AirNet, le modèle de virtualisation “ Edge-Fabric ” comme plan de contrôle pour les réseaux programmables

Messaoud Aouadj

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *08/11/2016* **par :**

Messaoud AOUADJ

AirNet : le modèle de virtualisation "Edge-Fabric" comme plan de contôle pour les réseaux programmables

---

JURY

Olivier FESTOR	Professeur d'Université Université de Lorraine	Rapporteur
Guy PUJOLLE	Professeur d'Université Université Pierre et Marie Curie	Rapporteur
Guillaume DOYEN	Maître de Conférences Université de Technologie de Troyes	Examinateur
Philippe OWEZARSKI	Directeur de Recherche LAAS-CNRS	Examinateur
Emmanuel LAVINAL	Maître de Conférences Université Paul Sabatier	Encadrant
Michelle SIBILLA	Professeur d'Université Université Paul Sabatier	Directeur de thèse

---

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture
Unité de Recherche :
Institut de Recherche en Informatique de Toulouse (UMR 5505)
Directeur(s) de Thèse :
Michelle SIBILLA et *Emmanuel LAVINAL*
Rapporteurs :
Guy PUJOLLE et *Olivier FESTOR*

Résumé

Les travaux de cette thèse s'inscrivent dans le contexte général des réseaux logiciels, dits "*Software-Defined Networking*" (SDN). Ce paradigme récent est l'une des initiatives les plus notables pour rendre les réseaux actuels programmables ou, en d'autres termes, plus simple à configurer, à tester, à corriger et à faire évoluer. Dans un écosystème SDN, l'interface nord (*Northbound API*) est utilisée par l'administrateur réseaux pour définir ses politiques et programmer le plan de contrôle, elle représente donc un enjeu majeur. Idéalement, cette interface nord devrait permettre aux administrateurs de décrire, le plus simplement possible, des services réseaux et leurs interactions, plutôt que de spécifier comment et sur quels équipements physiques ils doivent être déployés. Des travaux existants montrent que cela peut être notamment réalisé grâce à des solutions de virtualisation de réseaux et des langages de programmation dédiés de haut niveau.

L'objectif de ce travail de thèse est de proposer une nouvelle interface nord qui, d'une part, exploiterait la virtualisation de réseau et, d'autre part, exposerait ses services sous la forme d'un langage de programmation dédié. Actuellement, plusieurs langages intégrant des solutions de virtualisation de réseau existent. Néanmoins, nous pensons que les modèles d'abstraction qu'ils utilisent pour construire des réseaux virtuels restent inappropriés pour assurer des critères de simplicité, modularité et flexibilité des topologies virtuelles et des programmes de contrôle.

Dans ce contexte, nous proposons un nouveau langage de contrôle de réseaux nommé AirNet. Ce dernier intègre un modèle d'abstraction dont la principale caractéristique est d'offrir une séparation nette entre les équipements de bordure (*Edge*) et de cœur de réseau (*Fabric*). Cette idée est bien connue et acceptée dans le domaine des architectures réseaux. L'originalité de notre contribution étant de faire remonter ce concept au niveau du plan de contrôle virtuel et non de le restreindre au seul plan physique. Ainsi, des frontières logiques entre les différents types de politiques existeront (fonctions de contrôle et de données vs. fonctions de transport), garantissant ainsi la modularité et la réutilisabilité de tout ou partie du programme de contrôle. De plus, dans l'approche proposée, la définition du réseau virtuel et des politiques peut être totalement dissociée de l'infrastructure physique cible, favorisant ainsi la portabilité des applications de contrôle.

Une implémentation du langage AirNet a également été réalisée. Ce prototype inclut en particulier une bibliothèque des primitives et opérateurs du langage, ainsi qu'un hyperviseur qui assure la composition des politiques de contrôle sur un réseau virtuel, et leur transposition (mapping) sur l'infrastructure physique. Afin de s'appuyer sur des contrôleurs

SDN existants, l'hyperviseur inclut des modules d'intégration des contrôleurs POX et RYU. Une validation expérimentale a été menée sur différents cas d'étude (filtrage, répartition de charge, authentification dynamique, limitation de bande passante, etc.) dont les résultats attestent de la faisabilité de la solution. Enfin, des mesures de performances ont montré que le surcoût apporté par cette nouvelle couche d'abstraction est parfaitement acceptable.

Abstract

The work of this thesis falls within the general context of software-defined networking (SDN). This new paradigm is one of the most significant initiatives to enable networks programmability or, in other words, to make current networks easier to configure, test, debug and evolve. Within an SDN ecosystem, the Northbound interface is used by network administrators to define policies and to program the control plane, it thus represents a major challenge. Ideally, this northbound interface should allow administrators to describe, as simply as possible, network services and their interactions, rather than specifying how and on what physical device they need to be deployed. Current related works show that this can be partly achieved through virtualization solutions and high-level domain specific languages (DSL).

The objective of this thesis is to propose a new Northbound interface which will, on the one hand, rely on network virtualization and, on the other hand, expose its services as a domain specific programming language. Currently, several languages that include network virtualization solutions exist. Nevertheless, we believe that the abstract models they are using to build virtual networks remain inadequate to ensure simplicity, modularity and flexibility of virtual topologies and control programs.

In this context, we propose a new network control language named AirNet. Our language is built on top of an abstraction model whose main feature is to provide a clear separation between edge and core network devices. This concept is a well-known and accepted idea within the network designer community. The originality of our contribution is to lift up this concept at the virtual control plane, not limiting it solely at the physical plane. Thus, logical boundaries between different types of policies will exist (control and data functions vs. transport functions), ensuring modularity and reusability of the control program. Moreover, in the proposed approach, the definition of the virtual network and policies is totally dissociated from the target physical infrastructure, promoting the portability of control applications.

An implementation of the AirNet language has also been done. This prototype includes in particular a library that implements the primitives and operators of the language, and a hypervisor that achieves the composition of the control policies on the virtual network, and their mapping on the physical infrastructure. In order to rely on existing SDN controllers, the hypervisor includes integration modules for the POX and RYU controllers. An experimental validation has been also conducted on different use cases (filtering, load balancing, dynamic authentication, bandwidth throttling, etc.), whose results demonstrate the feasibility of our

solution. Finally, performance measurements have shown that the additional cost brought by this new abstraction layer is perfectly acceptable.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Contexte et problématique

Avec l'avènement des innovations technologique récentes telles que la virtualisation des machines et des systèmes, le cloud computing ou encore l'Internet des objets, les limites actuelles des architectures réseaux deviennent de plus en plus problématiques pour les opérateurs et les administrateurs réseaux. En effet, depuis déjà plusieurs années, il est communément admis que les architectures IP traditionnelles sont, d'une part, particulièrement complexes à configurer à cause de la nature distribuée des protocoles réseaux et, d'autre part, difficile à faire évoluer en raison du fort couplage qui existe entre le plan de contrôle et le plan de données des équipements d'interconnexion existants.

Le paradigme SDN (*Software Defined Networking*) est une nouvelle approche qui a pour ambition de répondre à cette rigidité architecturale des réseaux IP actuels, notamment en les rendant plus programmables. Pour ce faire, le paradigme SDN préconise une architecture où tout le plan de contrôle du réseau est logiquement centralisé dans un composant détaché du plan de données. La configuration d'un réseau reviendra alors à programmer ce composant, appelé contrôleur SDN, en utilisant son interface nord (*Northbound API*).

Actuellement, il existe plusieurs versions et types de contrôleurs SDN, chacun exposant une interface nord qui lui est propre. Malheureusement, ces interfaces présentent des limites importantes, particulièrement le fait qu'elles soient des APIs spécifiques, de bas niveau et qui n'offrent que très peu de fonctionnalités avancées telles que la composition des politiques de contrôle. Ainsi, l'utilisation d'un contrôleur SDN a rendu possible l'opération de programmation d'un réseau, mais actuellement sa mise en œuvre concrète n'est pas nécessairement plus facile que certaines solutions existantes de reconfiguration de réseaux.

