uses the SDN technology to virtualize data plane functionalities in SDN networks. Moreover, the combination of SDN with NFV allows to provide optimized service chaining e.g. [Opea]. OpenContrail [Opea], for example, combines both SDN and NFV. It virtualizes all network functionalities using an SDN virtual network controller, and provides service chaining. Furthermore, network function virtualization using SDN allows virtual network isolation, dynamic resource control and placement of the middleboxes [CRS+13, JKM+13].

In addition to providing network function virtualization, and service chaining, SDN allows to optimize the placement and migration of network resources, virtual networks, and virtual machines based on user traffic and physical network dynamics. In [RLL12], the authors propose a novel algorithm that is network aware and dynamically computes the placement of VMs based on real time network monitoring. In [LLJ15], the authors leverage SDN technology to optimize VM migration across datacenters by using the controller to select the best path and devices that enable the fastest and most resilient transfer of the migration traffic and they also use it to update the network resources. LIME [KGCR12], uses the SDN controller to perform live migration of a full network (VMs, network and their management system) without disrupting the network resources by cloning the data plane states to a new set of virtualized forwarding devices.

2.3 Conclusion

In this chapter, we identified the main performance and scalability challenges that SDN devices face. We also showed that the numerous proposed solutions require to modify the SDN switch and southbound interface design, or require to change the hardware used by the SDN switches which is very intrusive. Moreover, in this chapter, we have stated various solutions that use the SDN technology to enhance the network performance, resilience and energy efficiency. Energy efficient solutions focus only on optimizing the energy consumption of the networks without taking into consideration the possible degradation in network performance or data loss. As for the proposed

solutions that use SDN to enhance the network performance, they require either to change the packet header which causes the packets to be processed by the central CPU which increases the processing delay, or they require modifications at the end host which is also intrusive.

In the following chapters, we present our solutions that enhance the scalability at the SDN switch level by optimizing the usage of the flow table. We also present our solutions that leverage the switches statistics to enhance the network performance and resilience. We then present our energy efficient solutions that mainly focus on maintaining the network performance while turning off network devices. All our solutions are built as controller application modules, and do not require the modification of the data packet header or the end hosts.

Chapter 3

Flow Scalability: Minnie

Contents

Nowadays, the number of connected devices and connected services to the network, is increasing, hence the number and the variety of flows is increasing [Cis12]. Moreover, the increase in the variety of services provided, leads to the increase in the complexity of the traffic management to be provided. This increase in the number of flows and their variety naturally leads to the increase in the number of forwarding rules to be installed in the SDN forwarding devices when fine grained flow routing is performed. In this chapter, we present a solution called MINNIE that

allows flow scalability in SDN networks, while providing the best switch performance for all the flows installed on the device.

SDN forwarding devices aim at applying flow-based forwarding rules instead of destination-based rules (as in legacy routers) to provide a finer control of the network traffic. For instance, in OpenFlow 1.0[1], forwarding decisions can be made taking into account from zero up to a maximum of 12 packet header fields (IP, TCP, ICMP and UDP etc.), in addition to the incoming port field. When any of these fields should be ignored when forwarding a packet, such a field is set to “don’t care bits". Due to the complexity of SDN forwarding rules, SDN forwarding devices need Ternary Content Addressable Memory (TCAM) to store their routing table as they allow to store “don’t care bits" and can perform parallel lookups (classical CAM can only perform binary operations). However, TCAMs are more power hungry, expensive and physically bigger than binary CAMs available in legacy routers. Consequently, the available TCAM memory in routers is limited. Indeed, a typical switch supports between around a couple of thousands to no more than 25 thousands of 12-tuples forwarding rules, as reported in [SCF+12a].

Undoubtedly, emerging switches will support larger forwarding tables [BGK+13b], but TCAM still introduces a fundamental trade-off between forwarding table size and other concerns like cost and power. The maximum size of routing tables is thus limited, and represents an important concern for the deployment of SDN technologies. This problem has been addressed in previous works, as we will discuss in Section 3.1, using different strategies, such as routing table compression [GMP14, HCW+15], or distribution of forwarding rules [CLENR14].

In this work, we examine a more general framework in which *table compression* using wildcard rules is possible. Compression of SDN rules was previously a work conducted by the COATI team and discussed in [GMP14]. They proposed algorithms to reduce the size of the tables, but only by using a default rule. In collaboration with them, we developed a stronger compression methodology in which any packet header field may be compressed. Considering *multiple field aggregation* is an important improvement as it allows a more efficient compression of routing tables, leaving more space in the TCAM to apply advanced routing policies, like load-balancing and/or to implement quality of service policies. In the following, we focus on compression of rules based on sources and destinations using our solution called MINNIE.

Our testbed results demonstrate that even with a small number of clients, the limit in terms of number of rules is reached if no compression is performed, increasing the delay of new incoming flows. Our experimental and simulation results show that MINNIE allows scalability where it allows to save an average of 80% of the TCAM space without negatively affecting the network traffic (no packet loss nor detectable extra delays if routing lookups are done in the ASIC). Moreover, we show here that both simulations and experimental results suggest that MINNIE can be safely deployed in real networks, providing compression ratios between 70% and 99%.

In this chapter, we explain the existing work that enables to optimize the usage of the flow

[1] https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf

table and their disadvantages (Section 3.1) and we prove the need to install all the rules in the TCAM to provide the best performance (Section 3.2). We describe in detail our MINNIE solution in Section 3.3 and 3.4. And we present, our experimental and simulation results (Section 3.5.5 and 3.6) that proves the efficiency, scalability and adaptability to network dynamics of MINNIE. Then, in Section 3.7 we discuss possible extensions to our solution.

3.1 Related work

To support a vast range of network applications, SDN has been designed to apply flow-based rules, which are more complex than destination-based rules in traditional IP routers. As explained before, the complexity of the forwarding rules are well supported by TCAMs. However, as TCAMs are expensive and power-hungry, the on-chip TCAM size is typically limited.

Many existing studies in the literature have addressed this limited rule space problem. For instance, the authors in [KB13a] and [BK14] try to compact the rules by reducing the number of bits describing a flow within the switch by inserting a small tag in the packet header. This solution is complementary to ours, however, it requires a change in: (i) packet headers and (ii) in the way the SDN tables are populated. Also, adding an identifier to each incoming packet is hard to be done in the ASIC since this is not a standard operation, causing the packets to be processed by the central CPU of the router (a.k.a. the slow-path) strongly penalizing the performance and the traffic rate. Another approach is to compress policies on a single switch. For example, the authors in [ACJ+07, MLT10, MLT12] have proposed algorithms to reduce the number of rules required to realize policies on a single switch.

Several works have proposed solutions to distribute forwarding policies while managing rule-space constraints at each switch [CLENR14, KHK13, KLRW13, NSBT15, KARW16]. In [CLENR14] the authors model the problem by setting constraints on the nodes that limit the maximum size of routing table. The rules are spread between the switches to avoid QoS degradation in case of full forwarding tables. For the case of [KHK13], we note that the replacement of a routing policy, to follow the dynamicity of the network, can be hard since it implies to rebuild the forwarding table in (potentially) several switches. However, no compression mechanisms are added to those solutions. In [NSBT15], the authors propose OFFICER. OFFICER creates a default path for all communications, and later, some deviations are introduced from this path using different policies to reach the destination. According to the authors, the Edge First (EF) strategy, where the deviation is performed to minimize the number of hops between the default path and the target path, offers the best trade-off between the required Quality of Service and forwarding table size. Note however, that applying this algorithm could unnecessarily penalize the QoS of flows when the routers' forwarding tables are rarely full. In [KARW16], the authors propose CacheFlow which introduces a CacheMaster module and a shared section of software switches per TCAM (available in hardware switches only). The CacheMaster constructs the dependency tree of the rules to be installed and then distributes the rules between the TCAM and the software

switches, placing the most popular rules in the hardware switch, thus enabling fast forwarding for the biggest possible amount of traffic. When a packet needs a forwarding rule not available in the TCAM, such a packet is forwarded to the software switches, which send back the packet to the hardware switch in a predetermined input port, to be resent at a specific output port. If the software switches do not have a matching rule, the SDN controller is called. The weaknesses of CacheFlow relies in its inherent architecture, as this solution requires the installation of a software switch for every hardware switch, which might need a reorganization of the network cabling and additional resources to host software switches. Secondly, the optimal number of needed software switches can be difficult to determine, due to the fact that for performance reasons, software switches must only keep forwarding rules (whose number depends on the traffic characteristics) in the kernel memory space, which is limited. Lastly, the two-layer architecture of CacheFlow (i.e. software switches over a hardware switch) increases the delay to contact the controller and install missing rules.

To the best of our knowledge, the closest papers to our work are [HCW+15, BM14, GMP14]. In [HCW+15] the authors introduce XPath which identifies end-to-end paths using path ID and then compresses all the rules and pre-install the necessary rules into the TCAM. We compare our results with the ones of XPath in Section 3.6.2.4. MINNIE uses fewer rules even in the case of an all-to-all traffic as XPath codes the routes for all shortest paths between sources and destinations. This is at the cost of less path redundancy which is useful for load-balancing and fault tolerance. Network operators should consider this trade-off when choosing which method to use. In [BM14] the authors suggest SDN rule compression by following the concept of longest prefix matching with priorities using the Espresso [TNW96] heuristics and show that their algorithm leads to 17% savings only. We succeed in reaching better compression ratios using MINNIE.

MINNIE is an extension of the previous work of the COATI team, [GMP14] which addressed the problem of compressing routing tables using default rule only in case of Energy-Aware Routing.

3.2 Motivation: Software vs. hardware rules

As mentioned before, SDN forwarding devices can install the SDN rules in hardware or software. However, as explained before the TCAM memory is usually small as it is expensive and power hungry. Hence, in current SDN networks, most of the flow rules are installed in software and only the minority of the flows are installed in the hardware memory. Thus, one question naturally rises at this point: What is the real impact of the *slow path* on the switch performance? Do we really need to compress the routing table to enable the installation of all the rules in TCAM?

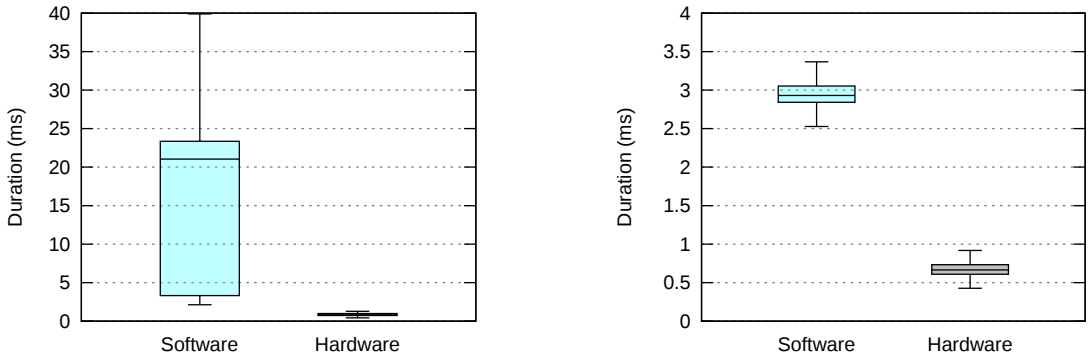
To answer this question, we have devised an experimental testing scenario using our SDN hardware switch (HP 5412zl) switch. Our SDN switch can support up to 65535 rules installed in hardware and software all-together, with only 3000 rules that can be installed in hardware. We have emulated in this switch a full k=4 Fat-Tree topology (see Figure 3.2) with one client per

access switch, and each client sends one flow to every other client in the network. A flow in this testing experiment is composed of a train of 5 ICMP request / reply packets, which is the default behavior of the `ping` command. To understand the difference in the performance of the hardware and software memories, we have devised two testing scenarios :

1. All rules are installed in hardware
2. All rules are forced to be installed in software

With this configuration, we can observe in Figure 3.1a that installing rules in software increases the first packet delay by a factor of 20 from a median of $1\,\mathrm{ms}$ to $20\,\mathrm{ms}$ as compared to hardware rules. Moreover, the average matching delay of the remaining packets (Figure 3.1b) features a 6-fold increase in software as compared to hardware ($3\,\mathrm{ms}$ compared to $0.5\,\mathrm{ms}$).

These results thus **justify the necessity of using only TCAM**. Thus, we developed MINNIE which is a solution that allows to compress the SDN rules to allow all flows to use the hardware memory and thus benefit of the increase in network performance.



(a) First packet's delay

(b) Remaining packet's delay

Figure 3.1: Packet delay boxplot

3.3 Description of MINNIE algorithm

MINNIE aims at reducing the number of routing entries that are installed on every node while respecting the link and node capacities (i.e. number of hardware rules installed). Hence, in collaboration with the COATI team, the MINNIE problem was formally defined as follows:

- *Given a set of flows $\mathcal{D}$, the problem we consider is to find sets of routing rules (aggregated or not) such that each flow is well routed from its source to its destination while respecting the* **link capacity constraints** *and the* **table size constraints.**

To solve this problem we represent the network as a directed graph $G = (V, A)$. A vertex is a router and an arc represents a link between two routers. Each router u has a maximum rule

Flow	Output port
(0, 4)	Port-4
(0, 5)	Port-5
(0, 6)	Port-5
(1, 4)	Port-6
(1, 5)	Port-4
(1, 6)	Port-6
(2, 4)	Port-4
(2, 5)	Port-5
(2, 6)	Port-6

(a) Without Compression

Flow	Output port
$(0, 4)$	Port-4
$(1, 5)$	Port-4
$(2, 4)$	Port-4
$(2, 5)$	Port-5
$(0, *)$	Port-5
$(*, *)$	Port-6

(b) MINNIE: Source table

Flow	Output port
(1, 4)	Port-6
$(1, 5)$	Port-4
$(0, 6)$	Port-5
$(*, 4)$	Port-4
$(*, 5)$	Port-5
$(*, *)$	Port-6

(c) MINNIE: Destination table

Flow	Output port
(0, 5)	Port-5
(0, 6)	Port-5
(1, 4)	Port-6
(1, 6)	Port-6
(2, 5)	Port-5
(2, 6)	Port-6
$(*, *)$	Port-4

(d) MINNIE: Default only

Table 3.1: Examples of routing tables: (a) without compression, (b) compression by the source, (c) compression by the destination, (d) default rule only. Rules' reading order: from top to bottom.

space capacity S_u given by the size of its routing table and expressed in number of rules. Each link $(u, v) \in A$ has a maximum capacity C_{uv}. A flow is identified as a triplet (s, t, d), in which $s \in V$ is the source of the flow, $t \in V$ its destination, and $d \in \mathbb{R}^+$, its load.

Then, we define a routing rule as a triplet (s, t, p) where s is the source of the flow, t its destination and p the outgoing port of the router for this flow. To aggregate the different rules, we use *wildcard rules* that can merge rules by source (i.e., $(s, *, p)$), by destination (i.e., $(*, t, p)$) or both (i.e., $(*, *, p)$, the default rule). Table 3.1 shows an example of a routing table and its compressed versions using different strategies. Table 3.1(a) gives the routing table without compression and Table 3.1(d) the table using default port compression.

To solve the problem, we propose an algorithm called MINNIE. MINNIE is composed of two modules: the compression module which compresses the routing tables using wildcard rules, and the routing module which finds paths (and routing rules) for the flows using a shortest-path algorithm with adaptive metrics to spread flows over the network and to avoid overloading a link or a table.

MINNIE, presented in Algorithm 1, works as follows: For every flow to be routed, MINNIE iteratively finds a path using the routing module described in Algorithm 3 (Algo 1, line 4). For every node in the path, it then adds a forwarding rule if no matching wildcard rule already exists. When any switch routing table reaches its rule space capacity, MINNIE calls the compression module, described in Algorithm 2, on (Algo 1, line 10). We refer to this table compression as a **compression event**. The total load of flows on each link of the path is then updated to account for the new flow (Algo 1, line 12). We now provide more details about the compression and routing modules.

Algorithm 1 MINNIE Algorithm

Input:

a digraph $G = (V, A)$
link capacity $C_{uv}, \forall (u, v) \in A$
rule space capacity $S_u, \forall u \in V$
a set of flows $\mathcal{D}$

Output:

a path (with the corresponding rules) for all flows in $\mathcal{D}$

flow $\mathcal{F}_{uv} = 0, \forall (u, v) \in A$
set of rules $R_u = \emptyset, \forall u \in V$
for each $(s, t, d) \in \mathcal{D}$ **do**
  Find path P_{st} between s and t using Algorithm 3
  for each $(u, v) \in P_{st}$ **do**
    if $(*, t, p_v), (s, *, p_v), (*, *, p_v) \notin R_u$ **then**
      add (s, t, p_v) at the top of R_u
    end if
    if $|R_u| == S_u$ **then**
      Compress R_u using Algorithm 2
    end if
    $\mathcal{F}_{uv} = \mathcal{F}_{uv} + d$
  end for
end for

3.3.1 MINNIE: compression module

Based on the COATI's study [GHM15], the compression of a single table is NP-Hard, hence, we use the following greedy algorithm (Algorithm 2). The theoretical basis of this algorithm was studied by the COATI team in [GHM15] and was proven to be efficient, as it provides a 3-approximation of the compression problem. We show here that it is also efficient in practice.

The algorithm first computes three compressed routing tables: (i) aggregation by source , (ii) by destination and (iii) by the default rule and then chooses the smallest one, as explained in more details below.

Given a routing table such as the one given in Table 3.1(a), the algorithm first considers an aggregation by source (Table 3.1(b)) using $(s, *, p_s^*)$ rules (Algorithm 2 line 1-24). We first consider the sources one by one and choose (one of) the most occurring port(s) in the rules with this source. It corresponds to the port allowing to compress the most rules using a rule of aggregation by source. Then, we use the default rule to reduce the number of aggregated rules.

The main principle is simple, but there is a small technicality to break ties when there are several most occurring ports. As a matter of fact, the choice taken of aggregation ports for each source affects the default port chosen. We thus postpone the choice of the source aggregation port in case of ties in order to choose the default port compressing the largest number of aggregation rules, as explained in details below.

For each source s, we need to find the port p_s^* such that we can aggregate using the rule $(s, *, p_s^*)$, and the port p^* to aggregate with the default rule $(*, *, p^*)$. First, we compute the set of most occurring ports for each source s, noted $\mathcal{P}_s^*$. The default port p^* is thus the most occurring port in all sets $\mathcal{P}_s^*$. If multiple ports can be chosen, one is selected at random. Then, for each source s, the port p_s^* is equal to p^* if $p^* \in \mathcal{P}_s^*$. Otherwise we choose at random among $\mathcal{P}_s^*$. Once the ports for the aggregated rules are chosen, we build the compressed table. First, we add rules that cannot be aggregated (line 16), i.e., $(s, t, p \neq p_s^*)$. Then, we add all the aggregation rules by source that do not use the default port p^* (line 21), i.e., $(s, *, p_s^* \neq p^*)$. Finally, we add the default rule $(*, *, p^*)$ (line 24). The order of insertion in the routing gives the order for the matching, i.e., non aggregated rules, then source aggregation rules and then default rule.

For example, the sets of the most occurring ports of sources 0, 1, and 2 in Table 3.1(a) are {Port-5}, {Port-6}, {Port-4, Port-5, Port-6}, respectively. Since Port-5 and Port-6 appear two times each, we choose at random Port-6 to be the default port. The ports used for the aggregation by source for 0, 1, 2 are then Port-5, Port-6, Port-6, respectively. Port-6 is chosen for the source 2, because it is the default port. We can now build the compressed table by adding all rules that have ports different than their corresponding aggregate rules: $(0, 4, 4)$, $(1, 5, 4)$, $(2, 4, 4)$, $(2, 5, 5)$. Then, we add all aggregate rules with a port different from the default port: $(0, *, 5)$. Finally, we add the default rule $(*, *, 5)$. This gives us the compressed table in Table 3.1(b).

For the second compressed routing table (Table 3.1(c)), we do the same compression consid-

ering the aggregation by destination with $(*, t, p_t^*)$ rules (Algorithm 2 line 25-46). As for the third table (Table 3.1(d)) a single aggregation using the best default port is performed (Algorithm 2 line 47-55), i.e., one of the most occurring port in the routing table becomes the default port (tie broken uniformly at random). We then choose the smallest routing table among the three computed ones.

In the current version of MINNIE, when the algorithm can no longer compress a table, it uses the default action to forward the new traffic to the controller. This could be enhanced to evict the least recently used rule from the table. It should be noted that based on our simulation results (Section 3.6.2) all flows can be forwarded using a rule space capacity of 1000 rules. Thus, using advanced eviction rules seem unnecessary.

3.3.2 MINNIE: routing module

We propose an efficient routing heuristics using a weighted shortest-path algorithm with an adaptive metrics. When several routes are possible for a flow, we select the one using the less loaded equipments, links and routers, as measured by our metrics. The intuition is two-fold: (i) we want to avoid sending new flows to a router with a very loaded routing table, if there exists an alternative path using routers with less loaded routing tables (ii) load balancing the traffic over the multiple possible paths is currently done in data centers to avoid overloading links.

For every flow (s, t, d), we first build a weighted directed graph (digraph) $G_{st} = (V, A_{st}, w)$, where, for every $(u, v) \in A_{st}$, w_{uv} is the weight of link (u, v). G_{st} represents the residual network after having routed the previously routed flows:

- G_{st} is a subgraph of G where an arc (u, v) is removed if its capacity is less than d or if the flow table of the router u is full and does not contain any wildcard rule for (s, t, p_v) (where p_v represents the output port of u towards v). Note that, when a table is full and compressed, a node u has only one outgoing arc (to the node v), corresponding to the first existing rule of the form $(s, *, p_v)$, $(*, t, p_v)$ or to the default rule $(*, *, p_v)$. As more tables get full, the number of nodes with only one outgoing arc increases, reducing the size of the graph.

- The weight w_{uv} of a link depends on the overall flow load on the link and the table's usage of router u. We note w_{uv}^c the weight corresponding to the link capacity and w_{uv}^r the weight corresponding to the rule capacity. They are defined as follows:

$$w_{uv}^c = \frac{\mathcal{F}_{uv}}{C_{uv}}$$

  where C_{uv} is the capacity of the link (u, v) and $\mathcal{F}_{uv}$ the total flow load on (u, v). The more the link is used, the heavier the weight is, which favors the use of lower loaded links allowing load-balancing. And

$$w_{uv}^r = \begin{cases} \frac{|R_u|}{S_u} & \text{if } \nexists \text{ wildcard rule for } (s, t, v) \\ 0 & \text{otherwise} \end{cases}$$

Algorithm 2 Compressing a table

Input:
Set of rules R

Output:
Compressed rules

$Compression\ by\ source$
list of rules C_r {order of insertion = order of matching}
for each $s \in V$ **do**
  $\mathcal{P}_s^*$, set of most occurring ports p in $\{(s,t,p) \mid \forall t \in V\}$
end for
$p^* =$ most occurring port in all $\mathcal{P}_s^*$ {ties are broken at random}
for each $s \in V$ **do**
  if $p^* \in \mathcal{P}_s^*$ **then**
    $p_s^* = p^*$
  else
    p_s^* = most occurring port in $\mathcal{P}_s^*$ {ties are broken at random}
  end if
end for
for each $(s,t,p) \in R$ **do**
  if $p \neq p_s^*$ **then**
    add (s,t,p) to C_r
  end if
end for
for each $s \in V$ **do**
  if $p_s^* \neq p^*$ **then**
    add $(s,*,p_s^*)$ to C_r
  end if
end for
add $(*,*,p)$ to C_r
$Compression\ by\ destination$
list of rules C_c {order of insertion = order of matching}
$\mathcal{P}_t^*$, set of most occurring ports p in $\{(s,t,p) \mid \forall t \in V\}, \forall s \in V$
$p^* =$ most occurring port in $\mathcal{P}_t^*$ {ties are broken at random}
for each $t \in V$ **do**
  if $d \in \mathcal{P}_t^*$ **then**
    $p_t^* = p^*$
  else
    p_t^* = most occurring port in $\mathcal{P}_t^*$ {ties are broke at random}
  end if
end for
for each $(s,t,p) \in R$ **do**
  if $p \neq p_t^*$ **then**
    add (s,t,p) to C_c
  end if
end for

41: **for each** $t \in V$ **do**
42: **if** $p_t^* \neq p^*$ **then**
43: add $(s, *, p_t^*)$ to C_c
44: **end if**
45: **end for**
46: add $(*, *, d)$ to C_c
47: $Default\ port\ compression$
48: list of rules C_d
49: $p^* =$ most occurring port in R {ties are broke at random}
50: **for each** $(s, t, p) \in R$ **do**
51: **if** $p \neq p^*$ **then**
52: add (s, t, p) to C_d
53: **end if**
54: **end for**
55: add $\{(*, *, p^*)\}$ to C_d
56: **return** smallest set of rules between C_r, C_c, and C_d

where R_u is the current set of rules for router u. Recall that S_u is the maximum number of rules which can be installed in the routing table of router u. The weight is proportional to the usage of the table. Note that $w_{uv}^c \in [0, 1]$ and $w_{uv}^r \in [0, 1]$. They measure the percentages of usage of link uv and the routing table of router u.

The weight w_{uv} of a link (u, v) is then given by:

$$w_{uv} = 1 + 0.5\ w_{uv}^c + 0.5\ w_{uv}^r.$$

The additive term 1 is used to provide the shortest path in terms of number of hops when links and routers are not used (i.e., when $w_{uv}^c = 0$ and $w_{uv}^r = 0$ for all $(u, v) \in A_{st}$). This term could be replaced by the delay to traverse link (u, v) to obtain the shortest paths in terms of delay. When the links and routers are used, we take into account their usage. Moreover, we wanted to give the same importance to network link load and table load. Thus, we choose an equal weight of 0.5 for w_{uv}^c and w_{uv}^r. Note that $w_{uv} \leq 2$. This ensures that $l(p) \leq 2 \times l(p^*)$, where p is the path found by the routing module, p^* is the unweighted shortest path and $l(p)$ the number of hops of path p (indeed, $l(p) \leq w(p)$ as $w_{uv} \geq 1$, $w(p) \leq w(p^*)$ as p was selected, and $w(p^*) \leq 2\ l(p*)$, where $w(p)$ is the sum of the weights of the links of path p). This means that no path longer than twice the current available shortest path is selected.

When (G_{st}, w) is built, we compute a route for the flow by finding a shortest path between s and t in the digraph minimizing the weight w.

Algorithm 3 Finding a path for a flow

Input:

A flow (s, t, d)
a digraph $G = (V, A)$
rule space capacity $S_u \; \forall u \in N$
set of rules $R_u \; \forall u \in N$
link capacity $C_{uv} \; \forall a \in A$
flow $\mathcal{F}_{uv}, \forall a \in A$

Output:

A path for (s, t, d)

Create a weighted digraph $G_{st} = (V, A_{st} = \emptyset, W)$
for each $(u, v) \in A$ **do**
  if $C_{uv} - \mathcal{F}_{uv} \geq d$ **then**
    if $\exists$ wildcard rule for (s, t, p_v) **then**
      add edge (u, v) to G_{st}
      $w^r_{uv} = 0$
      $w^c_{uv} = \mathcal{F}_{uv}/C_{uv}$
      $W_{uv} = 1 + 0.5w^r_{uv} + 0.5w^c_{uv}$
    else if $|R_u| < S_u$ **then**
      add edge (u, v) to G_{st}
      $w^r_{uv} = |S_u|/R_u$
      $w^c_{uv} = \mathcal{F}_{uv}/C_{uv}$
      $W_{uv} = 1 + 0.5w^r_{uv} + 0.5w^c_{uv}$
    end if
  end if
end for
return weighted shortest path between s and t in $G_{st} = (V, A', W)$

3.4 Implementation: MINNIE in SDN controller

When the controller compresses a table, the MINNIE SDN application[2] will first execute the routing phase and then the compression phase. Hence, in a dynamic setting, when a new flow must be routed with a new entry in the router, and when the threshold of X rule will be reached, the X^{th} entry is first pushed to the switch (to allow the new flow to travel to the destination), and right after that, the compression is executed. Once the compression module is launched at the controller, a single OpenFlow command is used to remove the entire routing table from the switch. Then the new routes are sent immediately to limit the *downtime* period, that we define as the period between the removal of all old rules and the installation of all new compressed rules. When two or more switches need to be compressed at the same time, the compression is executed sequentially.

After MINNIE compresses an SDN switch rules, the controller must install all the SDN rules in the switch in the order specified by MINNIE. When implementing this action in the controller two problems need to be considered: (i) How to make sure that the SDN device is following the rule order given by MINNIE? (ii) How to install the rules in the SDN switch quickly?

As stated in Section 3.3, the order given by the MINNIE algorithm is the order that should be used to match a packet. We leverage the usage of SDN rule priority to enforce the exact rule order given by MINNIE. SDN rules have a 16 bit priority field that enables 65535 priority numbers. When a packet matches multiple rules which have different priorities, the switch will forward the packet based on the highest priority rule.

In its current version, MINNIE compresses the table based on the source only or destination only. MINNIE routing table will thus end up with 3 types of rules: (i) Normal forwarding rules which match on source and destination (ii) Aggregated forwarding rules that match either on source or on destination (iii) Default rule. It should be noted here that the final routing table can not have at the same time aggregated forwarding rules by source and by destination. Hence, with all these constraints in mind, we need to use 3 priorities when installing compressed rule tables.

In order to minimize the downtime when compressing and pushing its compressed table to an SDN device, we decided to delete all the rules and install the new rules instead of updating existing rules. This decision was motivated by the fact that updating the SDN rules in TCAM is time consuming and an update operation is considered as two operations (delete + insert) [KPK14]. Our methodology leads to a single delete action for the whole table and then a batch of rule insertions. These rules are going to be inserted without waiting for the barrier reply message in order not to provoke high delay (see [KPK14] for details). In case one rule was not installed in the SDN switch, the controller will be notified of this problem and it will then reinstall the required rule. As we will see in a later section, this strategy did not have any negative impact on the network traffic delay or packet loss (Section 3.5.5).

[2] Available at: https://sites.google.com/site/nextgenerationsdndatacenters/our-project/minnie

3.5 Experimental results using an SDN testbed

In this section, we demonstrate the effectiveness of MINNIE using an SDN testbed. The characteristics of our experimental network is described in Section 3.5.1. More specifically, we explain how with a single hardware switch and OvS switches we deploy a full k=4 Fat-Tree topology with enough clients to exceed the routing table size of the hardware switch (Section 3.5.3), as well as a simple traffic pattern that fits the needs of our cases of study. The obtained results are shown in Section 3.5.5, where we discuss the impact of MINNIE over the traffic delay and the loss rate.

3.5.1 TestBed description

Our testbed consists of an HP 5412zl SDN capable switch (K15.15) with 4 modules installed, each with 24 GigaEthernet ports, and 4 DELL servers. Each server has 6 quad-core processors, 32 GB of RAM and 12 GigaEthernet ports. On each server, we deployed 4 VMs with 8 network interfaces each. Hence, we provide up to 32 different IP addresses per physical machine. Each VM is connected to a dedicated OvS switch. Each OvS switch is further connected using one physical port (of the server's 12 ports) to the HP access switch.

The topology of our data center network is a full k=4 Fat-Tree topology (see Figure 3.2), which consists of 20 SDN hardware switches. To emulate those 20 SDN hardware switches, we configured 20 Virtual Local Area Network (VLAN)s on the physical switch (referred to as Vswitches). Since each VLAN possesses an independent OpenFlow instance, each VLAN behaves as an independent SDN-based switch with its proper isolated set of ports and MAC addresses. The VLAN configuration and the consequently port isolation prevents the physical switch from routing traffic among VLANs through the backplane. The Vswitches are then interconnected on the HP switch using Ethernet cables.

Each access switch (Figure 3.2) interconnects a single IP subnet with 16 clients, the latter emulated by two VMs, featuring up to 8 Ethernet ports each one. We detail in subsection 3.5.3 the reason for choosing 16 clients per subnet. Hence, in this network architecture, there is in total 8 subnets, with 16 different IP addresses (i.e. clients) per subnet.

The HP SDN switch can support a maximum of 65536 (software + hardware) rules to be shared among the 20 emulated SDN switches. Software rules are handled in the RAM and processed by the general-purpose CPU (slow path) while hardware rules are stored in the TCAM (fast path) of the switch. The number of hardware rules that can be stored per module in our switch being equal to 750, the total switch capacity is equal to 3000 hardware rules maximum. Those 65536 (software + hardware) available entries are not equally distributed among the 20 switches as the concept of first flow arrived-first served policy is used where the SDN rules are going to be installed on the HP switch in the order of arrival.

In one of the physical servers, we also deployed an additional VM hosting a Beacon [Eri13]

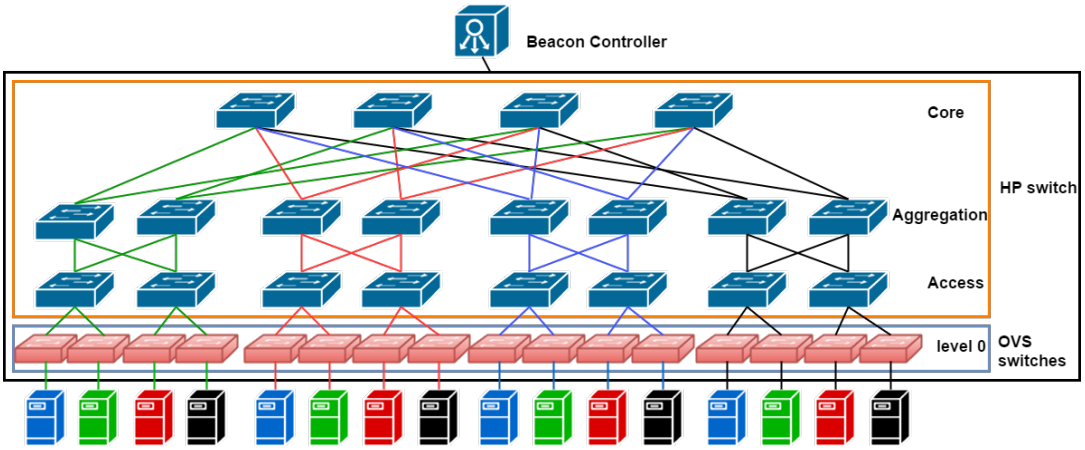


Figure 3.2: Our k=4 Fat-Tree architecture with 16 OvS switches, 8 level 1, 8 level 2, and 4 level 3 switches.

controller to manage all the switches (HP Vswitches or OvS switches) in the data center. According to [SZZ+13], Beacon features high performance in terms of throughput and ensures a high level of reliability and security. To prevent the controller from becoming the bottleneck during our experiments, we configured it with 15 vCPUs (i.e., 15 cores) and 16 GB of RAM.

In the following, we justify our choice of 16 clients per access (level 1) switch and why we have decided to add virtual OvS switches between clients and level 1 switches.

3.5.2 The need of level-0 OvS

OvS switches are used to make the controller aware of every new flow arriving in the fabric. Their routing tables are never compressed.

Without those switches, compressing at access switches with MINNIE may lead to possibly wrong routes. This phenomenon can be explained by considering the case where clients would be directly connected to access switches and MINNIE would be used at those switches. Suppose that a correct routing imposes at one of the access switches that to reach destinations d_1 and d_2, packets must be forwarded to port p_1 while for destination d_3, they should flow through port p_3. Without compression, we have three rules. Now suppose that MINNIE imposes compression when the rules for destination d_1 and d_2 are present but the one for d_3 has not been installed yet. This leads to entries (s_1, d_1, p_1) and (s_1, d_2, p_1) being replaced by $(s_1, *, p_1)$. When packets from s_1 to d_3 are sent later, they will match the compressed forwarding rule and will reach d_3 using a longer path (or no path at all), as they will be forwarded to port p_1 instead of p_3. In order to avoid this behavior, the controller should be contacted for every new flow to take the best routing decision for this flow.

This is the role of the OpenFlow enabled OvS switches that we introduced. They enable the controller to perform compression with an exact knowledge of the set of active flows. The net

result of using those OvS switches is to enable us to perform compression starting from the access switches, giving us more opportunity to use hardware rules at these switches. In the first version of this work, we did not use level-0 OvS switches, and dealt with this problem by not compressing at access switches, leading to lower compression ratios, and overloading of these switches.

Here, one could think that we just migrated the problem from the edge devices to the physical server. We believe however, that this architecture represents an important step towards the solution of limited TCAM space because of the following reasons:

1. While for physical SDN-capable devices, the TCAM size is a real problem, placing one OvS switch per server, even without compressing the flow table, should not introduce major performance problems. Indeed, the number of rules to be processed by each OvS switch should remain modest[3] while an OvS switch can handle 1000 rules in the kernel space, and up to a maximum of 200,000 rules [PPK+15].

2. Virtualization is a common service in modern data centers. Hence, virtual switches are routinely used to provide network access to the virtual machines. OpenvSwitch is natively supported by Xen 4.3 and newer releases. VMware offers support to OpenvSwitch through the NSX service for Multi-Hypervisor, which is the natural choice for large data centers. KVM, due to its native integration in Linux environments, can easily be deployed using OvS switches.

3.5.3 Number of clients chosen for the experimentations

In our Fat-Tree architecture, we can easily deduce the number of rules corresponding to a valid routing assuming that each VM talks to all other VMs not in its IP subnet. Considering no compression at all, one rule is needed for every flow passing through each switch along the path from a source to a destination. The set of flows that a switch "sees" depends on its level in the Fat-Tree. Note that here, a flow is identified by the couple IP source and IP destination addresses. Hence, for every pair of nodes A and B there are two unidirectional flows: $A \rightarrow B$ and $B \rightarrow A$, i.e. two rules per switch on the path from A to B.

For any flow between two servers, the path goes first through the access switches to which the servers are connected. Assuming n servers per access switch ($n = 2$ in Figure 3.2), then each of the n servers connected to an access switch communicates with the other $7 \times n$ servers in other subnets via outgoing and incoming flows. Overall, this represents $14n^2$ flows going through any access switch.

Using the same argument to find the number of flows for switches at the higher levels, we

[3] As reported in [CMT+11], a typical rack of 40 servers generates around 1300 flows per second. Therefore, each server is generating on average around 32.5 flows per second. Assuming a worst case where every per flow rule is unique and that the expiration interval for unused rules is the default value of 10 seconds of inactivity, then, an OvS switch in a single server will need to store 325 forwarding rules roughly (plus the default route to reach the local VM). This value is pretty small as compared to the 1000 rules in the fast path of an OvS switch.

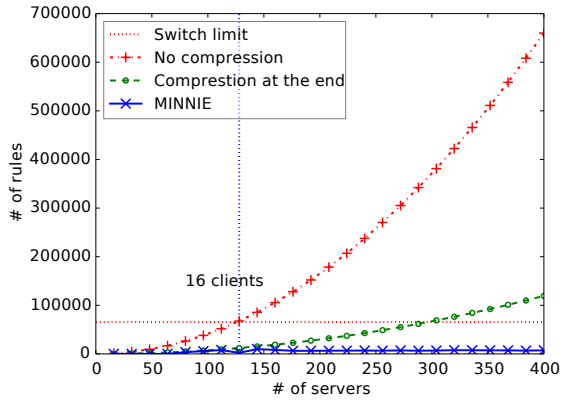


Figure 3.3: Total number of rules installed as a function of the number of servers, in a $k = 4$ Fat-Tree configuration.

have a total of $13n^2$ flows at each aggregation switch and $12n^2$ flows for a core switch. In total, $264n^2$ rules are needed for the entire network.

In Figure 3.3, we compare the total number of rules with no compression at all, and with compression (obtained via simulation) on all switches. Without compression, only 15 clients per subnet can be deployed without running out of space in the forwarding table of our entire data center (65536 entries), while up to 36 clients can be deployed with the compression at the end. Therefore, Figure 3.3 explains our choice of installing 16 clients per subnet. Indeed, it is the first value for which the number of rules exceeds our total limit of number of rules (67584 rules) when no compression is achieved.

3.5.4 Experimental scenarios

We aim at assessing the performance of MINNIE with a high number of rules and with a high load. Those two objectives are contradictory in our testbed. Indeed, stressing the SDN switch in terms of rules, i.e., getting close to the limit of 65536 entries, imposes to have software rules. As software rules are handled, by definition, by the general purpose CPU of the switch (the so-called slow path), a safety mechanism has been implemented by HP to limit the processing speed to only 10000 pps (packets per second) per VLAN. Assuming an MTU of 1500 bytes, we could not go beyond 120 Mbps, shared between all ports in a VLAN. This is why we designed a second scenario where only hardware rules are used. In this scenario, we can fully use the 1 Gbps link but we are limited to the 3000 hardware rules that have to be shared among the 20 switches. We thus built two scenarios to assess the performance and the feasibility of deploying MINNIE in real networks:

- **Scenario 1: Low Load with (large number of) software rules (LLS).** This scenario enables to test the behavior of the switch when the flow table is full.