Pour atteindre au mieux les objectifs du paradigme SDN, les contrôleurs doivent donc fournir des interfaces de programmation métiers et modernes qui permettront d'abstraire les détails de bas niveau de l'infrastructure physique et ceux de l'implémentation des contrôleurs, introduisant ainsi plus de modularité et de flexibilité dans les programmes de contrôle. Ces exigences sont essentielles pour les approches SDN, étant donné que la *Northbound API* est l'interface qui sera utilisée par les administrateurs et les opérateurs afin de spécifier leurs applications de contrôle et leurs services à valeur ajoutée.

Partant de ce constat, les travaux les plus récents sur la *Northbound API* d'une architecture SDN considèrent la virtualisation de réseau comme une approche qui permettra d'atteindre mieux ces objectifs de simplification, de modularité et de flexibilité des programmes de

contrôle. En effet, cette approche permet de créer des visions abstraites de l'infrastructure physique qui n'exposent que les informations les plus pertinentes pour les politiques de contrôle de haut niveau, masquant ainsi la nature complexe et dynamique de l'infrastructure physique. De plus, les programmes de contrôle, ou du moins une grande partie d'entre eux, peuvent être facilement réutilisés sur différentes topologies physiques, étant donné que leur spécification a été réalisée au-dessus de topologies virtuelles.

Cependant, l'utilisation de cette démarche présente trois défis majeurs :

- Le modèle d'abstraction qui sera utilisé pour masquer les détails et la complexité de l'infrastructure physique.
- La spécification des politiques virtuelles qui décrivent les services réseaux et leur orchestration.
- Le moteur d'exécution (l'hyperviseur) qui devra assurer la composition des politiques virtuelles et leur transposition sur l'infrastructure physique.

Le contexte de notre travail s'inscrit donc dans cette problématique précise qui est la proposition d'une *Northbound API* qui exposerait ses services sous la forme d'un langage de programmation de haut niveau supportant de la virtualisation de réseau.

Contributions

Etat de l'art sur les langages de contrôle de l'interface nord de SDN : afin d'identifier les différentes approches de configuration d'un contrôleur SDN, nous avons mené une étude dans le but d'analyser les différents langages de programmation réseaux qui ont été proposés. Le premier enseignement que nous avons retenu de cette étude est que cette problématique reste encore une question ouverte, où chaque nouveau langage a apporté des nouvelles abstractions tout en construisant sur les apports des travaux existants. Parmi ces travaux, nous nous sommes particulièrement intéressés aux langages qui offrent une solution de virtualisation de réseau. De notre point de vue, ces langages utilisent des modèles d'abstraction qui sont bien connus, cependant ces derniers restent inappropriés pour la spécification de politiques réseau de haut niveau.

Proposition du modèle Edge-Fabric : le modèle d'abstraction représente un enjeu majeur dans une approche basée sur la virtualisation de réseau, étant donné que ses propriétés intrinsèques (facilité d'utilisation, modularité, réutilisation et flexibilité) se répercuteront, in fine, sur les propriétés de la *Northbound API*. Ainsi, nous avons porté notre attention sur la proposition d'un nouveau modèle d'abstraction qui permettrait de mieux atteindre ces critères. Cette réflexion a donné lieu à la proposition du modèle Edge-Fabric, qui est basé sur l'idée clé d'une claire séparation entre les politiques de transport et les services réseaux plus complexes. Le modèle Edge-Fabric étant un modèle très connu de conception d'infrastructures physiques, l'originalité de ce travail réside donc sa réutilisation au niveau du plan de contrôle virtualisé afin d'assurer les exigences d'expressivité, modularité et flexibilité d'une *Northbound API*.

Définition du langage AirNet : un autre pilier de notre contribution de thèse est la définition du langage AirNet. Ce dernier a été construit autour du modèle Edge-Fabric et offre un ensemble de primitives qui permettent d'une part de définir des topologies virtuelles suivant le modèle Edge-Fabric et, d'autre part, de spécifier les politiques de contrôle qui seront exécutées au-dessus de ces dernières. Ces politiques sont divisées en quatre principaux types, suivant la nature des problèmes ciblés : politique de transport, fonction de contrôle statique, fonction de contrôle dynamique et fonction de traitement des données d'un paquet réseau. Par la suite, AirNet permet la composition de ces services, parallèlement ou en séquence, afin de définir la politique globale de gestion.

Conception et développement d'un système d'exécution : nos contributions incluent aussi un prototype de système d'exécution que nous nommons *AirNet hypervisor*. Ce dernier assure la composition des politiques virtuelles et leur transposition sur l'infrastructure physique, notamment en implémentant ces cinq fonctionnalités essentielles :

- Abstraction de la nature complexe et dynamique de l'infrastructure physique.
- Composition et résolution d'intersections entre politiques virtuelles.
- Transposition des politiques virtuelles sur les équipements physiques.
- Possibilité d'installation, en cours d'exécution (*at runtime*), de nouvelles politiques.
- Adaptation des règles physiques en cas d'une défaillance ou d'une mise à jour du réseau réel.

Validation expérimentale : la dernière partie de ce travail de thèse consiste en une validation expérimentale du langage AirNet et de son hyperviseur. Pour ce faire, nous avons mené divers tests et mesures de performance, sur divers cas d'utilisation et divers contrôleurs SDN, afin d'évaluer le coût de cette nouvelle couche d'abstraction. Les résultats obtenus sont encourageants et ont montré que le coût de cette simplification reste parfaitement acceptable.

Organisation générale du manuscrit

Ce document est organisé en trois grandes parties, chacune étant composée de plusieurs chapitres :

- La première partie détaille le contexte général de notre travail. Pour ce faire, un premier chapitre est consacré au paradigme SDN dans lequel nous présentons ses concepts fondamentaux. Le deuxième chapitre est, quant à lui, consacré à l'interface nord d'une architecture SDN. Dans ce chapitre, nous présentons, d'abord, les exigences que nous avons identifiées et qu'un langage de programmation de haut niveau devrait satisfaire. Puis dans un second temps, nous présentons une étude détaillée des différents langages qui ont été proposés.

- Dans la deuxième partie, un premier chapitre présente les modèles d'abstraction les plus utilisés pour créer des topologies virtuelles, ainsi que notre proposition de modèle Edge-Fabric. Le chapitre suivant décrit le langage AirNet en détaillant ses primitives et son modèle de programmation à travers un ensemble de cas d'utilisation. Enfin, le dernier chapitre de cette partie présente la conception et l'implémentation de l'hyperviseur d'AirNet.
- La dernière partie est dédiée à la mise en pratique du langage AirNet et de son hyperviseur. Le premier chapitre de cette partie détaille le processus de spécification d'un cas d'étude du début jusqu'à la fin en montrant notamment les résultats d'exécution obtenus. Enfin, le dernier chapitre de ce manuscrit est consacré à diverses mesures de performances de notre prototype.

Première partie

Contexte

Chapitre 1
Réseaux programmables

Un réseau informatique consiste typiquement en l'interconnexion de plusieurs équipements de différents types tels que des routeurs, des commutateurs, des hôtes ou des *middleboxes* (pare-feu, traduction d'adresses, équilibrage de charge, etc.). Souvent, ces équipements embarquent, d'une part, des systèmes propriétaires et fermés et exécutent, d'autre part, des protocoles complexes et distribués afin d'assurer la transmission des paquets. Ainsi, en dépit de leur large adoption, les réseaux IP traditionnels sont de plus en plus complexes et difficiles à gérer [Benson et al., 2009]. En effet, pour (re)configurer un réseau, un opérateur doit configurer séparément différents protocoles sur chaque équipement en utilisant des interfaces en ligne de commande (CLI) de bas niveau et qui, de surcroît, sont souvent différentes d'un constructeur à un autre et d'un type d'équipement à un autre [Kreutz et al., 2014].