- **Scenario 2: High Load with (small number of) hardware rules (HLS)**. This scenario enables us to demonstrate that the impact of MINNIE remains negligible even when the switch transfers a load close to the line rate.

For each scenario, we consider three compression cases:

- **Case 1: No compression.** We fill up the routing tables of the switches and we never compress them. This test provides the baseline against which we compare results obtained with MINNIE.
- **Case 2: Compression at the end** (after installing the whole set of forwarding rules or when the forwarding table is full). This scenario illustrates the worst case and provides insights about the maximum stress introduced by MINNIE in the network. Indeed, in this case, we have the highest number of rules to be removed and installed after the compression executed by MINNIE which should be done as fast as possible.
- **Case 3: MINNIE** (Dynamic compression at a fixed threshold). We set a threshold to the table size and compress whenever we reach this value. We extend the third scenario of the simulations by considering three thresholds values for *LLS* , namely 500, 1000 and 2000 entries, and also three values for *HLS*: 15, 20 and 30 entries.

While *LLS* allows to test the scalability of MINNIE in terms of number of rules in real SDN equipments, this scenario might introduce, by default, an important jitter in the network because of the usage of the general-purpose CPU to process the traffic. *HLS* helps to better understand the impact of the compression and forwarding table replacement over the traffic. Since the traffic rate fills up to 75% of the access links, which is not enough to introduce congestion, and packets are processed by the ASIC, we expect to have a low jitter. Hence, any sudden increase of this last will immediately suggest an important impact of the compression mechanisms over the network stability.

3.5.4.1 Traffic pattern

We detail in the following how the two scenarios introduced in the previous section are actually implemented in our testbed.

Low Load with software rules Scenario - *LLS* In this scenario, the traffic is generated as follows: each client pings all other clients in every other subnet. This means that for each access switch, each of the 16 clients pings 112 other clients. There are no pings between hosts in the same subnet as we focus on the compression of classical IP-centric forwarding rules, which is used to route packets between different subnets, and not MAC-centric forwarding rules, as in legacy L2 switches.

We start with an initial client transmitting 5 ping packets to one other client. This train of 5

Level	No Comp	Comp 500	Comp 1000	Comp 2000	Comp full
access	3452	752	761	790	802
aggregation	3233	618	649	672	717
core	3014	97	97	97	97
total	65535	11346	11667	12087	12542

Table 3.2: Average number of SDN rules installed in a virtual switch at each level

ICMP requests forms a single flow from the SDN viewpoint. We wait for this ping to terminate before sending 5 other different ping packets to another client, and so on, until all the 112 clients are pinged. When the first client finishes its pings series, a second client (hosted in the same VM) starts the same ping operation. Hence, the traffic is generated during all the experiment in a round-robin manner, among the 8 clients of each VM. Moreover, VMs do not start injecting traffic at the same time. We impose an inter-arrival period of 10 minutes between them. Hence, VM 1 starts sending traffic at time zero, while VM 2 starts at minute 10, VM 3 at minute 20, and so on. This smooth arrival of traffic in the testbed is motivated by the fact that we do not wish to overload the physical switch with OpenFlow events. Indeed, as stated in [KREV+15], commercial OpenFlow switches can handle up to 200 events/s. Since in our testbed we have 20 virtual switches emulated in th same physical openflow switch, each virtual switch handling its own *flow_mod* (message for sending rules), *packet_out* (message with packet to be sent) and other events, the critical number of events can be easily reached.

The experiment of this scenario ran for almost 3.5 hours. All the rules are installed in the first 2 hours and 45 minutes.

High Load with hardware rules Scenario - *HLS* In this scenario, we used 1 client per VM so that the total number of rules installed (1056 total rules) is less than the hardware limit (3000 rules). Each VM starts a 50 Mbps ICMP traffic with the other clients in a round robin manner. After starting the first client machine, we wait for 75s and then start the outgoing connections for the second VM and so on, until all the machines establish connections with one other client. In this scenario, we have chosen 50 Mbps per connection in order to have a maximum of 800 Mbps load on a 1 Gbps link when all connections are established.

Each experiment of this scenario ran for 1 hour and all the rules are installed in the first 20 min. As mentioned earlier, all the rules were installed in hardware in order to reach high loads.

3.5.5 Experimental results

3.5.5.1 Scenario 1: Compression with *LLS*

Level	Comp 500	Comp 1000	Comp 2000	Comp full
access (8 switches)	79%	78.75%	77.95%	77.61%
aggregation (8 switches)	81.43%	80.51%	82.14%	78.45%
core (4 switches)	96.84%	96.84%	96.84%	96.83%
total (20 switches)	83.21%	82.19 %	81.55 %	81.44%

Table 3.3: Average percentage of SDN rules savings at each level

Number of rules with/without compression As explained in Section 3.5.3, in this scenario and without compression, the limit of 65536 entries in our HP switch is reached. On the other hand, compressing the table with MINNIE allows to install all the required rules without reaching the limit when compressing at a given threshold (500, 1000 or 2000 entries) or when the flow table is full. Indeed, as shown in Table 3.2, the total number of installed rules does not exceed 13000 in all compression cases. This represents a total saving higher than 80% of the total forwarding table capacity (Table 3.3) with a saving larger than 96% at the third level and a minimal saving over 76%. We also notice that the percentage of savings decreases as we go from level 1 (access) to level 3 (core), this is mainly due to the number of routing path options to reach a destination that exist on a node. In a fat tree, using only Dijktsra's algorithm the number of routing variations decrease by $k/2$ as we move from one level to the other, the routing variation between level 2 and level 1 is $k/2$ and between level 1 and level 3 it is $k/4$. Hence, as the number of routing variation decrease the rule saving percentage increases.

Figure 3.4 depicts how the number of rules evolves over time with and without compression. Please, note that this figure takes into account the total number of forwarding rules in the network, including both OvS and the HP switch. The number of rules increases at the same pace in all 3 scenarios during the first 30 minutes. When the compression is triggered, the number of rules decreases. Later, for compression at 500, 1000, and 2000 entries, the number of rules increases at a lower pace than in the non compression case. This is because (i) the controller has installed some wildcard rules and so no new rules at level 1, 2 or 3 need to be installed for new flows, and (ii) other compression events are triggered. We further notice here that the presence of wildcard rules influences the routing scheme as the new incoming flows will follow these paths ($i.e.$ corresponding wildcard rules path) in priority which explains the difference between the results of compression when the forwarding table is full and compression with fixed thresholds as can be observed in Figure 3.4.

Compression time Figure 3.5 shows the compression duration seen by the controller, which consists of: (i) the computation time of compressed rules that can be seen in Figure 3.6, (ii) the removal of the current forwarding table, (iii) the formatting of the compressed rules to the OpenFlow standards, and (iv) the injection of the new rules to the switch.

We notice that the compression time per switch remains in the order of a few milliseconds. Indeed, compression takes about $5\,\mathrm{ms}$ (resp. $7\,\mathrm{ms}$) for compression at 500 and 1000 entries (resp.

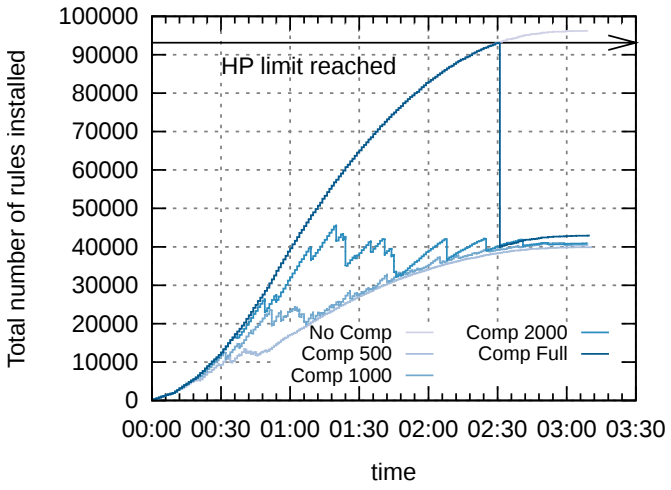


Figure 3.4: Total number of rules installed in the whole network

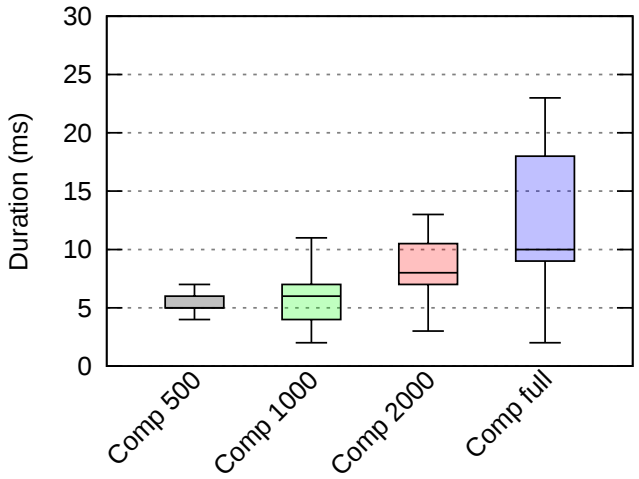


Figure 3.5: Average duration of compression period.

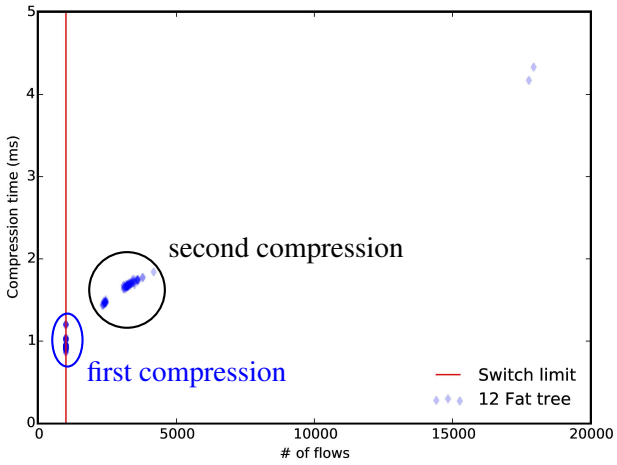


Figure 3.6: Scatter plot of the time to compress a routing table of a $k = 12$ Fat-Tree.

2000 entries). Even the worst case – compressing when the table is full – represents less than $18\,\mathrm{ms}$ for most of the switches with a median at $9\,\mathrm{ms}$. Moreover, in this latter case, sequentially compressing all switches requires no more than $152\,\mathrm{ms}$. For any switch, the first compression is done when reaching 1000 flows corresponding to 1000 forwarding rules (as aggregation rules are only introduced at the first compression). We then see that the second compression for a switch is done for around 2500 flows followed by compression when reaching 3000 to 4000 flows. These compression results show that previous compressions were efficient and that a large number of new flows are routed via aggregated rules.

This compression period is mainly due to the time needed to delete all the routing table using one delete request and install all the new rules in the switch. Indeed, as we can see in Figure 3.6, even for a larger scale $k = 12$ Fat-Tree with 432 servers with an all-to-all traffic, the average compression computation time is $1.29\,\mathrm{ms}$.

Our total compression time results are inline with the results shown in [KPK14] (Figure 3). The reason why we have smaller delays is the fact that, as stated before, we do not wait for the barrier reply before sending the next flow insertion rule (hence we ignore the time it takes to receive the barrier reply message); moreover, we delete all the rules using a single action instead of deleting each rule one by one.

SDN control path In the SDN paradigm, the controller-to-switch link is a sensitive component as the switch is CPU bounded and cannot handle events at a too high rate. Figure 3.7 represents the network traffic between the switch and the controller in the different scenarios. We can observe that the load increases highly when the switch limit in terms of number of software+hardware rules is reached and we do not compress the routing tables. After time t=2:30h, the limit is reached and for every packet of every new flow, each switch along the path has to ask the controller for the output port. These traffic peaks vanish when we compress the routing tables for the 1000 and

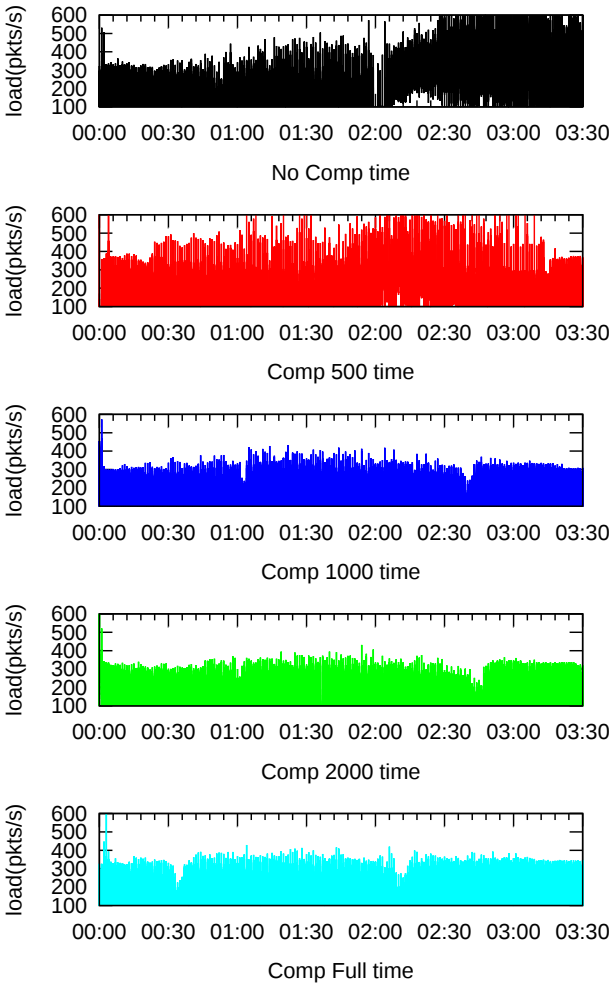


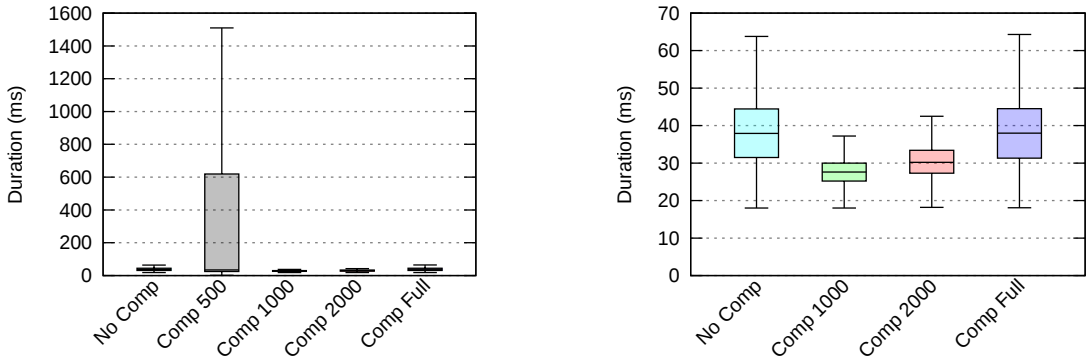
Figure 3.7: Network traffic between the switches and the controller.

2000 limits or for the case of compression when full. As for the compression at 500 scenario, we notice the occurrence of high peaks after the first hour. They result from successive compression events (over 16000 in our experiments as can be seen in Table 3.4) that are triggered by any new packet arrival. Indeed, in this scenario, most of the switches will perform a compression for every new flow, since the total number of rules after compression remains higher than the threshold.

To understand the impact of the control plane on the data plane, we have to look at three key metrics that we detail in the following sections: (i) the loss rate for all scenarios; (ii) the delay of the first packet of new flows that should be higher when there is no compression (at least after t=2:30h) or during compression at 500 and (iii) the delay of subsequent packets (packets 2 to 5) that should be larger for the case of no compression when the table is full. We ruled out a precise study of the loss rate as the load in this set of experiments (*LLS*) is low. We report in Table 3.4 the loss rates observed for all scenarios. Though there exist some significant differences between the different scenarios, the absolute values are fairly small. We therefore focus on delays hereafter.

Threshold	No Comp	Comp 500	Comp 1000	Comp 2000	Comp full
# of compressions	NA	16594	95	28	20
% pkt loss	6.25×10^{-6}	0.003	5.65×10^{-4}	2.83×10^{-5}	3.7×10^{-4}

Table 3.4: Total number of compressions and packet loss rate.



(a) First packet delay boxplot with compression 500

(b) First packet delay boxplot without compression 500

Figure 3.8: First packet delay boxplot

New rules installations: Impact on first packet delay The first packet delay provides insights on the time needed to contact the controller and install the rules when a new flow arrives. Indeed, the round trip delay seen by the first packet of a new flow includes the network propagation delay, the queuing delay, and the time needed by a switch to obtain a new rule.

We observe in Figure 3.8 that for the scenarios with compression at 1000 rules and compression at 2000 rules, the first packet delay ranges from $25\,\mathrm{ms}$ to $35\,\mathrm{ms}$. This increase as compared to subsequent packets of the same flow- which can reach a factor of 10 as we will see in the next section- highlights the price to pay to obtain and install a forwarding rule in software. The results can be significantly worse if the controller is frequently modifying the forwarding rule, like in the compression at 500 rules case. Indeed, for that special case, the third quantile reaches up to $600\,\mathrm{ms}$ for the first packet delay.

Surprisingly, the cases without compression and compression at the table limit lead to similar results. Compressing when the table is full should intuitively lead to better performance as in a number of cases, a limited number of new rules are needed and can be installed as compared to the no compression case. However, in our tests, the table becomes full after 2 hours and 30 min of experiment (out of 3 hours). Hence the similarity of results in Figure 3.8. In fact, when the table is full, the impact is striking, as can be seen in the time series of Figure 3.9a, which shows the evolution of the first packet delay per new flow when no compression is executed. Indeed, after 2:30 hours - when the table is full- we can observe a jump in the delay for no compression while when compressing at the table limit the trend is the opposite and the delay decreases (Figure

3.9e) after compression. As for the case of compression at 500, the first packet delay features a chaotic behavior (Figure 3.9b) due to its high compression frequency as expected. Regarding the scenarios of compression at 1000 (Figure 3.9c) and compression at 2000 (Figure 3.9d), the benefits of compressing periodically are stricking: the first packet delay shows a constant trend during the whole experiment.

Eventually, note that the results obtained here are impacted by the fact that we use software rules, which increase the delay to install rules.

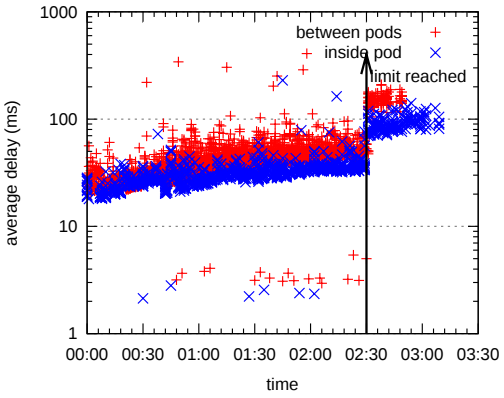
Subsequent packets delay As explained previously, we expect to observe higher delays for subsequent packets for the case of no compression (when the table is full) and also possibly for the case of compression at 500 as the switches have to reinstall new rules at a high frequency.

In our experiments, the delay seen by packets 2 to 5 of each flow is shorter than $4\,\mathrm{ms}$ most of the time for scenarios without compression, compression at 1000, compression at 2000 and compression at the forwarding table size limit, as we can see in Figure 3.10. Compression at 500 is slightly different (the third quartile reaches up to $5\,\mathrm{ms}$), highlighting the negative impact of the high frequency of compression events on the data path of the switches.

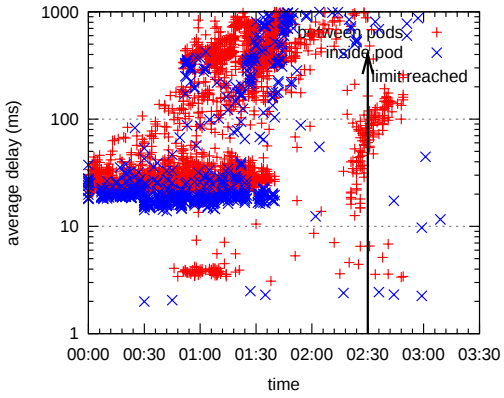
Figure 3.10 aggregates all the results together and we have again to resort on the time series to observe specific effects. When all needed forwarding rules are successfully installed and the compression frequency is low (which is the case for compression at the limit, compression at 1000 and compression at 2000), the delay of packets 2 to 5 is consistently comprised between $2\,\mathrm{ms}$ and $6\,\mathrm{ms}$ (Figures 3.11d, 3.11e and 3.11f).

Without compression, while most of the packets experience a delay between $2\,\mathrm{ms}$ and $6\,\mathrm{ms}$ before the table limit is reached, all new incoming packets will see a delay equal or higher than $40\,\mathrm{ms}$ afterwards (Figure 3.11b). As for the case of compression with small table limit (500 rules), we remark in Figure 3.11c a time interval between 1:45 hour and 2:15 hour, where the delay increases suddenly from $2\,\mathrm{ms}$ to $100\,\mathrm{ms}$. This is because some switches are unable to reach a forwarding table smaller than 500 rules even after compression, and hence, the controller executes a compression after every new flow arrival. After 2:15 hours, the frequency of new incoming flows that need to be installed decreases (Figure 3.9b), leading to a stabilization of the delay.

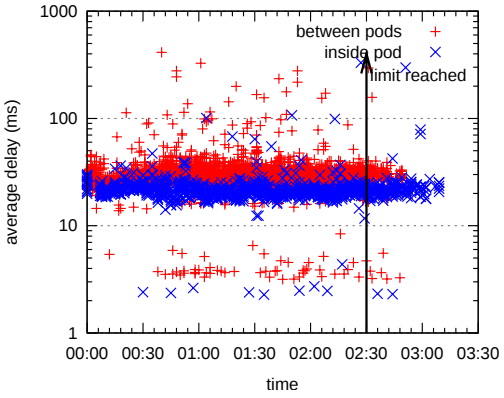
From all the results shown above, we notice that putting a low table limit (e.g. 500) has a bad impact over the traffic passing through the network, whereas setting it to 1000 and 2000 provided enhances performance for network traffic. This is due to the fact that in our scenarios, the compressed table had a size larger than 500 while it was always less than 1000. Hence, in order to leverage always the benefit of MINNIE we advise to set a dynamic threshold that will change based on the compressed table size - see Section 3.7.



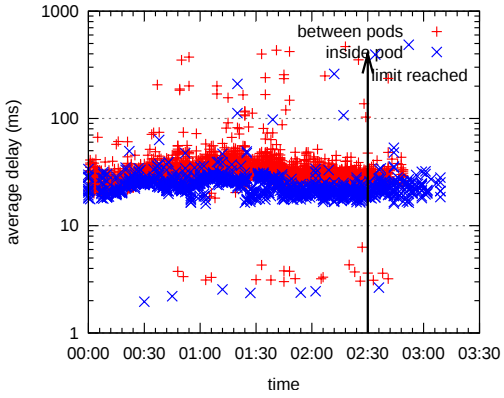
(a) Without compression



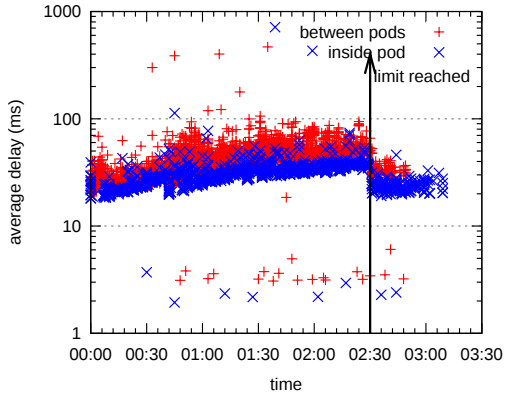
(b) Compression 500



(c) Compression 1000



(d) Compression 2000



(e) Compression when full

Figure 3.9: First packet average delay with low load

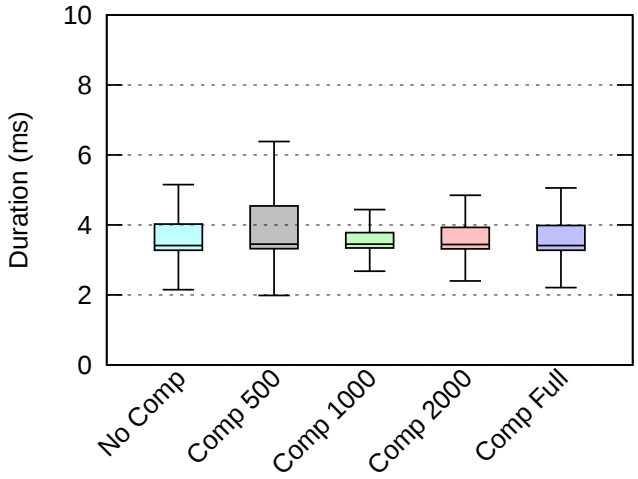


Figure 3.10: Average packet's delay boxplot for packets 2 to 5

3.5.5.2 Scenario 2: compression with *HLS*

We have so far investigated the behavior of MINNIE in an environment where the flow table can be full. The latter scenario involves the use of software rules and thus the slow path of our HP switch.

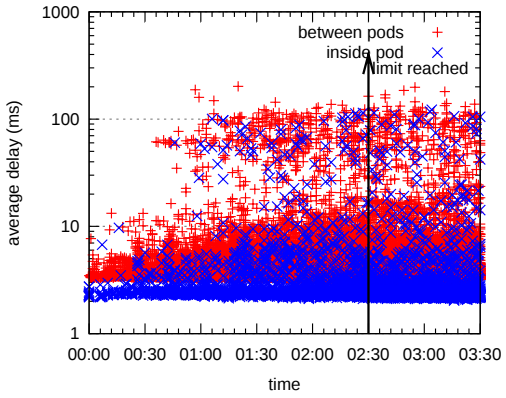
We now turn our attention to the case where the load on the data plane is as high as 80%. This entails using hardware rules only and we are limited to 3000 such rules with our HP switch, shared among the 20 switches of our k=4 Fat-Tree topology. The experiments in this section are consequently performed with 1 client per access switch (16 clients in total) and an all-to-all traffic pattern with 50Mb/s per flow.

As expected, the first packet round trip delay decreases to around $1\,\mathrm{ms}$, while packets 2 to 5 experience a round trip delay of around $0.55\,\mathrm{ms}$[4]. The compression duration, in all scenarios, is equal to $1\,\mathrm{ms}$ only, which is understandable given the small total number of flows. **More importantly, we noticed no packet losses and no drastic effects on delay even during compression events, which proves that MINNIE is a viable and realistic solution**. Indeed, the maximum variation of delay between the delays of no compression and all compression scenarios is less than $0.1\,\mathrm{ms}$, a value which might be observed even in non-SDN networks (see Figure 3.12).

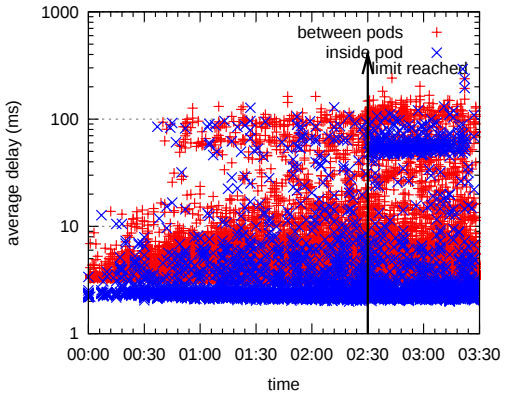
The compression ratio in Table 3.5 demonstrates that even with a low number of rules, MINNIE can achieve a high compression ratio, over 70%. Figure 3.13 which represents the evolution of the forwarding table size for all cases – no compression, compression at 15, 20, 30 and when full (after installing all the needed rules)– highlights that MINNIE maintains a similar low number of rules in all compression scenarios.

A last question that we aim at investigating is the impact of compression on TCP connections.

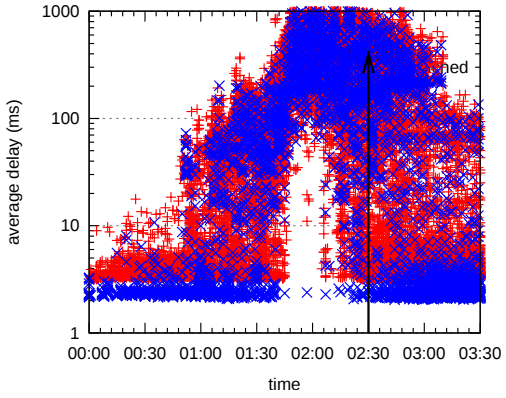
[4]A direct comparison between these delays and the one for the low load and software rules scenario is not straightforward. Section 3.2 will present a fair comparison of these two modes.



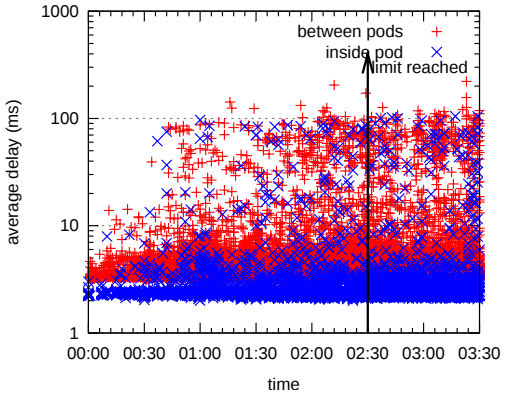
(a) Without compression 7 IPs



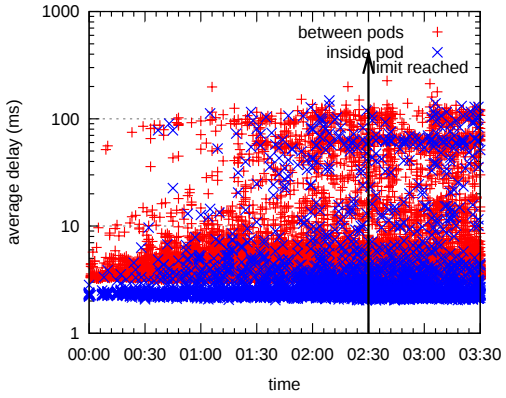
(b) Without compression 8 IPs



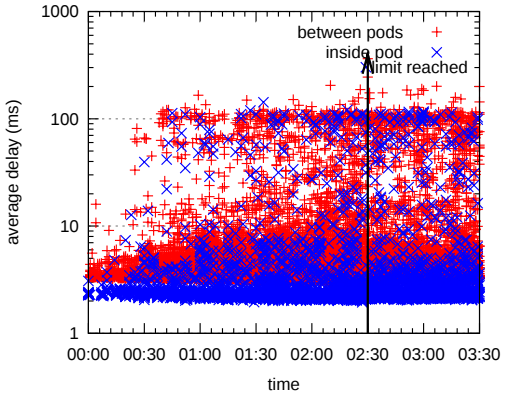
(c) Compression 500



(d) Compression 1000

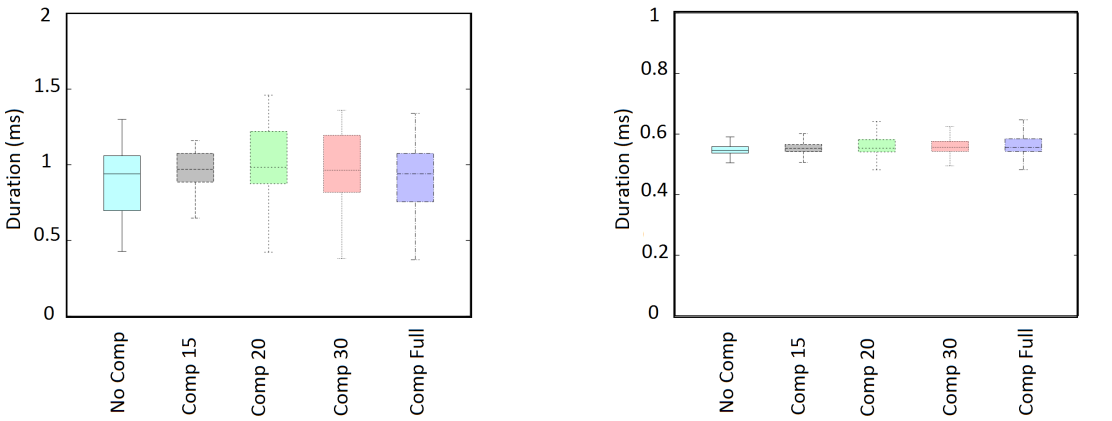


(e) Compression 2000



(f) Compression when full

Figure 3.11: Average packet delay of pkts 2 to 5 with low load



(a) First packet's delay

(b) Average packet's delay except first packet

Figure 3.12: Packet delay boxplot under high load

Level	Comp 15	Comp 20	Comp 30	Comp full
level 1 (8 switches)	76.56%	75.66%	75%	72.76 %
level 2 (8 switches)	75.48%	73.31%	71.87%	69.71 %
level 3 (4 switches)	76.04%	76.56%	74.47%	73.95 %
total (20 switches)	76.04%	74.9 %	73.67 %	71.78 %

Table 3.5: Average percentage of SDN rules savings at each level under high load

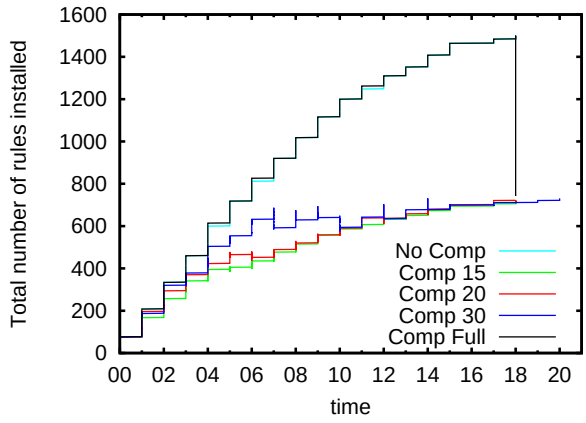


Figure 3.13: Total number of rules installed in the network under high load

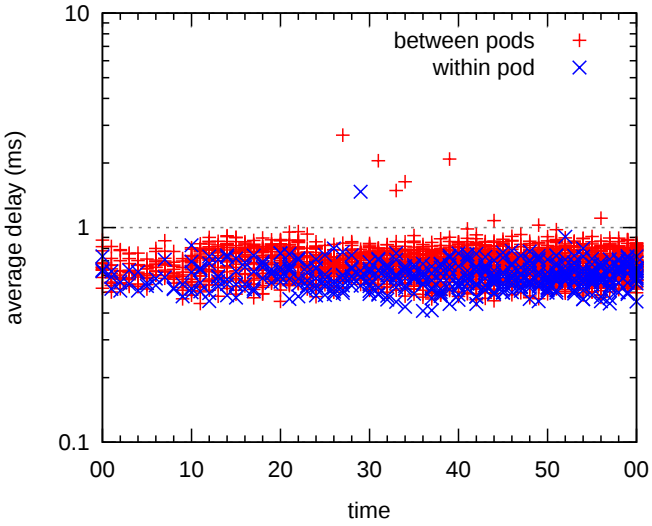


Figure 3.14: High load and hardware rules: Delay of packets 2 to 5 - Compression at 20 entries

The high load scenario is especially relevant as data centers are in general operated at high loads. The variation of the round trip delay of most of the packets is less than $0.1\,\mathrm{ms}$ (Figure 3.12) for compression at 20 entries with the highest variability. For compression at 20 entries and during the first 20 minutes of the experiment (compression events occur during that period), the minimum and maximum round trip delays between servers in the same pod is around $0.4\,\mathrm{ms}$ and $0.6\,\mathrm{ms}$ respectively, while the minimum and maximum round trip delays between servers in different pods is around $0.55\,\mathrm{ms}$ and $0.8\,\mathrm{ms}$ respectively (see Figure 3.14). Those observed delays will not produce any problem to TCP connections. Indeed, the minimum RTO value (the time needed to trigger a TCP timeout and retransmit a non Acked packet), is equal to $200\,\mathrm{ms}$ in Linux systems (and defined to be 1 second in the RFC 2988 [PA00]), which is far from our observed delays (lower than a millisecond). A recent draft submitted to the TCPM Working Group [BEBJ15] appeals for a decrease of the minimum RTO value to $10\,\mathrm{ms}$. Once again, the maximum delay observed during the compression events is still far from that proposed minimum RTO. Hence, compression operations should not lead to any spurious TCP time out. Note eventually that results obtained in the simulations on the computational time (last column of Table 3.6 of Section 3.6.2) confirm that the impact of MINNIE on the delay experienced by the packets of the flow should be limited in general.

3.6 Simulations scalability results

In order to assess the scalability of MINNIE, we study its behavior through simulations for a wide variety of data center architectures. We first present the different scenarios, performance metrics and data center architectures in Section 3.6.1. We then demonstrate that MINNIE works well for topologies of different sizes and structures in Section 3.6.2.

3.6.1 Simulation settings

We present in this section the different scenarios studied via simulations, the traffic patterns and metrics that will be evaluated. All simulations were carried out on a computer equipped with a 3.2 GHz 8 Core Intel Xeon CPU and 64 GB of RAM.

3.6.1.1 Scenarios

We ran simulations under the same different cases as in the experimental section (i.e. No compression, Compression at the end of the simulation and MINNIE). And for all scenarios, we consider an all-to-all traffic in which every single server establishes a connection to all other servers. Each flow is constantly sending traffic. We consider this situation to test MINNIE in the most extreme scenario in terms of number of flows, and thus, in terms of number of rules. Each flow is represented by a unique source-destination pair.

3.6.1.2 Data center architectures

To test the efficiency of MINNIE, we considered state-of-the-art data center architectures: Fat-Tree [AFLV08], VL2 [GHJ+09], BCube [GLL+09] and DCell [GWT+08]. For each family of architecture, we considered topologies of different sizes hosting from few units to about 3000 end points. These end points can be either servers or IP subnets, grouping hundreds of different machines. In the following, for simplicity, we often use the term *server* for both cases. The number of flows routed in the topologies can thus reach a few millions.

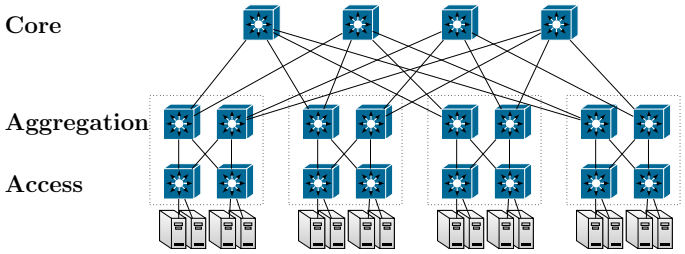
The architectures considered during these simulations can be classified into two different groups:

- **Group 1**, in which servers only act as end hosts includes Fat-Tree and VL2.
- **Group 2**, in which servers also act as forwarding devices (similarly to switches) includes BCube and DCell.

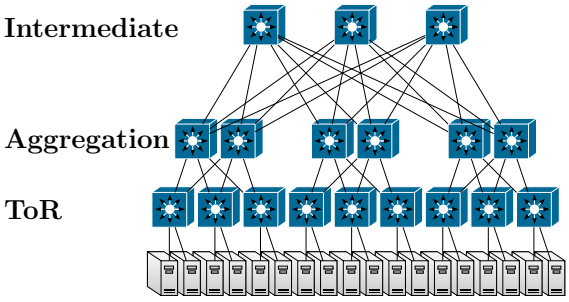
We detail below how we chose the different set of parameters to build these topologies like the number of switches or level of recursion.

Fat-Tree. The Fat-Tree is one of the most well-known architectures. The switches are divided into three categories: core, aggregation and access (or ToR for Top of the Rack) switches. A k Fat-Tree is composed of k pods of k switches and $k^2/4$ core switches. Every switch possesses k ports. Inside a pod, aggregation and edge switches form a complete bipartite graph. Each core switch is connected to every pod via one of the $k/2$ aggregation switches. Every ToR switch has a rack composed of $k/2$ servers. A $k = 4$ Fat-Tree is shown as example in Figure 3.15a.

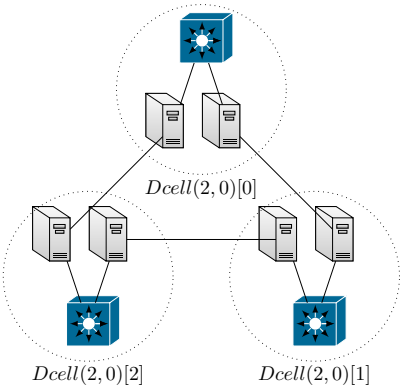
For our simulations, to build Fat-Trees with up to 3000 servers, we considered k values



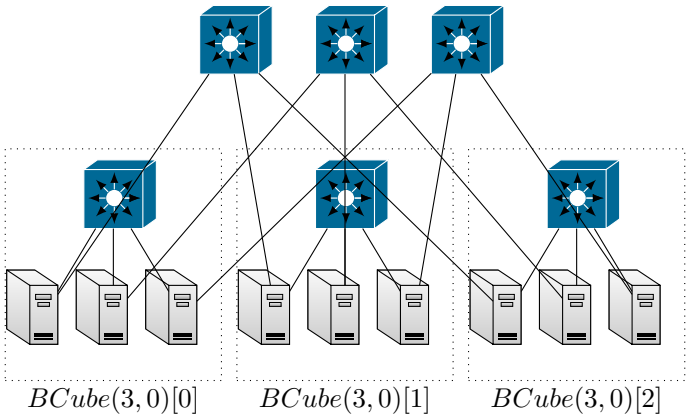
(a) Group 1: A Fat-Tree with $k = 4$ pods



(b) Group 1: A VL2 network with $D_i = 6$-ports intermediate switches, $D_a = 6$-ports aggregation switches and $T = 2$ servers per ToR



(c) Group 2: A DCell(2, 1), composed of 3 DCell(2,0)



(d) Group 2: A BCube(3, 1), composed of 3 $BCube(3,0)$

Figure 3.15: Example of topologies studied.

between 4 and 22.

VL2. The VL2 architecture is also composed of three layers of switches: intermediate, aggregation and ToR switches. The intermediate and aggregation switches are connected together to form a complete bipartite graph. Each ToR is connected to two different aggregation switches. Three parameters control the number of switches of each layer and the number of servers of the architecture: D_a represents the number of ports of an aggregation switch, D_i the number of ports of an intermediate switch and T the number of servers in the rack of a ToR switch. Figure 3.15b shows a $VL2(D_a = 6, D_i = 6, T = 2)$. The topology has $D_a/2$ (3 in the example) aggregation switches, D_i (6 in the example) intermediate switches, $D_aD_i/4$ (9 in the example) ToR switches and $TD_aD_i/4$ (18 in the example) servers.

For our simulations, we chose the parameters of the topologies to ensure that every switch has the same number of ports, that is VL2$(2k, 2k, 2k - 2)$ for k between 2 and 11.

DCell. The DCell architecture is a topology in which both servers and switches act as forwarding devices. The topology is built recursively. The basic block is the level-0 DCell, $DCell(n, 0)$, where n servers are connected to a unique switch. From a $DCell(n, l-1)$, composed of $s(n, l-1)$ servers, a $DCell(n, l)$ can be build by connecting each server of a $DCell(n, l-1)$ to a different $DCell(n, l-1)$. This builds a $DCell(n, l)$ containing $(s(n, l)+1)$ $DCell(n, l-1)$. For example, a $DCell(2, 0)$ is composed of 2 servers ($s(n, 0) = n$) and to create a $DCell(2, 1)$, as shown in Figure 3.15c, 3 $DCell(2, 0)$ are interconnected.

In our simulations, we compare topologies with one level of recursion (referenced as $DCell(l = 1)$), with n between 1 and 54, and topologies with two levels of recursion (referenced as $DCell(l = 2)$), with n between 1 and 7.

BCube. BCube is another architecture in which the servers also act as forwarding devices. Again, it is a recursive construction. The building block is a $BCube(n, 0)$, composed of n servers connected to a single switch. The level l being composed from multiple $l - 1$ levels. Unlike in the construction of DCell, in which the recursion connect servers together, the construction of BCube is done by connecting the servers via new switches. The number of switches added to make a BCube of level l is equal to the number of servers in a BCube of level $l - 1$. Each switch is then connected to one server of every BCube of level $l - 1$ and each servers to $l + 1$ switches – see the BCube$(3, 1)$ in Figure 3.15d.

Like for DCell topologies, the same number of servers can be obtained with different levels of recursion. We consider up to 3 levels of recursion.

3.6.2 Simulation results

In this Section, we validate the scalability of MINNIE through simulations over the set of topologies described above. We demonstrate here that MINNIE works well for different topologies and different sizes of data centers. We first analyze the compression rates that can be obtained by compressing large tables. Then, we show that if tables are compressed all along the simulation as soon as the limit is reached, then the compression module is much more efficient and the compression ratio reaches 90% for some topologies. We then investigate the efficiency of MINNIE when considering around 1000 servers in multiple topologies. We show the efficiency of our method by comparing the results of MINNIE with XPath [HCW+15].

Metrics To assess the efficiency of MINNIE, we measure the following metrics:

- Average compression ratio of compressed tables:

  $$\textit{compression ratio} = 1 - \frac{\textit{number of rules of a switch}}{\textit{number of flows passing through the switch}}$$

  Note that the compression ratio measures the efficiency of the compression algorithm. We thus, do not consider tables on which no compression event was performed (in particular empty tables), when we compute the average compression ratio.

- Number of compression events performed by a switch during the simulation.

- Number of flows passing through a switch (maximum and average over all switches).

- Number of rules per switch (maximum and average over all switches).

- Computation time for compressing a table and for routing a flow.

- Maximum number of servers which can be installed on a data center topology without going beyond a forwarding table size of 1000 rules.

For each family of topologies, we present the results for the three scenarios described in Section 3.6.1.1, referenced respectively as *No compression*, *Compression at the end* and MINNIE.

3.6.2.1 Efficiency of the compression module

The efficiency of the *compression module* of MINNIE can be assessed from Figure 3.16 where we look at the average compression ratios of the *Compression at the end* scenarios. In this figure we observe that DCell, BCube and VL2 topologies follow a similar phenomenon. They all feature a sharp increase of the compression ratio when the number of servers is between 0 and 100: for example, the ratio raises from 62% to 84% for DCell(l=2). Then, for larger number of servers, the compression ratio levels off. On the other hand, Fat-Tree topologies have a different behavior and do not experience the increase phase; the curve is almost flat all along the simulation. The higher ratio shown on DCell topologies is explained by the aggregation of flows on the few switches

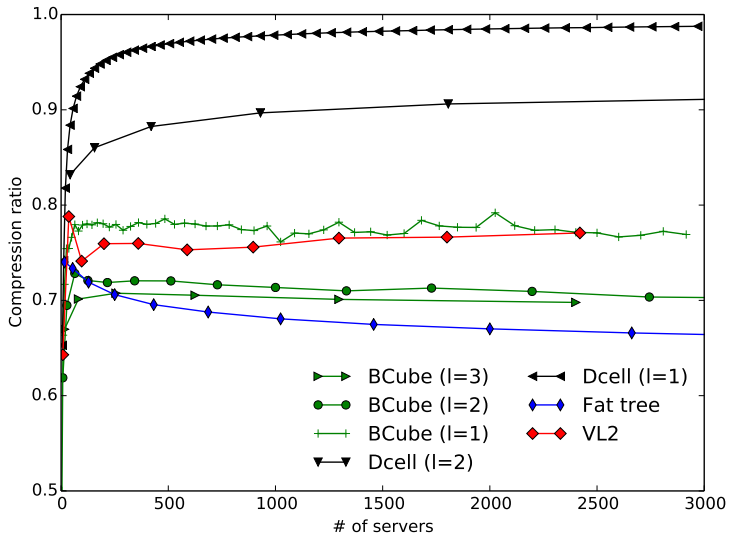


Figure 3.16: Compression ratio for the different topologies in Scenario 2.

available in the topology. Combined with a few number of outgoing ports, the compression module can attain a very high compression ratio.

In the flat phase, compression ratios are between 60% and 80% for the three families BCube, VL2 and Fat-Tree, and even reach values between 85% and 99.9% for DCell. **In summary, the compression module of MINNIE features a minimum of 60% savings in memory.**

Compression event frequency. In Figure 3.17, we observe the total number of compression events executed for the different topologies. Group 1 topologies reach a maximum of 516 compressions for the $k = 18$ Fat-Tree (and 301 for VL2(20, 20, 18)). This represents an average of about 1 compression event per switch for the Fat-Tree topology and less than 6 compression events for VL2. However, Group 2 shows a higher number of compression events, with a maximum of almost 6000 compression events for a $BCube(53, 1)$ (in average, 54 compression events per forwarding device). This difference is due to the near saturation of most of the switches in Group 2 topologies. In these nearly saturated tables, the compression leaves a table that is close to the 1000 limit and thus, the table is compressed only after few new flows are added.

3.6.2.2 Efficiency of MINNIE

MINNIE is composed of a routing and a compression module. When the number of rules reaches the 1000 limit, MINNIE triggers the compression module. This dynamic behavior allows to efficiently route traffic *without overloading the routing tables* on topologies where the number of servers increases. Figure 3.18 presents the maximum number of rules on a device (a router or a server depending on the family of topology) as a function of the number of servers for the different families of topologies. We remark that the curve for MINNIE first follows the *No compression* one

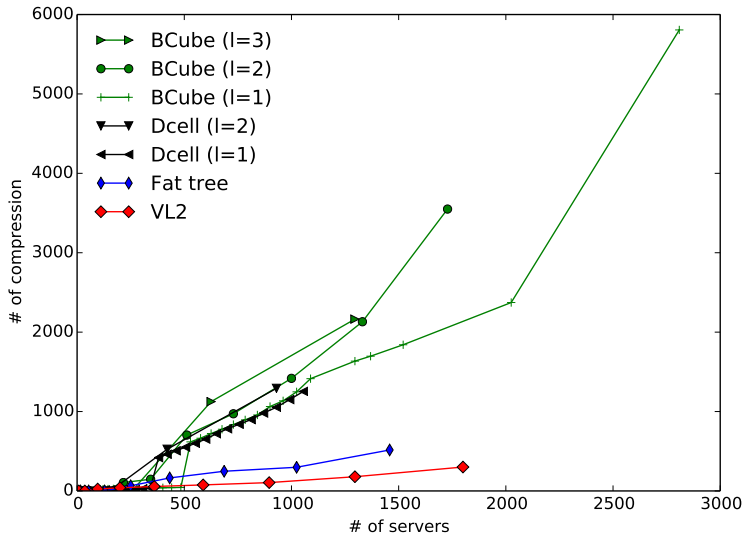


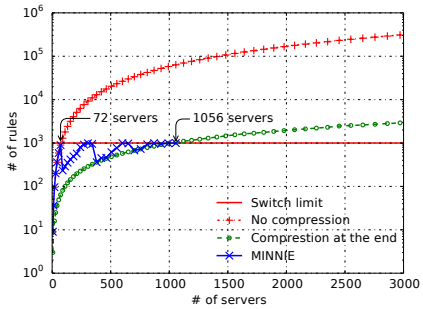
Figure 3.17: Number of compression executed for different topologies

until reaching the 1000 limit. Indeed, during this first phase, MINNIE performs no compression at all as the limit is not attained. Then, MINNIE triggers compression regularly and manages to keep all routers' table below the limit of 1000. When performing compression, MINNIE has introduced wildcard rules in the routing tables, and the new incoming flows will follow these paths in priority. Therefore, MINNIE deals with the same number of flows as *No Compression* with less than 1000 entries while *No Compression* needs between 10^4 and 10^6 entries. Note that some points for MINNIE are not depicted. Indeed, in Figure 3.18, we present only the results in which all the flows are routed without overloading the routing tables. As soon as one request cannot be routed and when the routing tables cannot be further compressed, the simulations are stopped.

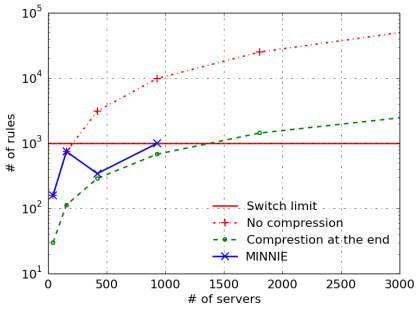
This phenomenon can be clearly seen for DCell(l=1) topologies in Figures 3.18a. Without compression, only 72 servers can be deployed in a DCell(8,1) without overloading tables while MINNIE allows to deploy 1056 servers with a DCell(32,1).

This represents a 15 fold increase compared to *No compression*. The number of servers which can be deployed with DCell topologies having two levels of recursion (Figure 3.18b) is similar: 930 with a DCell(5, 2) when running MINNIE and less than 200 with *No compression*.

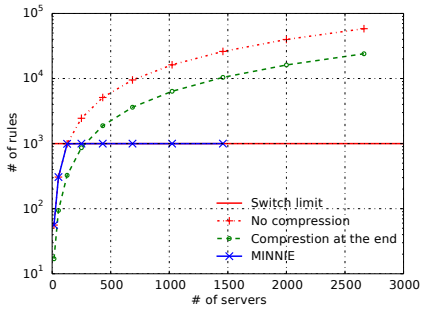
Another key observation is that MINNIE can reach or even outperform *Compression at the end* without exceeding the limit of number of rules. Indeed, if we consider for example Fat-Tree topologies in Figure 3.18c, without compression, the largest Fat-Tree which can be deployed with a rule limit of 1000 is a $k = 8$ Fat-Tree with 128 servers and 992 rules. With compression at the end, the number of servers which can be deployed would be around 256. However, we see that MINNIE succeeds in deploying a $k = 18$ Fat-Tree with 1458 servers without having overloading issues. This is a 6 fold increase compared to *Compression at the end*. **This is due to the fact that by compressing online, i.e., when flows are introduced, MINNIE impacts the routing of**



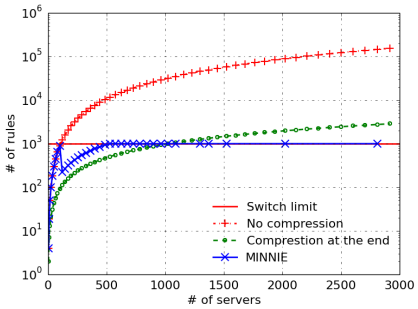
(a) DCell Topologies (l=1)



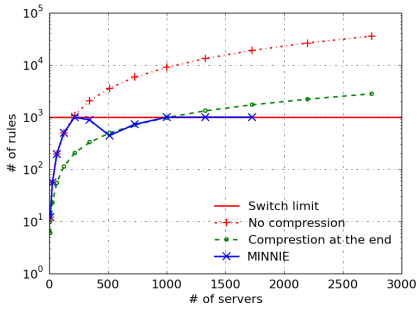
(b) DCell Topologies (l=2)



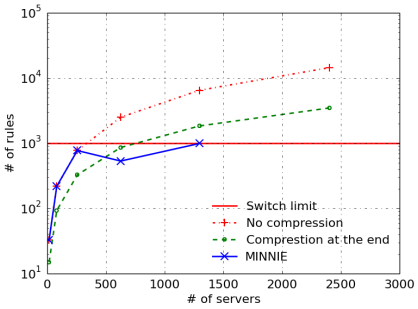
(c) Fat-Tree topologies



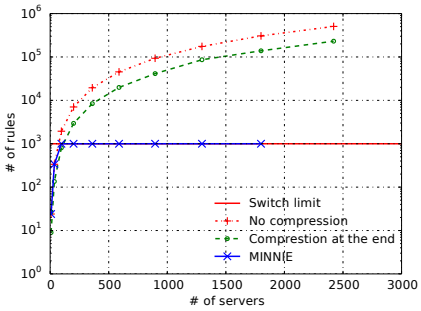
(d) BCube Topologies (l=1)



(e) BCube Topologies (l=2)



(f) BCube Topologies (l=3)



(g) VL2 Topologies

Figure 3.18: Maximum number of rules on a forwarding device as a function of the number of servers for different data center architectures.

the following flows. Because of the metrics used in the routing module, the algorithm will prefer to select shortest paths using wildcards as they do not increase the number of rules. This allows better compression ratios.

The phenomenon also appears for BCube topologies (Figures 3.18d, 3.18e, 3.18f) and with a striking intensity for VL2 topologies (Figure 3.18g). When compressing at the end, up to 96 servers can be deployed without reaching the table size limit (and only 36 without compression). With MINNIE, this number can be pushed up to 1800 servers which represents **36 fold increase**.

Difference of behavior inside a family of topologies. We notice in Figure 3.16 and 3.18 a difference of behavior inside a family of topologies. For a given family of data centers, different topologies can host a similar number of servers. For example, DCell(32,1) and DCell(5,2) host around 1000 servers, as well as BCube(32,1), BCube(10,2) and BCube(6,3). But the behavior of these topologies is significantly different: for example, the average number of rules is 113 for a DCell(32,1) compared to 642 for a DCell(5,2). We see that the compression ratio of the family DCell(l=1) is higher (more than 95% when the number of servers is greater than 200) than the one of DCell(l=2) (more than 85% when the number of servers is greater than 200). **Hence, the choice of the best set of parameters for a given family of topologies is very important.** In order to answer this question, we study in the following all these topologies with similar number of servers (around 1000).

3.6.2.3 Comparison of MINNIE effect on topologies with 1000 servers

Table 3.6 sums up the effect of MINNIE on the different topologies with a similar number of servers (around 1000), hence a similar number of flows to route. We detail below the different parts of the table, highlighting the key conclusions to draw.

Topology characteristics. The first part of the table provides basic information about the topologies. **Even with a similar number of servers, the topologies are very different** in terms of number of switches (between 20 and 903), links (between 1056 and 5184) and average number of ports per switch (between 2.9 and 54.4).

Flows in the network. The second part of the table reports the number of flows introduced in the network during the simulation. These topologies behave very differently in terms of number of flows per device: the average number of rules ranges from 3734 to 216000 and the maximum number of rules ranges from 7800 to 650000. Two explanations can be given for these differences. First, the topologies have very different numbers of switches (from 20 to 864). Secondly, in the topologies of Group 2, servers also act as switches, and thus also host some rules, leading to a lower average number per device.

Topology	servers #	switches #	links #	Avg ports #	# flow per switch		Rule w/ comp #		Average Comp.	Computation time in average (ms)	
					Max	Average	Max	Average	Ratio	Paths	Comp.
Group 1											
$k = 4$ Fat-Tree (64)	1024	20	1056	54.4	454244	216268	999	446	~ 99.60	0.17	13
$k = 8$ Fat-Tree (8)	1024	80	1280	19.2	649044	61030	999	323	~ 99.61	0.21	7
$k = 16$ Fat-Tree (1)	1024	320	3072	16	630998	15897	999	303	~ 98.42	0.30	5
VL2(16, 16, 14)	896	88	384	16	261266	42906	1000	673	~ 97.90	0.15	4
VL2(8, 8, 64)	1024	28	612	~ 41.1	423752	161499	1000	799	~ 99.45	0.19	11
VL2(16, 16, 16)	1024	88	1152	~ 17.5	276575	56040	1000	648	~ 98.39	0.18	4
Group 2											
DCell(32, 1)	1056	33	1584	~ 2.91	63787	4893	1000	113	~ 97.23	0.09	2
DCell(5, 2)	930	186	1860	~ 3.33	11995	5716	994	642	~ 87.84	0.19	2
BCube(32, 1)	1024	64	2048	~ 3.77	377	3734	999	329	~ 86.04	0.19	2
BCube(10, 2)	1000	300	3000	~ 4.62	10683	4153	998	653	~ 80.85	0.25	2
BCube(6, 3)	1296	864	5184	4.8	7852	5184	991	831	~ 83.18	0.49	4

Table 3.6: Comparison of the behavior of MINNIE for different families of topologies with around 1000 servers each. For the Fat-Tree topologies, we tweak the number of clients per server to obtain 1024 "servers".

Compressing with MINNIE. The third part of the table represents the effect of using MINNIE on the number of rules, average compression ratio and computation time. **MINNIE succeeds to route the traffic on all the topologies without exceeding the limit of 1000 rules per device** (maximum number of rules between 989 and 1000).

We also observe that with 1000 servers **MINNIE allows to attain an average compression ratio higher than 80%**. This shows that considering the state of the forwarding table when routing increases the compression done by the wildcard rules. Compared with the *Compression at the end* scenario, we see a ratio increase between 20% and 30% for the Fat-Tree and VL2 topologies, and a smaller increase between 5 and 10% for BCube. This difference comes from the smaller amount of shortest path available in BCube compared to the Group 1 topologies. DCell topologies display close to no differences since flows were already highly aggregated in the other scenario.

As for the computation time we notice that **MINNIE dynamically computes the route with a sub-millisecond delays** as the maximum average routing computation time is $0.49\,\mathrm{ms}$ for BCube(6,3). And finally, we can observe that **compressing the rules with MINNIE will cost less than** $13\,\mathrm{ms}$ **delay in all topologies**.

3.6.2.4 Comparison with XPath

We compare MINNIE with another compression method of the literature, XPath [HCW+15]. XPath combines re-labeling and aggregation of paths. Each path is assigned an ID. Two paths can share the same ID if they are either convergent or disjoint but not if they are divergent. The assignment of IDs is then based on prefix aggregation. This method requires that, for every request in the data center, an application contacts the controller to acquire the corresponding ID of the path to its destination.

DCNs	Number of rules	
	XPath	MINNIE
BCube(4, 2)	108	56
BCube(8, 2)	522	443

(a) Comparison with MINNIE for paths between servers

DCNs	Number of rules		
	ToR to ToR		Server to Server
	XPath	MINNIE	MINNIE
$k = 8$ Fat-Tree	116	27	272
$k = 16$ Fat-Tree	968	116	6351
$k = 32$ Fat-Tree	7952	482	113040
$k = 64$ Fat-Tree	64544	1925	-
VL2(20, 8, 40)	310	135	138354
VL2(40, 16, 60)	2820	1252	-
VL2(80, 64, 80)	49640	22957	-

(b) Comparison with MINNIE: for paths between servers and paths between level 1 switches

Table 3.7: Comparison of the maximum number of rules on a switch between XPath and MINNIE (between servers or ToRs).

In Table 3.7, we compare the maximum number of rules installed on a forwarding device between XPath and MINNIE for a larger variety of topologies. Numbers reported in the table for XPath are directly extracted from [HCW+15]. In MINNIE, we consider all the flows between servers even if they act only as end hosts but in XPath, only the path between ToRs are considered for the standard architecture (VL2, Fat-Tree). So for an accurate comparison, we apply the same principle to MINNIE by only considering flows between ToRs. Since in [HCW+15], they also consider a bigger table size of 144000 entries, the limit is set to 144000 for MINNIE too. MINNIE requires a lower number of rules to be installed than XPath on every architecture while both dealing with all possible (source, destination) flows. This can be explained by the fact that XPath installs rules for all possible paths for every source/destination pair before compressing while MINNIE only considers one path per flow.

3.7 Discussion

The results obtained in sections 3.5.5 and 3.6.2 via experimentation and simulation respectively demonstrate the feasibility, efficiency, and scalability of MINNIE. We discuss here several practical points and possible extensions of our algorithm.

Dealing with different workloads. We have used an all-to-all traffic pattern, which constitutes a worst case in terms of traffic workload that an application could possibly generate in the network. This was however achieved with 16 IPs per server in the experimentation part and 1 IP per server in the simulation part, which might seem fairly limited. However, in an operational network deployment, it is reasonable to admit that SDN rules are mainly installed on an IP subnet basis, while flow-based rules (created with the matching of all or several fields of the OpenFlow standard) might be rarely employed. Our results can thus be interpreted as routing all-to-all traffic between several IP subnets per server, as one expects to observe in a typical data center where virtualization is used. This means that MINNIE is able to deal with a worst case traffic scenario involving a large number of end hosts.

Rule deletion. All scenarios studied in this work considered flows with unlimited lifetime in order to obtain a worst case scenario regarding the total number of rules involved. However, in practice, flows are active for a limited amount of time as they come and go. We discuss here a possible extension of MINNIE that would handle the departure of flows.

OpenFlow enables the use of idle or hard timeouts to remove rules if no more packets are seen (idle) or after a fixed time interval (hard). Timeouts could be set on the level-0 switches, allowing the detection of inactive flows by MINNIE. Hard timeouts enable the controller to know the exact state of each level-0 switch without any feedback from the switch. With idle timeouts, the controller can specify (in OpenFlow) when a rule is inserted, that the switch must notify the controller when the rule expires. With the exact information of the currently active rules, MINNIE, which keeps an uncompressed version of all the rules in all switches, can delete any unused aggregated rules. As more and more rules are removed, the compression module could also be called to produce a smaller table to insert in place of the current one.

Impact of compression on rule update. We discussed the impact of rule compression on the performance of rule update in several parts of the paper. We summarize here the findings. We have 3 cases of rule update in the compressed tables:

- *Addition of a new simple rule (assuming the table sizes are below the compression threshold).* This event is due to the arrival of a new flow. In this case, there is no impact of compression on rule update. Note that, thanks to aggregated rules, a new flow arrival will require a new entry at the level-0 OVS switches, but might require no new entries at the access switches or higher switch levels, if the new flow is routed by already existing aggregated rules. In this case, we do not have to update the routing table.

- *Deletion of a rule.* This is done in particular when a flow finishes. This operation is discussed above and was not tested yet. However, the controller knows which flow uses which rule (simple or aggregated), and thus may easily know which rule to delete (or not) when an entry expires at the level-0 OVS switches, which is a quick operation.

- *Compression event.* If a table is full, we compress the table totally and we send the new compressed table to the corresponding switch. We then update the switch table by doing, first a delete operation to remove the old table, and then, we send the new rules to be inserted in the fewest number of packets[5]. We measured experimentally the duration of these operations and tested its impact on delay and packets losses. We first evaluated the time needed to carry out a compression event (compression duration, time duration to send a new table to the switch, and updates duration). We show that this time period is in the order of a few ms, as presented in Figure 3.5. Recall that, if a compression event is needed when a new flow arrives, we first send the forwarding rules for the new flow, and then we compress the routing table. Thus avoiding additional delay for a new flow due to a compression event. We also evaluated the impact of rule compression on the network thanks to our experiments. We report packet delay and loss rates in our experiments and compare scenarios with compression and without compression. We show that even with high load ($1\,\mathrm{Gbps}$) for the High Load Scenario (HLS), the loss rate and the delay are not impacted, see e.g. Figure 3.14.

Dynamic compression limit. Early compression helps maintaining the routing tables small. However, the threshold should not be set smaller than the actual compressed table size, as exemplified by the case of compressing at 500 entries in the experimentation part. To work around this potential issue and reap the full benefit of compression we advise to set a dynamic compression limit. We can start for example from a low limit (for example 100 rules) and once a certain percentage of our limit is reached (for example 80%), to trigger MINNIE to compress the routing table. This compression limit is then increased whenever the resulting compressed table is higher than the actual limit, e.g., to 150% of the current compressed table size.

Dealing with burstiness of traffic. A dimension that we have not explored during our tests is the burstiness of arrival of flows that could lead to stress the switch-controller communication, and hit the limit of a few hundreds events/s that the switch is able to sustain. This could be the case of an application that generates a lot of requests towards a large set of servers at high rate. In this situation, MINNIE could help alleviating the load on the controller. Indeed, the sooner one compresses the flow table, the more likely we are to install rules that will prevent the switch from querying the controller for a rule for every new connection. One could argue that compressing entails complete modification of the flow table at the switch, i.e. a large number of events (deletion, insertion) related to the management of the table. However, in OpenFlow, those events can be grouped together: all insertions can be sent at once to the switch. **In summary, MINNIE should also help alleviating the stress of the switch-controller channel in case of flash-crowds of new connections.**

Security. Eventually, note that MINNIE does not alter the security level of the SDN network. Indeed, rules are not compressed in the level-0 switches that connect the VMs to the network.

[5]We have observed that several flow_mod operations are encapsulated in only a few TCP packets

This means that there is no possibility for a packet that belongs to one tenant to be seen or to be inserted in the network of another tenant, provided that the SDN rules at the edge are correctly written. Compressing at the edge could indeed give the opportunity to the traffic of one tenant to enter another tenant's network thanks to some wildcard effect. Note however, that we do not compress at the edge not because of any security concern, but to prevent any misbehavior in the routing process.

3.8 Conclusion

In this chapter, we introduced our solution MINNIE, which aims at providing SDN device's flow scalability by compressing the routing tables. MINNIE routes flows while respecting link and SDN routing table capacity constraints, it also compresses the routing table using table compression with aggregation by source, destination and by default rule. We validated the functionality of MINNIE using an experimental testbed and proved that it can provide an average of 80% of table compression ratio with no negative impact on the end-user traffic. We then showed, through simulation, the efficiency and the scalability of our solution. At the end of this chapter, we discussed the possible extensions of this work and its adaptability to various traffic patterns.

3.9 Publications

- **International Conferences**
  - M.Rifai, N.Huin, C.Caillouet, F.Giroire, D.Lopez, J.Moulierac ,G.Urvoy-Keller "Too many SDN rules? Compress them with Minnie", IEEE Globecom 2015.
- **National Conferences**
  - Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frédéric Giroire, Joanna Moulierac, et al.. MINNIE : enfin un monde SDN sans (trop de) règles. ALGOTEL 2016 - 18èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications, May 2016, Bayonne, France.
- **Journal**
  - Rifai, M., Huin, N., Caillouet, C., Giroire, F., Moulierac, J., Pacheco, D. L., & Urvoy-Keller, G. (2017). Minnie: An SDN world with few compressed forwarding rules. Computer Networks, 121, 185-207.

Chapter 4

Performance

Contents

In the previous chapter, we tackled the problem of flow scalability in SDN devices. In this chapter, we move on to enhance the flow performance in the network using SDN technology by decreasing the end-to-end delay of flows, and the number of packets or connections lost when link failures occur. In this chapter, we leverage the centrality of the controller explained in Section 4.1 to enhance the flow performance and the link failure resilience. We first focus on enhancing the end-to-end delay of short flows in the network by providing coarse grained-scheduling solutions that leverage the controller centrality and the switch flow statistics information to distinguish long flows from short flows in the network and then prioritize short flows. We propose in Section 4.2 two main scheduling methodologies: (i) a state-full and (ii) a scalable scheduler. We then study their efficiency, scalability and limitations.

In Section 4.3, we benefit from the lessons learnt when creating coarse grained scheduling solutions, and aim at decreasing the number of packets lost when a network failure occur in a non-SDN network segment. Our solution, called Providing Resilient Path in Hybrid SDN Networks (PRoPHYS), leverages the same characteristics as the coarse grained scheduling solutions: (i) the port statistics information of the SDN nodes, (ii) the centralization of the control plane at the controller and (iii) its capability of building a fully centralized network topology. PRoPHYS tries to detect network failures in less than 25ms. PRoPHYS, then, reroutes the network traffic according to the predicted failed network segment to decrease the number of packets lost.

4.1 Control Plane Centrality

Both coarse-grained scheduling and PRoPHYS solutions leverage the centrality of the control plane, its characteristics, and its capabilities to enhance the end-to-end delay of flows and decrease the link failure detection interval. In SDN, the controller communicates with all of the SDN nodes and can request the switch features, state and statistics. Based on the controller statistics request message, the switch can provide statistics information per port, flow table, flow or queue. Based on Openflow and OvS documentation [The12, PPK+15], statistics information pulling interval can be set to any value, nonetheless, the advised value for flow statistics information is 1s. However, the coarse grained scheduler tests have proved that pulling flow statistics below 1s pulling interval results in inaccurate statistics information and can increase the CPU overhead. Thus, first in our coarse-grained scheduling solution we leveraged the flow statistics. Then, we leveraged the port statistics information in both coarse-grained scheduling scalable solution and PRoPHYS.

In both solutions explained below, we set the statistics pulling interval– or monitoring interval as we will call it in the rest of this chapter– to a minimum of 10ms. The proposed 10ms minimum pulling threshold reflects the fact that, generally, SDN nodes support 200 events/s. Hence with this threshold, fetching port statistics requires $max(100)$ events/s, which is feasible [KREV+15] as the controller transmits a single statistics request event per switch.

After receiving the statistics information, in both solutions, the controller compares the transmitted and received statistics across ports to detect congestion or link failures. Then, to apply the new scheduling scheme or reroute the traffic, the controller transmits `flow_mod` event containing the new flow rule to be installed in the switch.

4.2 Coarse-grained Scheduling

In current networks, short flows (mice) constitute the majority of the network flows; however, they are hindered by the minority of long flows (elephants) that tend to use most of the network resources, degrading short flows' end-to-end performance [ETHG05]. Indeed, while various variants of TCP, e.g. Cubic [HRX08], have been devised to improve the performance of long flows

in typical high bandwidth-delay product environments like data centers, short TCP flows remain vulnerable and likely to time-out.

Our objective in this work is to demonstrate that despite the limited toolbox offered by SDN to directly manipulate the data plane, one can however implement some form of coarse grained scheduling with legacy SDN equipments. Our focus is on size-based scheduling [AABN04, RBU05], where priority is given to flows in their early stage. This approach is valuable in backbone and data centers networks where the bulk of traffic is carried by TCP, and consists of a majority of short flows.

Our approach takes advantage of the feedback loop offered in SDN where a switch exposes to the controller per rule statistics. The latter enables us to identify long flows and separate them from short flows using multiple queues per port that serve to implement 802.1p QoS mechanisms. Our objective is to minimize flow completion of the majority of flows.

We demonstrate the feasibility of building an SDN size-based scheduler using OvS switches with Beacon controller. We further propose a scalable version of our scheduler to avoid continuous monitoring of each active flow by the switch and the controller.

4.2.1 Related Work

Multiple research efforts have been conducted in order to decrease short flows delay with a limited impact on long flows performance such as [MQU+13, AKE+12, GYG12, JAG+14, XBNJ13, HCG12, ZIA+15, HUKDB10]. These efforts consist mainly of creating new protocols, architectures, algorithms, changing the host or the network devices. For example, the authors of [MQU+13, HCG12] propose to use new scheduling protocols mainly created to enhance flow completion time in data centers. The authors of [MQU+13] propose a new data center transport protocol called L^2DCT that decreases the flow completion time by 50% for DCTCP and 90% for TCP by approximating the Least Attained Service (LAS) scheduling discipline. In [HCG12], the authors propose to use Preemptive Distributed Quick (PDQ) flow scheduling protocol that uses a similar algorithm to shortest job first algorithm where short flows are prioritized, however, PDQ pauses large flows. On the other hand, in [AKE+12] the authors propose to use High bandwidth and Ultra-Low Latency (HULL) architecture that enforces a utilization less than link capacity by provoking phantom queues signal that allow to sacrifice almost 10% of bandwidth utilization at the sake of a factor of 10 to 40 decrease in average and tail latency. In [JAG+14], the authors propose to use tiny packet programs to provide low latency and increase network visibility for the end user.

Other solutions resolved to enhance the short flow network delay in data center but only for TCP flows such as [GYG12, ZIA+15]. The authors in [GYG12] propose to improve the performance of TCP in SDN networks, albeit at the cost of modifying the end-hosts protocol stack. As for [ZIA+15], the authors propose FastLane, a solution which provides an in-network drop notification mechanism to force end hosts to decrease their traffic transmission bandwidth

and hence throttle transmission rates. Fastlane thus decreases the flow completion time of short flows by 81%. However, FastLane requires in-network devices to communicate with the end hosts to notify them of packet drops which limits the solution prospect to TCP flows only that can retransmit lost packets.

As for SDN, a lot of efforts have been conducted to enhance the network performance by leveraging the control plane. In contrast, the data plane remains opaque and cannot be directly influenced by the controller. Few works have addressed the extension of SDN to the data-plane with scheduling in mind. A noticeable exception is [SWSB13], where the authors first advocate the need to use different scheduling or buffer management solutions in different scenarios as there is no one-fit-all solution. They exemplify the problem with different workload scenarios mixing bulk and interactive traffic for cellular or wired access network scenarios, showing that each scenario requires a different combination of buffer management/scheduling solution, e.g., FQ with Codel [THJ16] or FIFO with drop tail. They propose next an extension of Openflow with data plane primitives that they implement via a programmable FPGA.

To decrease network delay in SDN networks, most studies aimed at using the centrality feature of the controller and the capability to monitor the network traffic. For instance, in [THW+13], the authors advise to divert some flows on longer rarely used paths to decrease completion time. However, this solution might cause packet reordering problems at the end hosts. In [BCW+15], the authors rely on the capacity of (non SDN) commodity switches to offer several queues with a tagging process performed at the end hosts, complemented with Explicit Congestion Notification (ECN) in the network. However, tagging the packets could lead to packet processing using the slow path in multiple flow tables as tagging requires to use the CPU. The authors of [TP13] present a flow scheduling algorithm called Baatdaat that depends on the forwarding device reports of network utilization and then uses the spare data center network capacity to mitigate performance degradation of heavily used links. Their algorithm provides a decrease of link utilization by 18% while also decreasing flow completion time by 41%. In [CKY11], the authors propose Mahout where hosts will observe their socket buffers and will then try to detect the elephant flows. Upon detection of elephant flows, the end hosts will transmit this information to the Openflow controller using an in-band message. This solution, however, requires to insert modifications at the end host.

All of the previously stated solutions require the modification of either the network architecture, protocols or end hosts. In contrast, our solution is a purely network centric approach, that does not require any modification of the end-hosts. It schedules network traffic independently of the transport protocol used and is suitable for SDN networks. Similar to Baatdaat, our solution leverages SDN capability to monitor the network. However, our solution aims at scheduling flows within a link in case backup links are not available– or they are also highly utilized– and hence link utilization can not be reduced.

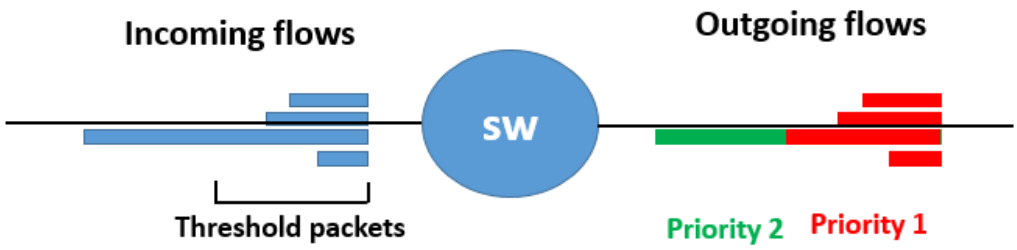


Figure 4.1: State-full scheduler mode of action.

4.2.2 Scheduling Methodologies

As explained in Section 4.1, our solution, leverages this centrality feature of the controller and its capability to perform online network traffic monitoring and adapt the network flow actions accordingly. To provide enhanced flow completion time, our solution attempts to use SDN forwarding devices queues to segregate the flows within the data traffic in it. While in legacy switches, the SDN packets are scheduled according to the Type of Service (ToS) field available in the IP header, in our solution, we queue packets per flow based on their source and destination addresses (MAC, IP or TCP/UDP port number).

Our solution requires a minimum of two queues per port (802.1p mandates 8 queues per port) the highest priority queue is for short and new flows, while the lower priority queue is for long flows. Both, hardware switches (e.g., HP, Pica8, CISCO) and software switches (e.g., OvS) typically propose several queue management schemes like priority queuing or some sort of token bucket. Since our basic scheduler consists of only two queues, we assume that those queues are managed with a strict priority scheduler where the high priority queue is served as long as it has packets– the other queue is served when the first one is empty. In this case, the short flows' packets are always prioritized over long flows' packets. In our proposition, all packets of new flows are treated as high priority traffic. Then, when the size of a flow reaches a given threshold, its packets are processed with a low priority policy.

Our first approach to decrease short flows completion time consisted of creating the **state-full scheduler**. This scheduler is a networking module that runs on the controller and communicates with the forwarding devices. It will first allow all new network flows to use the high priority queue ($e.g.$ Priority 1 in Figure 4.1). Then, it will monitor the flows on the SDN switches or forwarding devices every T_{monitor} interval. If a flow has exceeded $T_{\text{threshold\_pkts}}$ of transmitted packets, the scheduler will then require to modify the queue used by this flow from the highest priority queue to the lowest priority queue ($e.g.$ Priority 1 to Priority 2 in Figure 4.1). This solution allows to detect and react rapidly when a long flow appears. However, with the increase in the number of flows traversing a data center, this solution will not scale as it requires to monitor all the flows and could hence impact negatively the SDN switch performance.