Pour rendre les choses encore plus compliquées, les réseaux actuels sont verticalement intégrés. En effet, le plan de contrôle (la partie qui décide comment doit se faire la transmission des paquets) et le plan de données (la partie qui s'occupe de la transmission effective des paquets suivant les décisions prises par le plan de contrôle) sont fortement couplés à l'intérieur des équipements réseaux, rendant ainsi plus difficile l'évolution du matériel, des protocoles et des architectures [Raghavan et al., 2012]. La transition du protocole IPv4 vers le protocole IPv6, qui a commencé il y a déjà plus d'une décennie tout en restant largement inachevée, est un excellent exemple de ce manque de flexibilité dans les architectures réseaux actuelles.

Le paradigme SDN (*Software Defined Networking*) est une des plus récentes initiatives qui vise à répondre à cette problématique de rigidité architecturale [Ghodsi et al., 2011]. Pour ce faire, SDN **découple** le plan de contrôle du plan de données et le centralise dans un point logique **programmable** appelé contrôleur SDN [Gude et al., 2008]. Ainsi, en utilisant ce point de contrôle centralisé (logiquement), les administrateurs réseaux sont en mesure de définir et de mettre à jour rapidement le comportement de leur réseau, et cela, "simplement" en (re)programmant le contrôleur SDN via son interface de programmation applicative (API), communément appelée *Northbound API*.

Dans ce qui suit, nous présentons plus en détails ce nouveau paradigme ainsi que les différents concepts qui le définissent. Nous allons commencer par introduire brièvement les travaux initiaux ainsi que les enjeux qui ont motivés la proposition du paradigme SDN. Puis

nous présenterons le paradigme SDN selon une approche ascendante (*bottom-up*) du plan de données jusqu'à l'interface nord, interface qui nous intéresse tout particulièrement dans le contexte de nos travaux.

1.1 Les préludes de SDN

Dans cette section, nous présentons, brièvement, les principaux travaux académiques et projets industriels à partir desquels le paradigme SDN tire ses origines. En effet, le paradigme SDN n'est pas apparu soudainement, mais fait partie d'une longue histoire d'initiatives pour rendre les traditionnels réseaux IP plus programmables [Feamster et al., 2013], ou en d'autres termes, plus flexibles et plus simples à configurer et à faire évoluer. Ainsi, suivant la nature de leurs apports, ces travaux peuvent être divisés en deux principales contributions :

- Proposer un plan de données programmable pour simplifier le développement et le déploiement de nouveaux services réseaux.
- Introduire une séparation entre le plan de contrôle et le plan de données afin de faciliter la (re)configuration des réseaux.

1.1.1 Plan de données programmable

Le concept de plan de données programmable fait référence à la possibilité de disposer d'équipements matériels qu'on pourrait facilement modifier afin d'ajouter des opérations sur les flux de paquets, et cela dans le but faciliter et d'accélérer le déploiement de nouveaux services et technologies. Ainsi, les réseaux actifs (ou *active networks*) [Tennenhouse et al., 1997] représentent une des premières tentatives de construire des architectures qui disposent d'un plan de données programmable. En effet, la mise en réseau actif (ou *active networking*) est une approche qui est apparue dans le milieu des années 1990 et dont la principale motivation était de répondre à la lenteur de déploiement de nouveaux services réseaux et à la complexité des processus de standardisation et de normalisation.

A son apparition, l'*active networking* était une approche radicalement différente par le fait qu'elle proposait d'inclure, sur chaque nœud réseau, une API qui exposerait des ressources (calcul, stockage, file d'attente pour les paquets, etc.) afin de programmer des fonctions (le plan de données) qui seraient appliquées par la suite sur des flux de paquets. Aussi, les réseaux actifs ont connu deux principaux modèles de programmation :

- Le modèle capsule : le code à exécuter sur les nœuds est transporté dans la bande (*in-band*) par les paquets réseaux [Wetherall et al., 1998].
- Le modèle commutateur/routeur programmable : le code à exécuter sur les nœuds réseaux est envoyé par des moyens hors-bande (*out-of-band*) [Bhattacharjee et al., 1997].

Aussi, il est important de noter, que déjà à cette époque, on retrouvait dans la littérature [Tennenhouse et al., 1997] presque les mêmes motivations qui ont poussé plus tard le paradigme SDN, à savoir principalement :

- La difficulté qu'éprouvaient les opérateurs télécoms à développer puis à déployer des nouveaux services réseaux.
- Le désir de la communauté scientifique de disposer d'une plateforme d'expérimentation réaliste et qui soit capable de passer l'échelle.

1.1.2 Séparation entre le plan de contrôle et le plan de données

Au début des années 2000, les opérateurs télécoms étaient à la recherche de meilleures approches afin de mieux contrôler l'acheminement de leurs trafics (*traffic engineering*). En effet, avec le succès d'Internet et l'augmentation sans cesse croissante des volumes de données, les opérateurs avaient un besoin pressant de solutions de *traffic engineering* qui soient plus fiables, performantes et prédictibles, à l'inverse des approches basées sur l'utilisation des protocoles de routage conventionnels.

La complexité des solutions classiques était principalement due au fait que les routeurs IP intégrés un plan de contrôle et un plan de données étroitement liés. Cette intégration rendait les tâches de configuration, de déboggage et de prévision de comportement excessivement compliquées. Pour répondre à cette problématique, plusieurs efforts ont été menés afin de séparer le plan de contrôle du plan de données. Ces efforts ont permis notamment de catalyser deux principales innovations [Feamster et al., 2013] :

- Proposition d'une interface ouverte entre le plan de contrôle et le plan de données, tel le projet *ForCES* [Doria et al., 2010] qui définit un *framework* et un protocole pour standardiser les échanges d'informations entre le plan de contrôle et le plan de données des routeurs IP et des équipements similaires.
- Fournir un point de contrôle logiquement centralisé, comme cela a été le cas pour les architectures RCP [Caesar et al., 2005] qui implémentent une solution de routage centralisée. Ainsi, RCP permet de construire une carte de la topologie physique, d'exécuter des algorithmes sur cette carte, puis de sélectionner les routes BGP préférées pour chaque routeur dans un système autonome.

Par ailleurs, il est à noter que, à l'inverse des projets sur la programmabilité du plan de données, les projets cités ci-dessus ont su démontrer leurs utilité urgente pour les industriels. Cela est notamment dû au fait que ces projets [Feamster et al., 2013] :

- Sont axés sur des problématiques pressantes en matière de gestion (*traffic engineering*).
- Permettent la programmabilité du plan de contrôle en lieu et en place du plan de données, en mettant l'accent notamment sur l'innovation par et pour les administrateurs réseaux.
- Offrent un contrôle et une vision globale du réseau plutôt qu'une vision individuelle sur chaque équipement.

SDN (*Software Defined Networking*) [ONF, 2012] est un paradigme émergent qui reprend beaucoup des idées présentées ci-dessus. De plus, ce paradigme arrive à un moment ou les enjeux cités ci-précédemment sont encore plus pesants, notamment à cause de grands changements technologiques tels que la mobilité des utilisateurs, le big data ou le cloud

computing. Dans ce qui suit, nous présentons plus en détails cette nouvelle architecture réseau, ainsi que les principes fondamentaux qui la définissent.

1.2 Software Defined Networking

SDN est un paradigme qui décrit une architecture réseau dont le plan de données est contrôlé à distance par une entité logique centrale. Plus concrètement, on peut dire qu'une architecture réseau suit le paradigme SDN si, et seulement si, elle vérifie ces quatre points fondamentaux :

- Le plan de contrôle est complètement découplé du plan de données. Cette séparation est concrétisée au travers de la définition d'une interface de programmation (*Southbound API*) ouverte, comme cela est montré à la figure 1.
- Toute l'intelligence du réseau est externalisée dans un point logiquement centralisé appelé contrôleur SDN. Ce dernier offre une connaissance globale de l'infrastructure physique et des abstractions pour la configurer.
- Le contrôleur SDN est un composant programmable. Ce dernier peut être vu comme un système d'exploitation réseaux (NOS, *Network Operating System*) qui expose une API (*Northbound API*) pour spécifier des applications de contrôle.
- La transmission des paquets ne se fait plus suivant la destination, mais par flux, sachant qu'un flux est défini par un ensemble de champs d'en-têtes. Cela permet notamment d'unifier les différents types d'équipements réseaux qu'on retrouve dans une infrastructure physique tels que les commutateurs et les routeurs.