Hence, we devised the **scalable scheduler** as continuous monitoring of each active flow is

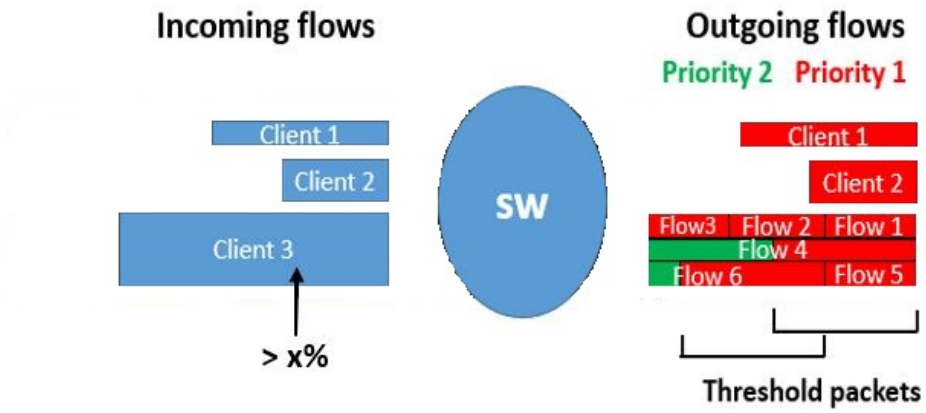
resource consuming, and installing per-flow rules could quickly overload the forwarding table of SDN devices. Also, when the load on the port is low, say lower than 50%, all schedulers typically offer the same performance as queues do not build up. There is thus no need to monitor the individual flows continuously. To address these concerns, we propose the scalable scheduler where the controller initially sets up one default rule for a set of flows (the default rule could aggregate a client's traffic or a specific service traffic)– see Figure 4.2. In this solution, we assume that in a data center the administrator usually specifies a certain percentage of resource utilization allowed per client or service etc, in order to provide traffic engineering.

Based on this, the scalable scheduler will work as follows. Similar to the state-full scheduler all new flows are considered as high priority traffic. Then, the scalable scheduler, will monitor the general routing rules (i.e. client/service forwarding rule) of the switches and for every switch it will calculate the total bandwidth used per client/service based on the amount of traffic sent during the last period of monitoring interval. If the bandwidth used for this client/service exceeds a certain threshold ($T_{\text{threshold\_utilization}} = X\% \times LinkCapacity$) of allowed link utilization of the service, the scheduler zooms in the client/service traffic to identify the large flows (Figure 4.2a). Assuming that large flows are responsible for load increase is a reasonable assumption in typical backbone and data center traffic. To zoom in the client/service traffic, the scheduler uninstalls the client/service forwarding rule triggering the state-full scheduler mechanism and installs per flow rule on traffic demand of the client/service. Afterwards, the scalable scheduler adopts the behavior of the state-full scheduler for this client/service and will monitor every T_{monitor} the flow size. If it is more than $T_{\text{threshold\_pkts}}$ packets it will modify the queue used by the flow to the lower priority queue. After a few T_{monitor} cycles, large flows are isolated and the general forwarding rule is reinstalled with a high priority (Figure 4.2b) while the long flow uses the lower priority queue. The scalable scheduler allows scalability regarding the number of flows to monitor, and reduces the number of events the scheduler requires from the forwarding devices. However, this comes at the cost of increased long flow detection delay.

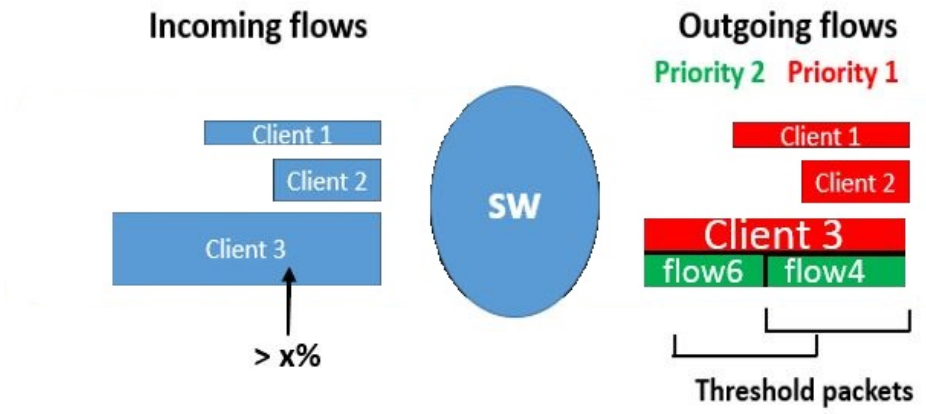
4.2.3 Results

To illustrate our approach, we have developed a prototype and considered a basic experimental set-up described in Figure 4.3. We have ten clients and one server that acts as a sink for traffic and an OvS switch connected to a Beacon controller. The traffic workload is generated using Impt [IMP] and consists of bulk TCP transfers. The distribution of flow size follows a bounded Zipf distribution with a flow size between 15 KB (10 packets) and 10 MB. The average flow size is around 100 packets, in line with typical average flow size in the Internet. The Zipf distribution is the discrete equivalent of a Pareto distribution, which is known as a reasonable model of Internet flow size [BAAZ10]. The load is controlled by tuning the flow inter-arrival time, which follows a Poisson process.

For both schedulers, we monitor the switch flows every $T_{\text{monitor}} = 10ms$ which is equal to the minimum statistics pulling interval advised in Section 4.1, and we set the threshold of short



(a) Step 1



(b) Step 2

Figure 4.2: Scalable scheduler mode of action.

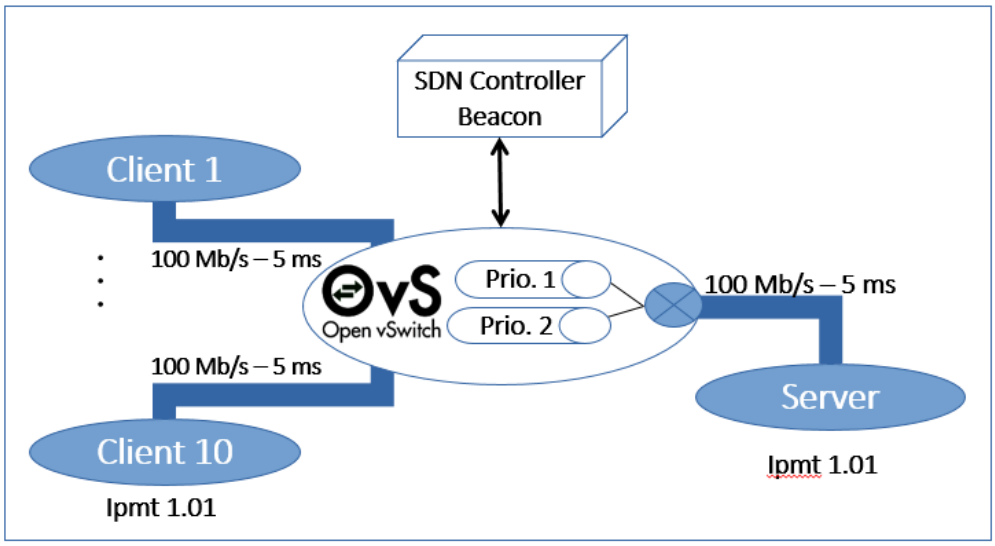


Figure 4.3: Experimental set-up

flows to $T_{\text{threshold\_pkts}} = 100\ packets$ which is the average flow size in the Internet [Dav]. For the scalable scheduler, we set the threshold utilization per client to $T_{\text{threshold\_utilization}} = 10\% \times LinkCapacity$.

Flow Completion Time We present in Figure 4.4 results obtained out of 10 experiments for a load of about 90%. We distinguish between small flows and large flows, where small flows are defined as flows smaller than the 90-th quantile of the flow size distribution. For the Zipf distribution we consider, the 90-th quantile that corresponds to about 95 packets, i.e. close to $T_{\text{threshold\_pkts}}$ used by the scheduler. It is important to note here that small flows represent about 15% of the load.

We observe in Figure 4.4 that the state-full scheduler offers the best response time for small flows and for the majority of large flows as their 100 first packets will take advantage of the scheduling mechanism. Only a minority of the largest flows suffer in the state-full scheduler. As for the scalable scheduler, it offers intermediate results between the state-full scheduler and the absence of scheduler as it needs an additional T_{monitor} to detect large flows and then convert to the state-full scheduler. As we cannot simultaneously obtain better response times for all flows, one can observe the longer tails of response times for the two schedulers as compared to the case with no scheduler. We also notice here, that without a size-based scheduler, most of the short flows experience longer completion time than some of the long flows. However, with the addition of the scheduler (scalable or state-full), we notice that more than 90% of the flows have a shorter response time than any long flow. We believe that the remaining 10% of the flows that have almost the same response time as 3% to 4% of the flows have close flow size (100 packets ± 10 packets). It should be noted here that these results are impacted by the Round Trip Time (RTT) between the controller and the SDN switches, in our case $RTT = 2ms$.

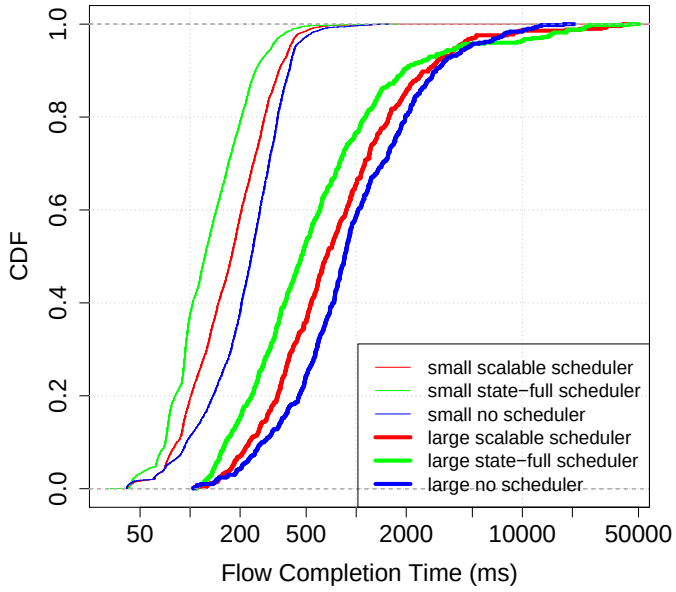
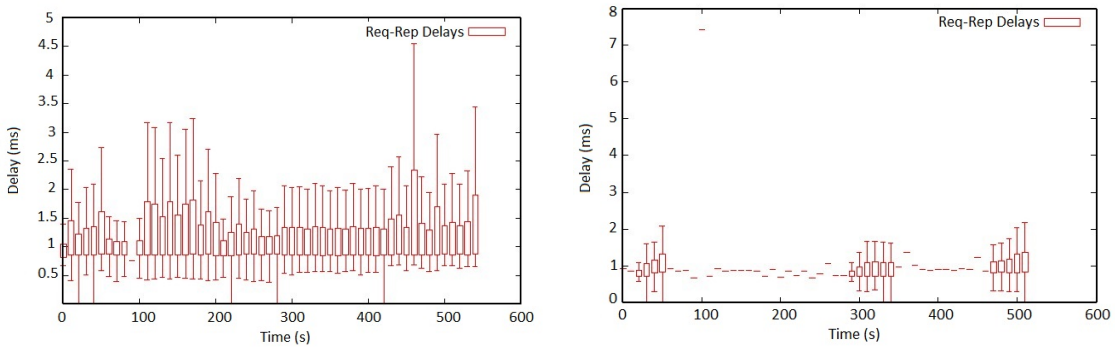


Figure 4.4: Flow completion time CDF for long and short flows

Switch Response Time Both of our schedulers (state-full and scalable) depend on the switch statistics information to detect long flows and change their priority dynamically. Hence, based on the schedulers description, the scheduler reactivity time depends on the monitoring interval set by the administrator and the reactivity of the switch to statistics requests. To study the reactivity of the scheduler we studied the reactivity duration of SDN hardware switches where the slow path might constitute the bottleneck when collecting network statistics (as compared to a virtual switch like the OvS switches in our testbed). We present here results collected, while I was visiting NII in early 2016, of 12 Pica8 hardware switches connected together to form the Sinet4 topology [Top] composed of 74 nodes and 76 edges . We generated an all-to-all network traffic between the end hosts connected to the access switches. We conducted two tests: (i) all-to-all TCP traffic with an aggregated throughput of 1Gbps traffic at the access ports and (ii) all-to-all UDP traffic with an aggregated throughput of 10Mbps at the access ports.

Our results, in Figure 4.5, show that the switch response time for both TCP and UDP traffic remains below 5ms (Figure 4.5a). This response time is even below 2ms for UDP traffic with a 10Mbps access port throughput (Figure 4.5b). This shows that, the request-reply delay is low and though it can be impacted by the network variations its variability remains in the order of 5ms.

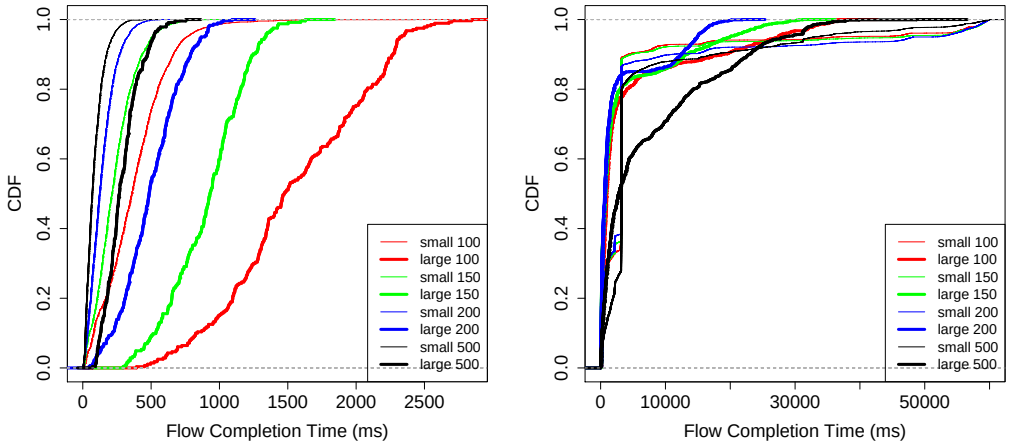
Scalability After studying the performance of both the state-full and scalable schedulers we put the scalable scheduler under test in Mininet [Min] using both star and fat tree topologies with different link bandwidth and link capacity used (100Mbps, 200Mbps up to 500Mbps)- results are reported in Figure 4.6. We notice in Figure 4.6 that the scalable scheduler provides a lower completion time for short flows than for long flows for all bandwidth (i.e. all throughput) used.



(a) TCP traffic with 1Gbps access port

(b) UDP traffic with 10Mbps access port

Figure 4.5: Switch response time.



(a) Star topology 1 node 10 hosts.

(b) Fat tree topology k=4.

Figure 4.6: Flow completion time with respect to bandwidth transmitted.

However, the scalable scheduler fails the promise to provide a short flow completion time for short flows in the fat tree topology. We suspect that the failure might be either due to: (i) the simulation environment when the memory utilization increases due to the increase in the total number of packets queued on all SDN switches in the hosting machine increases or (ii) the scheduling methodology does not take into consideration network heterogeneity and global network utilization. However, this study, was not completed and was interrupted due to my visit to NII lab in Japan where I started working on network resilience.

4.2.4 Scheduler Limited Scope

Though our primary results were promising on star and linear topologies, the results on other topologies, such as fat tree and VL2, would need to be confirmed either with a more powerful

testbed than Mininet. Note that we could not use the Minnie platform as our HP switch did not support OF rules specifying output link queues. A more worrisome problem is that, in our tests, we noticed that the flow statistics openflow database update threshold for some software switches– such as OvS in our case– is limited to 1 second (i.e. the advised value by the manufacturers). Going below this threshold (based on our tests) hindered the performance of the SDN switch and increased its response time. Thus, in this case, our solution is limited to the minimal statistics pulling interval the SDN switch can support without hindering its reactivity. Moreover, as stated in Section 4.1, current SDN switches have a limitation of 200 events/sec, even though, we believe that with the increase development of the SDN forwarding device capabilities, future forwarding device will be able to treat thousands of events/sec and would be able to support higher statistics polling interval. In the next section, we introduce our solution called PRoPHYS that also leverages the centrality of the control plane. However, in PRoPHYS, we leverage the lessons acquired from our work on the schedulers and build a solution capable of estimating link or network segments failure in hybrid SDN networks before the failure is declared by neighboring nodes, using online monitoring without negatively impacting the performance of the SDN switches.

4.3 PRoPHYS: Enhancing Network Resilience using SDN

In the previous section, we presented our coarse grained scheduling solutions that aim at decreasing the short flow completion time leveraging the centrality of the control plane. In this section, we leverage the centrality of the control plane to enhance flow resiliency in hybrid SDN networks, and hence, decrease the protection interval to reduce the number of packets lost when a failure occurs in a network using our solution called PRoPHYS: Providing Resilient Path in Hybrid Software-Defined Networks.

Nowadays, a 50ms protection interval is the norm for service providers [BBU+09] since most applications support the resulting loss with limited impact on the end-user's Quality of Experience (QoE). However, following the *increase* of required *broadband* speed per application [Cis16] especially for time-critical applications (such as video streaming, voice over IP, and virtual reality) the amount of packets lost during those 50ms will also increase, leading to disruptions in the buffer playout process, negatively impacting the QoE. In order to guarantee a good QoE for the end-user, it is essential to improve fault tolerance mechanisms so as to protect these network flows against large-scale packet loss and decrease the total downtime (detection and rerouting time) below the currently acceptable 50ms.

Existing fault detection techniques rely on *link failure detection*, which is achieved by detecting missing session packets. In OSPF, the session packets are called `HELLO` packets [Moy98], and the failure detection process takes a few seconds. The equivalent for SDN networks are Link Layer Discovery Protocol (LLDP) packets [Con02], and the failure detection process takes, by default, 100ms. The main difference between legacy and SDN networks is that in SDN networks, the controller has a global view of the network topology, and communicates with every SDN node.

Thus, the controller can change the routes of all nodes directly, without any delay needed for routing information propagation and convergence. With the help of additional services, such as Multi-protocol Label Switching (MPLS) [BBU+09], Bidirectional Forwarding Detection (BFD) [KW15], or Fast Rerouting (FRR) [SB10] explained in the following section (Section 4.3.1), the link failure detection time can be decreased to around 50ms for both legacy and SDN networks.

Since SDN networks do not need to reconverge upon changes in the control plane, the programmability of the SDN control plane is extremely help-full to methodologies that decrease the failure detection time. Unfortunately, the migration from an existing legacy network infrastructure to a full SDN infrastructure is known to be expensive and cumbersome [Bri14]. In the meantime, so-called *hybrid* networks, where SDN nodes are incrementally introduced in legacy networks, are the most likely scenario.

In this context, we propose PRoPHYS. PRoPHYS leverages the capabilities of the SDN controller in a hybrid network to detect network failure and reroute traffic. It features two modes of action: (i) the passive monitoring of the transmitted and received traffic of SDN nodes (Section 4.3.2), or (ii) the active probing of the paths connecting SDN nodes (Section 4.3.3). These two modes of actions can be deployed independently or together to provide the lowest detection time interval of both methods. Ideally, SDN nodes should be placed such that they can communicate using two disjoint paths. In our work, we focused on the necessary criteria and steps to ensure efficient flow protection between two SDN nodes. However, the placement of SDN nodes in a hybrid network to create disjoint paths is out of the scope of our study, it has been studied in $e.g.$ [HMBM16].

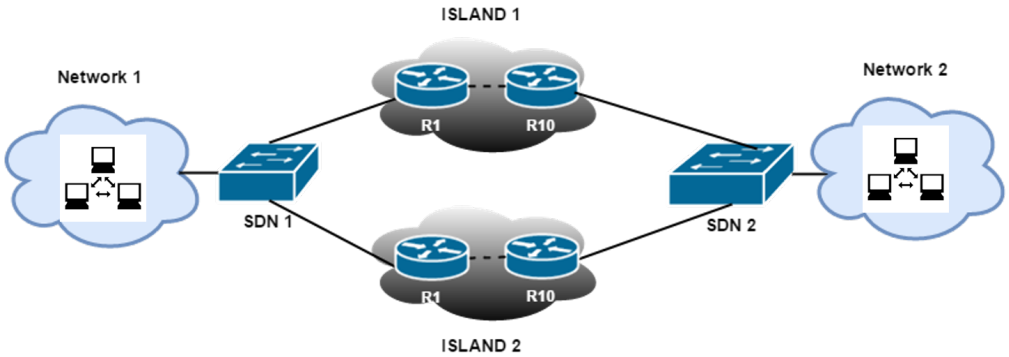


Figure 4.7: Example topology for PRoPHYS.

In its simplest form, using Figure 4.7 for illustration, upon detection of a segment failure within ISLAND 1, PRoPHYS reroutes traffic through the second-best shortest path (ISLAND 2), effectively decreasing the loss rate (Section 4.3.4). Our results (Section 4.3.5), show that the two varieties of failure detection module of PRoPHYS provide a minimal decrease of 50% in total network traffic loss compared to the standard 50ms protection interval while respecting the maximum number of events supported by SDN nodes.

4.3.1 Related Work

4.3.1.1 Hybrid SDN Networks

To allow full network programmability, some studies try to propagate SDN functionalities from within SDN to legacy nodes in hybrid networks. Panopticon [LCS+14] extend the benefits of SDN node in a hybrid network by forcing all the traffic transmitted from any source to any destination to pass through at least a single SDN node to have an abstraction of a logical SDN in a partially upgraded legacy network. However, even with the introduction of SDN nodes in the network, their system is only able to achieve failure detection in a minimum of 1s, as it depends on the Spanning Tree Protocol to detect a failure. Telekinesis [JLX+15] leverages the SDN controller to force legacy L2 switches to use the forwarding tree of the SDN nodes. Both [LCS+14] and [JLX+15] aim at extending SDN programmability to legacy devices, but they do not really tackle the problem of sub-50ms link failure detection, or of rerouting in hybrid L3 networks.

4.3.1.2 Total Downtime and Rerouting

Multiple works were conducted to *decrease the failure detection time*, such as [FFEB05, KW15, Ham15, KW10]. To detect a network failure, they mostly rely on active probing where a legacy/SDN node transmits a state packet to its neighbors to check the viability of the common link [Moy98, KW15, KW10] or the viability of the path between the nodes [KW10]. When this state packet is not received from the neighbor for more than a predefined, fixed $timeout$ duration, the link or path connecting the node to its neighbor is declared down. By default, legacy routing protocols, such as OSPF [Moy98], detect failures after 40 seconds. SDN nodes that use LLDP are able to detect link failures and transmit the information to the controller after around 110 ms. To decrease the default link or path failure detection time, Bidirectional Forwarding Detection (BFD) [KW15] and BFD-multihop [KW10] were proposed. Both of these methodologies also depend on the active link/path probing, and a session is established between the neighboring nodes across a physical link, tunnel or path. BFD allows the detection of link failure in around 40ms [SSC+13]. However, as stated in [Hew16], BFD packets are generated by the forwarding devices themselves, $i.e.$ the legacy nodes and SDN switches. Add to this that the minimal interval between two consecutive BFD transmission on SDN hardware are manufacturer dependent where some switches can support a minimal of 1s minimum interval [Hew16]. Indeed, very low BFD transmission interval increases the CPU overhead of the hardware SDN switch [Hew16]. In PRoPHYS, we provide a maximum of 25ms downtime using active or passive probing without increasing the CPU of the forwarding devices or being limited to the features provided by the SDN switch vendor.

Link failure prediction based on opportunistic scheduling and power signal measurements has been proposed in [Ham15]. The authors mainly measure the electric signal on every port, and then, based on the disruptions of this electrical signal, estimate the probability that a network

failure occurred. This method, however, is limited to the failure detection of directly connected links.

To *decrease the rerouting time*, two main methodologies exist: passive rerouting and active rerouting. Passive rerouting pre-installs backup routes or rules in the network, such as [YLS+14, CXLC15], and provides the shortest rerouting time. In this case, once the link failure has been detected, legacy nodes can use the backup route. For example, the MPLS-FRR [ASP05] extension of Resource Reservation Protocol (RSVP)-Traffic Engineering (TE) for Label Switched Path (LSP) tunnels labels the traffic, and reroutes it using pre-established MPLS tunnels. OpenFlow nodes can use backup rules through the notion of FAST-FAILOVER group rules. The authors of [YLS+14] showed that they can obtain a minimum of 99.7% resilience where 99.7% of the all-to-all connections can be established with k independent link failures. This methodology however, cannot scale for large networks, as it would require the installation of hundreds, or even thousands of backup routes or rules in the limited (and expensive) TCAM memory. Following the same concept, to avoid losses in the case of link failures in hybrid networks, [CXLC15] introduced pre-setup backup tunnels from legacy routers towards SDN routers, and SDN nodes reroute traffic through non damaged paths.

As for active rerouting, the router/controller recomputes the route needed to reach the destination based on the information received from the update messages such as [MLP+16]. RSDN of Recursive SDN [MLP+16] leverages the recursiveness of ISP networks, to create a set of aggregate routers called Logical Crossbars (LXBs). Route computation are done on LXBs for each level, and a summary of the results are sent to the parent LXBs. When a node fails (e.g., node b) on a path (e.g., the path a-b-c-d), the LXBs controller hierarchy will be able to virtualize b's routing table in a. RSDN will then compute recursively the paths to destinations. In SEaMLESS we use active rerouting, with previously-installed tunnels that could be used, but we only instruct the switch to use them when necessary. We chose this technique because passive rerouting cannot scale in ISP networks, due to the enormous number of routes that should be installed to cover all combinations of link failures, which prohibits scalability.

4.3.2 Passive Probing Failure Detection Methodology

The first methodology that PRoPHYS employs to detect the failure of a *segment*– either a link failure within a legacy island, or a link failure between an SDN router and the legacy island– is a passive monitoring of the network traffic at the set of transmitting and receiving SDN nodes. After receiving SDN ports statistics, it searches for discrepancies between the inbound and outbound traffic. Hence, the Passive Probing Failure Detection Methodology follows the following steps:

1. create the matrix of the communicating SDN ports;
2. monitor the ports;
3. upon reception of ports statistics, trigger the failure detection module.

We now detail each step successively in the remainder of this Section.

4.3.2.1 Matrix of Communicating SDN Ports

In SDN, when a new flow arrives in the network, the controller sets up the ad-hoc rules on every SDN switch used by the flow. Thus, the controller has a global view of all the segments used by the flows, and of the pairs of communicating SDN ports. In other words, the controller is capable of building a communication matrix $\mathcal{M}_{\text{ports}}$, indicating whether an SDN switch port is sending traffic to his neighboring SDN port, or not. For example, for the two flows depicted in Figure 4.8, the controller builds the matrix illustrated in Table 4.1. A value of 1 indicates a direct traffic transmission between two ports, a value of 0 indicates no transmission.

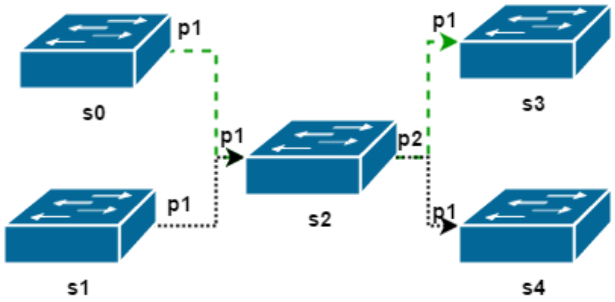


Figure 4.8: Flows passing through the network.

SENDERS \ RECEIVERS	s0 p1	s1 p1	s2 p1	s2 p2	s3 p1	s4 p1
s0 p1	0	0	1	0	0	0
s1 p1	0	0	1	0	0	0
s2 p1	0	0	0	1	0	0
s2 p2	0	0	0	0	1	1
s3 p1	0	0	0	0	0	0
s4 p1	0	0	0	0	0	0

Table 4.1: SDN ports communication matrix $\mathcal{M}_{\text{ports}}$ as built within the SDN controller for the flows depicted in Figure 4.8.

4.3.2.2 SDN Ports Monitoring

To detect traffic loss, ports statistics are fetched from the SDN nodes every few milliseconds. The polling interval could be set to a static value defined by the administrator, or it could be equal to the estimated one-way delay between two SDN nodes. We propose to use the value $\max(10\text{ms}, \text{estimated delay})$ for the scalability and performance reasons explained in Section 4.1. It should be noted that the SDN controller sends only one statistic request per SDN node to get

the statistics for all ports at once. Hence, the number of required events does not increase with the number of ports, allowing scalability.

Even though the granularity of port statistics is larger than the granularity of flow statistics, we chose to monitor ports because (i) monitoring the ports instead of flows enables to scale, and (ii) SDN switches update the flow counters of their OpenFlow database only once every second [HSS$^+$]. Going below this 1s flow counter update threshold increases the CPU consumption of the switch, and the results returned to the controller will be delayed and inaccurate.

4.3.2.3 Failure Detection Module

After receiving the port statistics from all the switches, the failure detection module checks if the outbound traffic on transmitting ports was received on their corresponding receiving ports. For example, if we take the matrix defined in Table 4.1 for the topology in Figure 4.8, the failure detection module will compare $T^t_{\text{s0p1}} + T^t_{\text{s1p1}}$ to $R^{t+\delta t}_{\text{s2p1}}$ and T^t_{s2p2} to $R^{t+\delta t}_{\text{s3p1}} + R^{t+\delta t}_{\text{s4p1}}$, where T^t_i is the traffic sent by port i at a given time t, and $R^{t+\delta t}_j$ is the traffic received by port j after δt. δt is equal to the maximum delay required by the transmitted packets to arrive to the receiving port $e.g.\ \delta t = max(\frac{RTT_{s0p1-s2p1}}{2}, \frac{RTT_{s1p1-s2p1}}{2})$ where $\frac{RTT_{s0p1-s2p1}}{2}$ is the average time needed for transmitted packets from port $s0p1$ to reach port $s2p1$.

However, directly comparing the value of the transmitted traffic and the value of the received traffic δt later ($e.g.\ T^t_{\text{s0p1}} + T^t_{\text{s1p1}} = R^{t+\delta t}_{\text{s2p1}}$) might lead to a large number of false positives since the network link delays vary and taking the maximum delay between the transmitting and receiving delay allows additional packets to arrive to the destination before the maximum delay is reached. Moreover, δt uses the average RTT value, and RTTs could vary when the network is highly loaded $i.e.$ congestion or buffering delay. Thus, in PRoPHYS, we suppose that it is possible that some transmitted traffic could not be received δt later without being lost. We represent this percentage of delayed network traffic as a fraction θ of transmitted packets ($e.g.\ \theta * (T^t_{\text{s0p1}} + T^t_{\text{s1p1}})$).

With all these ideas in mind, to check if all the transmitted traffic from $s0p1$ and $s1p1$ was received by $s2p1$ taking into consideration network jitter, the failure detection module does the comparison represented in Equation 4.1. If the given equation is not validated by the port statistics result, PRoPHYS detects a possible network failure between the transmitting and the receiving ports.

$$(1 - \theta) * (T^t_{\text{s0p1}} + T^t_{\text{s1p1}}) \leq R^{t+\delta t}_{\text{s2p1}} \tag{4.1}$$

However, to declare a failure and reroute the traffic, PRoPHYS makes sure that the disturbance of network traffic is not due to possibly moderate to high congestion rates or possible packets loss in large ISP Autonomous Systems (AS) networks. Thus, PRoPHYS checks the validity of Equation 4.1 each time the required statistics are received, and if Equation 4.1 is not fulfilled for the same set of ports $thresh\_count$ consecutively, PRoPHYS declares a failure in the segment

between these ports. The introduction of the $thresh\_count$ variable should be set in a conservative yet not intrusive way such that it allows to decrease the number of false positives while trying not to lose a huge amount of data traffic when the network failure is actually detected. It should be noted here that using this methodology high congestion periods that would cause a lot of losses frequently could be considered as link failures. However, we believe that mistaking high congestion rates that lead to huge packet losses is acceptable as the traffic would be routed to different segments, reducing congestion and enhancing the performance of the network.

4.3.3 Active Probing Failure Detection Methodology

The second methodology that PRoPHYS leverages to detect a segment failure is a variation to the classical active probing methodology which injects messages in the network. However, a key difference between classical methodologies and our active probing methodology lies in the fact that *PRoPHYS detects link failure when the probing is lost after a timeout that depends on the link or segment delay*.

Basically, the SDN controller transmits a probing packet to the SDN switches with the highest IP Precedence to provide highest priority, the switches then flood the probe to their SDN neighbors across the connected OSPF island. To achieve this, the controller generates a Segment Discrepancy Detection (SDDP) packet for each SDN node. This packet will be transmitted from the controller, flooded through the SDN network segments, and back to the controller (see Figure 4.9). SDDP packet is coined with LLDP packets [SSC+13]. However, SDDP can be transmitted over network segments instead of being limited to the directly connected link. SDDP is also an enhancement compared to LLDP- which features a slow failure detection time of over 100ms- as it uses a dynamic timeout which depends on link/segment delay. Once the SDDP message is sent on the network interface, the controller starts a timeout timer that is set dynamically based on the estimated delay of the link/segment in addition to the communication delay between the controller and the SDN nodes.

To calculate the SDDP $timeout$ threshold for every link/segment, we use an exponential moving average approach. Before the reception of the first SDDP packet, the delay is set to 1s. Then we use equations similar to TCP's Round-Trip Time's [SCPA11] to calculate SDDP $timeout$. The estimated delay of the link/segment, d_l, is updated as depicted in Equation 4.2 where α is set to 0.125. The variation of the link/segment, var_l, is computed following Equation 4.3, and we set the timeout threshold to Equation 4.4 with K=4 (α and K are set to the same values used in TCP timeout).

$$d_l[i] = (1 - \alpha)\ d_l[i-1] + \alpha\ \text{new\_}d_l \tag{4.2}$$

$$\text{var}_l[i] = (1 - \beta)\ \text{var}_l[i-1] + \beta\ |d_l[i-1] - \text{new\_}d_l| \tag{4.3}$$

$$SDDP\_timeout = d_l + K\ \text{var}_l \tag{4.4}$$

Then, similarly to other link failure detection active probing methods, if the message is not received back (by the controller) before the timer expires, the controller declares segment discrepancies. If the message is received *after* the timer has expired, the controller learns that the particular network segment is congested and modifies the timeout value.

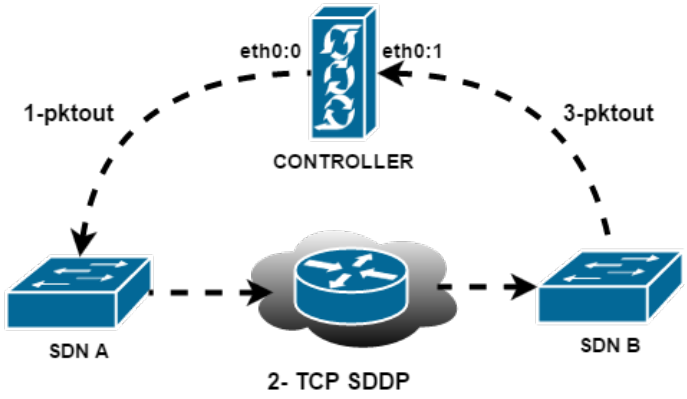


Figure 4.9: $Packet\_out$ transmission over the network.

The closest work to our SDDP packet is BFD Multipath [KW10]. Nonetheless, SDDP differs from BFD Multipath because it does not need a session for every link and does not depend on the architecture of the SDN switch. Indeed, the performance of BFD depends on the manufacturer. For example, the HP5412 switch provides a minimum of 1 second detection interval [Hew16]. Moreover, the use of BFD negatively impacts the performance of routers, because the creation of the transmitted packets and their processing must be done at the general purpose CPU of SDN switches, which has limited capacities in comparison to the CPU capacities of servers [Sch]. Our methodology solely relies on the controller to take decisions, it is manufacturer independent, and it allows to decrease the CPU overhead on the switch.

4.3.4 Rerouting

After detecting a failure in the network, PRoPHYS reroutes the traffic away from the lossy link, to other paths of the network. To select the best path to use, PRoPHYS computes a virtual topology graph by removing all lossy segments $G_{\text{virtual}} = G_{\text{real}} \setminus \{\text{lossy\_segment}\}$ from the initial topology G_{real}. Then, the controller selects the shortest path in the virtual topology, and sends the new rules to be installed on the SDN switches.

However, since we consider hybrid networks, we should also account for the fact that legacy routers, in-between SDN nodes, will keep their outdated view of the routing topology, since their time to detect the failure is longer. Hence, the conflicting views, between the SDN controller and the OSPF nodes, could cause routing loops and denial of service. For example, if we take a topology similar to Figure 4.7, and assume that the number of hops on Island 1 is 3, while the number of hops on Island 2 is 6, the first OSPF node on Island 2 (R1) will send traffic destined to Network 2 through SDN1 (shortest path). Once SDN1 detects failures on Island 1, and PRoPHYS

	Bandwidth(Mbps)	Time(ms)
ovs_bridge (SDN link)	943	8.9
ovs_stt (SDN tunnel)	934	9.1
ovs_gre (SDN tunnel)	919	9.1

Table 4.2: Bandwidth and time of transmission of 1000 MByte of data from a client to a server.

decides to reroute traffic through Island 2, R1 still has its (old) routing table since it did not detect any network updates. It will thus retransmit the traffic back to SDN1, or drop it altogether. Consequently, PRoPHYS will not be able to route the traffic from SDN1 to SDN2 via Island 2. To force OSPF routers to route traffic via the (new) path selected by PRoPHYS, instead of the old lossy path, we use tunnels to route traffic between SDN nodes and the first OSPF/SDN node that has the appropriate routing that allows it to reach the destination (in our example we use the tunnel between SDN1 and R2 in Island2). To know which nodes can route the traffic properly to the destination without any tunnels, we listen to OSPF update packets at the SDN nodes to rebuild the topology structure at the controller.

Using tunnels to reroute traffic has been suggested in [CXLC15] to handle single link failures, and avoid the failure detection period. In PRoPHYS, we leverage this solution and reroute traffic through tunnels, until reaching the closest node with the correct routing to the destination, where the traffic is decapsulated and transmitted to reach the end host.

However, it should be noted that routing data through tunnels can decrease the end-to-end bandwidth attainable by the client application. For instance, Table 4.2 reports the maximum attained throughput, and the delay required to route 1GB of traffic between two machines, through a tunnel active on these machines using either Generic Routing Encapsulation (GRE) [FHMT00], or Stateless Transport Tunneling (STT) [BDG12] deployed over OpenVSwitches-based routers.

The end hosts are connected using 1Gbps Ethernet links. We observe that the bandwidth decreases, and that the delay of transmission increases, once we use tunnels (ovs_gre and ovs_stt) compared to no tunnels (ovs_bridge). Moreover, the choice of the tunneling protocol impacts the results: GRE obtains a lower bandwidth than STT, 919Mbps instead of 934Mbps. Hence, in SEaMLESS we use STT tunnels to reroute the traffic through OSPF islands.

4.3.5 Performance Evaluation

To test the performance and the impact of PRoPHYS on network traffic, we conducted a series of tests using a Floodlight v2 controller [Iza15] connected remotely to a Mininet [Min] network. The test network, depicted in Figure 4.10, consists of 5 SDN switches connected through OSPF islands. We set the delay between the network nodes and between SDN nodes and the controller to 2ms. The 2ms delay was motivated by the average link delay between nodes in a medium sized ISP topology in SNDlib [Zus] such as germany50. The delay was calculated based on the

geographical coordinates of the links provided taking into consideration fiber optic connections. In this network, h0 communicates with h1 and with h3; and h2 communicates with h3. The traffic is generated by iperf3 and is either bidirectional Constant Bit Rate (CBR) UDP traffic or TCP traffic. Five seconds after starting iperf3 on the hosts, we simulate a link failure in the OSPF island connecting the nodes s0 and s1 and observe the impact of PRoPHYS. We repeat each experience 100 times to obtain statistical significance.