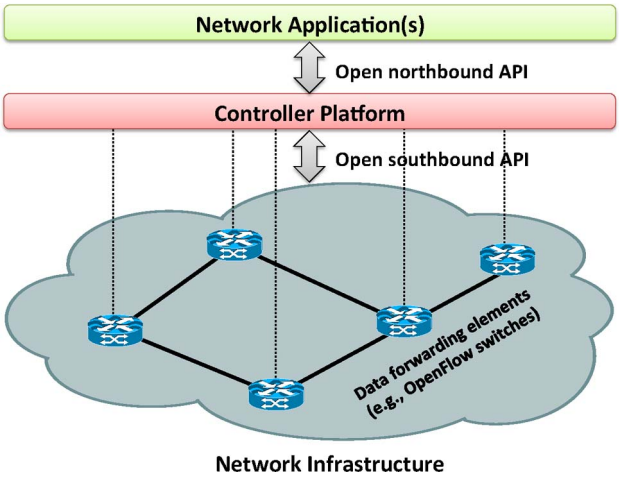

FIGURE 1. Vision simplifiée d'une architecture SDN [Kreutz et al., 2014]

Une conséquence directe de cette conception est que la configuration d'un réseau devient une tâche plus simple et moins sujette aux erreurs. En effet, programmer un unique contrôleur en utilisant des API ou des langages de programmation haut-niveau est une tâche

significativement moins compliquée que celle de configurer individuellement plusieurs équipements physiques en utilisant leur interface de bas niveau. Ainsi, un aspect important du paradigme SDN, si ce n'est le plus important, est qu'il permet de configurer un réseau à travers des applications (services) qui sont installées au-dessus du contrôleur SDN.

De plus, la complexité des algorithmes utilisés se verra elle aussi significativement diminuée, étant donné que les algorithmes de contrôle n'auront plus besoin de considérer la nature distribuée des réseaux, mais ils seront directement appliqués sur une vision globale de l'infrastructure qui est fournie par le contrôleur SDN. Comme exemple concret, on peut citer les algorithmes de routage où il sera plus aisé de construire un graphe qui représente la topologie en utilisant l'état global du contrôleur, en lieu et place des protocoles distribués.

Dernier point, mais pas le moindre, en cassant l'intégration verticale des réseaux, SDN a permis d'introduire plus de flexibilité et d'innovation dans les architectures réseaux. En effet, cela a permis une stricte séparation entre deux problèmes distincts, à savoir : la définition des politiques de contrôle et la manière avec laquelle elles sont mises en œuvre sur les équipements physiques. Cette séparation permet à chacune de ces deux parties d'évoluer séparément et de se concentrer sur leur problématique propre telle que la proposition d'équipement plus simple, moins onéreux et plus performant pour le plan de données ou encore l'introduction de nouveaux services réseaux pour le plan de contrôle.

Dans ce qui suit, nous présentons, en suivant un schéma *bottom-up*, chacune de ces briques qui constituent une architecture SDN.

1.2.1 Le plan de données

Dans une architecture SDN, le plan de données représente l'infrastructure physique qui s'occupe de l'acheminement effectif des paquets. Cette infrastructure est similaire aux infrastructures traditionnelles, par le fait qu'elle est constituée d'un ensemble d'équipements interconnectés entre eux. La différence principale est que les équipements d'un plan de données SDN n'intègrent aucune fonctionnalité de contrôle qui pourrait leur permettre de prendre des décisions d'une manière autonome. En effet, toute l'intelligence de l'infrastructure physique est externalisée dans le contrôleur SDN.

Une autre caractéristique des commutateurs SDN est qu'ils sont conceptuellement construits autour d'une interface ouverte et standard. Ce critère est fondamental étant donné qu'il permet d'assurer la configuration de ces équipements, mais, plus important encore, il garantit l'interopérabilité entre différentes conception et implémentation des plans de données et de contrôle.

Actuellement, la norme OpenFlow [McKeown et al., 2008] est la technologie la plus répandue, on parle alors de commutateur OpenFlow. Dans la suite de cette section (et dans le reste de ce document), nous considérons principalement cette norme. En effet, malgré l'existence d'autres technologies comme ForCES [Doria et al., 2010], POF [Song, 2013], ou OVSDB [Pfaff and Davie, 2013], force est de constater que OpenFlow est actuellement le standard de facto.

Le fonctionnement d'un équipement OpenFlow est basé sur l'utilisation d'une ou plusieurs tables de flux (en *pipeline*). Chaque entrée (règle) d'une table OpenFlow contient

cinq principales parties :

- un filtre qui permet de capturer des paquets suivant un ensemble de champs d'en-têtes.
- une liste d'actions à appliquer sur les paquets retournés par le filtre.
- une priorité pour définir quelle règle doit être appliquée dans le cas où plusieurs règles d'une même table s'appliquent sur un paquet entrant.
- des compteurs qui tiennent des statistiques sur le nombre de paquets et d'octets traités.
- une durée de vie, qui peut être éventuellement positionnée à l'infinie.

Le filtre d'une règle OpenFlow, quant à lui, peut être construit en utilisant plusieurs champs d'en-têtes, nous en citons ci-dessous les plus importants :

- Ethernet : source, destination, type, VLAN ID.
- IPv4 : source, destination, protocole supérieur, ToS.
- TCP/UDP : port source, port destination.

Par ailleurs, il est à noter qu'à chaque sortie d'une nouvelle version d'OpenFlow, de nouveaux champs d'en-têtes ont été ajoutés, tels que les champs du protocole MPLS (label, classe de trafic) dans la version 1.1 ou le support du protocole IPv6 dans la version 1.2.

Concernant les actions, un commutateur OpenFlow expose des actions similaires à celles retrouvées dans les équipements IP traditionnels, comme cela est montré dans la liste ci-dessous :

- Commuter le paquet vers un ou plusieurs ports de sortie.
- Rejeter un paquet.
- Modifier les champs d'en-tête d'un paquet.
- Encapsuler un paquet puis l'envoyer vers le contrôleur SDN (action spécifique aux commutateurs OpenFlow).

Ainsi, à l'intérieur d'un équipement OpenFlow, le chemin que va suivre un paquet à travers les différentes tables de flux définit la manière avec laquelle le commutateur va traiter ce paquet ainsi que ceux qui appartiennent au même flux que lui (qui ont les mêmes champs d'en-tête). En effet, à la réception d'un paquet, le commutateur OpenFlow déroule l'automate qui est schématisé à la figure 2. Le commutateur commence d'abord par effectuer une recherche dans sa première table et si un traitement en pipeline est défini, il effectue des recherches supplémentaires dans ses autres tables. Si le commutateur ne trouve aucune règle qui correspond et qu'aucune règle par défaut n'a été spécifiée, il rejète le paquet. Néanmoins, il est à noter qu'il peut exister une règle par défaut qui consiste soit à envoyer tous les paquets qui ne correspondent à aucune règle vers le contrôleur, soit à rejeter tous les paquets inconnus.

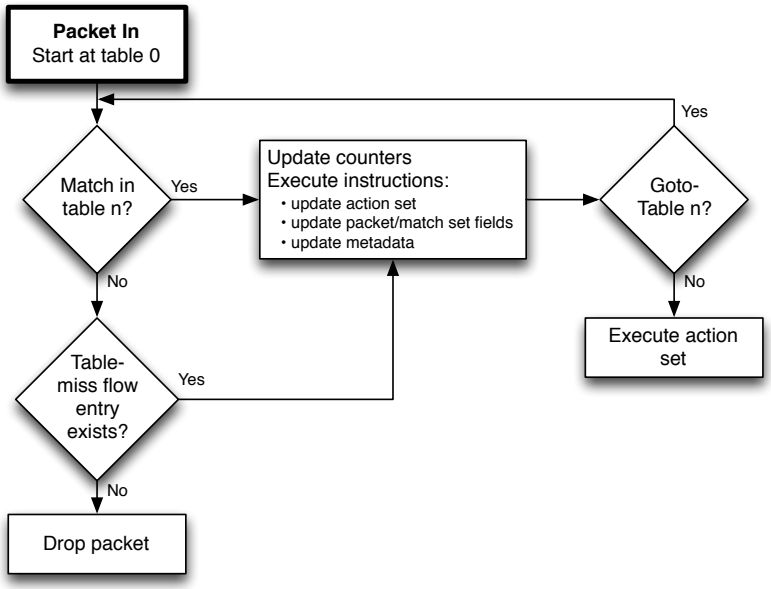

FIGURE 2. Traitement d'un paquet dans un commutateur OpenFlow [Ope, 2012]

1.2.2 SDN *Southbound API*

La *Southbound API* représente l'interface qui permet de connecter chaque commutateur du plan de données au contrôleur SDN. En effet, à travers cette interface, le contrôleur configure et gère l'ensemble des commutateurs qui sont sous son autorité. Actuellement, il existe plusieurs types de *Southbound API* tels que ForCes [Doria et al., 2010], Open vSwitch Database (OVSDB) [Pfaff and Davie, 2013] ou POF [Song, 2013], mais comme cela a été mentionné auparavant, nous nous appuyons dans notre contexte sur OpenFlow, étant donné que c'est un standard qui est largement accepté et répandu dans les réseaux SDN.

OpenFlow définit trois principaux types de messages qui peuvent être échangés entre les commutateurs et le contrôleur SDN :

- messages de commande : représentent les messages de configuration qui sont envoyés du contrôleur en direction des commutateurs.
- messages asynchrones : ces messages sont envoyés depuis les commutateurs vers le contrôleur SDN, sans solicitation de ce dernier.
- messages symétriques : des messages de maintenance échangés entre les commutateurs OpenFlow et le contrôleur SDN.

1.2.2.1 Messages asynchrones

Le contrôleur SDN reçoit plusieurs informations à partir de l'infrastructure physique, notamment grâce aux messages asynchrones qui sont envoyés par les commutateurs. La norme OpenFlow en distingue essentiellement deux types :

- *Packet In* : message qui permet de transférer le contrôle d'un paquet au contrôleur SDN. Communément cela se produit dans deux cas : *i*) le commutateur n'a aucune règle dans ses tables qui s'applique à ce paquet, *ii*) une règle avec une action de retransmission vers le contrôleur a été expressément installée.
- *Port Status* : informe le contrôleur d'un changement de l'état (*UP/DOWN*) d'un port.

1.2.2.2 Messages de commande

Les messages de commande permettent au contrôleur SDN de configurer le fonctionnement des commutateurs OpenFlow. Comme cela a été décrit dans la section précédente, configurer un commutateur OpenFlow revient à configurer la manière avec laquelle un paquet est acheminé à travers les différentes tables de flux d'un équipement. Dans cette catégorie de messages, le protocole OpenFlow en distingue trois principaux sous-types :

- *Modify-state* : ces messages sont envoyés par le contrôleur afin de modifier l'état d'un commutateur. Leur objectif premier est d'ajouter, de supprimer ou de modifier des règles dans les tables des commutateurs.
- *Read-state* : ces messages permettent au contrôleur de collecter diverses informations sur l'infrastructure physique telles que la configuration actuelle d'un commutateur ou des statistiques sur des flux.
- *Packet-out* : ce type de messages permet au contrôleur de relayer un paquet sur un port de sortie d'un commutateur. Ce message est communément utilisé pour répondre à des messages de type *Packet In*.

1.2.2.3 Messages symétriques

Les messages symétriques sont envoyés sans sollicitation dans un sens ou dans l'autre. Ces messages permettent essentiellement de garder en vie le canal de communication qui est établi entre le contrôleur et les commutateurs OpenFlow. On distingue deux principaux messages symétriques :

- *Hello* : ces messages sont échangés entre un commutateur et le contrôleur à l'établissement d'une connexion.
- *Echo request/reply* : ces messages sont principalement utilisés pour vérifier l'état de la connexion entre un commutateur et le contrôleur SDN. Potentiellement ils peuvent être aussi utilisés pour mesurer le délai et la bande passante du canal de communication.

1.2.3 Le plan de contrôle

Communément, un système d'exploitation offre une suite d'abstractions (API, langage de programmation, etc.) pour accéder aux ressources (calcul, mémoire, stockage, etc.) d'une machine physique (ou virtuelle). Ces abstractions sont primordiales considérant le fait que, d'une part, elles introduisent plus de possibilités d'innovation pour les programmeurs et, d'autre part, elles facilitent et accélèrent le processus de développement et de déploiement de nouveaux services et applications.

Dans un contexte réseaux, le contrôleur SDN ambitionne d'atteindre ces mêmes objectifs. En effet, les contrôleurs SDN centralisent toute l'intelligence du réseau et fournissent, eux aussi, plusieurs abstractions et services tels que la découverte de la topologie. Actuellement, la conception et l'implémentation de ces contrôleurs reste une démarche très libre, ce qui fait qu'en quelques années plusieurs types de contrôleurs ont été proposés [Gude et al., 2008, Nippon Telegraph and Telephone Corporation, 2012, Erickson, 2013, Project Floodlight, 2012, Tootoonchian et al., 2012, Koponen et al., 2010, Berde et al., 2014], chacun suivant des choix architecturaux différents. Néanmoins, malgré les différences de conception, tous les contrôleurs assurent ces deux principaux services :

- Découverte de topologie : service qui permet aux applications qui s'exécutent au-dessus du contrôleur de disposer d'une connaissance globale de l'infrastructure physique.
- *Northbound API* (NI) : désigne l'interface de programmation qui est destinée aux développeurs d'application et de services réseaux. Typiquement, cette interface abstrait (plus ou moins fortement) les détails bas niveau de la *Southbound API*.

En plus de ces fonctionnalités de base, un contrôleur SDN peut fournir des abstractions et des services supplémentaires tels que la découverte des hôtes ou des utilisateurs authentifiés, des algorithmes de parcours à appliquer sur la topologie physique [Kreutz et al., 2014] ou même des fonctionnalités qui garantissent l'isolation entre les applications qui s'exécutent au-dessus de lui comme cela est fait dans le contrôleur Rosemary [Shin et al., 2014].

Par ailleurs, il est important de souligner qu'un contrôleur qui est logiquement centralisé n'implique pas forcément qu'il le soit physiquement. Effectivement, parmi les divers choix de conception et d'implémentation, il en existe un qui a impact important sur la conception et le dimensionnement d'un réseau physique à savoir le support d'un mode de fonctionnement distribué. Ainsi, l'aspect centralisé ou distribué de ce dernier représentera un des points les plus important dans les choix de conception et d'implémentation d'un contrôleur SDN. En effet, une conception centralisée d'un contrôleur suppose, de facto, deux principales limites :

- Tolérance aux fautes : en cas d'erreur, c'est tout le plan de contrôle du réseau qui devient non disponible.
- Passage à l'échelle : un seul contrôleur peut ne pas être suffisant pour gérer des larges réseaux avec un grand nombre d'équipements physiques.

Pour pallier les problèmes de passage à l'échelle, des contrôleurs centralisés comme NOX-MT [Tootoonchian et al., 2012], Beacon [Erickson, 2013] ou Floodlight [Project Floodlight, 2012] ont été conçus d'une manière hautement parallèle afin d'exploiter au mieux les ressources des machines multi-cœurs. Par exemple, le contrôleur Beacon est capable de traiter plus de 12 millions de flux par seconde en utilisant des nœuds d'un fournisseur cloud comme Amazon [Erickson, 2013].