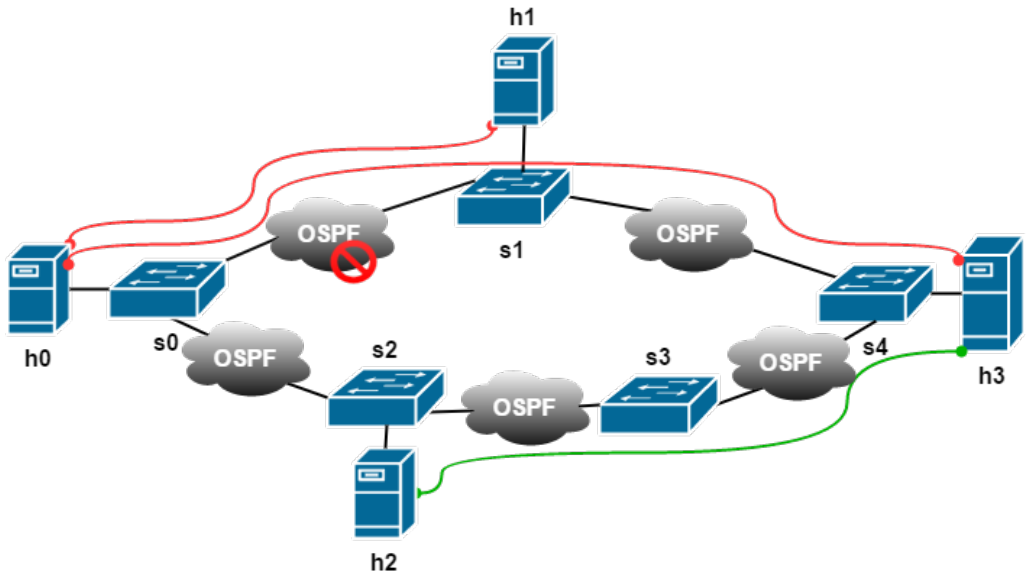


Figure 4.10: The SDN testing network topology in Mininet.

In all of these experiments, we set the segment monitoring interval to 10ms which is equal to the minimal monitoring interval, as discussed in Section 4.3.2. We run two versions of PRoPHYS that differ based on the failure detection technique they use, either Passive Probing (Section 4.3.2) or Active Probing (Section 4.3.3).

For the Passive Probing, we declare segment failure when the following two conditions apply:

- the number of packets lost or not yet arrived (θ) is higher than X% of the traffic transmitted in the last iteration, where $X \in \{25\%, 50\%, 75\%, 90\%\}$.
- the previous condition should be valid $thresh\_count = 2$ consecutive intervals.

For the Active Probing methodology, we declare a failure when the delay between transmitting and receiving the packet on the other interface of the controller is higher than the SDDP timeout delay introduced in Section 4.3.3. For the sake of briefness, in the remainder of this Section, we refer to the Passive Probing methodology as PortStats, and to the Active Probing methodology as PktOut.

METHODOLOGY	BANDWIDTH			
	1Mbps	10Mbps	20Mbps	40Mbps
50ms threshold	5	42	84	168
PktOut	2	16	30	63
PortStats 25%	3	20	35	71
PortStats 50%	5	23	35	83
PortStats 75%	3	32	35	79
PortStats 90%	2	22	43	84

Table 4.3: Maximum number of packets lost.

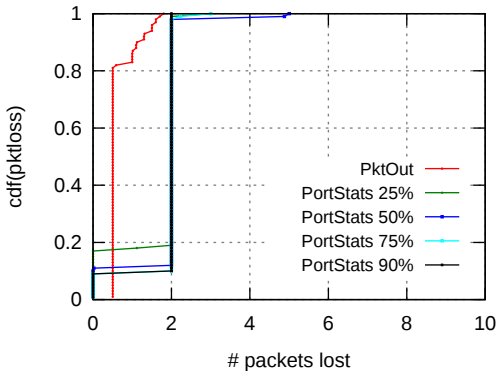
4.3.5.1 Impact on Network Traffic

UDP experiments We assess the performance of PRoPHYS variants first by comparing them with the *reference case* of 50ms. Specifically, we look at the maximum number of packets lost per connection (over the 100 experiments) for different connections bandwidths. The results are shown in Table 4.3, where we notice that the maximum number of packets lost using PRoPHYS, for all speeds, and for both detection methodologies, is 50% less than the reference case, which can be translated by a maximum downtime of 25ms.

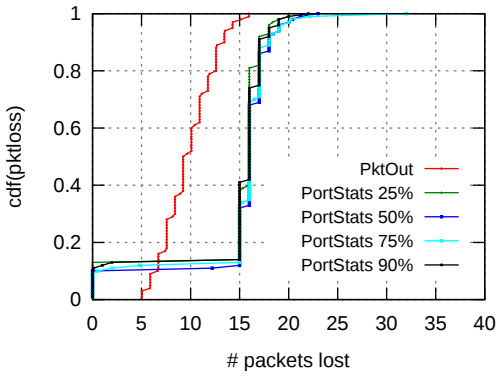
In Figure 4.11, we go beyond the maximum value and plot the Cumulative Distribution Function (CDF)s of the number of packets lost with PRoPHYS, for both detection methods, and each bandwidth. We notice first that as the connection bandwidth increases, PortStats provides better results than PktOut. For example, for 1Mbps (Figure 4.11b), only 10% of the connections obtained less packet lost using PortStats than PktOut. However, as the bandwidth speed increases from 1Mbps to 40Mbps, (see Figures 4.11b-4.11d), PortStats tends to have better performance than PktOut. Specifically, for 40Mbps, 60% of the connections using the PortStats 50%, PortStats 75%, and PortStats 90%, as well as up to 90% of the connections using PortStats 25% feature less lost packets than PktOut. These results are in line with intuition. Indeed, the higher the rate, the less likely PortStats is to make errors. In contrast, the higher the traffic rate, the larger the average delay in Equation 4.2 (and also the variance in Equation 4.3, even if here it has less impact with CBR traffic) computed by PktOut, which increases its detection time.

We also notice in Figure 4.11, that the exact value of `thresh` (i.e., the tolerable percentage of packets lost or not received that ranges between 25 and 90%) for PortStats has little impact on the number of packets lost. This is because even if a lower `thresh` might lead to detect a failure quicker (potentially doing a false positive), it is counterbalanced by the fact that two such consecutive observations must be made (`thresh_count` $= 2$).

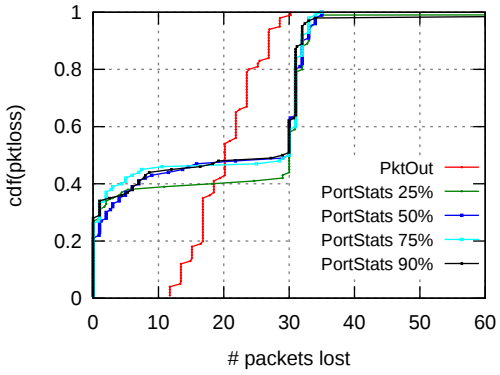
In Figure 4.11, we also notice that, using PortStats, we can obtain 0% packet loss. This is due to false positives detections. Indeed, as shown in Figure 4.12, PortStats tends to have between 1 and 5 false positives in our tests. Some of these false positives can occur before the actual link failure, hence all the traffic would have been rerouted from the original path before the



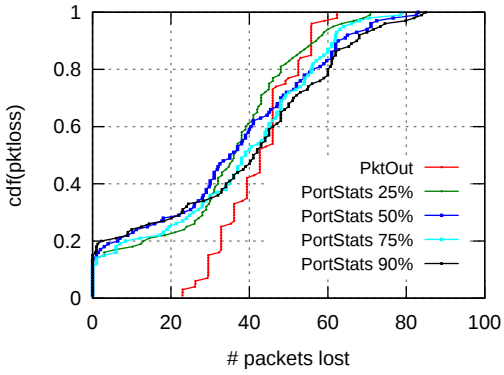
(a) 1Mbps bandwidth



(b) 10Mbps bandwidth

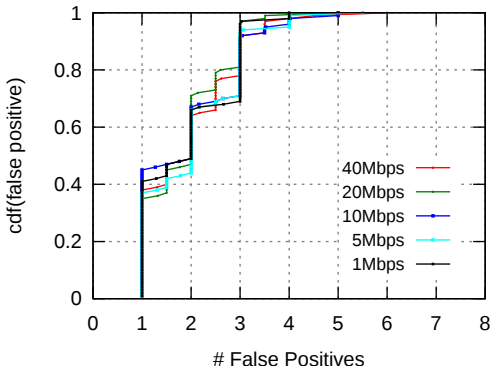


(c) 20Mbps bandwidth

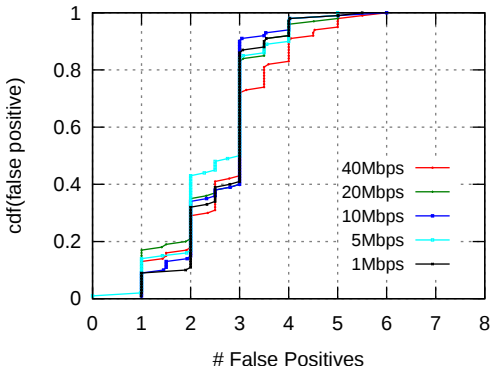


(d) 40Mbps bandwidth

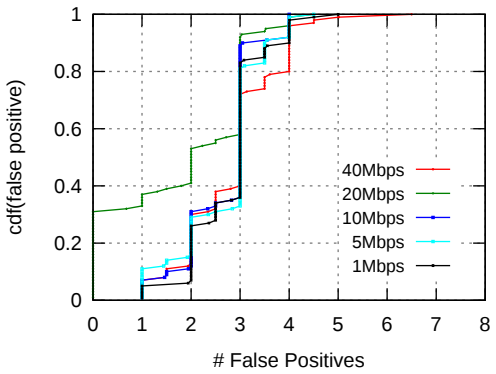
Figure 4.11: Packets loss of connections using the failing link.



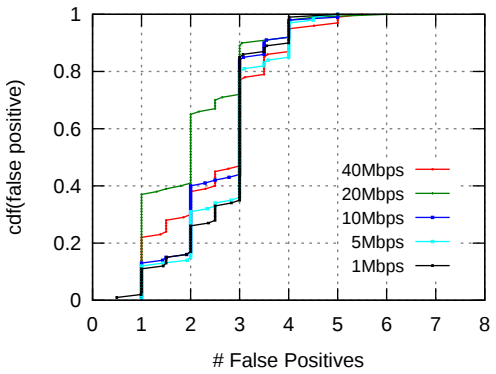
(a) Threshold=25%



(b) Threshold=50%



(c) Threshold=75%



(d) Threshold=90%

Figure 4.12: Number of false positive detections of segment failures with PortStats.

segment actually fails, leading to 0% loss rates. In contrast, in all of our tests PktOut had zero false positives, which proves its stability. From these results, we conclude that PktOut is more stable than PortStats, however, PortStats shows better results as the bandwidth increases.

As a final note on our UDP experiments, it is important to point out that the variability observed in all figures is due to the OvS switches and the PRoPHYS implementation (within Floodlight), rather than in iperf3 or Mininet, since the UDP sources emit CBR traffic and the network is well provisioned and the server on which Mininet is running is far from saturation.

TCP Experiments After studying the impact of PRoPHYS using CBR traffic and having proven its effectiveness in that context, we study the impact of PRoPHYS on TCP traffic. We thus repeat the experiments with the PktOut method, and with the PortStats 50% method for TCP connections as PortStats varieties gave similar results with $thresh\_count = 2$. One UDP connection is replaced by one TCP connection, with an average speed of 10Mbps.

Without PRoPHYS, TCP connections loose connectivity, as they time out before OSPF reconverges. With PRoPHYS, all connections were able to survive the link failure, and the number

of retransmissions was low, as seen in Figure 4.13. In this figure, we see that the number of retransmissions reached a maximum of 25 packets for PortStats 50%, and of 20 packets for PktOut. These values are far below the 42 packets lost that a 50ms recovery time would generate (see Table 4.3).

We observe that in a few percentage of cases, both methods achieve a 0% retransmission rate. This means that both methods do false positives, i.e., traffic was rerouted before the failure actually occurs. Still, the rate of false positives of PktOut remains below the one of PortStats, in line with the observations made for UDP traffic. We however observed that the false positives of PktOut are made in the warm up phase of the tests, when the TCP window opens widely and delay is the most difficult to track. We can expect that with real ISP traffic where the level of multiplexing is high and delay variations smoother, the false positive rate of PktOut should remains small. This observation of false positives for both methods hints towards combining them to minimize the false positive rate. We discuss this option in Section 4.3.6.

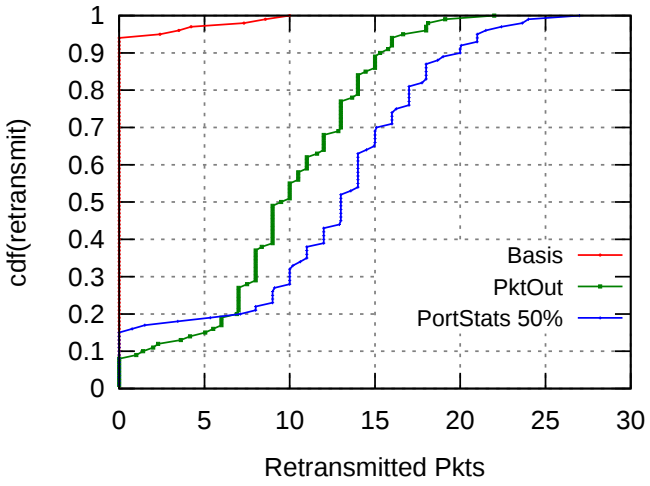
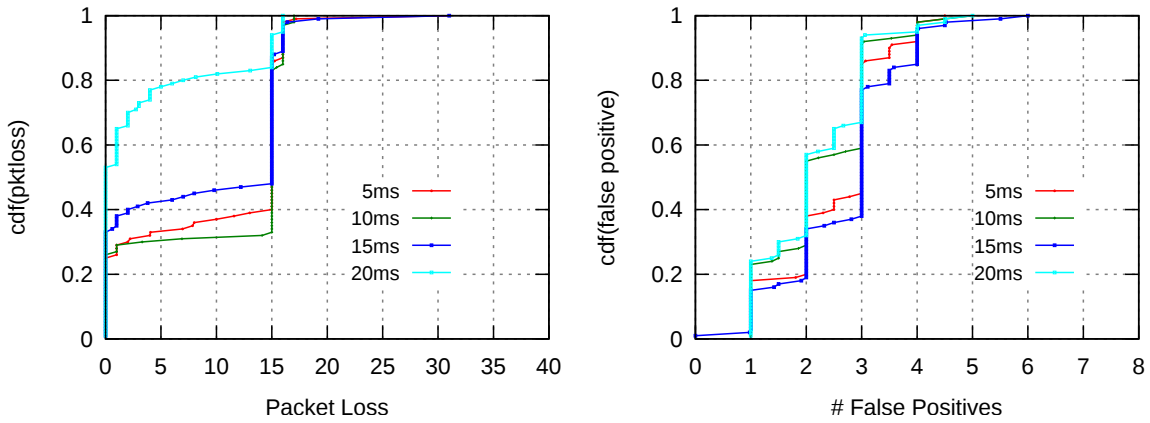


Figure 4.13: Number of packets retransmitted by TCP.

4.3.5.2 Impact of the Segment Delay on PortStats

In order to study the impact of the segment delay on the 10ms monitoring interval that we have set for the previous experiments, we repeat the PortStats 50% UDP experiment with a 10Mbps bandwidth, while varying the segment delay. We set the delays to 5ms, 10ms, 15ms and 20ms. Figure 4.14a shows that the number of packets lost is the lowest when the segment delay is 20ms and `thresh`=50%. This is due to the fact that, with a 20ms segment delay, we tend to receive, after 10ms, exactly 50% of the transmitted traffic. However, the number of packets lost increases slightly as the segment delay decreases. Hence we advise to set the monitoring interval, and the variable `thresh` based on the delay obtained on the segments connecting the SDN nodes. We further discuss this notion in Section 4.3.6.



(a) CDF of packets lost

(b) CDF of false Positives

Figure 4.14: Packet loss and false positive variation with the variation of delay on the failed island using PRoPHYS PortStats 50% methodology

4.3.6 Discussion

From the results obtained in the previous section, we notice that the passive PortStats method outperforms the PktOut method when the connection speed increases, but at the cost of exhibiting false positives. On the other hand, PktOut outperforms PortStats for low connection speeds, and maintains a stable performance. Another difference that we observed between the two methods is that the SDDP packet sent by the controller across a network segment connecting two SDN nodes only enables the measurement of the performance of one shortest path between the two SDN nodes. If multiple shortest paths exist between the two nodes (e.g., in the case of load balancing), PktOut will not be able to test the viability of all existing paths simultaneously, which could increase the detection delay. However, PortStats enables to test all paths used by real traffic simultaneously, i.e., the passive approach measures a wider variety of paths across the legacy islands.

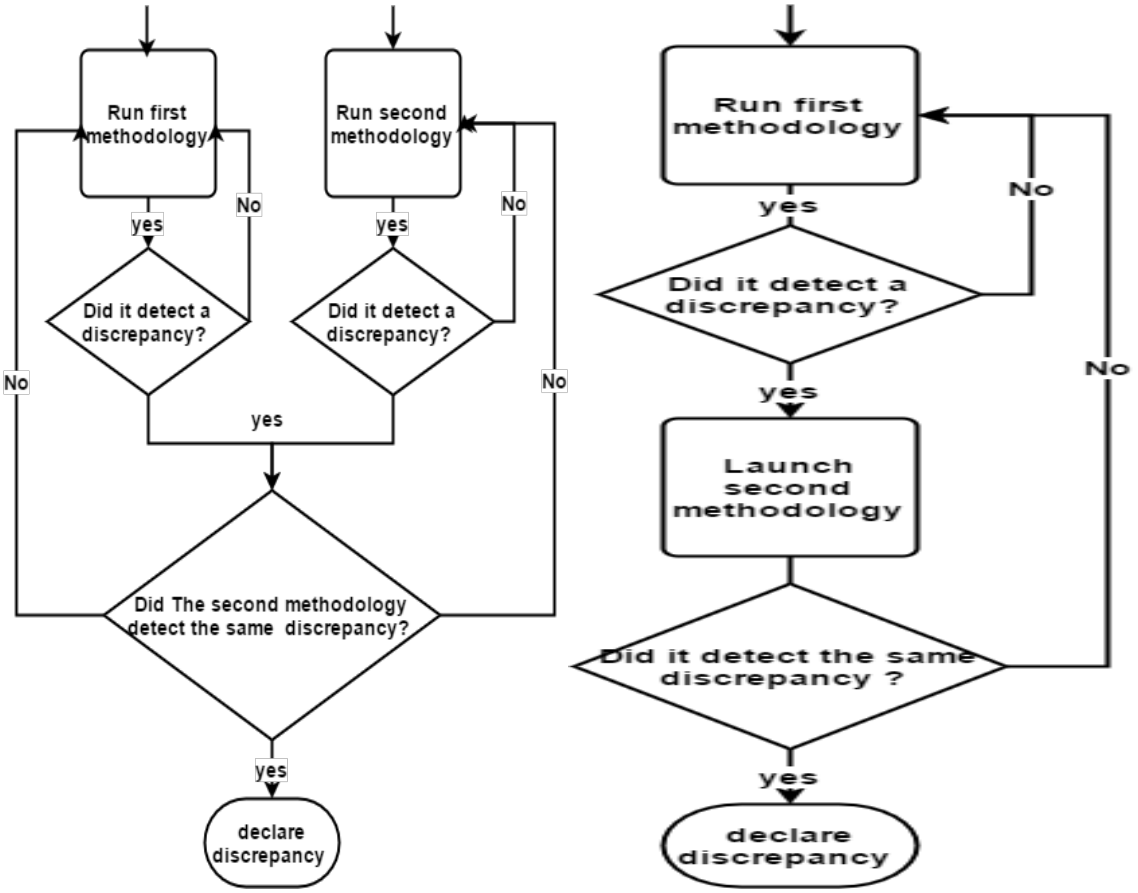
It thus sounds natural to use a combination of both methods. In order to decrease the number of false positives, we can use 3 possible combinations of PortStats and PktOut:

(i) **PortStats and PktOut in parallel**. In this scenario both methods are running in parallel and whenever a method detects failures, it validates/rejects its results by comparing them with the results of the other method. (Figure 4.15a)

(ii) **PortStats then PktOut.** PortStats is used, and when it detects a segment failure, it launches PktOut to validate/refute the result. (Figure 4.15b)

(iii) **PktOut then PortStats.** PktOut is used, and when it detects a segment failure, it launches PortStats to validate/refute the result. (Figure 4.15b)

The parallel approach (case (i)) would allow to validate the detection of segments failure



(a) Method A and B. **(b)** Method A then B.

Figure 4.15: Flowcharts of combination of methodologies in PRoPHYS.

with the minimum detection interval. Moreover, it allows to adapt `thresh`, `thresh_count`, and the monitoring interval variables of PortStats based on the estimated delay information that are obtained by PktOut. The main advantage of one method followed by the other (cases (ii) and (iii)), is the limited number of events sent at the same time by the controller to the switch. Indeed, with a monitoring interval of 10ms, the parallel method induces a higher pressure on the SDN control plane as illustrated in Figure 4.16, where we observe that the parallel approach reaches the 200 event/s limit that a typical hardware SDN can sustain.

4.4 Conclusion

In this chapter, we described in details our solutions to enhance flow performance in SDN and hybrid networks. We first presented our coarse grained scheduling algorithms that leverage the centrality of the control plane and the online trafic monitoring to detect large flows in the network.

We proposed two schedulers, the state-full and the scalable schedulers, which detect large

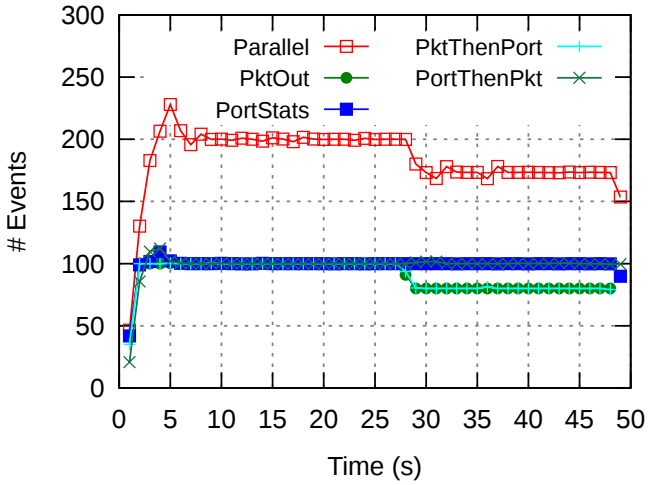


Figure 4.16: Total number of events triggered per second over every switch in the network.

flows and modify their queue to use the lowest priority queue so as to allow short flows to finish quickly. The state-full scheduler monitors all the flows in the network to detect large flows quickly, however, this scheduler does not scale. On the other hand, the scalable scheduler monitor the ports of the switches and then zooms in the port traffic once the port utilization bypasses the predefined threshold. Our primary results, showed that both schedulers enhance the end-to-end delays of short flows in small networks. However, they failed to maintain similar results in fat-tree and VL2 networks.

In the second part of this chapter, we explained in details our solution called PRoPHYS that detects network failures in non-SDN networks and reroutes the traffic away from these non-SDN segments once the failure is detected. PRoPHYS solution can decrease the downtime from 50ms to 25ms in hybrid SDN/OSPF networks. This is achieved by using the SDN switches to quickly detect failures within the SDN and non SDN part (OSPF islands) of the network and also by carefully interacting with OSPF routers during their re-convergence period using tunnels.

We proposed two techniques to detect failures for PRoPHYS, PortStat and PktOut, and evaluated their relative merits in terms of detection time and false positive rate for key scenarios involving UDP or TCP traffic. We also discussed various options to combine these methods. We further highlighted their potential impact on the control plane of OpenFlow, which is the bottleneck of OpenFlow as control messages between SDN switches and the controller are handled in the slow path of the SDN switches.

In the following chapter, we will discuss how SDN nodes can be leveraged in order to increase network's energy efficiency while keeping the network performance intact ($i.e.$ no packet loss resulting from turning off links or nodes).

4.5 Publications

- **Poster**
  - Myriana Rifai, Dino Lopez, Guillaume Urvoy-Keller,"Coarse-grained Scheduling with Software-Defined Networking Switches", ACM Sigcomm 2015.
- **Research Report**
  - Myriana Rifai, Dino Lopez, Quentin Jacquemart, Guillaume Urvoy-Keller "PRoPHYS: Providing Resilient Path in Hybrid Software Defined Networks".

Chapter 5

Energy Efficiency

Contents

Due to the high increase of energy consumption and its negative impact on the global carbon footprint and the environment, in 2016, the European Union decided to cut its energy consumption by 20% by 2020 [Eur]. In 2013, the worldwide energy consumption of Information and Communication Technologies (ICT) was around 1,253 Twh which amounts to 10% of the world's energy consumption [SW]. This energy consumption is increasing yearly and is expected to almost double by 2020 [EA15] due to the increase in Internet traffic, network services and number of connected devices.

As most of the power hungry ICT can be found in the backbone and data center networks, multiple solutions were created to decrease the energy consumption of the: (i) backbone networks notably using *energy aware routing* where some network devices are put in sleep mode e.g. [CMN12], or the network device speed and link capacity is adapted to the traffic load to decrease energy consumption e.g. [CMN09] and (ii) the data centers by applying *energy aware routing* solutions to decrease the energy consumption of the network, server and network virtualization [ENE], VMs energy aware placement algorithms e.g. [KGB13], *server consolidation* e.g. [SKZ08, BB10a] to minimize the number of running servers at any time, or even cooling strategies such as free cooling [KRA12] that uses the external air temperature to cool the air that circulates in the data center. In this thesis, we mainly focused on the limitations of current energy aware routing and server consolidation solutions.

Energy aware routing decreases the energy consumption of the network by forwarding traffic so as to maximize the number of unused network devices that can be shutdown. It can also require a dynamic adaptation of network resources to the network load. However, in legacy networks, operators are reluctant to change network configurations as they are frequently manually set which makes dynamic changes to routing configurations almost impossible. On the other hand, by placing the control plane in a central programmable controller, the SDN paradigm allows the dynamic control of a network. SDN, thus bears the promise of enabling those energy efficient solutions. The problem with existing energy aware routing solutions for backbone networks [CMN12, CMN09, CEL$^+$12] is that energy efficiency comes at the cost of performance degradation especially when sudden traffic peaks or link failures occur.

Server consolidation mainly aims at maximizing the number of virtual instances running on the least possible number of physical hosts, thereby lowering the required number of running servers, and enabling to power off a part of the data center. However, when VMs are *idle*, even though the least possible number of physical servers are used, the power they consume is not necessarily used to do any useful work. Instead, it might be used to maintain the background services of a large number of idle VMs afloat. Unfortunately, these solutions e.g. [dSdF16, PLBMAL15, BAB12] offer limited benefits when active VMs exhibit frequent idle periods.

In this chapter, we present an energy efficient solution called Smooth ENergy Aware Routing (SENAtoR) for hybrid ISP networks (Section 5.2). Then, we represent an energy efficient enterprise data center/cloud architecture SEaMLESS (Section 5.3). SENAtoR is an energy efficient solution that uses SDN nodes to turn off network devices without incurring data loss at any moment even when traffic peaks or network failures occur. SENAtoR leverages three main features to ensure zero traffic loss: (i) tunneling for fast rerouting, (ii) smooth node disabling and (iii) detection of both traffic spikes and link failures. SEaMLESS is an energy efficient architecture, that is still in its infancy but with promising result, which migrates idle virtual machine (VM) to lightweight virtual network function (VNF) so as to empty the RAM and allow enhanced server consolidation.

5.1 Related Work

Reducing the energy consumption in data center and backbone networks has been widely studied. In backbone networks, researchers aimed at using energy efficient devices that can adapt their energy consumption based on the traffic load [CMN09], or using energy aware routing algorithms that can turn off or put in sleep mode unused network devices [CMN12]. As for data centers, researchers opted to obtain energy efficiency by virtualizing networks and servers [ENE], using renewable energy power sources [ZWW11] or cooling strategies such as free cooling [KRA12], using VM energy aware placement algorithms [KGB13], and server consolidation [dSdF16, PLBMAL15, BAB12].

5.1.1 Backbone Networks

Backbone network energy efficiency consists mainly of using energy aware routing, e.g. [CMN09, SLX10, LLW+11], or traffic engineering and power aware network protocols, which in some cases might leverage the benefits of SDN [HSM+10b, HJW+11, HSM+10a, WZV+14, JP12, XSLW13].

Energy aware routing has been studied for several years, see for example [CMN09] for backbone networks, [SLX10] for data center networks, or [DRGF12] for wireless networks. The proposed algorithms allow to save from 30% to 50% of the network energy consumption. However, as stated earlier, they imply to do on the fly routing changes.

After SDN was introduced into the networking community, multiple works proposed and investigated SDN solutions to implement energy aware routing. For instance, in [GMP14], the authors propose algorithms to minimize the energy consumption of routing by shutting down links while taking into account constraints of SDN hardware such as the size of TCAM memory. Authors in [HSM+10b] implemented and analyzed ElasticTree, an energy aware routing solution for data center networks. They showed that saving up to 50% can be achieved while still managing traffic spikes. However, these solutions require a complete migration of the network to the SDN paradigm and are not adapted for ISP hybrid networks.

The most realistic scenario for the introduction of the SDN paradigm is a progressive migration using hybrid networks. As explained in Chapter 1, in hybrid networks, legacy and SDN hardware stand alongside and the difficulty is to make different protocols coexist. Opportunities and research challenges of Hybrid SDN networks are discussed in [VVB14]. Routing efficiently in hybrid networks has been studied in [AKL13] where the authors show how to leverage SDN to improve link utilization, reduce packet losses and delays. We extend this work by considering energy efficiency.

However, turning off SDN devices in hybrid SDN networks, can be interpreted as link or node failures by legacy network devices and might decrease the network ability to drain sudden,

yet not malicious, traffic surges (due, for instance, to exceptional events such as earthquakes). Consequently, our energy-aware solution SENAtoR implements some features to correctly cope with link failures and flash crowds. The network community has addressed such problems, with the help of SDN, as follows:

- **Link Failure Detection and Mitigation.** As in legacy devices, SDN devices can rely on the legacy BFD to detect link failures [KW15]. Once the link failure has been detected, OpenFlow already offers a link failure mitigation through the notion of FAST-FAILOVER group rules, where several rules per flow can be installed. Protection of the link and control channel of OpenFlow requires however more complex solutions, as the one proposed in [SSC+16]. To avoid losses in case of link failures in hybrid networks, [CXLC15] proposes to introduce pre-set tunnels from a legacy router towards an SDN router, which form backup paths. Later, SDN nodes reroute traffic through non damaged paths. We borrow this idea and propose to use pre-set tunnels, which are used when a node is turned down. This is an adaptation and a generalization of the solution proposed in [CXLC15] to handle a link failure. Indeed, we use it for energy efficiency when multiple links are turned off. We also allow tunnels to be set between any (OSPF or SDN) pair of nodes and we carry out practical experimentations to validate the method.

- **Detecting Traffic Variations in SDN Networks.** Traffic variations of backbone networks are usually smooth as the network traffic is an aggregation of multiple flows [RGK+02, IDGM01]. However, abrupt variations happen in case of link failures or flash crowds [LCD04]. Methods have been proposed to detect them in legacy networks, see for example [LCD05, AHM+03]. Netfuse [WZS+13] has been proposed in SDN-based data centers to mitigate the effect of traffic variations where it detects traffic surges and adapt the traffic flow and load to prevent traffic overload. In SENAtoR, we propose a method to detect such abrupt variations in a hybrid SDN network.

5.1.2 Data Center

A data center is composed of three major components: (i) the servers, (ii) the network, and (iii) the cooling systems. Thus, decreasing the energy consumption of any of the three components or all of the components all-together decreases the energy consumption of the data center.

To decrease the energy consumption of the servers, some researchers suggest to use new electronic components to quickly enter a power saving mode when the device is underutilized [CRN+10, VSZ+11, CSB+08, D-L09, GTB+14]. Others suggested to use virtualization techniques such as virtual machine (VM) and containers, and migrate existing servers to VMs or containers using KVM [KVM], Qemu [Qem], Xen [Xen] or Docker [Doc] to allow server consolidation, energy efficient VM placement and relocation.

Server consolidation and energy efficient VM placement and relocation methods allow to decrease or minimize the number of servers used at any time to host all of the needed services as in

average 30% of the servers are unused at a single time [UR10]. Multiple server consolidation and VM placement solutions exist such as [BB10b, dSdF16, PLBMAL15, BAB12]. These solutions take into consideration, in general, the memory, number of CPU used, and disk space of the VM limitations when relocating or placing the VM. They also take into account the rate of changes of the memory pages of the VM to be migrated. However, when the VM is idle these solutions turn off or suspend the VM disconnecting its services which causes performance degradation when the service needs to be always available such as in enterprise data canter networks e.g. a consulting company or a ski resort website.

To decrease the energy consumption resulting from running machines hosting idle services, the authors of [DPP+10] propose to save energy from idle desktop machines without any service disruption by live migrating the user's desktop environment from the personal computer to a server VM in a remote data center. This strategy indeed saves energy of the personal computer, however, on the side of the data center, energy is wasted to keep idle VMs on.

To decrease the energy consumption of the data center network, energy aware routing and energy efficient networking devices are used such as [SLX10], in addition to traffic engineering and power aware network protocols, which in some cases might leverage the benefits of SDN [HSM+10b, HJW+11, HSM+10a, WZV+14, JP12, XSLW13]. Moreover, to decrease the energy consumption of data center networks, researchers suggest to leverage network virtualization techniques [ENE] such as VLANs and VNF techniques to virtualize full network and network devices decreasing the number of physical network components required to host the network.

In addition to decreasing the energy consumption of the data center servers and networks, energy consumption can be reduced by using renewable energy resources [ZWW11] or cooling strategies such as free cooling [KRA12]. Multiple researchers have studied methodologies to integrate the renewable energy resources such as the solar panels in the data centers without negatively impacting the performance of the latter [LCB+12]. Some others suggested free cooling [KRA12] which uses the external natural air to cool down the temperature of the air circulating in the data center.

5.2 SENAtoR: Reducing Energy Consumption in Backbone Networks

In this work, we consider the problem of *energy aware routing in a hybrid SDN network.* To provide energy optimization in hybrid networks, we introduce SENAtoR - **S**mooth **EN**ergy **A**ware **R**outing which turns off network devices with no negative impact on data traffic. Since in real life, turning off network equipments is a delicate task as it can lead to packet losses, SENAtoR provides several features to safely enable energy saving services: (i) tunneling for fast rerouting, (ii) smooth node disabling and (iii) detection of both traffic spikes and link failures.

In SENAtoR, the controller first chooses the set of routes that minimizes the number of used network equipments for the current traffic, and then we put SDN nodes in sleep mode (a.k.a by

putting them in power save mode) which turns off its network interfaces and some inner networking modules. We consider a typical dynamic traffic of an operator, and hence, our solution adapts the numbers of active and inactive network equipments during the day.

To maintain the network performance while saving energy, traffic has to be rerouted dynamically and automatically when the SDN nodes are put in sleep mode and their links are turned off. It is thus impossible to wait for the convergence of the legacy protocols (e.g. OSPF). Moreover, if ISP network traffic usually shows smooth variations of throughput, it also experiences sudden changes which may correspond to (link or node) failures or to flash crowds [RGK+02]. In these cases, the energy efficient solution should be able to react very quickly and switch on previously turned off devices– in this chapter, we use the terms turn off and sleep mode interchangeably. We thus propose three mechanisms (detailed below):

- First, we use pre-set *tunnels* as backup routes in case of link failure or turned off devices.
- Second, we use the SDN controller to suppress any incoming OSPF packet to simulate a link disconnection on the network interface to be disabled– this forces OSPF nodes to converge to different Shortest Path Tree (SPT)s.
- Last, we use SDN monitoring capabilities to detect quickly large unexpected traffic peaks or link failures.

Using preset tunnels is inspired by the solution proposed in [CXLC15] to handle single link failure. The goal was to avoid waiting for the convergence of legacy routing protocols by using tunnels from a node with a failing link to an SDN node which can reach an alternative OSPF shortest path in one hop. We reused this idea to reroute from any node, with a turned off link, to any other node with a direct path towards the destination which does not include a disabled link.

5.2.1 Energy Aware Routing for Hybrid Networks

Routing in a Hybrid Network. We consider that a network is modeled as a directed graph $D = (V, A)$ where a node represents a Point of Presence (PoP) and an arc represents a link between two PoPs. A PoP consists of several routers linked together [GNTD03]. Each link $(u, v) \in A$ is connected to a specific router in PoP u and in PoP v, see Figure 5.1. A link (u, v) has a maximum capacity C_{uv}.

We consider hybrid networks in which SDN capable equipments are deployed alongside legacy routers. We consider a scenario in which PoPs do not contain heterogeneous equipments, i.e, all routers are either SDN capable (in this case, we use the term SDN switch) or legacy. Legacy routers follow a legacy routing protocol, such as OSPF. We denote the next hop to the destination t on a legacy router u by $n^t(u)$. SDN switches are controlled by one or several central controllers and can be configured, dynamically, to route to any of its neighbors.

Power Model and Energy Aware Mechanism. To model the power consumption of a link,

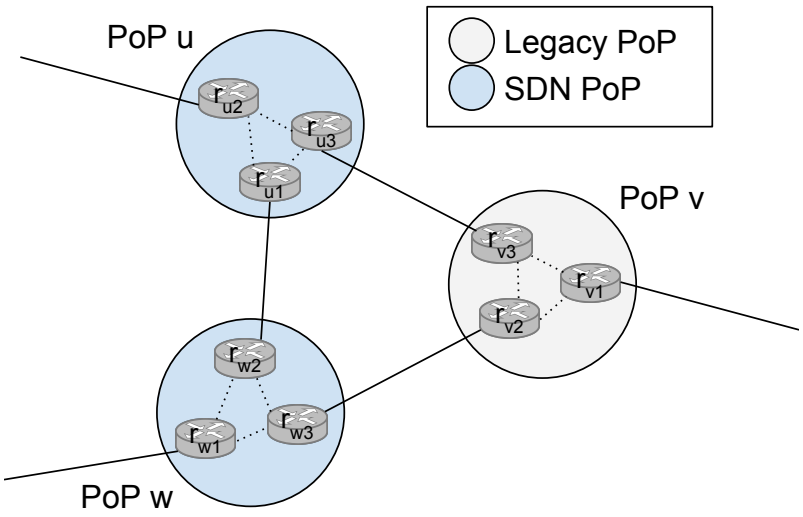


Figure 5.1: 3 PoPs interconnected in a hybrid network.

we use a hybrid model comprised of a baseline cost, representing the power used when the link is active, and a linear cost depending on its throughput. This allows, depending on the value of the parameters, to express the different power models (between ON-OFF and energy proportional) found in the literature, see [ICC+16] for a discussion. The power usage of a link is expressed as follows

$$P_l(u, v) = x_{uv} * (U_{uv} + \mathcal{F}_{uv} L_{uv})$$

where x_{uv} represents the state of the link (ON or OFF), U_{uv} is the baseline power consumption of an active link, $\mathcal{F}_{uv}$ the total amount of bandwidth on the link, and L_{uv} the power coefficient of the link.
Routers have two power states: active or sleep, and their total consumption $P_n(r)$ is given by

$$P_n(u) = B_u + A_u + \sum_{v \in N^+(u)} P_l(u, v)$$

where B_u is the sleep state power usage and A_u the additional power used when the equipment is active.

To save energy, links must be powered down and routers put to sleep. Only SDN switches can be put into sleep mode without negative impact on the network (this is discussed in more details in Section 5.2.2). As it should be done dynamically according to the network traffic, the decision is taken by the SDN controller. Thus, only links with an SDN switch as one of their end point can be shutdown. Since PoPs are interconnected using dedicated routers inside their infrastructure, if a link between two SDNs is shutdown, then each router of the link can be shutdown, if it is SDN capable. For example, in Figure 5.1, shutting down the link between PoP u and PoP v will set router r_{u3} to sleep mode, as it is an SDN switch, but r_{v3} will remain active. Shutting down the link between PoP u and PoP w will put r_{u1} and r_{w2} to sleep.

When an SDN node has to be put in sleep mode and links have to be shutdown, the mechanism is the following: (i) the SDN controller first reroutes the traffic so that no flows are passing through

this node or link (this is discussed in details in Section 5.2.2), then (ii) the SDN controller sends the order to the SDN switch to enter into sleep mode or to disable the interface corresponding to the link. Since no more data packets are using the link, the interface of the legacy router can automatically enter into sleep, using for instance IEEE 802.3az Energy-Efficient Ethernet [CRN+10].

Tunneling. To avoid loosing packets during the re-convergence phase, we use pre-set tunnel backup paths to redirect traffic that would otherwise be lost after a link or node is down. The idea is to reroute the traffic that would use this down link or node to an intermediate node whose shortest path to destination does not use down links. We now consider the following problem.