Concernant les contrôleurs distribués (par exemple ONIX [Koponen et al., 2010], ONOS [Berde et al., 2014]), la répartition peut être aussi bien au sein d'un *cluster* ou sur des site physiques différents [Jain et al., 2013]. Les contrôleurs distribués ont une architecture

particulière du fait qu'ils intègrent des API supplémentaires, à savoir la *Eastbound* et la *Westbound* API, comme cela est illustré dans la figure 3. Comme dans tous les systèmes distribués, ces interfaces fournissent principalement des fonctionnalités d'import/export entre les contrôleurs et des algorithmes pour assurer la consistance des modèles de données.

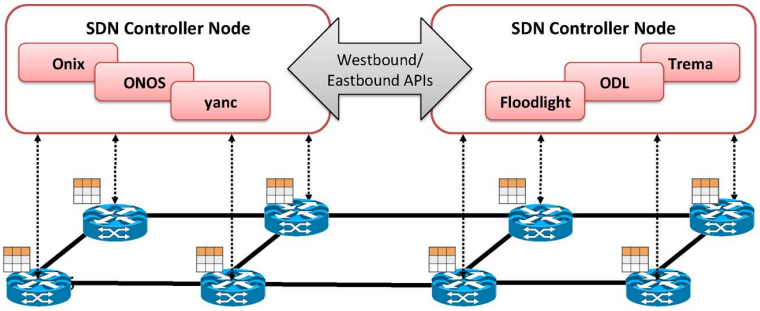

FIGURE 3. Contrôleurs distribués : l'interface est/ouest [Kreutz et al., 2014]

1.2.4 SDN *Northbound API*

La *Northbound API* représente une des abstractions clés de l'écosystème SDN, étant donné que c'est l'interface qui sera utilisée par les administrateurs et les opérateurs afin de spécifier leurs applications de contrôle. En effet, l'interface nord a pour objectif principal d'abstraire les détails de bas niveau de l'infrastructure physique ainsi que ceux de l'implémentation du contrôleur SDN (tels que les spécificités de l'interface sud).

Ainsi, la *Northbound API* devra permettre aux administrateurs de spécifier des applications qui décrivent le comportement du réseau souhaité et non pas comment il est mis en œuvre. Ceci peut être notamment réalisé grâce à des solutions de virtualisation de réseaux et des langages de programmation dédiés et haut niveau. En effet, ces approches permettent de spécifier des applications sur des visions simplifiées (réseaux virtuels) de l'infrastructure physique, suivant un certain modèle d'abstraction. Par la suite, ces politiques seront compilées et transposées sur l'infrastructure physique.

La figure 4 illustre cette évolution de l'architecture SDN (SDNv2 [Scott Shenker, 2011]) où les applications de contrôle sont toujours exécutées au-dessus de réseaux virtuels, les rendant ainsi plus facile à spécifier, étant donné que seules les informations les plus pertinentes sont exposées, et plus simple à réutiliser, étant donné qu'elles sont complètement découplées de l'infrastructure physique.

En dernier, il est à noter que, contrairement à la *Southbound API* qui dispose déjà d'un standard largement accepté, l'interface nord reste encore une question ouverte. Ainsi, on retrouve aujourd'hui plusieurs implémentations différentes de cette interface, souvent sous la forme d'un langage de programmation haut niveau. Chacune de ces implémentations offre des fonctionnalités bien différentes allant du simple masquage des détails de l'interface sud jusqu'à des fonctionnalités avancées telles que la vérification formelle des règles de contrôle ou la virtualisation de réseau. Aussi, comme pour la *Southbound API*, on s'attend à voir émerger dans les prochaines

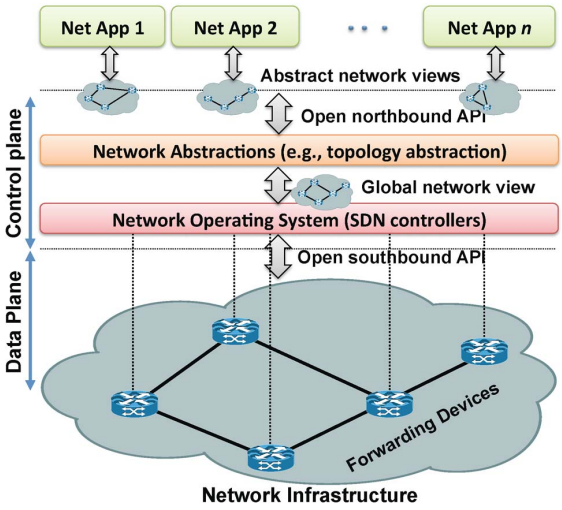

FIGURE 4. Architecture SDNv2 [Kreutz et al., 2014]

années un standard de facto, du fait d'une large adoption d'une ou plusieurs *Northbound API* [Robert Sherwood, 2013, Isabelle GUIS, 2012, Brent Salisbury, 2012, Greg Ferro, 2012, Ivan Pepelnjak, 2012, Sally Johnson, 2012, Rivka Gewirtz Little, 2013].

1.3 Conclusion

Ce chapitre a introduit le paradigme SDN et ses concepts les plus fondamentaux. L'un des objectifs était de montrer que ce paradigme consiste en un ensemble de couches d'abstraction (**acheminement**, **distribution**, et **spécification**) destinées à cacher la nature complexe et dynamique d'un réseau physique. Cela nous a permis également de présenter le contexte et les enjeux qui entourent la dernière couche d'abstraction de ce paradigme, à savoir la *Northbound API*. Le chapitre suivant aborde plus en détail cette dernière, notamment en présentant les différents travaux qui ont porté sur la conception et l'implémentation de langages de contrôle de réseaux de haut niveau.

Chapitre 2
État de l'art des langages de contrôle sur l'interface nord de SDN

Après avoir présenté le paradigme SDN ainsi que les principaux concepts qui le définissent, nous nous intéresserons plus particulièrement dans ce deuxième chapitre aux différents travaux de recherche qui ont porté sur la conception et le développement d'une *Northbound API*. Ainsi, ce chapitre commence par présenter les principaux enjeux et motivations qui sont à l'origine de propositions de nouvelles *Northbound API*, principalement sous la forme de langages de programmation de haut-niveau. Puis, il introduit avec un certain niveau de détails l'ensemble de ces langages, leurs motivations ainsi que leurs principales contributions. La dernière section du chapitre propose une analyse de l'ensemble de ces contributions, sans pour autant essayer de sélectionner un "gagnant". En effet, nous pensons que chaque langage a apporté des contributions qui ont permis de mieux définir les exigences qui doivent être satisfaites, et les services qui peuvent être fournis par une *Northbound API*.

2.1 Quels usages ?

Durant des décennies, le monde informatique a connu une prolifération de langages de programmation. Au départ, la principale problématique était d'essayer de remplacer les langages-machine, bas-niveau, spécifiques à chaque constructeur, tel l'assembleur pour les machines x86, par des langages plus fonctionnels et modulaires tels que Java et Python. Ainsi, industriels et académiques étaient toujours sans cesse à la recherche de nouvelles abstractions qui permettraient de cacher ou, à la rigueur, de simplifier les détails de conception et d'implémentation des plateformes sous-jacentes. Aujourd'hui, nous observons le même phénomène pour les langages de programmation réseau, où l'objectif est de remplacer les instructions de bas-niveau (communément des règles OpenFlow) utilisées par les contrôleurs SDN par des langages de plus haut niveau capablent de fournir des fonctionnalités avancées qui permettront aux programmeurs réseaux d'innover, d'être plus productifs et de se concentrer davantage sur leur problématique métier (sécurité, routage, QoS, etc.) plutôt que sur la maîtrise de la complexité du réseau sous-jacent et des différents équipements hétérogènes qui le composent.

Ainsi, dans ce contexte des réseaux SDN, nous pensons que la conception et

l'utilisation d'un langage de programmation réseau de haut niveau peuvent être motivées, principalement, par ces quatre points :

- Disposer d'abstractions de haut niveau qui permettent de simplifier la configuration des équipements réseaux.
- Améliorer la modularité des programmes de contrôle et permettre aussi leur composition et réutilisation.
- Avoir des modèles de programmation qui permettent de penser des problématiques métiers afin de faciliter le développement des programmes de contrôle.
- Fournir des fonctionnalités plus avancées telles que la vérification formelle des politiques ou la virtualisation de réseaux.

Nous détaillons ces motivations dans les sous-sections qui suivent.

2.1.1 Faciliter la configuration des équipements d'un réseau SDN

Pour configurer un réseau SDN, un administrateur doit utiliser l'API qui est fournie par un contrôleur SDN. Le point commun entre ces API est que toutes reprennent, avec un certain niveau d'abstraction, les principales fonctionnalités du standard OpenFlow. Néanmoins, l'implémentation et l'utilisation d'une API diffèrent suivant le type et la version du contrôleur utilisé. En effet, chaque API définit des modules, fonctions et structures de données qui lui sont propres. De plus, la nature bas-niveau de ces API fait que beaucoup de détails du standard OpenFlow subsistent, tels que la priorité des règles ou le type de protocole utilisé, détails qui sont souvent non pertinents pour la politique de configuration globale.

Ainsi, la plus-value que peut apporter un langage de programmation est de permettre aux administrateurs de s'affranchir de tous ces détails de bas-niveau et de faire en sorte que leur programme de contrôle soit expressif et complètement agnostique au type et à la version du contrôleur utilisé. Pour ce faire, un langage de programmation doit idéalement fournir des abstractions qui permettent de cacher ou de simplifier toutes ces interactions entre les modules de contrôle et l'infrastructure physique comme les messages de commandes qui sont construits puis envoyés pour configurer les équipements OpenFlow ou les requêtes qui remontent des statistiques ou des informations sur la topologie physique.

2.1.2 Modularité et réutilisation

Dans la communauté du génie logiciel, il est communément admis que la modularité et la réutilisabilité sont des propriétés primordiales d'un bon programme informatique, étant donné les gains considérables que cela peut apporter en termes de temps et d'argent. En effet, décomposer un programme en plusieurs composants facilite grandement les phases de développement, de test et de maintenance des programmes. De plus, cela permet leur réutilisation dans d'autres contextes ou sur des infrastructures différentes, simplement en les composant avec d'autres modules existants.

Malheureusement, les API actuelles que fournissent les contrôleurs SDN n'offrent que très peu de possibilités de modularité et de réutilisabilité. En effet, ces API, de bas niveau,

sont dépourvues de concepts qui permettraient de regrouper au sein d'un même conteneur logique des actions qui collaborent pour réaliser une même tâche de plus haut-niveau.

De ce fait, un langage de programmation réseau permettrait d'écrire des programmes de contrôle qui soient plus modulaires et réutilisables. Un administrateur pourra ainsi développer des fonctions et des modules de contrôle indépendants qui représenteront ses principaux services réseaux (contrôle d'accès, transport, équilibrage de charge, etc.). Par la suite, ces modules pourrant être composés ensemble afin d'obtenir des politiques de contrôle plus élaborées et réalistes.

2.1.3 Modèle de programmation métier

Un aspect primordial d'un langage de programmation réseau est le fait qu'il doit être un langage métier. Concrètement, le langage doit fournir un modèle de programmation et des structures de données qui permettent aux administrateurs de penser leurs problématiques réseaux d'une manière simple et intuitive. Ce modèle de programmation doit, entre autre, permettre de décrire les divers types de services rencontrés, la manière dont ces services seront composés et chaînés entre eux et cela dans divers contextes d'utilisations (réseaux d'entreprise, réseaux d'opérateurs, etc.).

2.1.4 Fonctionnalités avancées

En plus de ces fonctionnalités de base qui sont assurées par presque tous les langages qui ont été proposés (de façon et à des niveaux différents), un certain nombre de fonctionnalités plus avancées peuvent être aussi fournies, permettant ainsi d'ajouter une plus-value au langage. Nous en citons-ici les plus importantes :

- *verification formelle* : être capable de vérifier, formellement, la sémantique d'un programme de contrôle.
- *virtualisation de réseau* : permettre de créer et de spécifier des topologies et des programmes virtuels afin de se détacher complètement de l'infrastructure physique. A noter que notre travail et contribution s'inscrivent dans ce contexte bien précis.
- *performance* : faire en sorte de pousser le plus possible le traitement au niveau des équipements qui sont présents dans le plan de données et n'exécuter au niveau du contrôleur que ceux qui sont expressément souhaités.
- *tolérance aux fautes* : le langage et son système d'exécution doivent être capables, à partir d'une défaillance qui se produit dans le réseau, d'installer des chemins de secours soit d'une manière proactive (à l'initialisation), soit d'effectuer des modifications en cours d'exécution sur la configuration courante.

2.2 Langages de contrôle existants

2.2.1 FML

FML (*Flow-Based Management Language*) [Hinrichs et al., 2009] est un des premiers langages à avoir proposé des abstractions de haut-niveau pour la gestion de configurations de

réseaux SDN, et plus particulièrement les réseaux d'entreprises. FML est basé sur le langage Datalog [Huang et al., 2011], mais sans récursivité entre les règles. Ainsi, une politique FML consistera en un ensemble de clauses (c.-à-d. règles), chacune représentant une relation de type **si-alors** appliquée sur un flux unidirectionnel.

L'implémentation de FML a été réalisée en se basant sur le contrôleur NOX [Gude et al., 2008] et en utilisant deux langages de programmation, à savoir C++ et Python. Le langage C++ a été utilisé pour implémenter les modules les plus essentiels et ceux qui nécessitent le plus de performance comme la comparaison d'un flux par rapport aux politiques existantes. Le langage Python, quant à lui, a été utilisé pour les modules les moins exigent en termes de performance comme la compilation des politiques.

2.2.1.1 Structure d'une règle FML

Etant donné que FML est basé sur Datalog, chacune de ses règles prend la forme d'une clause qui est composée d'une tête et d'un corps. En effet, comme on peut le voir sur l'extrait de programme ci-dessous, la tête de la clause (La partie à gauche de la flèche) est un prédicat qui représente souvent une action à réaliser sur un flux, alors que le corps (La partie à droite de la flèche) représente une composition de prédicats ou leur négation. Le flux sur lequel s'applique un prédicat (c.-à-d. une action FML) est identifié grâce aux variables qui sont passées en paramètres au prédicat, sachant que si une variable apparaît dans le corps de la clause, elle doit aussi nécessairement apparaître dans la tête de la clause. L'extrait ci-dessous présente un exemple basique d'une politique FML.

```
allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) ⟵ superuser(U_s)
superuser(Arnaud)
superuser(Pierrick)
```

Ainsi, on observe que la politique ci-dessus est constituée de trois règles ou clauses. La première règle stipule que les flux d'un super utilisateur doivent être toujours autorisés. Quant à la deuxième et la troisième règle, elles stipulent qu'`Arnaud` et `Pierrick` sont des super utilisateurs.

Par ailleurs, si on examine de plus près la première règle, on peut constater que le flux sur lequel s'applique l'action `allow` est identifié grâce aux arguments qui sont passés à ce prédicat. Ces arguments peuvent être des constantes ou des variables. Sachant que l'environnement d'exécution de FML est supposé être capable de remonter les informations suivantes concernant les flux qui circulent au sein d'un réseau, en s'appuyant notamment sur le contrôleur NOX et des protocoles tels que le standard de contrôle d'accès 802.1X [802, 2010] :

- U_s et U_t : nom d'utilisateur source et destination.
- H_s et H_t : hôte source et destination.
- A_s et A_t : points d'accès physique source et destination (numéro de port).
- Prot : protocole de communication utilisé.
- Req : booléen pour dire si le flux est de type requête ou réponse.

Par ailleurs, en plus de la contrainte `allow`, qui permet d'autoriser un flux, FML définit aussi un ensemble d'autres actions qui permettent de couvrir divers cas d'utilisations, nous en citons, ci-dessous les plus importantes :

- `allow` : autorise l'accès à un flux.
- `deny` : refuse l'accès à un flux.
- `waypoint` : route un flux à travers un nœud du réseau.
- `avoid` : route un flux de sorte à ne pas passer par un nœud du réseau.
- `ratelimit` : limite la bande passante maximale pour un flux.
- `band` : fixe le minimum de bande passante à assurer pour un flux.
- `latency` : fixe le délai maximum à garantir pour un flux.