Hybrid Energy Aware Routing (hEAR) with tunnels Problem. We consider an SDN budget k, i.e., a number of PoPs which can be transitioned to SDN equipments. The hEAR problem is:

- to deploy k SDN PoPs in the network
- to route a set of demands $\mathcal{D}$
- to choose a set of tunnels

while

- minimizing the total power consumption of the network
- respecting the link capacities
- ensuring that the traffic can be rerouted quickly through tunnels when network equipments are turned off

5.2.1.1 Heuristic Algorithm (SENAtoR)

Along with the COATI team, we propose here SENAtoR (Smooth ENergy Aware Routing) which can be used to solve the hEAR problem. The problem can be naturally divided into three subproblems: (i) first, *SDN node placement*, to define the subset of SDN nodes in the network (ii) then, *path assignment*, to find a path for every demand in $\mathcal{D}$, (iii) and last, *off link selection*, to select the links/nodes we power off and reroute the affected traffic.

SDN node placement The SDN node placement is multi-criteria. Indeed, SDN nodes have several functionalities. First, they allow a better control of the routing as the next hop of a flow passing through an SDN node can be chosen dynamically by the controller. Thus, it is important to select SDN nodes which are central and route a large amount of traffic. A way to do this is for example to choose nodes according to their centrality, e.g., *betweenness centrality* or *closeness centrality*. Second, recall that only links adjacent to an SDN node can be turned off. Thus, to be able to reduce efficiently the network energy consumption, we want to cover the maximum number of links with the available budget of k SDN nodes. If we consider this second criterion independently, we can optimize it by solving a MAX k-VERTEX COVER. This problem is known

to be NP-hard. However, it can be solved optimally for the topology sizes considered, using for example a simple ILP and CPLEX.

The COATI team tested different methods to select the SDN nodes and choose a *simple criterion to express the importance of a node (centrality and covering): the node degree*. The resulting heuristics is: first sort all nodes according to their degree; second, choose the k first nodes. This method gives similar results to the other ones and has the advantages of being simple and to allow a good incremental upgrade to SDN hardware (on the contrary to solving MAX k-VERTEX COVER).

Path Assignment To assign a path to a demand, we build a weighted residual graph $H_{st} = (V, A')$ and then search for the shortest path between s and t in H_{st}. We build H_{st} using the following method:

Nodes in H_{st} are the ones of D and correspond to network routers. For links, we only consider links and tunnels which (i) have enough residual capacities to satisfy the demand D_{st} (ii) can be used by a feasible routing of the demand between s and t. For Condition (ii), we consider each node u and construct its set of out-neighbors as follows:

If u is a legacy node, the routing is done by the legacy routing protocol towards the next hop $n^t(u)$ if the link to $n^t(u)$ is active. In this case, the only neighbor of u in H_{st} is $n^t(u)$. Otherwise, if the link to $n^t(u)$ is inactive, the routing is done through a tunnel. We have several cases: (i) If a tunnel is already defined for the destination t, the end of the tunnel is the only neighbor of u. (ii) If no tunnel is defined, the next step depends on the variant of the problem. For hEAR-with-tunnel-preset, tunnels are already selected. Thus, u has no neighbor in H_{st}. In the hEAR-with-tunnel-selection variant, we have to set a tunnel in this case. We thus add all the potential tunnels by adding any node that can reach the destination t, using direct forwarding (OSPF or OpenFlow) or existing tunnels. The decision of which tunnel will be really selected is done later when we compute the shortest path in the residual graph.

If u is an SDN node, the routing is done by OpenFlow rules installed by the controller. We have two cases: if no OpenFlow rule is set for the demand in node u, any neighbor can be the next hop. The neighbors of u in H_{st} are the same as in the original digraph D. Otherwise, we only add as neighbor of u in H_{st} the node designed as the next hop by OpenFlow. As for a legacy node, if the link to the next hop given by OpenFlow is inactive, we also consider tunnels in the same way. The decision of installing or not a new OpenFlow rule in u is done later by the algorithm when the shortest path in the residual graph is computed. The same applies for the selected tunnel.

When the residual graph H_{st} is built, we select the shortest path from s to t to route the demand. If this path uses new tunnels or new OpenFlow rules, we add them to the current solution.

Off Link Selection Once all demands have been assigned a path, we try to power off links to save energy. We consider SDN links one by one, i.e., links with at least one SDN endpoint. We

select the active link with the smallest amount of traffic on both arcs. We then try to reroute all the demands flowing through that link. If no valid routing can be found, the link is set as *non-removable* and the previous routing is restored. If a valid routing is found, the link is set as *inactive* and powered off. We then consider the remaining active links. The heuristics stops when all SDN links are either powered off or *non-removable*.

5.2.2 OSPF-SDN interaction and traffic spikes/link failures

In SENAtoR we consider a hybrid ISP network that is composed of multiple PoPs that are connected together (see Figure 5.1). The SDN nodes communicate with the OSPF nodes using the OSPF protocol control packets. To prevent data loss when we enable the energy efficient solutions, we created several methodologies that will:

- Allow the smooth shutdown of the network devices without loosing data.
- React dynamically to link failures and traffic bursts.

5.2.2.1 Lossless link turn-off.

Before putting an SDN PoP switch in powersave mode which turns off its interfaces, the Floodlight controller demands the switch to stop sending any OSPF packet to its neighbors. This allows neighboring OSPF routers to converge to a network view excluding this node. Indeed, in OSPF protocol, the nodes transmit a `Hello` packet every `hello_interval` to maintain the connection with its neighbors. If a node does not receive a `Hello` packet from its neighbor after the default `dead_interval` of 3 $\times$ `hello_interval`, an OSPF router declares its neighbor as dead and stops using the link. However, until the end of the `dead_interval`, the link is considered to be active and traffic flows over this link. Thus, after the `dead_interval` , plus a safety margin of 10 additional seconds, and if no traffic is received through its links (that we define as the OSPF expected convergence period), the SDN PoP switch is put in powersave mode. To avoid any losses when a node should be put in powersave mode, SENAtoR thus stops sending `Hello` packets during the expected convergence time, before actually putting the node in powersave mode.

5.2.2.2 Traffic bursts mitigation.

Sudden traffic spikes are relatively rare due to the high statistical multiplexing in the backbone of ISPs. However, exceptional events (such as earthquakes) can lead to flash crowds [RGK$^+$02]. Therefore, we complement SENAtoR with a safeguard mechanism that aims at reactivating inactive SDN PoP switches in case of a sudden traffic spike. The latter event is defined on a per link basis as follows: the controller is collecting the traffic load on each interface of every SDN active switch at a small time scale (in our experiments, once per minute). We then compare the real traffic level received at interface i, $E_i(t)$, to the estimated rate, $E_i^{ES}(t)$. In case the real traffic rate is 50%

higher than the estimated rate, $E_i(t) \geq 1.5 \times E_i^{ES}(t)$, for any interface i, all inactive SDN routers are re-enabled to prevent the over-congestion of existing links. The value of 50% was chosen in a conservative manner, since, in general, ISP networks are over-provisioned. There is thus little need taking actions unless the rate fluctuation is severe. After the OSPF expected convergence period, the controller reruns the SENAtoR solution to obtain a new green architecture if possible.

5.2.2.3 Link failure mitigation.

We employ a mechanism similar to the traffic spike mitigation mechanism in case of link failures where SENAtoR undoes any previous action, i.e., it turns on again any inactive SDN node when a link failure is detected. A link failure is discovered by the SDN nodes instantaneously if the link is directly connected to an SDN node. However, if the failed link is between two OSPF PoP nodes, SDN nodes detect it by observing a decrease of the rate of one interface as compared to what the traffic matrix predicts $E_i(t) \leq 0.5 \times E_i^{ES}(t)$. We benefit from the fact that in typical ISP networks, traffic is all-to-all, i.e., from one PoP to any other PoP. Hence, any SDN router in the network is likely to detect the link loss, as a fraction of the traffic it handles is affected by the failure. Again we use a conservative threshold of 50%, i.e., an SDN switch must detect a decrease of 50% of any of its links' load to trigger the link failure mitigation mechanism. After link failure detection, packets are rerouted through a different path if possible (including the pre-set tunnels). Once again, after the OSPF convergence expected period, the controller reuses the SENAtoR solution to obtain a new green architecture if possible.

5.2.3 Experimentations

In this section, we present results obtained on a Mininet testbed with the SENAtoR solution. Our objective in this section is twofold. First, we aim at demonstrating that SENAtoR can indeed turned off links and put SDN switches in power save mode without loosing packets thanks to a smooth integration with OSPF to anticipate link shutdown. Second, we evaluate our link failure and traffic spike detection algorithm that enable SENAtoR to cope with those small time scale events, as compared to energy saving, which is performed at a larger time scale.

5.2.3.1 Testbed

We built a hybrid SDN testbed using Mininet [Min] and a Floodlight controller. OSPF routers are materialized as host nodes in Mininet and run the Quagga software [Qua] while OvS switches act as SDN switches. Our Floodlight controller is able to parse and respond to OSPF `Hello` packets received and forwarded by the SDN OvS switches (through adequate Openflow rules installed in the SDN switches) ; hence ensuring the correct functioning of the adjacent OSPF routers. The very same code implementing the heuristics proposed in Section 5.2.1 and 5.2.4 respectively, is used by the Floodlight controller. Tunnels are implemented as simple GRE tunnels and the interplay

between the tunnel interface and the regular interfaces is controlled by tuning the administrative distance so that regular interfaces have a higher priority. When SENAtoR notifies to put into sleep mode an SDN PoP switch, we turn off all of its interfaces and disconnect it from the rest of the network. We believe this is a fair simulation of powersaving mode, as our tests done on our HP5412zl switch revealed that putting an SDN switch in powersave mode (inactive SDN switch) is equivalent to shutting down all of the network interface modules and background modules, that are not used anymore, and to decrease the energy consumption of the fan, while keeping the set of rules previously installed by the controller. *Keeping the previously installed rules in memory enables a quick recovery from powersave mode to normal active mode.*

We consider in this section the Atlanta topology (15 PoP nodes, 22 links between PoPs) of SNDlib with 50% SDN deployment. As stated in Section 5.2.1, a PoP is composed of several routers. Each router connects the PoP X to another PoP Y, and the routers in a PoP are connected together using a central node such that they form a star topology.

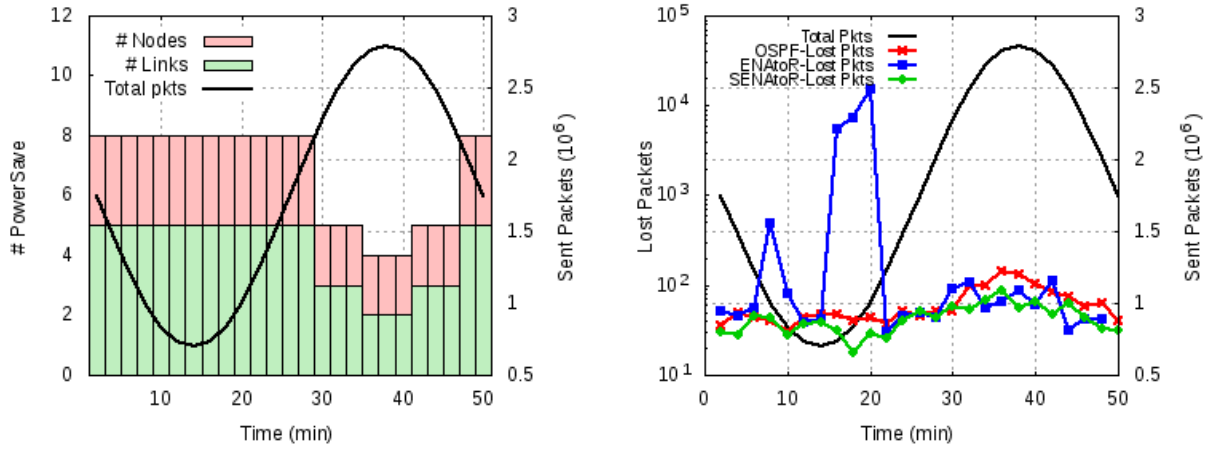
For a given traffic matrix, each source-destination pair corresponds to one CBR UDP connection in our experiments.

5.2.3.2 Results

Lossless link turn-off. In Figure 5.2a we vary the traffic over time (continuous black curve), so that we have a factor of 6 in between the minimal and the maximal total traffic in the network. This is achieved by taking one traffic matrix and scaling it using a sinusoidal function to impose smooth variations on the average rate, similarly to what is expected in a typical ISP network. The bars in the figure correspond to the number of links that are turned off and the number of nodes that are put in powersave mode by the SENAtoR algorithm.

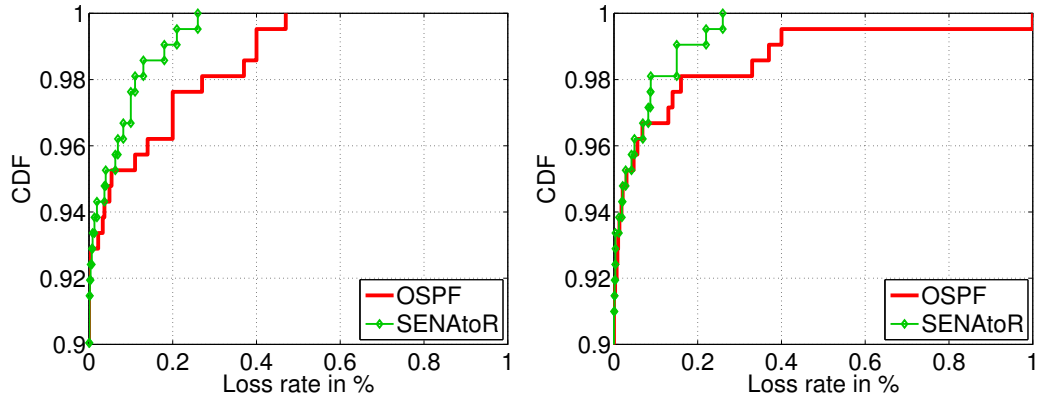
The experiment enables to highlight that the interplay between SDN and OSPF is effective, i.e., that our smooth link shutdown approach effectively avoids data losses. Figure 5.2b portrays the time series of packet loss with pure OSPF (OSPF operates the complete network and no link is turned off in this case), SENAtoR and ENAtoR (SENAtoR without the smooth link shutdown). The figure shows the importance of anticipating the link shutdown (resulting from putting SDN switch in sleep mode) as is done in SENAtoR as losses explode to 10^4 packets when this feature is disabled (ENAtoR). In this case, the high loss rate of ENATOR is proportional to the amount of time it takes for OSPF to declare the link down multiplied by the traffic intensity. In contrast, SENAtoR manages to maintain the same packet loss as a full OSPF network without any links shutdown, with negligible loss rates ($10^{-4}\%$), even though it is using less links and nodes in the network.

Traffic spikes To illustrate the traffic spike mitigation mechanism, we consider a fixed traffic matrix (no scaling) and we induce a traffic spike either at an OSPF node directly connected to an SDN switch (Figure 5.3a) or between OSPF nodes (Figure 5.3b). We report the CDF of loss rates



(a) Number of turned off links **(b)** Packet loss

Figure 5.2: Senator impact on `atlanta` topology using sinusoidal traffic flow.

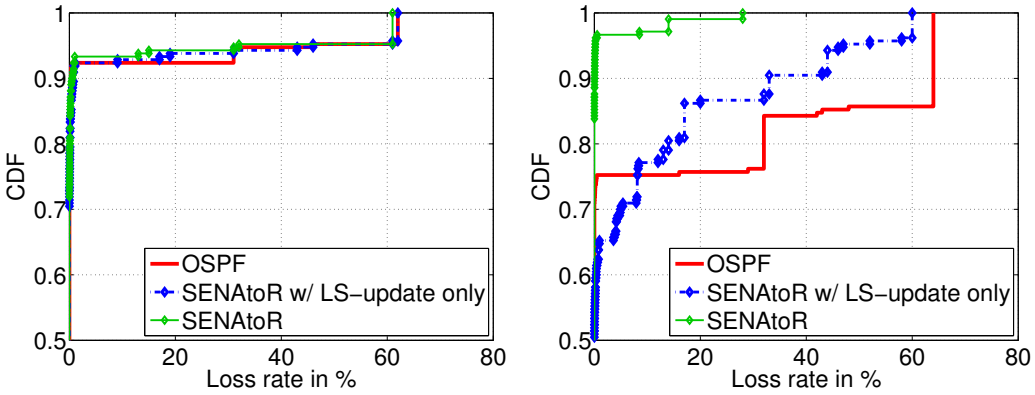


(a) In an SDN-OSPF link **(b)** In an OSPF-OSPF link

Figure 5.3: Traffic spike experiment with the `atlanta` topology

of all connections (source-destination pairs). Clearly, the spike detection algorithm of SENAtoR allows it to outperform OSPF, even though it is using less active links and nodes. One of the reasons of such a phenomenon is that regular OSPF nodes have no mechanisms to automatically load balance packets in case of traffic spikes.

Link failure We consider again a fixed traffic matrix (no scaling) and we induce a link failure either between an SDN switch and an OSPF router or in between two OSPF routers. We present in Figures 5.4a and 5.4b the loss rates for the former and latter case respectively. We compare here three protocols: (i) the legacy OSPF scenario, in which the link failure is handled by the OSPF protocol (with a long convergence time), (ii) the SENAtoR solution using OSPF Link State (LS) Updates only to detect network changes; and (iii) the SENAtoR solution with its *link failure* detection and mitigation mechanism.



(a) On an SDN-OSPF link **(b)** On an OSPF-OSPF link

Figure 5.4: Link failure experiment with the `atlanta` topology

We first observe that even without link failure mitigation, *SENAtoR does not experience higher loss rates than the legacy OSPF protocol* (and significantly lowers loss rates in the OSPF-OSPF case), *even though some of the switches and links were down at the time of the failure*, and had to be switched on. The explanation is that SDN switches do not need to wait for the OSPF convergence before rerouting traffic through the pre-established set of tunnels. The link failure mitigation mechanism further improves the situation.

We further observe a counter intuitive result, which is that the loss rates using SENAtoR are smaller when the failure occurs on an OSPF-OSPF link rather than an SDN-OSPF link. Two factors contribute to this result. First, SDN nodes are placed at key locations in the network such that they convey more traffic. Hence, a failure at these nodes induces higher loss rates. Second, as soon as a downstream SDN node detects a link failure in an OSPF-OSPF link, SENAtoR limits the traffic flowing on this link, which it can do by instructing upstream SDN nodes to reroute their traffic. So, while OSPF convergence time is slow, this is mitigated by the fact that less traffic is sent over the lossy link as soon as SENAtoR detects the failure.

5.2.4 Numerical evaluation

In this section, we evaluate numerically the solutions proposed on different ISP topologies. This allows us to solve the hEAR problem on larger networks of SNDLib than the Mininet testbed used previously. We show that *energy savings up to 35% can be obtained for different levels of SDN hardware installation.*

For the parameters of the power model, we considered the cases of two different hardware: our HP5412zl SDN switch and an *ideal energy efficient* SDN switch as discussed in [VLQ+14]. In the first case, we measured the power consumption using a wattmeter:

The switch uses 95W when in sleep mode and 150W if it is active ($B_u = 95, A_u = 55$).

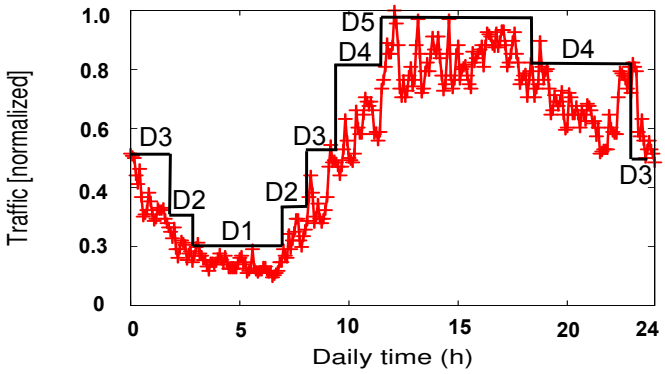


Figure 5.5: Daily traffic in multi-period

According to Cisco specifications [Cis], links are using 30W as a baseline and go up to 40W when at full capacity ($U_{uv} = 30, L_{uv} = 10$). In the second case, we consider an *ideal energy efficient* switch. In order to have a fast recovery from sleep mode, the TCAM must remain powered on to preserve the forwarding rule. According to [CMFA14], TCAM represents 30% of the consumption of a high end router, and considering results from [VLQ+14], we can safely assume that an *ideal energy efficient* switch could save up to 60% of energy in sleep mode.

5.2.4.1 Simulations on larger networks

We first look at the performance of the heuristics on `atlanta` and on larger networks such as `germany50` (50 nodes and 88 links), `zib54` (54 nodes and 81 links) and `ta2` (65 nodes and 108 links).

Traffic Model In this work, we assume that an ISP is able to estimate the traffic matrix of its network using (sampled) netflow measurements [B. 04] or, in the case of hybrid networks, by combining SDN and OSPF-TE data [AKL13]. Estimation errors can be handled by our traffic variation detection algorithms (Section 5.2.2). Since ISP traffic is roughly stable over time with clear daily patterns, a few traffic matrices would be enough to cover a whole day period. Consequently, a relatively small number of routing reconfigurations allows operators to obtain most of the energy savings [ICC+16] and avoid making frequent reconfigurations.

Indeed, as exemplified by the daily variations for a typical link in the Orange ISP network, see Figure 5.5, five traffic matrices (labeled D1 – D5) are enough to represent the daily variations. Inspired from this observation, we select these 5 different traffic matrices as baseline for our simulations and experiments.

We next compute the best hybrid energy aware routing for them. Whenever SENAtoR detects that the amount of traffic has changed, it relaunches its routing heuristic. We rely on the mechanisms discussed in Section 5.2.2 to do this operation without loss.

Daily savings In Figure 5.6, we compare the energy savings during the day for the four topologies. The top figures represent the savings with HP switches and the bottom ones the savings with *ideal energy efficient* switches. We look at 4 different levels of SDN deployment: 10%, 25%, 50% and 100% of upgraded nodes in the network. For each period, we compare the energy used to the one of a legacy network at the same period.

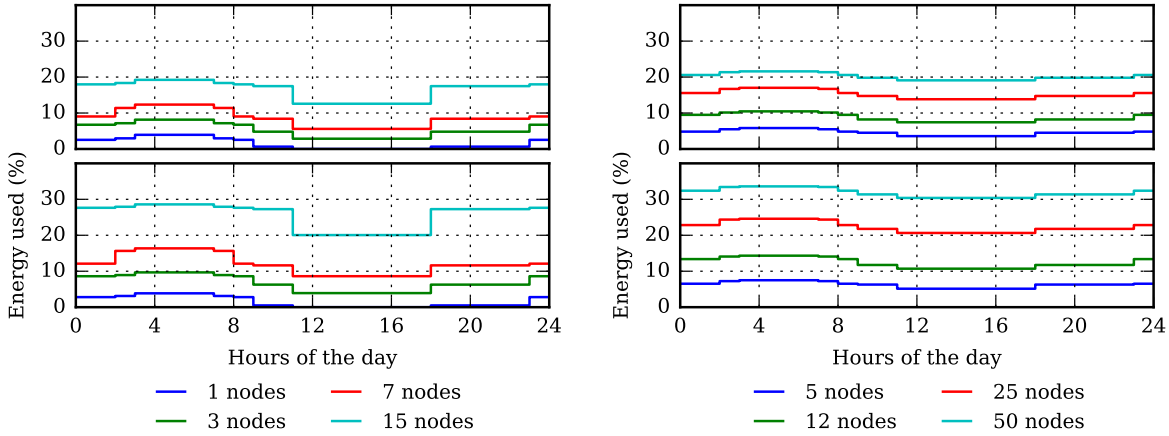
On a full SDN network, the difference between night and day energy savings is between 2% and 7% (3.5% and 9% with *ideal switches*). With HP switches, we can save up to 19% on `atlanta`, 22% on `germany50`, 17% on `zib54` and 21% on `ta2` with a full SDN networks. With *ideal* switches, we obtain higher savings, between 25% and 35%.

Number of tunnels We look at the number of tunnels used in Figure 5.7. For small SDN budgets (up to 30% of the network for `atlanta`, 20% for larger networks), the average number of tunnels greatly increases with the number of SDN nodes. The reason is that more network links may be turned off, and thus, more backup tunnels are needed. The number of tunnels then levels off and decreases. Indeed, with a large penetration of SDN in the network, SDN nodes can dynamically forward the traffic regardless of OSPF and the traffic can be rerouted before arriving to the turned off link. Thus, less backup tunnels are needed. The maximum average number of tunnels needed per node is proportional to the size of the network (3 for `atlanta`, 8 for `germany50`, 9 for `zib54` and 15 for `ta2`). Finally, while the number of tunnels needed may seem high, we see in the next section that the impact of this overhead on the network performance (packet loss or delay) is not noticeable.

Stretch and delay By nature, Energy Aware Routing has an impact on the length of the route in the network. As we turn off links, we remove shortest paths. Moreover, tunnels can also increase the path length. In Figure 5.8, we show the stretch ratio of the paths for four levels of SDN deployment. We only show the stretch for the period with the lowest amount of traffic, as it is the period with the largest number of turned off links and thus the one with the largest stretch.

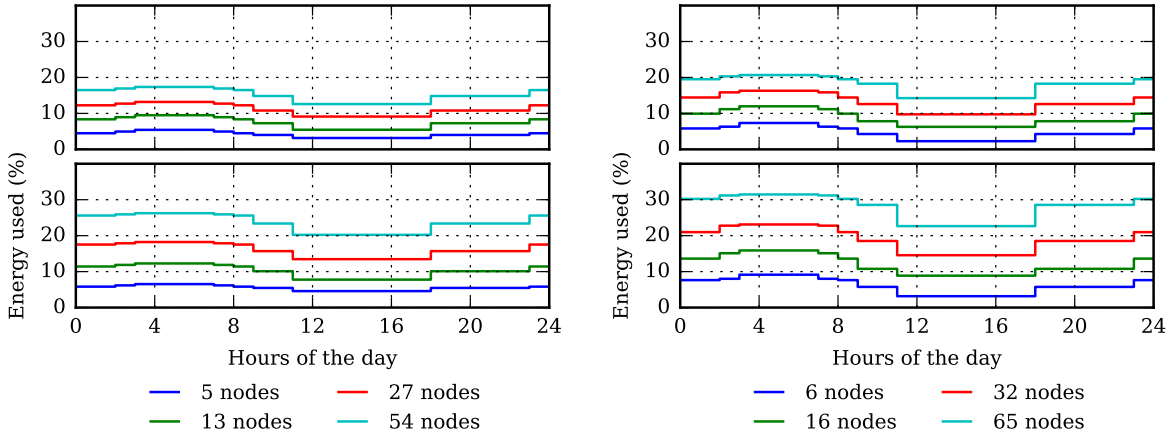
Most of the demands are barely affected by SENAtoR. The median stays around a ratio of 1 with a maximum of 1.25 for `atlanta` at 100% deployment, 1.25 for `germany50` at 50% deployment, 1.33 for `zib54` at 10%, and 1.25 for `ta2` at 25%. 90% of the paths have at most a ratio less than or equal 3. The stretch of the paths follows the same behavior as the number of tunnels needed for a valid hEAR. Below a 50% deployment, we need an increased number of tunnels to forward the traffic, and thus, we also increase the length of the paths. On a full SDN network, we only see the stretch due to powered off links.

Even though some paths reach a stretch ratio of 14 on `germany50` and 9 on `zib54`, we can see in Figure 5.9 that the delay on the network stays relatively low. Indeed, the paths with a big stretch are mostly one-hop paths that used to be on currently inactive links. To compute the delays, as the delay is proportional to the distance in an optical network [CMZ+07], we use



(a) `atlanta`

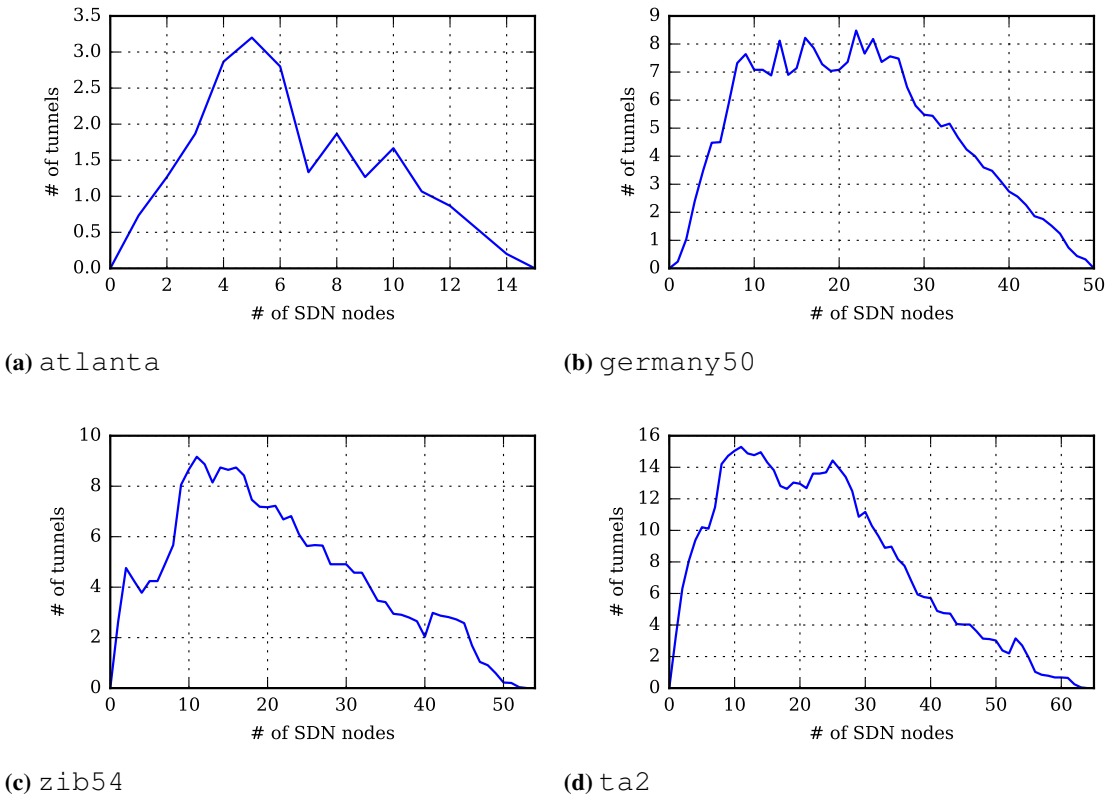
(b) `germany50`



(c) `zib54`

(d) `ta2`

Figure 5.6: Daily energy savings over the day for the (a) `atlanta`, (b) `germany50`, (c) `zib54` and (d) `ta2` networks. with 10, 25, 50 and 100% SDN nodes deployment. Top plots: power model of the HP switch. Bottom plots: power model of an ideal energy efficient SDN switch.



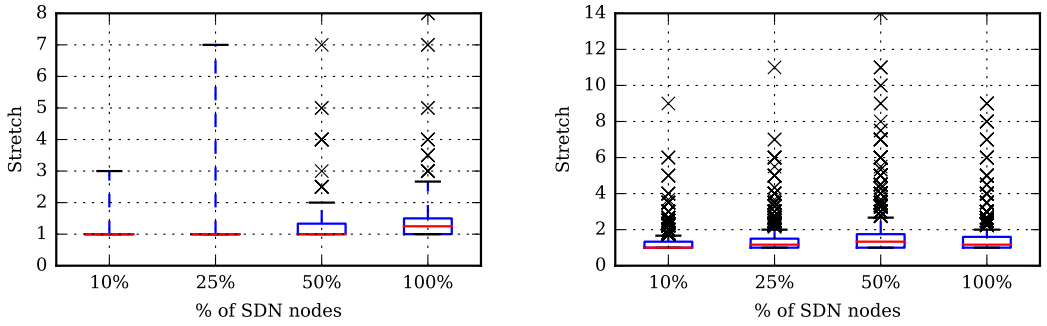
(a) `atlanta` **(b)** `germany50`

(c) `zib54` **(d)** `ta2`

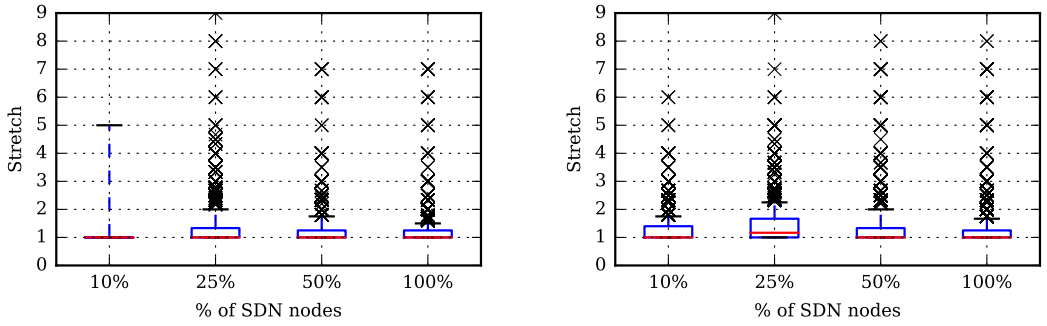
Figure 5.7: Number of average tunnels installed per node on the (a) `atlanta`, (b) `germany50`, (c) `zib54`, and (d) `ta2` networks

the distances given by the geographical coordinates in SNDlib for the `germany50` network. We got an average value of 1.8 ms per link. Since the coordinates are not given for the other two topologies, we used the same average value for `atlanta`, `zib54` and `ta2`. The median delay rarely goes above 10 ms for all four networks. The `zib54` network experiences the worst delay with a half SDN deployment, with almost 35ms of delay. *The bottom line is that using SENAtoR, we stay below a delay of 50ms.* This is important, as this value is often chosen by Service Level Agreements (SLAs) as the maximum allowed delay for a route in a network [FDA+04] Thus, even if new routes computed by our algorithms may experience sometimes a high value of stretch, this will not be a problem for network operators.

As explained above SENAtoR saves energy by turning off SDN nodes network interfaces or putting the nodes into sleep mode. Turning off only SDN nodes in a hybrid environment that contains 50% of SDN nodes can provide between 5% to 35% of energy savings. Though this energy saving percentage might seam low, it can be aggregated with other energy aware routing techniques (e.g. [RPUK14]) for non-SDN nodes to increase energy savings. In addition, SENAtoR saves energy while preserving network performance, by using lossless link/node turn-off, spikes, and traffic failure detection services.

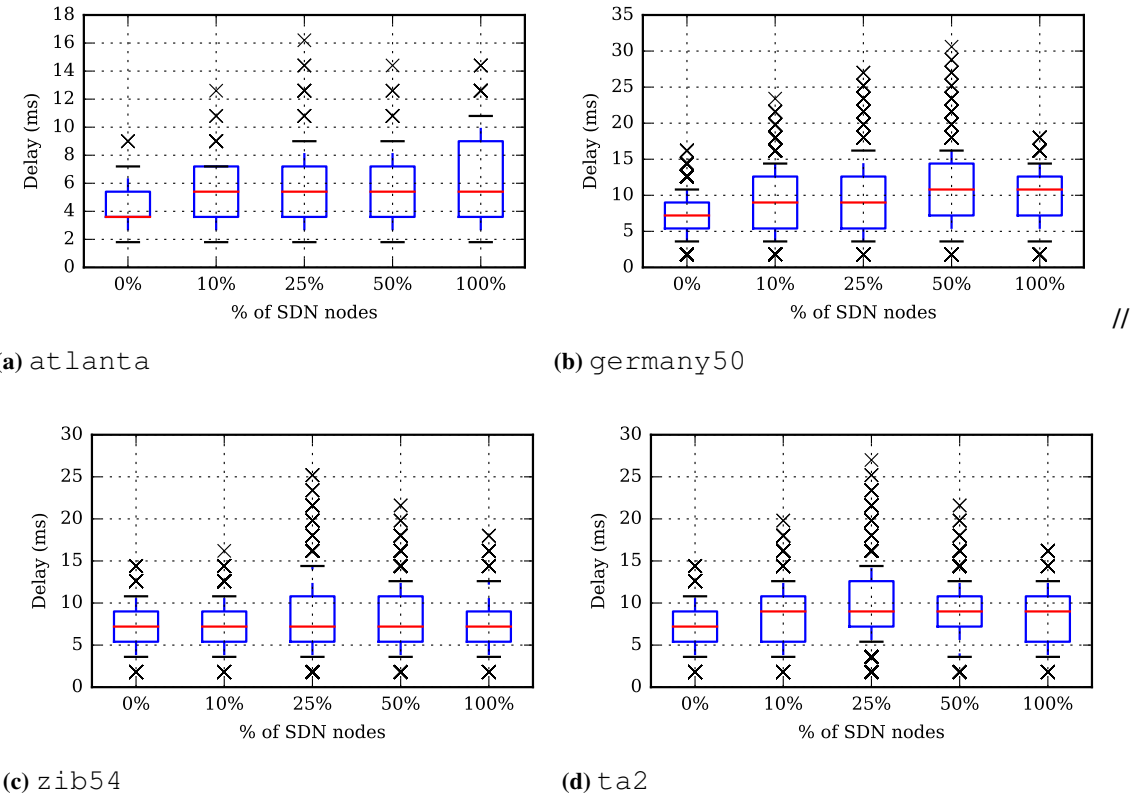


(a) `atlanta` **(b)** `germany50`



(c) `zib54` **(d)** `ta2`

Figure 5.8: Stretch ratio for four different levels of SDN deployment on (a) `atlanta` (b) `germany50`, (c) `zib54`, and (d) `ta2` networks. The box represents the first and third quartiles and whiskers the first and ninth deciles.



(a) `atlanta` **(b)** `germany50`

(c) `zib54` **(d)** `ta2`

Figure 5.9: Delays for the demands in the (a) `atlanta` (b) `germany50`, (c) `zib54`, and (d) `ta2` networks.

5.3 SEaMLESS: Reducing Energy Consumption in DataCenters

In the previous section, we presented an energy aware routing solution that allows to improve the energy efficiency of ISP networks while preserving network performance, by using lossless link/node turn-off alongside spikes and traffic failure detection services. In this section, we move on to enhance the energy efficiency of data centers.

As we stated previously in the introduction of this chapter, the *power consumption* of Data Centers (DC) is one of the major problems of ICT. Almost 50% of the power consumption used by the datacenters is used by the physical servers that host the VMs that provide the services (e.g. Web server, SQL server, etc). To limit this energy consumption, algorithms for power-aware placement of VMs aim at minimizing the number of active physical servers. However, these solutions do not consider the state of the virtual machine (active or idle), which could definitely lead to an optimized solution that minimizes the number of physical servers needed to host active VMs. Moreover, as we can see in Figure 5.10 which shows the energy gain when turning off idle VMs, idle virtual machines increase the power consumption of the physical server even though they are not doing any action. Based on the results of Figure 5.10, we notice that the energy gain increases as the number of idle VM hosted in the physical server increase, and as the load of the running VM increase to attain a maximum of 20%.

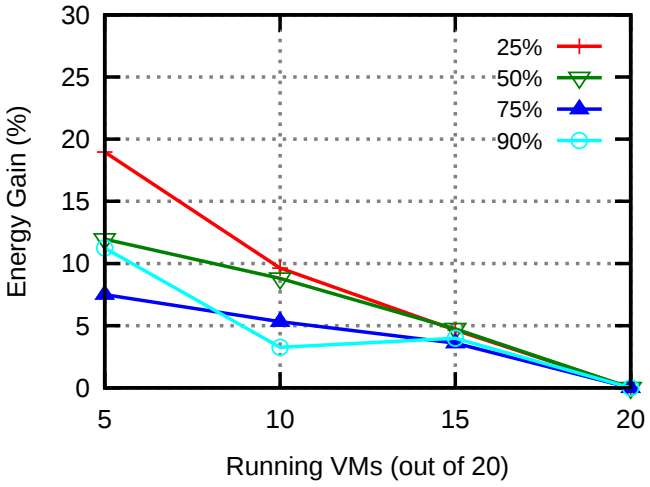


Figure 5.10: Energy gain when turning off idle virtual machines on a physical server.

Hence, in Signet Team we started working on SEaMLESS, a solution *that tackles the problem of idle virtual machines in data centers, in particular, for private enterprise clouds*. SEaMLESS is a novel strategy that migrates an idling service from within a VM to an external *Sink Server*. SEaMLESS solution was developed for data centers where the VMs enter frequently short or long idle periods e.g. during breaks or outside office hours etc. However, those VMs services need to be always available due to possible user remote connection at any time [ZLC+16]. Our work is based on the general assumption that a service e.g. a web server in the VM, which we call the *Gateway Process*, is actively listening to one or multiple port numbers, waiting for incoming connections

or requests.

To enable VM suspension while maintaining Gateway Process availability, SEaMLESS migrates the Gateway Process from the original hosting VM to a lightweight virtual network function (VNF) in Sink Server (Section 5.3.1) and then it suspends the original VM. Next, when the Sink Server detects user activity on the VNF service (Section 5.3.4), the original VM will be resumed, and the Gateway Process will migrate from the VNF to the VM (Section 5.3.2) and continue its execution in its original environment, in a transparent manner for cloud users, i.e. SEaMLESS-ly.

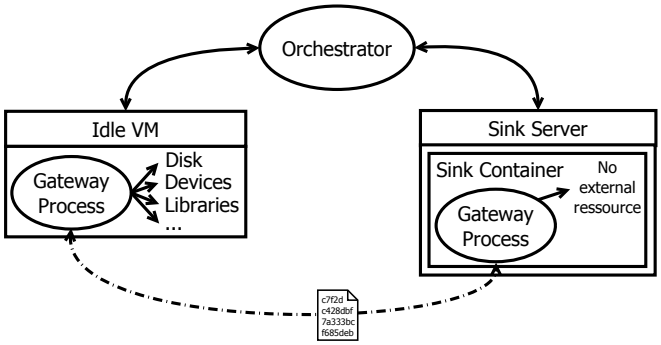


Figure 5.11: Components and architecture of SEaMLESS

In more details, using Figure 5.11 as illustration, a virtual machine runs many processes, including the Gateway Process, which accesses several resources. Once the virtual machine has been detected as idle, the SEaMLESS *Orchestrator* – an agent responsible for the synchronization of the migration – will create a lightweight version of the Gateway Process and transfer it to the Sink Server, effectively turning the Gateway Process into a VNF. At this point, the VNF form of the Gateway Process does not have access to its external resources anymore. This peculiarity enables the VNF to run as a much more lightweight environment, hence the Sink Server can host hundreds to thousands of Gateway Process VNFs, avoiding the need for additional energy to host the corresponding idle VMs across multiple physical servers. In this section, we present in brief the main working blocks of SEaMLESS.

Our analysis and experiments show that SEaMLESS, combined with server re-consolidation solutions, provides a way to put physical servers in standby while introducing little impact on the quality of experience. We even show a way to power off physical servers in cases where VMs are expected to remain idle for a long period of time.

5.3.1 Migrating from the VM to the Sink Server

When a virtual machine is known to be idle, or is planned to be idle for a long period of time, the Orchestrator triggers the Gateway Process migration to the Sink Server. The complete procedure for a process migration from the VM to the Sink Server shall be done as follows. Note that these steps *must* be executed sequentially. When a single step requires the execution of several actions, these actions should be parallelized. The complete process migration procedure between the VM

and the Sink Server is graphically depicted in Figure 5.12.

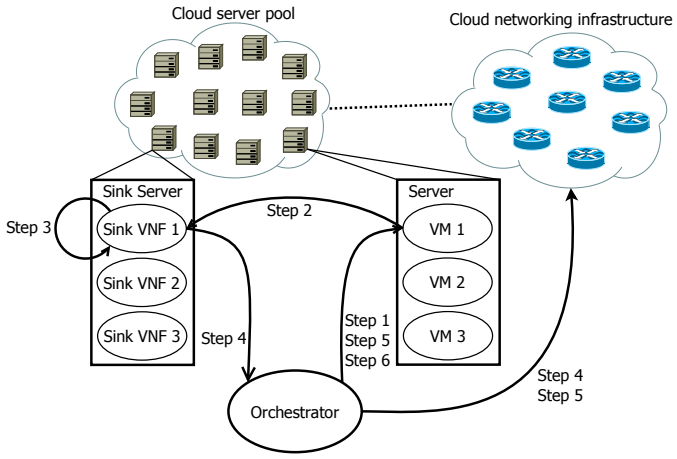


Figure 5.12: Migration procedure from a working virtual machine to a Sink Server

Step #1: The Orchestrator asks the working VM to dump the Gateway Process to a file. It keeps the process running, which allows the VM to detect any user activity. Should any user activity be detected, the VM sends an abort message to the Orchestrator to cancel the migration, without any message loss. **Step #2:** Once the process image has been generated, the working VM sends the process image file to the Sink Server. **Step #3:** The Sink Server deploys the Gateway Process. **Step #4:** Once the Sink Server has deployed the Gateway Process, the Sink Server sends a ready message to the Orchestrator. The Orchestrator then modifies the networking devices to forward a *copy* of any incoming packet destined to the VM to the Sink Server. This packet duplication is useful to avoid splitting possible train of packets with a user's requests, in the case that some of such packets would reach the VM when the rerouting mechanisms occurs. **Step #5:** After a few milliseconds (which ensures that no packet has been received by the VM during either, the setting up Sink Container period, or the network reconfiguration period), the networking devices are modified to forward packets to the Sink Container *exclusively*. At the same time, the Orchestrator asks the working VM to kill the Gateway Process. **Step #6:** The Orchestrator freezes the working VM.

Note that any user activity detected during process migration will result in a roll-back mechanism that is completely transparent from the user perspective.

5.3.2 Migrating from the Sink Server to the VM

The procedure to move the Gateway Process back from the Sink Server to the VM is close to the reverse procedure, described earlier in Section 5.3.1, except that when a process is moved from the Sink Server to the VM, it is because the user already issued a request to the cloud service. Hence, a TCP connection is most likely already set up and packets can flow at any time.

SEaMLESS provides a simple, fast, but still powerful solution to avoid packet losses during

the restoring service period. Like before, steps must be executed sequentially, and multiple actions in a single step can be parallelized. This procedure is depicted in Figure 5.13.

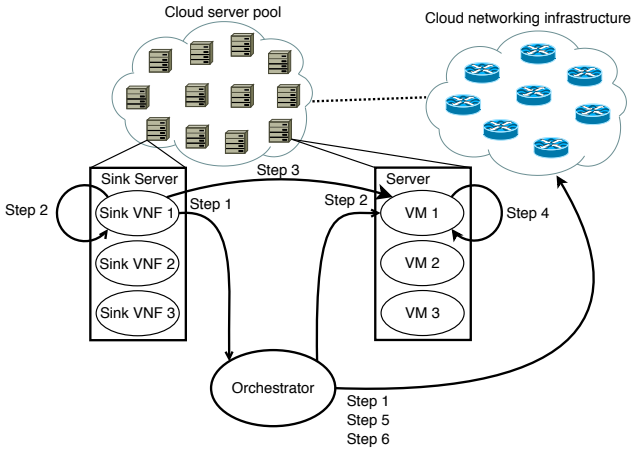


Figure 5.13: Migration procedure from a Sink Server to a working virtual machine

Step #1: Upon detection of user activity on the Sink Server, the Sink Server notifies the Orchestrator that the corresponding VM must be waken up. **Step #2**: The Orchestrator deploys a buffer where any incoming packet destined to the Gateway Process (identified by the IP destination address and port destination number) are delayed. At the same time, the routing tables are modified to send any packet to the VM instead of the Sink Container. The buffering of packets is to ensure that no packet will be lost before having a completely deployed Gateway Process at the VM. **Step #3**: The Gateway Process at the Sink Server is dumped. The VM is resumed. **Step #4**: Once the VM is on, the process image is sent from the Sink Server to the VM. **Step #5**: The process is deployed at the VM. **Step #6**: Buffered packets are unblocked, and the buffer is destroyed. The communication is now handled by the VM.

5.3.3 Addressing Routing Issues

To properly reroute packets to the Gateway Process VNF when the Gateway Process is available inside a Sink Container, SEaMLESS entirely relies on SDN forwarding devices. Hence, when packets must be duplicated or only rerouted to the Sink Server (as detailed in Sections 5.3.1 and 5.3.2), the Orchestrator asks the SDN controller to inject the ad-hoc rules into the SDN switches.

In modern data centers, where SDN hardware devices are available, the forwarding table of the needed SDN switch(es) will be modified by the SDN controller. In this case, the veth interface of the Sink Container should either be configured with the MAC address of the working VM, or the SDN switch should replace the destination MAC address of any packet destined to the VM by the MAC address of the Sink Container.

In the case where no SDN hardware is available in the data center, multi-tenancy and IaaS is frequently provided with the deployment of a Networking as a Service (NaaS), such as Neutron

[Neu] in OpenStack, where a full virtual SDN network is deployed by means of software switches, like OvS [PPK+15]. This way, even in presence of legacy network infrastructure, IaaS providers offer migration and server reconsolidation capabilities. In such a case, SEaMLESS will need to only interact with the virtual network devices to replace a fully-fledged VM by a Sink Container and inversely.

5.3.4 Detecting User Activity

Once the Gateway Process VNF is deployed, the VNF is responsible of processing incoming packets and initiating the communication with the remote peer. Then, after processing the packets and if user activity is detected, the VNF should hand back the connection to the VM. However, it should be noted here that not all incoming packets carry user data as some packets can be processed at the kernel level (e.g. ICMP or ARP requests) without requiring the VM intervention, hence the VNF must reply to such packets. However, data packets must be handed back to the VM as the VNF does not have the external data resources of the Gateway Process.

Packets which correspond to user activity (i.e. data packets) have a different fingerprint depending on the type of Gateway Process: (i) *stateless* Gateway Process such as HTTP request-and-response protocol or (ii) *stateful* Gateway Process such as SSH protocol where a session is established between the server and a client, and the communication channel remains open (using keep-alive packets) until either party decides to tear it down.

Our strategy to detect user activity relies on *tracking syscalls* made by the Gateway Process VNF inside the Sink Container. For a stateless Gateway Process, when a request is triggered by a client and a TCP socket is open, an `accept()` syscall is raised by the server. SEaMLESS will track this `accept()` syscall in order to detect user activity.

The same principle applies to stateful gateway processes, but the user activity detection is more complex. With stateful gateway processes, peers are already connected and, from time to time, it is possible to receive TCP keep-alive messages, which are directly replied to by the kernel. However, application-level keep-alives, like the ones used by SSH clients to keep a connection open, involve some code execution within the Gateway Process, but require no (external) resource access. Therefore, it is better to allow the Gateway Process VNF to directly reply to such requests. To implement the keep-alive-reply feature without resuming the VM, SEaMLESS relies on the numbers of the file descriptors that the Gateway Process wants to access in the following way. When a new message is received, a `recvfrom()`, or a similar syscall (e.g. `read()`) is raised, the number of the invoked file descriptor is recorded as the message may either be a keep-alive or a user command. If the received message is a keep-alive, the Gateway Process will raise a `write()` or a similar syscall (e.g. a `writev()` syscall). When entering these syscalls, SEaMLESS compares the number of the file descriptor where the write will occur to the number of the file descriptor from which the message was received. If the number of the file descriptors are the same, it means that the message was a keep-alive, and the Gateway Process has been able to build

an appropriate reply. Conversely, if the file descriptor numbers are different, it means that the message is sent to an external resource, which is not available within the Sink Container. In this case, the Gateway Process must be restored within the working VM, and the communication with the end user can no longer be handled by the Gateway Process VNF.

A key advantage of the above approaches is that they are generic, i.e., agnostic to the details of the application hosted in the sink.

5.3.5 Energy Saving Strategies

In this Section, we discuss how to deploy SEaMLESS to provide energy savings, while simultaneously meeting the required QoS for the data center.

To increase the energy savings when multiple idle VMs are replaced by sink containers, physical servers should enter deeper sleep modes than a purely operating system based idle state. By relying on external cloud manager plugins that execute server re-consolidation and migrate all active VMs in the least possible number of physical servers, such as the BtrPlace project [The], SEaMLESS can allow important energy saving. We propose two strategies to save energy, by making the physical servers hosting idle VMs go into deep idle modes. The first one places the servers in standby mode while the second one enables to shut them down.

5.3.5.1 Servers in Standby Mode

Most physical servers in data centers are designed to enter into either S1 or S2 power states. In power state S1, the processor clock is off and bus clocks are stopped. In power state S2, the processor is not powered and the CPU context and contents are lost. Data centers servers can save a few tens of Watts when entering one of these states. We measured an immediate gain of 10W when entering the S1 modes on a Dell PowerEdge R730, and 18.8W on a Dell PowerEdge R410. These correspond to a relative gain of 10% and 16.3%, respectively, over the total energy consumption of a server.

This approach also introduces very little impact on the perceived QoS, as seen by virtual machine users. Indeed, a physical server is capable of resuming from S1 or S2 within a few seconds (we measured an average of 3 seconds with our servers), which implies that the delay between the reception of a user request, and the user's VM response will be a total of around 4 seconds. Indeed, our experiments with the modules of SEaMLESS show that the amount of time necessary to migrate and deploy a Gateway Process from the Sink Container back to the VM is, in the worst case, below one second– see Section 5.3.6. The idle VM is still available when the physical server is woken up, and the server wake up process occurs in parallel to the checkpointing process of the Gateway Process. Consequently, the total migration time is the sum of the transmission time, deployment time of the Gateway Process image and the server wake-up delay.

Note that according to [CFY04], for applications with very short expected response delay (no more than 2 seconds, like in Telnet), this strategy can still negatively impact the user QoS. However, for applications where a response delay between 2 to 5 seconds is expected by the user (e.g. Web browsing), this strategy will have only limited impact on the perceived QoS. This solution is therefore suitable in working environments where the VMs must be resumed as soon as possible e.g. during working office periods.

5.3.5.2 Powered-Off Servers

When VMs are expected to remain idle for long periods of time (e.g. outside working hours), they can be suspended to disk. In this situation, the physical servers can be completely powered off. Indeed, from the N servers hosting only idle VMs, $N - 1$ servers can be powered off, while one will remain powered on. If physical servers possess hard drives, servers to be powered off must transfer the VM images to the server remaining awake. Otherwise, the images of VMs must be stored on network storage (e.g. SAN devices). When user activity is detected, and the Gateway Process must be migrated back from the Sink Server to its VM, the remaining powered on server will restore the required VM (by recovering the VM image from the network storage if needed), and the Gateway Process will be restored in its original VM, on that particular server.

Assuming the case where the VM image was transfered on the local hard drive of the powered on server, the time needed to restore the VM must be taken into account to estimate the total elapsed time between the reception of the user's request by the Gateway Process VNF and the appropriate response from the VM. According to our experiments, the time needed to resume a VM from disk, excluding the network transfer time, depends on the size of the image of the VM's RAM, which depends on the particular implementation of the used hypervisor. For instance, KVM needs around 7 seconds to restore an image of a VM using 1GB of its assigned RAM. This time increases linearly as the VM RAM image size increases. In VMWare, we measured the time needed to restore a VM to be around 3 seconds, independantly of the image size. We believe that VMWare hypervisor uses a memory-mapped file to assign the RAM to the VM before it is completely restored to the physical memory, whereas KVM fully copies the RAM image's contents to the physical RAM before resuming the VM services.

5.3.6 Performance Evaluation

5.3.6.1 Impact on the Quality of Experience

In this Section, we run the key modules of SEaMLESS, and show that the amount of time needed to migrate the Gateway Process from the Sink Container to the working VM is below 1 second.

To evaluate SEaMLESS, several experiments where conducted on mainstream, unmodified applications. The different phases that are triggered in case of user activity/inactivity were tested

individually: (i) dumping of the image; (ii) image compression; (iii) transferring of the image; (iv) image decompression; (v) processes restoring; and (vi) VM resume. The time needed by those six phases provides insights about the migration period of the service.

The testbed consists of two physical hosts equipped with 1Gbps network interface cards, connected through one 1Gbps switch. One of the physical hosts runs a VM with Ubuntu 16.04, a single processor, 1GB of RAM, and 8GB of virtual disk. The dumping and restoring mechanisms are executed within the VM. Note that the time to dump and restore a process within the Sink Container is similar to the time required to do the same process at the VM as in such a case, there is no overhead introduced by a hypervisor layer and the image size of the Gateway Process remains similar (e.g. the image size at the Sink Container exceeds the one from the VM by around 10KB in our experiments). The transfer of the process image occurs between the two physical servers, through the SDN switch.

From our tests, available in Table 5.1, we see that the application with the biggest image file is the Apache 2 application with PHP enabled (7.508MB), followed by Apache 2 without PHP and OpenSSH (1.832MB and 1.216MB respectively). Apache 2 and OpenSSH are the most deployed softwares in servers where the performance is more important than preserving computational resources.

Application	Image Size (MB)	Delay (secs)					
		Transfer	Dumping	Restoring	Comp.	Decomp.	Total
lighttpd	0.236	0.137	0.010	0.014	0.007	0.007	0.175
Apache 2 (without PHP)	1.832	0.183	0.089	0.082	0.024	0.025	0.403
Apache 2 (with PHP)	7.508	0.358	0.190	0.153	0.110	0.150	0.961
vsftpd	0.128	0.169	0.083	0.052	0.006	0.005	0.315
OpenSSH	1.216	0.166	0.192	0.081	0.019	0.019	0.477

Table 5.1: Size of the archived (`tar.lzo`) image of real-world applications.

The lightest image is for the `vsftpd` application (128KB), which deploys a single master process listening on TCP port 21. Please note that in our configuration, the number of SSH worker processes for OpenSSH (and, therefore, the size of the process image) depends on the number of opened SSH session.

From those tests, we see that the total delay, from the dumping of the Gateway Process to its deployment, is always smaller than 1 second.

5.3.6.2 Scalability and Energy Consumption of the Sink Server

The promise of possible energy gains with SEaMLESS is that the Sink Server is able to host way more Gateway Process VNFs than it would be able to host virtual machines. In this Section, we measure how the Sink Server scales with numerous Sink Container.

We use a Dell PowerEdge R520 to host a Sink Server VM that has been assigned 1 CPU and

1GB of RAM. Table 5.2 shows that it is capable of hosting up to 53 Dorpbear SSH Sink Container, if there are no active SSH sessions. In the case that all Dropbear SSH VNFs have 1 active SSH session, the Sink Server is still capable of hosting 40 Sink Container. And as shown in Figure 5.14, the memory used and the percentage of CPU used increases as a function of the number of Sink Container deployed. However, we notice that the number of VNF that can be installed is limited by the RAM capacity: 48 Apache 2 VNFs with PHP module would saturate 1GB of RAM, while consuming only 19% of the CPU. These results show that a typical data center server configured with 32GB of RAM can host around 1312 Dropbear SSH Sink Container with active sessions, and up to 1696 Dropbear SSH Sink Container without connections. These figures are much higher than the number of idle VMs that could possibly be running simultaneously on a server with only 32GB of RAM.

Application	Maximum Number of VNFs
Apache 2 (with PHP)	48
Dropbear SSH	53
Dropbear SSH with cnx	40

Table 5.2: Maximum number of VNFs that can be configured in a Sink VM with 1 CPU and 1GB of RAM.

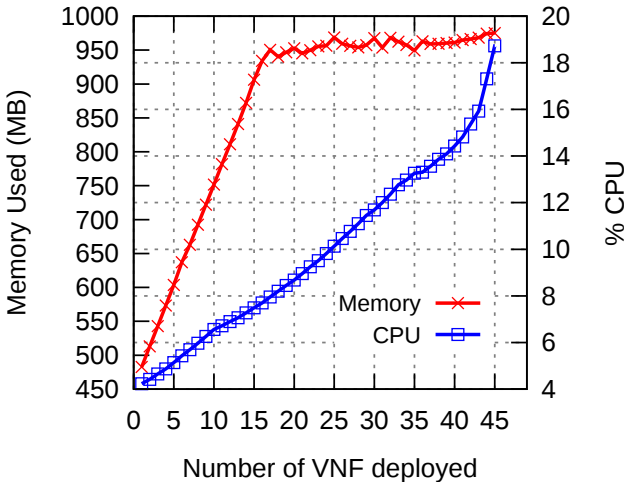


Figure 5.14: RAM and CPU used as a function of number of deployed Apache 2 with PHP module VNF .

Figure 5.15, compares the energy consumption of the same server with Gateway Process VNFs, compared to the same server with full-fledged VMs. We notice that even with 40 VNFs configured on the Sink Server, each one maintaining an active SSH session, the energy consumption of the server remains in average 10W lower than the energy consumption of the same server running 20 idle VMs, each one keeping also a single SSH session.

This shows that, even without suspending or powering off the physical servers, SEaMLESS provides a minimal gain of 10W per server.

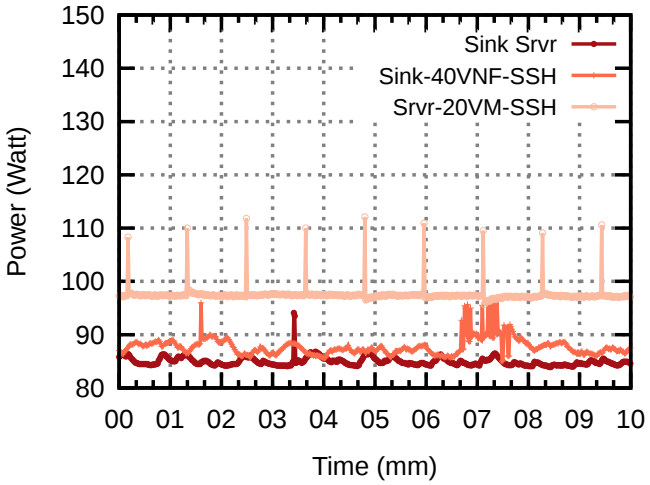


Figure 5.15: Energy consumption of the sink server with VNFs compared to the same server with VMs.

5.4 Conclusion

Providing energy saving services in current networks and data centers is a challenging task as care must be taken to avoid traffic disruption, preserve failure tolerance capabilities, and maintain service availability even when the network operates with a reduced number of devices. In this chapter, we have presented our solutions that permit to maintain the network performance intact when enabling energy efficiency (SENAtoR) and maintain energy efficiency and service availability even when the service is idle (SEaMLESS).

We first presented our energy efficient solution SENAtoR that allows to deploy energy efficient solutions in backbone hybrid networks without degrading the network performance. We showed that SENAtoR can decrease the energy consumption of backbone networks by 35% with only 50% deployment of SDN nodes. We have shown through our SENAtoR implementation and experimentation with emulated devices that we can deal with unexpected network events correctly. More strikingly, our experiments show that even when green services are enabled and traffic spikes occur in a non SDN capable node, SENAtoR provides loss rates lower than the all-OSPF case, since the SDN controller can provide most appropriate routes. Moreover, our numerical results show, in an all-to-all traffic matrix, an incremental deployment of SDN can provide between 5% to 35% of energy savings.

Then, we presented the Signet project SEaMLESS in which I participated. SEaMLESS provides enhanced functionalities which allow to optimize the energy efficiency of existing server consolidation solutions. We have explained how SEaMLESS converts a *Gateway Process* inside an idle VM into a VNF running on a *Sink Server* to enable to turn off idle VMs and release all resources used by this memory. We showed that converting idle VMs to VNFs allows to release a huge amount of memory space where tens of VNFs require almost the same memory space as a

single VM. Thus, when server consolidations solutions are used, active servers utilization would be optimized as memory is arguably the most scarce resource in data centers and clouds.

5.5 Publications

- **International Conference**
  - N.Huin, M.Rifai, F.Giroire, D.Lopez Pacheco, G.Urvoy-Keller, J.Moulierac , "Bringing Energy Aware Routing closer to Reality with SDN Hybrid Networks", IEEE Globecom 2017.
- **Poster**
  - D. Lopez Pacheco, Q. Jacquemart, M. Rifai, A. Segalini, M. Dione, G. Urvoy-Keller "SEaMLESS: a lightweight SErvice Migration cLoud architecture for Energy Saving capabilitieS", ACM SoCC 2017.

Chapter 6

Conclusion

Contents

In this thesis, I presented the work done during my three years of PhD which tackled the scalability, performance and energy efficiency problems to build next-generation SDN based virtualized networks. We first presented MINNIE in Chapter 3, a solution developed in collaboration with the COATI team, which allows to enable flow scalability in Software Defined Networks by optimizing the usage of the TCAM memory in the hardware SDN forwarding devices. We then presented our performance solutions: a coarse grained scheduler and PRoPHYS in Chapter 4 that allow to enhance the flow performance and resilience in SDN and hybrid networks respectively. Finally, we created energy efficient solutions in Chapter 5 where we presented SENAtoR and SEaMLESS. SENAtoR allows to decrease the energy consumption in backbone hybrid networks while preserving network performance even when link failures and/or traffic peaks occur. SEaMLESS allows to migrate the Gateway Process of idle virtual machines to a virtual network function in order to release unused idle resources and shutdown idle VMs to preserve energy while maintaining service availability. We detail those contributions, pinpoint their limitations, and provide potential ideas for future work in the next sections. At the end of this chapter we provide final remarks regarding the current state of SDN, its potentials and the biggest remaining challenges that SDN networks face in our opinion.

6.1 Scalability

To permit SDN physical devices that feature limited and power hungry TCAM memory to scale we created MINNIE. MINNIE is a solution built as a module of the Beacon controller that maximizes

the usage of the limited TCAM space in hardware SDN forwarding devices. When the forwarding rule TCAM limit is reached on an SDN node, MINNIE automatically compresses the SDN device forwarding table on the controller side based on the source or the destination IP addresses. Then, MINNIE transmits the new compressed forwarding table to the SDN device to replace the old forwarding table which used the full TCAM space. Using experimentations and simulations we demonstrated that MINNIE provides a minimum of 70% compression ratio and does not negatively impact the network performance.

When MINNIE transmits the new compressed routing table of a switch, multiple flow insertion events are transmitted to the switch simultaneously which could impact the switch performance in case a peak of new flows is arriving at the switch. In such a case, MINNIE can be modified to send the new compressed routing table when the network utilization is low in order to respect the number of events supported by the switch. MINNIE has been tested on an experimental testbed with either a large number of flows or a high bandwidth due to the limitation of our testbed. Extra tests need to be done to assess the performance of MINNIE with high bandwidth and large number of flows and with traffic peaks to validate the performance of MINNIE before integrating it in the network. This could be done with a testbed featuring several physical SDN switches as, in our case, we had to split our physical switch into several virtual switches to carry out experiments.

6.2 Performance

In this thesis, we tried to enhance the flow performance and resilience in SDN and hybrid networks leveraging the controller centrality and online monitoring of statistics information with our coarse-grained scheduling prototype and our failure resilient solution PRoPHYS.

Our Coarse Grained Scheduling solution pulls the SDN statistics information from the forwarding devices to detect large flows. We devised two schedulers: (i) a State-full and (ii) a Stateless Scheduler. The state-full scheduler detects large flows by pulling every flows' statistics from the SDN devices. On the other hand, the stateless scheduler monitors the client bandwidth utilization. Then, when the client traffic utilization reaches a predefined bandwidth threshold, the scheduler zooms into the client's traffic and uses the state-full scheduler to detect large flows. The large flows are detected by monitoring the number of transmitted packets of every flow. Any flow that has transmitted more than the defined packet threshold set by the administrator is considered as a large flow. After detecting large flows, both variants of this prototype de-prioritize large flows (change from highest priority queue to lowest priority queue) in order to enable short flows to finish quickly. The test results have shown that this solution was efficient on small linear topologies, where all short flows end before long flows using the state-full and stateless schedulers. However, as explained before this scheduler failed on more complex topologies such as fat tree and VL2 topologies.

The current version of our coarse grained scheduling prototype requires to pull the statistics information for all the flows from all the switches in the network and applies the same policy in

all switches. Applying the same policy can lead to degradation of performance flows that have to pass through multiple switches in the network if the flow is either always prioritized on all switches or the flow is always de-prioritized in the network. To solve this, we estimate that the controller scheduling module has to take into consideration the network topology structure, the flow path, the Layer 4 port used by this application and the full network utilization (i.e. utilization of the SDN forwarding nodes and their links). Moreover, for a complete coarse grained scheduling solution, one should take intro consideration multiple queues instead of two, rerouting when high network congestion is detected to use the backup links, plus selective SDN port or flow monitoring approach to decrease the number of control packets transmitted by the controller to the switches and packet delay between SDN nodes.

After creating our coarse grained scheduling prototype we continued leveraging the controller centrality and the statistics information provided by Openflow to enhance the network performance in case of link failure by introducing SDN nodes that will enable to monitor the network using our PRoPHYS solution. PRoPHYS features two techniques (active and passive) to enhance network resilience by decreasing the link failure detection time in hybrid SDN networks. The passive technique estimates link failures by detecting discrepancies between the transmitted and received port statistics of the same set of flows on SDN nodes. This technique first constructs a matrix of communicating ports. It then monitors the transmitted and received statistics of this set of ports. Once this methodology detects that the transmitted statistics are less than the received statistics across a network segment, it supposes the existence of link or node failure. This methodology proved to decrease the link/network failure detection interval by 50%, however, it presents some false positives. The active technique depends on the transmission of a path monitoring packet created by the controller, instead of the switches, which decreases the CPU overhead in the switches. This methodology is faster than traditional failure detection methodologies such as (BFD) since it detects a link or segment failure if the packet is not received after a dynamic moving average timeout that is calculated based on the real delay between the nodes. This methodology also proved to provide a 50% decrease in the number of packets loss compared to classical failure detection methodologies.

Both of our performance solutions have been studied on small scale prototype networks which enabled us to gain a first insight on their performance and their limitations. Again, both solutions would benefit from being tested on large scale ISP topologies using both software and hardware SDN switches with real traffic patterns and distributed controllers to be able to better study the impact of pulling switch statistics on the SDN switches performance, the statistics reply delay and the accuracy of the statistics reply information delivered to the controller.

Our current version of PRoPHYS, also depends on multiple control packets to detect link failure where it depends on the statistics information or the controller packet transmission across the network to detect link failure. This could hence increase the control overhead which is the bottleneck of SDN technology. However, we estimate that in large scale hybrid ISP network, we would have multiple distributed controllers and the controller would be placed in-band (i.e. uses the same network topology constructed by the SDN nodes to communicate with the SDN

switches). Thus, in such a case, one could leverage the controller default control packets to detect link failures and estimate the link delays. The only control packets that would thus have to be transmitted to detect link failures is for the nodes controlled by different controllers that share a link.

Moreover, PRoPHYS currently uses tunnels to reroute the traffic when link failures are detected, which degrades the flow performance due to encapsulation overhead. To maintain the flow performance without incurring routing loops due to the routing table difference between the SDN switches and the legacy routing devices, one could try to propagate the controller routing table scheme used by the SDN devices to the legacy layer 3 routing devices. Moreover, since legacy nodes would require a few milliseconds to propagate the routing information and converge, one could consider buffering the packets either at the SDN switch or on the controller if the switch does not feature buffering.

6.3 Energy Efficiency

Finally, after creating solutions that allow SDN networks to scale to millions of flows and enhance the network performance and resilience, we focused on energy efficient solutions. In Chapter 5, we presented our energy efficient solutions that allow energy efficiency without negatively impacting the network performance (SENAtoR) or service availability (SEaMLESS).

SENAtoR is an energy efficient solution for backbone hybrid networks developed with the COATI team. Similarly to classic energy aware routing algorithms, SENAtoR turns off/on SDN devices based on the traffic utilization. SENAtoR features however three main aspects that allow to preserve network performance and avoid packet loss while turning off network devices, and when sudden link failures or traffic spikes occur. First, to avoid packet loss when turning off network devices, SENAtoR asks the controller to stop sending OSPF hello packets from the SDN switch to its neighboring devices, once the SDN switch is to be turned off or put in sleep mode. After, a duration larger than the OSPF failure detection and convergence period, the controller then puts the corresponding SDN switch in sleep mode which allows to save energy without loosing the previously installed forwarding rules. Second, SENAtoR monitors the network traffic to detect possible link failures or sudden traffic spikes when SDN nodes are turned off. If link/node failures are discovered or if traffic peaks appear, SENAtoR turns on again all previously turned off SDN nodes to prevent network disconnection or data packet loss. Third, SENAtoR uses tunnels to reroute from the SDN nodes neighbors to the correct destination to prevent routing loops due to the differences in the routing table of OSPF and SDN nodes.

Similarly to PRoPHYS, SENAtoR uses tunnels to perform rerouting in hybrid networks. However, as shown in Chapter 4 Table 4.2 tunnels degrade the throughput and increase the delay as they require packet encapsulation and additional processing delay. Thus, as stated in the previous section to maintain the flow performance, one could try to propagate the controller routing table scheme from the SDN nodes to the layer3 legacy nodes so that all the nodes in the network will

have the same routing scheme.

Moreover, SENAtoR performance features have been tested on a small ISP topology using simulation and with the assumption that in an ISP network we can estimate the traffic pattern which gives a first insight on the performance of SENAtoR. However, for a complete view, SENAtoR has to be tested in a real medium to large size ISP network where the network traffic varies and multiple link failures might occur in order to test its efficiency and decipher the threshold increase or decrease in the network traffic that allows to detect possible traffic peaks or link failures. As a first approach, one could consider the dynamic threshold based on the full network current and estimated utilization information, and the network topology and probably flows information.

After describing SENAtoR in Chapter 5 we presented SEaMLESS. SEaMLESS is an architecture created by the Signet team that is still in its infancy. It aims at solving the idle VM problem in current data centers and enterprise clouds. The key issue is mainly how to release all the resources used by the idle VM (e.g. memory and energy) while maintaining the availability of the service provided.

In order to release the memory used by these VMs and to allow server consolidation SEaMLESS migrates the idle VM Gateway Process to a VNF service which allows to turn off the VM machine while maintaining its services reachable. When users try to connect to this VM service, the VNF establishes the connection first, then in case of an attempt to access the VM service data, the orchestrator resumes the idle VM and the VNF would then handle the session back to the VM to be treated. This allows an availability close to 100% of the VM services in data centers and enterprise clouds while optimizing the usage of memory and energy resources.

SEaMLESS holds the promise of allowing a better sever consolidation scheme and liberating memory space in the physical servers. The total energy gains and memory that can be saved in a real enterprise cloud have not been tested yet. Thus additional large scale tests should be done to study better the efficiency of SEaMLESS. Indeed, SEaMLESS is still a project under study, the Signet team is continuing to build the whole SEaMLESS architecture and automatize all the steps required to migrate the idle VM to a VNF. We also want to study its performance in a simulated cloud network using Grid 5000 [Gri] and its integration with data center management solutions like OpenStack or VM consolidation algorithms like Btrplace [The].

Moreover, the current version of SEaMLESS suggests to dump the idle VM to disk to increase the energy saving. However, this increases the data response time of the VM when new connection is established when the VM is dumped. A first approach to decrease the VM response time is to better analyze the connection and user information once a user is connected to the VNF in order to estimate the probability to access the VM resources. Thus, one could consider that upon some connections to the VNF, the VM could be resumed before the user asks for the VM resources.

6.4 Final Remarks

In this chapter, we summarized our solutions that enhance SDN nodes scalability and use SDN to enhance flow performance, resilience and network energy efficiency. SDN is a promising technology that can be used to replace current network devices to enhance the network performance, decrease the configuration, maintenance and energy costs. With the insertion of Data Plane Development Kit (DPDK), simple data plane actions such as forwarding, buffering, and queuing could be modified and adapted to user requirements. Moreover, the programmability of the network and the usage of a centralized controller allows SDN to introduce Machine Learning (ML) and Artificial Intelligence (AI) into the data center, ISP and cloud networks which moves the network into a new era of auto programmable configurable networks that do not require any human intervention.

SDN alongside NFV can already be deployed in the data center and cloud networks to decrease the cost of the network as a single SDN node could replace the action of the switch, router, firewall, DHCP, intrusion detection systems and intrusion prevention systems. Moreover, centralized controllers and distributed controllers– in a small geographical area– can allow online network analysis and auto management. It can also be configured to allow to perform automated decisions regarding VM or VNF migration based on the network and VM or VNF utilization and the global network view. SDN can also be used with OpenStack to automatically add or delete network services or controllers servers based on the online network analysis.

As for the introduction of SDN in ISP networks, it is more cumbersome due to the large geographical distance that might separate the controller from the switches. Though this delay might not negatively impact the performance of the SDN switch as shown in Chapter 4, it could impact the distributed controllers performance that needs to communicate to exchange the network status and information. In such a case, we imagine that ISP networks would not leverage all of the benefits of SDN technology achievable for data center and cloud networks such as online full network analysis and adaptation. Thus, the delay of the communication channel between the control plane and the data plane remains a challenge for SDN.

However, the biggest challenge that SDN still faces, in my opinion, is security. The centralization of the control plane and the use of the Openflow channel to communicate all the topology information, control features and configuration, and the flow information renders the communication channel and the controller the weakest point in the network. Thus, as explained in [KREV+15] new security and encryption methodologies should be created to protect the communication channel and the controller nodes.

Acronyms

AN Active networks. 1

AS Autonomous Systems. 80

ASIC Application-Specific Integrated Circuit. 14, 24, 25, 40

BFD Bidirectional Forwarding Detection. 76, 77, 82, 96, 125

CAB CAching in Buckets. 16

CBR Constant Bit Rate. 84, 86, 104

CDF Cumulative Distribution Function. xix, 73, 86, 104

CLI Command Line Interface. ix, 1, 2

CPU Central Processing Unit. 3, 19, 21, 36, 39, 40

DC Data Centers. ix, x, 1, 112

DPDK Data Plane Development Kit. 17, 128

DRAM Dynamic Random-Access Memory. 16

ECN Explicit Congestion Notification. 19, 68

EF Edge First. 25

FIS flow instruction set. 4

FPGA Field Programmable Gate Array. 16, 17, 68

FRR Fast Rerouting. 76, 78

GPU Graphics Processing Unit. 16

GRE Generic Routing Encapsulation. 83, 103

HLS High Load with (small number of) hardware rules. xvi, 40, 41, 50

HULL High bandwidth and Ultra-Low Latency. 67

ICT Information and Communication Technologies. 93, 94, 112

ISP Internet Service Providers. ix, xiii, 1, 8, 9, 20, 78, 80, 83, 87, 94, 95, 98, 102–104, 106, 107, 112, 125, 127, 128

LAS Least Attained Service. 67

LLDP Link Layer Discovery Protocol. 75, 77

LLS Low Load with (large number of) software rules. xvi, 39–41, 45

LS Link State. 106

LSP Label Switched Path. 78

LXBs Logical Crossbars. 78

MPLS Multi-protocol Label Switching. 76, 78

NaaS Networking as a Service. 116

NETFPGA NETwork Field Programmable Gate Array. 17

NFV Network Function Virtualization. 20, 128

NOS Network Operating System. 5

OvS OpenvSwitch. xvi, xix, 3, 23, 36–38, 42, 66, 69, 70, 74, 103, 116

PDQ Preemptive Distributed Quick. 67

PoP Point of Presence. 98–100, 102–104

PRoPHYS Providing Resilient Path in Hybrid SDN Networks. xi, 8, 9, 66, 74–76, 80, 83, 84, 86, 87, 90, 123–126

QoE Quality of Experience. iii, 75

QoS Quality of Service. 13, 18, 25, 67, 117, 118

RSVP Resource Reservation Protocol. 78

RTT Round Trip Time. 72

SDC Software-Defined Counters. 14

SDDP Segment Discrepancy Detection. 81, 82, 84, 89

SDN Software Defined Networking. iii, ix–xiii, xv–xvii, xix–xxi, 1–9, 11–21, 23–27, 35, 36, 38–42, 44, 65–70, 72, 74–83, 88–91, 93–110, 112, 115, 116, 119, 121, 123–126, 128

SENAtoR Smooth ENergy Aware Routing. xi, xii, 8, 94, 96, 97, 100, 102–104, 106, 107, 110, 112, 121, 123, 126, 127

SLAs Service Level Agreements. 112

SoC System-On-Chip. 17

SPT Shortest Path Tree. 98

SRAM Static Random-Access Memory. 16

STP Spanning Tree Protocol. 18

STT Stateless Transport Tunneling. 83

TCAM Ternary Content Addressable Memory. iii, x, 3, 7, 8, 15, 16, 19, 24–27, 35, 36, 38, 78, 95, 107, 123, 124

TE Traffic Engineering. 78, 107

ToS Type of Service. 69

VLAN Virtual Local Area Network. 36, 39, 97

VM virtual machine. iii, xii, xvii, xx, xxi, 8, 9, 20, 36, 38, 41, 93–97, 112–123, 127, 128

VMs virtual machines. 8, 9

VNF virtual network function. iii, xii, xx, xxi, 8, 9, 94, 97, 112, 113, 115–118, 120–123, 127, 128

Bibliography

[SW] SWI Venture LLC. Green ict: Sustainable computing, media, e-devices. `http://www.vertatique.com/ict-10-global-energy-consumption`.

[AABN04] Konstantin Avrachenkov, Urtzi Ayesta, Patrick Brown, and Eeva Nyberg. Differentiation between short and long TCP flows: Predictability of the response time. In *Proceedings of IEEE INFOCOM 2004*, 2004.

[ACJ+07] D. L. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang. Compressing Rectilinear Pictures and Minimizing Access Control Lists. In *ACM-SIAM SODA*, 2007.

[ADRC14] Kanak Agarwal, Colin Dixon, Eric Rozner, and John Carter. Shadow macs: Scalable label-switching for commodity ethernet. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 157–162. ACM, 2014.

[AFLV08] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.

[AFRR+10] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[AHM+03] Ismail Ari, Bo Hong, Ethan L Miller, Scott A Brandt, and Darrell DE Long. Managing flash crowds on the internet. In *IEEE/ACM MASCOTS 2003*, 2003.

[AKE+12] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 19–19. USENIX Association, 2012.

[AKL13] Sugam Agarwal, Murali Kodialam, and TV Lakshman. Traffic engineering in software defined networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2211–2219. IEEE, 2013.

[ALCP14] Joao Taveira Araújo, Raúl Landa, Richard G Clegg, and George Pavlou. Software-defined network support for transport resilience. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–8. IEEE, 2014.

[ASP05] Alia Atlas, George Swallow, and Ping Pan. Fast Reroute Extensions to RSVP-TE for LSP Tunnels. RFC 4090, May 2005.

[B. 04] B. Claise. RFC 3954 - Cisco Systems NetFlow Services Export Version 9, 2004.

[BAAZ10] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *Computer Communication Review*, 40(1), 2010.

[BAAZ11] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 8. ACM, 2011.

[BAB12] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems*, 28(5):755 – 768, 2012. Special Section: Energy efficiency in large-scale distributed systems.

[BAM09] Theophilus Benson, Aditya Akella, and David A Maltz. Unraveling the complexity of network management. In *NSDI*, pages 335–348, 2009.

[BB10a] Anton Beloglazov and Rajkumar Buyya. Energy efficient allocation of virtual machines in cloud data centers. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 577–578. IEEE, 2010.

[BB10b] Anton Beloglazov and Rajkumar Buyya. Energy efficient resource management in virtualized cloud data centers. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 826–831, Washington, DC, USA, 2010. IEEE Computer Society.

[BBD+13] Raffaele Bolla, Roberto Bruschi, Franco Davoli, Lorenzo Di Gregorio, Pasquale Donadio, Leonardo Fialho, Martin Collier, Alfio Lombardo, Diego Reforgiato Recupero, and Tivadar Szemethy. The green abstraction layer: A standard power-management interface for next-generation network devices. *IEEE Internet Computing*, 17(2):82–86, 2013.

[BBDL15] Raffaele Bolla, Roberto Bruschi, Franco Davoli, and Chiara Lombardo. Fine-grained energy-efficient consolidation in sdn networks and devices. *IEEE Transactions on Network and Service Management*, 12(2):132–145, 2015.

[BBK15] Andreas Blenk, Arsany Basta, and Wolfgang Kellerer. Hyperflex: An sdn virtu-

alization architecture with flexible hypervisor function allocation. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 397–405. IEEE, 2015.

[BBRF14] Fábio Botelho, Alysson Bessani, Fernando MV Ramos, and Paulo Ferreira. On the design of practical fault-tolerant sdn controllers. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 73–78. IEEE, 2014.

[BBU+09] Deborah Brungard, Malcolm Betts, Satoshi Ueno, Ben Niven-Jenkins, and Nurit Sprecher. Requirements of an MPLS Transport Profile. RFC 5654, September 2009.

[BCW+15] Wei Bai, Kai Chen, Hao Wang, Li Chen, Dongsu Han, and Chen Tian. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 455–468, Oakland, CA, May 2015. USENIX Association.

[BD14] R Bifulco and M Dusi. Reactive logic in software-defined networking: Accounting for the limitations of the switches. In *Third European Workshop on Software Defined Networks*, page 6, 2014.

[BDG12] Ed. B. Davie and J. Gross. A Stateless Transport Tunneling Protocol for Network Virtualization(STT). IETF draft-davie-stt-01, March 2012.

[BEBJ15] S. Bensley, L. Eggert, D. Thalerand P. Balasubramanian, and G. Judd. Microsoft's Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters draft-bensley-tcpm-dctcp-05. Internet-Draft, July 2015.

[BGK+13a] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 99–110. ACM, 2013.

[BGK+13b] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM*. ACM, 2013.

[BK14] S. Banerjee and K. Kannan. Tag-in-tag: Efficient flow table management in sdn switches. In *CNSM*, 2014.

[BM14] W. Braun and M. Menth. Wildcard compression of inter-domain routing tables for openflow-based software-defined networking. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 25–30, Sept 2014.

[BRC+13] Md Faizul Bari, Arup Raton Roy, Shihabur Rahman Chowdhury, Qi Zhang, Mohamed Faten Zhani, Reaz Ahmed, and Raouf Boutaba. Dynamic controller provisioning in software defined networks. In *Network and Service Management (CNSM), 2013 9th International Conference on*, pages 18–25. IEEE, 2013.

[Bri14] ONF Solution Brief. SDN Migration Considerations and Use Cases. Technical Report ONF TR - 506, Open Network Foundation, November 2014.

[BRL+14] Jeremias Blendin, Julius Rückert, Nicolai Leymann, Georg Schyguda, and David Hausheer. Position paper: Software-defined network service chaining. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 109–114. IEEE, 2014.

[CAJ+12] Carlos HA Costa, Marcelo C Amaral, Guilherme C Januário, Tereza CMB Carvalho, and Catalin Meirosu. Sustnms: Towards service oriented policy-based network management for energy-efficiency. In *Sustainable Internet and ICT for Sustainability (SustainIT), 2012*, pages 1–5. IEEE, 2012.

[CEL+12] Antonio Cianfrani, Vincenzo Eramo, Marco Listanti, Marco Polverini, and Athanasios V Vasilakos. An ospf-integrated routing strategy for qos-aware energy saving in ip backbone networks. *IEEE Transactions on Network and Service Management*, 9(3):254–267, 2012.

[CFY04] Yan Chen, Toni Farley, and Nong Ye. Qos requirements of network applications on the internet. *Inf. Knowl. Syst. Manag.*, 4(1):55–76, January 2004.

[Cha13] Charter: Forwarding Abstractions Working Group, 2013.

[Cis] Cisco 1941 series integrated services routers data sheet.

[Cis12] I Cisco. Cisco visual networking index: Forecast and methodology, 2011–2016. *CISCO White paper*, pages 2011–2016, 2012.

[Cis16] Cisco visual networking index: Forecast and methodology, 2016–2021. Technical report, Cisco, June 2016.

[CKY11] Andrew R Curtis, Wonho Kim, and Praveen Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM, 2011 Proceedings IEEE*, pages 1629–1637. IEEE, 2011.

[CLENR14] R. Cohen, L. Lewin-Eytan, J.S. Naor, and D. Raz. On the effect of forwarding table size on sdn network utilization. In *INFOCOM*. IEEE, 2014.

[CMFA14] Paul T Congdon, Prasant Mohapatra, Matthew Farrens, and Venkatesh Akella. Simultaneously reducing latency and power consumption in openflow switches. *IEEE/ACM Transactions on Networking (TON)*, 2014.

[CMN09] Luca Chiaraviglio, Marco Mellia, and Fabio Neri. Energy-aware backbone networks: a case study. In *IEEE ICC*, 2009.

[CMN12] Luca Chiaraviglio, Marco Mellia, and Fabio Neri. Minimizing isp network energy cost: Formulation and solutions. *IEEE/ACM Transactions on Networking (TON)*, 20(2):463–476, 2012.

[CMT+11] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):254–265, August 2011.

[CMZ+07] Baek-Young Choi, Sue Moon, Zhi-Li Zhang, Konstantina Papagiannaki, and Christophe Diot. Analysis of point-to-point packet delay in an operational network. *Computer networks*, 51(13), 2007.

[COA] Coati. `https://team.inria.fr/coati/`.

[Con02] Paul Congdon. Link layer discovery protocol. RFC 2922, July 2002.

[COS] Sdn system performance.

[CRN+10] Ken Christensen, Pedro Reviriego, Bruce Nordman, Michael Bennett, Mehrgan Mostowfi, and Juan Antonio Maestro. Ieee 802.3 az: the road to energy efficient ethernet. *IEEE Communications Magazine*, 48(11), 2010.

[CRS+13] Prasad Calyam, Sudharsan Rajagopalan, Arunprasath Selvadhurai, Saravanan Mohan, Aishwarya Venkataraman, Alex Berryman, and Rajiv Ramnath. Leveraging openflow for resource placement of virtual desktop cloud applications. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 311–319. IEEE, 2013.

[CSB+08] J. Chabarek, J. Sommers, P. Barford, C. Estan, D. Tsiang, and S. Wright. Power awareness in network design and routing. In *IEEE INFOCOM*, 2008.

[CSS14] C Jasson Casey, Andrew Sutton, and Alex Sprintson. tinynbi: Distilling an api from essential openflow abstractions. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 37–42. ACM, 2014.

[CXLC15] Cing-Yu Chu, Kang Xi, Min Luo, and H Jonathan Chao. Congestion-aware single link failure recovery in hybrid sdn networks. In *IEEE INFOCOM*, 2015.

[D-L09] D-Link. Green computing and dlink (last accessed sep 2016), Feb 2009.

[Dav] Dave Plonka. Internet traffic flow size analysis. `http://net.doit.wisc.edu/data/flow/size/`.

[DDW+10] Avri Doria, Ligang Dong, Weiming Wang, Hormuzd M. Khosravi, Jamal Hadi

Salim, and Ram Gopal. Forwarding and Control Element Separation (ForCES) Protocol Specification. RFC 5810, March 2010.

[dep15] Facebook, google use sdn to boost data center connectivity. http://searchsdn.techtarget.com/tip/Facebook-Google-use-SDN-to-boost-data-center-connectivity, March 2015.

[DHM+14] Advait Abhay Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Kompella. Elasticon: an elastic distributed sdn controller. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 17–28. ACM, 2014.

[DLP17] M. Rifai A. Segalini M. Dione G. Urvoy-Keller D. Lopez Pacheco, Q. Jacquemart. Seamless: a lightweight service migration cloud architecture for energy saving capabilities. 2017.

[Doc] Docker. Build, ship, and run any app, anywhere. https://www.docker.com/.

[Dpd14] Intel Data Plane Development Kit (Intel DPDK) with VMware Vsphere , 2014.

[DPP+10] Tathagata Das, Pradeep Padala, Venkat Padmanabhan, Ramachandran Ramjee, and Kang G. Shin. Litegreen: Saving energy in networked desktops using virtualization. In *Annual Technical Conference*. USENIX, June 2010.

[DRGF12] Floriano De Rango, Francesca Guerriero, and Peppino Fazio. Link-stability and energy aware routing protocol in distributed wireless networks. *IEEE Transactions on Parallel and Distributed systems*, 23(4), 2012.

[dSdF16] Rodrigo A. C. da Silva and Nelson L. S. da Fonseca. Topology-aware virtual machine placement in data centers. *Journal of Grid Computing*, 14(1):75–90, 2016.

[EA15] Can Eyupoglu and Muhammed Ali Aydin. Energy efficiency in backbone networks. *Procedia-Social and Behavioral Sciences*, 195:1966–1970, 2015.

[EBBS11] Rob Enns, Martin Bjorklund, Andy Bierman, and Jürgen Schönwälder. Network Configuration Protocol (NETCONF). RFC 6241, June 2011.

[EFSM+11] Omar El Ferkouss, Ilyas Snaiki, Omar Mounaouar, Hamza Dahmouni, Racha Ben Ali, Yves Lemieux, and Cherkaoui Omar. A 100gig network processor platform for openflow. In *Network and Service Management (CNSM), 2011 7th International Conference on*, pages 1–4. IEEE, 2011.

[ENE] ENERGY STAR. Top 12 ways to decrease the energy consumption of your data center. https://www.energystar.gov/sites/default/files/asset/document/DataCenter-Top12-Brochure-Final.pdf.

[Eri13] David Erickson. The beacon openflow controller. In *HotSDN*, pages 13–18. ACM, 2013.

[ETHG05] Shirin Ebrahimi-Taghizadeh, Ahmed Helmy, and Sandeep Gupta. Tcp vs. tcp: a systematic study of adverse impact of short-lived tcp flows on long-lived tcp flows. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 2, pages 926–937. IEEE, 2005.

[Eur] European Union. eurostat statistics explained- consumption of energy. `http://ec.europa.eu/eurostat/statistics-explained/index.php/Consumption_of_energy`.

[FDA+04] Wissam Fawaz, Belkacem Daheb, Olivier Audouin, M Du-Pond, and Guy Pujolle. Service level agreement and provisioning in optical networks. *IEEE Communications Magazine*, 42(1), 2004.

[FFEB05] Pierre Francois, Clarence Filsfils, John Evans, and Olivier Bonaventure. Achieving sub-second IGP convergence in large IP networks. *ACM SIGCOMM CCR*, 35(3):35–44, 2005.

[FHMT00] Dino Farinacci, Stanley P. Hanks, David Meyer, and Paul S. Traina. Generic Routing Encapsulation (GRE). RFC 2784, March 2000.

[GB13] Minzhe Guo and Prabir Bhattacharya. Controller placement for improving resilience of software-defined networks. In *Networking and Distributed Computing (ICNDC), 2013 Fourth International Conference on*, pages 23–27. IEEE, 2013.

[GC06] Chamara Gunaratne and Ken Christensen. Ethernet adaptive link rate: System design and performance evaluation. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 28–35. IEEE, 2006.

[GHJ+09] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, volume 39:4, pages 51–62. ACM, 2009.

[GHM15] Frédéric Giroire, Frédéric Havet, and Joanna Moulierac. Compressing two-dimensional routing tables with order. In *INOC*, 2015.

[GLL+09] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: A high performance, server-centric network architecture for modular data centers. *SIGCOMM Comput. Commun. Rev.*, 39(4):63–74, August 2009.

[GMP14] Frédéric Giroire, Joanna Moulierac, and T Khoa Phan. Optimizing rule placement

in software-defined networks for energy-aware routing. In *GLOBECOM*. IEEE, 2014.

[GNTD03] Frédéric Giroire, Antonio Nucci, Nina Taft, and Christophe Diot. Increasing the robustness of ip backbones in the absence of optical level protection. In *IEEE INFOCOM*, 2003.

[Gri] Grid 5000. `https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home`.

[GTB+14] W. Godycki, C. Torng, I. Bukreyev, A. Apsel, and C. Batten. Enabling realistic fine-grain voltage scaling with reconfigurable power distribution networks. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 381–393, Dec 2014.

[GWT+08] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: A scalable and fault-tolerant network structure for data centers. *SIGCOMM Comput. Commun. Rev.*, 38(4):75–86, August 2008.

[GYG12] Monia Ghobadi, Soheil Hassas Yeganeh, and Yashar Ganjali. Rethinking end-to-end congestion control in software-defined networks. In *Proceedings of HotNets-XI*, 2012.

[Ham15] B. Hamzeh. Network failure detection and prediction using signal measurements. US Patent 9,100,339, August 4 2015.

[HCG12] Chi-Yao Hong, Matthew Caesar, and P Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review*, 42(4):127–138, 2012.

[HCW+15] Shuihai Hu, Kai Chen, Haitao Wu, Wei Bai, Chang Lan, Hao Wang, Hongze Zhao, and Chuanxiong Guo. Explicit path control in commodity data centers: Design and applications. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 15–28, Berkeley, CA, USA, 2015. USENIX Association.

[Hew16] Hewlett Packard Enterprise Development LP. Bidirectional forwarding detection (BFD), 2016.

[HHG+13] David Hock, Matthias Hartmann, Steffen Gebert, Michael Jarschel, Thomas Zinner, and Phuoc Tran-Gia. Pareto-optimal resilient controller placement in sdn-based core networks. In *Teletraffic Congress (ITC), 2013 25th International*, pages 1–9. IEEE, 2013.

[HJW+11] L. Huang, Q. Jia, X. Wang, S. Yang, and B. Li. Pcube: Improving power efficiency in data center networks. In *Cloud Computing (CLOUD), 2011 IEEE International*

Conference on, pages 65–72, July 2011.

[HKGJ+15] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. Measuring control plane latency in sdn-enabled switches. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 25. ACM, 2015.

[HMBM16] David Ke Hong, Yadi Ma, Sujata Banerjee, and Z. Morley Mao. Incremental deployment of SDN in hybrid enterprise and ISP networks. In *ACM Symposium on SDN Research*, 2016.

[hp5] Hp procurve switch 5400zl series.

[HPO13] Hp openflow and sdn technical overview.e. `http://h17007.www1.hpe.com/docs/networking/solutions/sdn/devcenter/02_-_HP_OpenFlow_and_SDN_Technical_Overview_TSG_v1_2013-10-01.pdf`, October 2013.

[HRG+17] Nicolas Huin, Myriana Rifai, Frédéric Giroire, Dino Lopez Pacheco, Guillaume Urvoy-Keller, and Joanna Moulierac. *Bringing Energy Aware Routing closer to Reality with SDN Hybrid Networks*. PhD thesis, INRIA Sophia Antipolis-I3S; I3S, 2017.

[HRX08] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[HSM+10a] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 17–17, Berkeley, CA, USA, 2010. USENIX Association.

[HSM+10b] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In *Nsdi*, volume 10, pages 249–264, 2010.

[HSM12] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 7–12. ACM, 2012.

[HSS+] Luuk Hendriks, Ricardo de O Schmidt, Ramin Sadre, Jeronimo A Bezerra, and Aiko Pras. Assessing the quality of flow measurements from openflow devices. *TMA 2016*.

[HUKDB10] Martin Heusse, Guillaume Urvoy-Keller, Andrzej Duda, and Timothy X Brown. Least attained recent service for packet scheduling over wireless lans. In *World of*

Wireless Mobile and Multimedia Networks (WoWMoM), 2010 IEEE International Symposium on a, pages 1–9. IEEE, 2010.

[HWG+13] Yannan Hu, Wang Wendong, Xiangyang Gong, Xirong Que, and Cheng Shiduan. Reliability-aware controller placement for software-defined networks. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 672–675. IEEE, 2013.

[HYG+15] Chengchen Hu, Ji Yang, Zhimin Gong, Shuoling Deng, and Hongbo Zhao. Desktopdc: setting all programmable data center networking testbed on desk. *ACM SIGCOMM Computer Communication Review*, 44(4):593–594, 2015.

[ICC+16] Filip Idzikowski, Luca Chiaraviglio, Antonio Cianfrani, Jorge López Vizcaíno, Marco Polverini, and Yabin Ye. A survey on energy-aware design and operation of core networks. *IEEE Communications Surveys & Tutorials*, 18(2), 2016.

[IDGM01] Gianluca Iannaccone, Christophe Diot, Ian Graham, and Nick McKeown. Monitoring very high speed links. In *ACM IMW*, 2001.

[IMP] Impt. `http://ipmt.forge.imag.fr/`.

[Int11] Software Defined Networking and Softwarebased Services with Intel Processors , 2011.

[Int12] International Computer Science Institute, UC Berkeley . The nox controller. `https://github.com/noxrepo/nox`, 2012.

[Iza15] Ryan Izard. Floodlight documentation. `https://floodlight.atlassian.net/wiki/`, march 2015.

[JAG+14] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 3–14, New York, NY, USA, 2014. ACM.

[JCPG14] Yury Jimenez, Cristina Cervello-Pastor, and Aurelio J Garcia. On the controller placement for designing a distributed sdn control layer. In *Networking Conference, 2014 IFIP*, pages 1–9. IEEE, 2014.

[JKM+13] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):3–14, August 2013.

[JLX+15] Cheng Jin, Cristian Lumezanu, Qiang Xu, Zhi-Li Zhang, and Guofei Jiang. Telekinesis: Controlling legacy switch routing with openflow in hybrid networks.

In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 20:1–20:7, New York, NY, USA, 2015. ACM.

[JP12] Michael Jarschel and Rastin Pries. An openflow-based energy-efficient data center approach. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 87–88, New York, NY, USA, 2012. ACM.

[KARW14] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Infinite cacheflow in software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 175–180. ACM, 2014.

[KARW16] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proc. ACM Symposium on SDN Research (SOSR)*, 2016.

[KB13a] K. Kannan and S. Banerjee. Compact TCAM: Flow Entry Compaction in TCAM for Power Aware SDN. In *ICDCN*, 2013.

[KB13b] Kalapriya Kannan and Subhasis Banerjee. Compact tcam: Flow entry compaction in tcam for power aware sdn. In *International Conference on Distributed Computing and Networking*, pages 439–444. Springer, 2013.

[KCG+10] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[KCGJ14] Anand Krishnamurthy, Shoban P Chandrabose, and Aaron Gember-Jacobson. Pratyaastha: An efficient elastic distributed sdn control plane. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 133–138. ACM, 2014.

[KGB13] Atefeh Khosravi, Saurabh Kumar Garg, and Rajkumar Buyya. Energy and carbon-efficient placement of virtual machines in distributed cloud data centers. In *European Conference on Parallel Processing*, pages 317–328. Springer, 2013.

[KGCR12] Eric Keller, Soudeh Ghorbani, Matt Caesar, and Jennifer Rexford. Live migration of an entire network (and its hosts). In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 109–114. ACM, 2012.

[KHK13] Yossi Kanizo, David Hay, and Isaac Keslassy. Palette: Distributing tables in software-defined networks. In *INFOCOM*. IEEE, 2013.

[KLRW13] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *CoNEXT*. ACM,

2013.

[KPK14] Maciej Kuzniar, P Perešıni, and Dejan Kostic. What you need to know about sdn control and data planes. *EPFL, TR*, 199497, 2014.

[KRA12] Jungsoo Kim, Martino Ruggiero, and David Atienza. Free cooling-aware dynamic power management for green datacenters. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 140–146. IEEE, 2012.

[KREV+15] D. Kreutz, F.M.V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1), Jan 2015.

[KSP+14] Masayoshi Kobayashi, Srini Seetharaman, Guru Parulkar, Guido Appenzeller, Joseph Little, Johan Van Reijendam, Paul Weissmann, and Nick McKeown. Maturing of openflow and software-defined networking through deployments. *Computer Networks*, 61:151–175, 2014.

[KVM] Kvm. `https://www.linux-kvm.org/page/Main_Page`.

[KW10] Dave Katz and David Ward. Bidirectional Forwarding Detection (BFD) for Multihop Paths. RFC 5883, June 2010.

[KW15] Dave Katz and David Ward. Bidirectional Forwarding Detection (BFD). RFC 5880, October 2015.

[KZFR15] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 4. ACM, 2015.

[LCB+12] Zhenhua Liu, Yuan Chen, Cullen Bash, Adam Wierman, Daniel Gmach, Zhikui Wang, Manish Marwah, and Chris Hyser. Renewable and cooling aware workload management for sustainable data centers. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 175–186. ACM, 2012.

[LCD04] Anukool Lakhina, Mark Crovella, and Christophe Diot. Characterization of network-wide anomalies in traffic flows. In *ACM IMC*, 2004.

[LCD05] Anukool Lakhina, Mark Crovella, and Christophe Diot. Mining anomalies using traffic feature distributions. In *ACM Computer Communication Review*, volume 35, 2005.

[LCMO09] Yan Luo, Pablo Cascon, Eric Murray, and Julio Ortega. Accelerating openflow switching with network processors. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 70–71.

ACM, 2009.

[LCS+14] Dan Levin, Marco Canini, Stefan Schmid, Fabian Schaffert, Anja Feldmann, et al. Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks. In *USENIX Annual Technical Conference*, pages 333–345, 2014.

[LGZ+15] Stanislav Lange, Steffen Gebert, Thomas Zinner, Phuoc Tran-Gia, David Hock, Michael Jarschel, and Marco Hoffmann. Heuristic approaches to the controller placement problem in large scale sdn networks. *IEEE Transactions on Network and Service Management*, 12(1):4–17, 2015.

[Lin15] Onos joins the linux foundation, becoming an opendaylight sibling. `https://www.sdxcentral.com/articles/news/onos-joins-the-linux-foundation-becoming-an-opendaylight-sibling/2015/10/`, 2015.

[LLJ15] Jiaqiang Liu, Yong Li, and Depeng Jin. Sdn-based live vm migration across datacenters. *ACM SIGCOMM Computer Communication Review*, 44(4):583–584, 2015.

[LLW+11] Zhenhua Liu, Minghong Lin, Adam Wierman, Steven H Low, and Lachlan LH Andrew. Greening geographical load balancing. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 233–244. ACM, 2011.

[LZH+14] Yanbiao Li, Dafang Zhang, Kun Huang, Dacheng He, and Weiping Long. A memory-efficient parallel routing lookup model with fast updates. *Computer Communications*, 38:60–71, 2014.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2), March 2008.

[MC11] Mehrgan Mostowfi and Ken Christensen. Saving energy in lan switches: New methods of packet coalescing for energy efficient ethernet. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8. IEEE, 2011.

[MC12] Jeffrey C Mogul and Paul Congdon. Hey, you darned counters!: get off my asic! In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 25–30. ACM, 2012.

[Min] Mininet. `http://mininet.org/`.

[MLP+16] James McCauley, Zhi Liu, Aurojit Panda, Teemu Koponen, Barath Raghavan, Jennifer Rexford, and Scott Shenker. Recursive SDN for carrier networks. *ACM*

SIGCOMM CCR, 46(4), 2016.

[MLT10] C. R. Meiners, A. X. Liu, and E. Torng. TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs. In *IEEE/ACM Transaction in Networking*, 2010.

[MLT12] C.R. Meiners, A.X. Liu, and E. Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in tcams. *Networking, IEEE/ACM Transactions on*, 20(2), April 2012.

[MOS] What are sdn southbound apis? `https://www.sdxcentral.com/sdn/definitions/southbound-interface-api/`.

[Moy98] John T. Moy. OSPF Version 2. RFC 2328, April 1998.

[MQU+13] Ali Munir, Ihsan A Qazi, Zartash A Uzmi, Aisha Mushtaq, Saad N Ismail, M Safdar Iqbal, and Basma Khan. Minimizing flow completion times in data centers. In *INFOCOM, 2013 Proceedings IEEE*, pages 2157–2165. IEEE, 2013.

[MR17] Quentin Jacquemart Guillaume Urvoy-Keller Myriana Rifai, Dino Lopez. Prophys: Providing resilient path in hybrid software defined networks. 2017.

[MVL+13] G Memon, M Varvello, R Laufer, T Lakshman, J Li, and M Zhang. Flashflow: a gpu-based fully programmable openflow switch. *University of Oregon, Tech. Rep*, 2013.

[NCC10] Eugene Ng, Z Cai, and AL Cox. Maestro: A system for scalable openflow control. *Rice University, Houston, TX, USA, TSEN Maestro-Techn. Rep, TR10-08*, 2010.

[NEC+08] Jad Naous, David Erickson, G Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an openflow switch on the netfpga platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 1–9. ACM, 2008.

[Neu] Neutron. Neutron. `https://wiki.openstack.org/wiki/Neutron`.

[NMN+14] Bruno Astuto A Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014.

[NSBT15] Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti. OFFICER: A general Optimization Framework for OpenFlow Rule Allocation and Endpoint Policy Enforcement. In *INFOCOM*. IEEE, April 2015.

[Ono] Onos. `http://onosproject.org/`.

[Opea] Opencontrail. `http://www.opencontrail.org/`.

[Opeb] Opendaylight. https://www.opendaylight.org/.

[PA00] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), November 2000. Obsoleted by RFC 6298.

[PIC] Pica8 white box sdn.

[PLBMAL15] J. A. Pascual, T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano. Towards a greener cloud infrastructure management using optimized placement policies. *Journal of Grid Computing*, 13(3):375–389, 2015.

[PMK13] Gergely Pongrácz, Laszlo Molnar, and Zoltán Lajos Kis. Removing roadblocks from sdn: Openflow software switch performance on intel dpdk. In *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pages 62–67. IEEE, 2013.

[POX15] Pox wiki. https://openflow.stanford.edu/display/ONL/POX+Wikix, 2015.

[PPK+15] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[Qem] Qemu. http://www.qemu.org/.

[Qua] Quagga. http://www.nongnu.org/quagga/.

[RBU05] Idris A. Rai, Ernst W. Biersack, and Guillaume Urvoy-Keller. Size-based scheduling to improve the performance of short TCP flows. *IEEE Network*, 19(1), 2005.

[RGK+02] Matthew Roughan, Albert Greenberg, Charles Kalmanek, Michael Rumsewicz, Jennifer Yates, and Yin Zhang. Experience in measuring backbone traffic variability: Models, metrics, measurements and meaning. In *ACM IMW*, 2002.

[RHC+15] Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frédéric Giroire, D Lopez-Pacheco, Joanna Moulierac, and Guillaume Urvoy-Keller. Too many sdn rules? compress them with minnie. In *Global Communications Conference (GLOBECOM), 2015 IEEE*, pages 1–7. IEEE, 2015.

[RHC+17] Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frederic Giroire, Joanna Moulierac, Dino Lopez Pacheco, and Guillaume Urvoy-Keller. Minnie: an sdn world with few compressed forwarding rules. *Computer Networks*, 2017.

[RJK+12] Ahmad Rostami, Tobias Jungel, Andreas Koepsel, Hagen Woesner, and Adam Wolisz. Oran: Openflow routers for academic networks. In *High Performance*

Switching and Routing (HPSR), 2012 IEEE 13th International Conference on, pages 216–222. IEEE, 2012.

[RLL12] Ramya Raghavendra, Jorge Lobo, and Kang-Won Lee. Dynamic graph query primitives for sdn-based cloud network management. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 97–102. ACM, 2012.

[RLPUK15] Myriana Rifai, Dino Lopez-Pacheco, and Guillaume Urvoy-Keller. Coarse-grained scheduling with software-defined networking switches. *ACM SIGCOMM Computer Communication Review*, 45(4):95–96, 2015.

[RMR13] Ramon Marques Ramos, Magnos Martinello, and Christian Esteve Rothenberg. Slickflow: Resilient source routing in data center networks unlocked by openflow. In *Local Computer Networks (LCN), 2013 IEEE 38th Conference on*, pages 606–613. IEEE, 2013.

[RPUK14] Myriana Rifai, Dino Lopez Pacheco, and Guillaume Urvoy-Keller. Towards enabling green routing services in real networks. In *Green Communications (Online-Greencomm), 2014 IEEE Online Conference on*, pages 1–7. IEEE, 2014.

[RRJ+15] Bruno B Rodrigues, Ana C Riekstin, Guilherme C Januário, Viviane T Nascimento, Tereza CMB Carvalho, and Catalin Meirosu. Greensdn: Bringing energy efficiency to an sdn emulation environment. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 948–953. IEEE, 2015.

[RSKK16] Piotr Rygielski, Marian Seliuchenko, Samuel Kounev, and Mykhailo Klymash. Performance analysis of sdn switches with hardware and software flow tables. In *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools (ValueTools 2016)*, 2016.

[RSV87] Richard L Rudell and Alberto Sangiovanni-Vincentelli. Multiple-valued minimization for pla optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–750, 1987.

[Ryu17] Ryu SDN Framework Community. Build sdn agilely. `https://osrg.github.io/ryu/`, 2017.

[SB10] Mike Shand and Stewart Bryant. IP Fast Reroute Framework. RFC 5714, January 2010.

[SCF+12a] Brent Stephens, Alan Cox, Wes Felter, Colin Dixon, and John Carter. Past: Scalable ethernet for data centers. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 49–60, New York, NY, USA, 2012. ACM.

[SCF+12b] Brent Stephens, Alan Cox, Wes Felter, Colin Dixon, and John Carter. Past: Scal-

able ethernet for data centers. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 49–60. ACM, 2012.

[Sch] Tom Scholl. BFD: Does it work and is it worth it? NANOG 45.

[SCPA11] Matt Sargent, Jerry Chu, Vern Paxson, and Mark Allman. Computing TCP Retransmission Timer. RFC 6298, June 2011.

[SGC+13] Andrea Sgambelluri, Alessio Giorgetti, Filippo Cugini, Francesco Paolucci, and Piero Castoldi. Openflow-based segment protection in ethernet networks. *Journal of Optical Communications and Networking*, 5(9):1066–1075, 2013.

[SIG] Signet. `http://signet.i3s.unice.fr/`.

[SK14] Arne Schwabe and Holger Karl. Using mac addresses as efficient routing labels in data centers. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 115–120. ACM, 2014.

[SKZ08] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. Energy aware consolidation for cloud computing. In *Proceedings of the 2008 conference on Power aware computing and systems*, volume 10, pages 1–5. San Diego, California, 2008.

[SLX10] Yunfei Shang, Dan Li, and Mingwei Xu. Energy-aware routing in data center network. In *ACM workshop on Green networking*, 2010.

[SO14] N. M. Sahri and Koji Okamura. Fast failover mechanism for software defined networking: Openflow based. In *Proceedings of The Ninth International Conference on Future Internet Technologies*, CFI '14, pages 16:1–16:2, New York, NY, USA, 2014. ACM.

[Son13] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 127–132, New York, NY, USA, 2013. ACM.

[SSC+13] Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester. Openflow: Meeting carrier-grade recovery requirements. *Elsevier Computer Communications*, 36(6):656–665, 2013.

[SSC+16] Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester. In-band control, queuing, and failure recovery functionalities for openflow. *IEEE Network*, 30(1), 2016.

[SSD+14] M Said Seddiki, Muhammad Shahbaz, Sean Donovan, Sarthak Grover, Miseon Park, Nick Feamster, and Ye-Qiong Song. Flowqos: Qos for the rest of us. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 207–208. ACM, 2014.

[SWSB13] Anirudh Sivaraman, Keith Winstein, Suvinay Subramanian, and Hari Balakrishnan. No silver bullet: Extending sdn to the data plane. In *Proceedings of HotNets-XII*, 2013.

[SZZ+13] Alexander Shalimov, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, and Ruslan Smeliansky. Advanced study of SDN/openflow controllers. In *CEE-SECR*. ACM, 2013.

[TAG16] Mehmet Fatih Tuysuz, Zekiye Kubra Ankarali, and Didem Gözüpek. A survey on energy efficiency in software defined networks. *Computer Networks*, 2016.

[TG10] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pages 3–3, 2010.

[TGG+12] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. On controller performance in software-defined networks. *Hot-ICE*, 12:1–6, 2012.

[The] The BtrPlace Project. An open-source flexible virtual machine scheduler. `http://www.btrplace.org/`.

[The12] The Open Networking Foundation. OpenFlow Switch Specification, Jun. 2012.

[THJ16] D. Taht J. Gettys T. Hoeiland-Joergensen, P. McKenney. The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm. draft-ietf-aqm-fq-codel-06, 2016.

[THW+13] Fung Po Tso, Gregg Hamilton, Rene Weber, Colin S. Perkins, and Dimitrios P. Pezaros. Longer is better: Exploiting path diversity in data center networks. In *Proceedings of IEEE ICDCS '13*, 2013.

[TNW96] Michael Theobald, Steven M. Nowick, and Tao Wu. Espresso-hf: A heuristic hazard-free minimizer for two-level logic. In *Proceedings of the 33rd Annual Design Automation Conference*, DAC '96, pages 71–76, New York, NY, USA, 1996. ACM.

[Top] The internet topology zoo. `http://www.topology-zoo.org/dataset.html`.

[TP13] Fung Po Tso and D.P. Pezaros. Baatdaat: Measurement-based flow scheduling for cloud data centers. In *Computers and Communications (ISCC), 2013 IEEE Symposium on*, July 2013.

[TSS+97] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35:80–86, 1997.

[UR10] Mueen Uddin and Azizah Abdul Rahman. Server consolidation: An approach to make data centers energy efficient and green. *arXiv preprint arXiv:1010.5037*, 2010.

[VLQ+14] TH Vu, VC Luc, NT Quan, T Thanh, NH Thanh, and PN Nam. Sleep mode and wakeup method for openflow switches. *Journal of Low Power Electronics*, 10(3):347–353, 2014.

[VRV14] Allan Vidal, Christian Esteve Rothenberg, and Fábio Luciano Verdi. The libfluid openflow driver implementation. In *Proc. 32nd Brazilian Symp. Comp. Netw.(SBRC)*, pages 1029–1036, 2014.

[VSZ+11] Arun Vishwanath, Vijay Sivaraman, Zhi Zhao, Craig Russell, and Marina Thottan. Adapting router buffers for energy efficiency. In *ACM CoNEXT*, 2011.

[VVB14] Stefano Vissicchio, Laurent Vanbever, and Olivier Bonaventure. Opportunities and research challenges of hybrid software defined networks. *ACM SIGCOMM Computer Communication Review*, 44(2), 2014.

[VVC+17] Stefano Vissicchio, Laurent Vanbever, Luca Cittadini, Geoffrey G Xie, and Olivier Bonaventure. Safe update of hybrid sdn networks. *IEEE/ACM Transactions on Networking*, 2017.

[WNS12] Guohui Wang, TS Ng, and Anees Shaikh. Programming your network at run-time for big data applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 103–108. ACM, 2012.

[WWW+14] Zhiming Wang, Jiangxing Wu, Yu Wang, Ning Qi, and Julong Lan. Survivable virtual network mapping using optimal backup topology in virtualized sdn. *China Communications*, 11(2):26–37, 2014.

[WZS+13] Ye Wang, Yueping Zhang, Vishal Singh, Cristian Lumezanu, and Guofei Jiang. Netfuse: Short-circuiting traffic surges in the cloud. In *IEEE ICC*, 2013.

[WZV+14] Lin Wang, Fa Zhang, Athanasios V. Vasilakos, Chenying Hou, and Zhiyong Liu. Joint virtual machine assignment and traffic engineering for green data center networks. *SIGMETRICS Perform. Eval. Rev.*, 41(3):107–112, January 2014.

[XBNJ13] Yunjing Xu, Michael Bailey, Brian Noble, and Farnam Jahanian. Small is better: Avoiding latency traps in virtualized data centers. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 7. ACM, 2013.

[Xen] Xen project. `https://www.xenproject.org/`.

[XSLW13] Mingwei Xu, Yunfei Shang, Dan Li, and Xin Wang. Greening data center networks with throughput-guaranteed power-aware routing. *Computer Networks*, 57(15):2880 – 2899, 2013.

[YLS+14] Baohua Yang, Junda Liu, Scott Shenker, Jun Li, and Kai Zheng. Keep forwarding: Towards k-link failure resilient routing. In *IEEE INFOCOM 2014*, pages 1617–1625, 2014.

[YRFW10] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with difane. *ACM SIGCOMM Computer Communication Review*, 40(4):351–362, 2010.

[YWR14] Minlan Yu, Andreas Wundsam, and Muruganantham Raju. Nosix: A lightweight portability layer for the sdn os. *ACM SIGCOMM Computer Communication Review*, 44(2):28–35, 2014.

[YXX+14] Bo Yan, Yang Xu, Hongya Xing, Kang Xi, and H Jonathan Chao. Cab: A reactive wildcard rule caching system for software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 163–168. ACM, 2014.

[ZHLL06] Kai Zheng, Chengchen Hu, Hongbin Lu, and Bin Liu. A tcam-based distributed parallel ip lookup scheme and performance analysis. *IEEE/ACM Transactions on Networking (TON)*, 14(4):863–875, 2006.

[ZIA+15] David Zats, Anand Padmanabha Iyer, Ganesh Ananthanarayanan, Rachit Agarwal, Randy Katz, Ion Stoica, and Amin Vahdat. Fastlane: making short flows shorter with agile drop notification. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 84–96. ACM, 2015.

[ZJP14] Shijie Zhou, Weirong Jiang, and Viktor Prasanna. A programmable and scalable openflow switch using heterogeneous soc platforms. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 239–240. ACM, 2014.

[ZLC+16] Liang Zhang, James Litton, Frank Cangialosi, Theophilus Benson, Dave Levin, and Alan Mislove. Picocenter: Supporting long-lived, mostly-idle applications in cloud environments. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 37. ACM, 2016.

[Zus] Zuse-Institute Berlin. Problem germany50–d-b-l-n-c-a-n-n). `http://sndlib.zib.de/home.action`.

[ZWW11] Yanwei Zhang, Yefu Wang, and Xiaorui Wang. Greenware: Greening cloud-scale data centers to maximize the use of renewable energy. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 143–164. Springer, 2011.