2.2.1.2 Résolution de conflits

Dans FML, l'ordre d'apparition des règles n'a pas d'incidence sur la logique d'exécution d'une politique. Cela permet, entre autres, aux administrateurs d'être capables d'appliquer plusieurs contraintes sur un même flux. Par exemple, au sein d'une même politique, on peut avoir une règle qui force un flux à passer par un nœud du réseau, et une deuxième qui impose au même flux d'éviter un autre nœud particulier.

Néanmoins, cette capacité de FML peut, éventuellement, générer des conflits entre les différentes règles d'une même politique. Par exemple, on ne peut pas appliquer sur un même flux une règle qui l'autorise et une autre qui le bloque. Afin de répondre à cette problématique, FML embraque deux mécanismes de gestions de conflits, le premier est au niveau des mots-clés du langage et le deuxième opère entre les politiques d'un même programme.

Le premier mécanisme est implémenté au niveau de l'environnement d'exécution de FML et permet d'introduire une relation de priorité entre les actions de FML. Par exemple, dans le cas d'une application de contrôle d'accès, l'action `deny` sera toujours plus prioritaire que l'action `allow`, alors que `waypoint` et `avoid` auront une priorité équivalente et seront tous les deux plus prioritaires que `allow`.

Ce mécanisme de priorité permet d'écrire des politiques plus concises. L'extrait ci-dessous représente une politique d'autorisation ouverte, où tout flux qui n'a pas été explicitement interdit est autorisé :

```
allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req)
deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) ⟵ blacklist(U_s)
```

Ainsi, on peut constater qu'écrire des politiques d'accès ouvertes est une tâche assez simple, par contre écrire une politique d'accès fermée où tous les flux sont refusés à part ceux explicitement autorisés est problématique. En effet, étant donné que l'action `deny` est plus prioritaire que l'action `allow`, tous les flux seront bloqués même ceux explicitement autorisés. Pour gérer ce type de cas de figure, FML embarque un deuxième mécanisme de gestion de conflits qui opère au niveau des politiques et permet d'introduire une notion d'ordre entre les diverses politiques qui constituent un programme de contrôle. Ainsi, l'administrateur a la capacité de spécifier ses politiques sous forme d'une cascade où leur ordre d'apparition

dans la cascade affecte leur priorité. Ainsi une cascade de politiques FML peut être définie comme un ensemble fini de politiques P_1, ..., P_n munis d'une relation d'ordre total (>) entre ces politiques. On désigne une cascade de politique par : $P_1 < \ldots < P_n$, et où la politique P_1 est la moins prioritaire et P_n est la plus prioritaire.

Pour reprendre l'exemple d'une politique de contrôle d'accès fermée, l'extrait ci-dessous décrit un programme composé de deux politiques (P_1 et P_2), et où les politiques sont présentées de la plus prioritaire à la moins prioritaire ($P_1 < P_2$)

```
Policy P2
# authorise http flows
allow(Us, Hs, As, Ut, Ht, At, Prot, Req) ⟵ Prot = http
Policy P1
#default refusal
deny(Us, Hs, As, Ut, Ht, At, Prot, Req)
```

En dernier, il est à noter que le langage FML a été expérimenté dans des conditions réelles pour la gestion de la configuration de deux réseaux de productions. Le premier est le réseau d'une petite entreprise qui est constituée de 70 machines et de quatre commutateurs. Quant au second, c'est le réseau d'une unité hospitalière qui inclut plus de 200 machines. Aussi, la configuration des deux réseaux a principalement consisté en la gestion du contrôle d'admission et d'accès.

2.2.2 Nettle

Nettle [Voellmy et al., 2010] est un langage, embarqué au sein de Haskell, dédié à la programmation de réseaux SDN. Le concept clé derrière Nettle est l'utilisation du paradigme FRP (*Functional Reactive Programming*) [Elliott and Hudak, 1997] afin de permettre aux administrateurs, d'une part, de modéliser leur réseau sous forme d'un système dynamique et réactif, et d'autre part, d'exprimer leurs politiques de contrôle d'une manière déclarative et concise. Aussi, Nettle dispose d'une couche d'abstraction pour OpenFlow qui permet d'unifier tous les messages échangés entre le contrôleur et l'infrastructure physique au sein d'un même flux d'événements.

La figure 5 représente l'architecture de Nettle. Tout en bas, on retrouve la première couche qui englobe l'ensemble des commutateurs OpenFlow. Un niveau plus haut, on retrouve le langage Haskell, qui est le langage hôte dans lequel Nettle est embarqué. Vient après la bibliothèque HOpenFlow qui permet de construire les messages OpenFlow. La couche suivante représente une implémentation du paradigme FRP sous forme d'un contrôleur SDN. En dernier, viennent les modules de gestion réseaux qui implémentent les primitives et les abstractions qui permettent d'interagir avec l'infrastructure physique.

2.2.2.1 Nettle/FRP

La programmation réactive fonctionnelle (FRP) permet de modéliser, d'une manière déclarative, les entrées et les variables d'un programme sous forme de flux d'événements qui s'écoulent dans le temps. Dans le paradigme FRP, la succession d'événements dans le temps est désignée comme étant un signal, et un programme réactif consistera, au final, en une composition de transformateurs de signaux (c.-à-d. fonction signal).

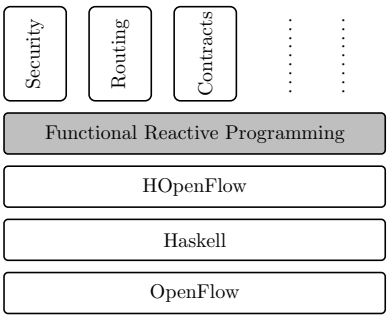

FIGURE 5. Architecture globale de Nettle [Voellmy et al., 2010]

Nettle embarque plusieurs fonctionnalités intégrées, incluant toutes les fonctions numériques évidentes qui peuvent être appliquées sur un signal. Un administrateur a aussi la possibilité de définir ses propres fonctions comme le montre l'exemple ci-dessous qui représente la définition de la fonction `sigfun` qui prend en entrée un signal `x` (un réel dans ce cas) et retourne toujours un nouveau signal avec des valeurs incrémentées de un.

```
sigfun :: SF Float Float
sigfun = proc x → do
y ← sin ≺ x + 1
returnA ≺ y
```

D'après les auteurs de Nettle, l'utilisation des signaux et des fonctions de transformation de signal permet de gérer plus facilement la (re)configuration d'un réseaux et cela d'une manière déclarative. Par exemple, un signal entrant peut représenter un flux de statistiques concernant un lien particulier dans le réseau, alors que le signal sortant peut être un flux de messages OpenFlow qui a pour objectif d'adapter la configuration de l'infrastructure physique. Ainsi, Nettle unifie tous les flux de messages dans des signaux continus qui sont défini à des points discrets dans le temps. Par ailleurs, il est à noter qu'un signal discret, qui transporte périodiquement des informations, est modélisé aussi comme un signal continu dont les valeurs peuvent être `Event`(données disponiblent) ou `NoEvent`(pas de données).

2.2.2.2 FRP pour OpenFlow

Le protocole OpenFlow permet d'assurer la communication entre le contrôleur SDN et les commutateurs OpenFlow. A travers le paradigme FRP, Nettle modélise tous ces flux de messages OpenFlow en deux grandes catégories de signaux : i) `SwitchMessages` pour les messages qui proviennent des commutateurs et partent vers le contrôleur, on peut citer par exemple le message de type `SwitchJoin` qui permet de signaler qu'un commutateur vient de se connecter à un contrôleur, et donc fait maintenant partie du réseau, ii) `SwitchCommand` pour les messages qui sont envoyés depuis le contrôleur à destination des commutateurs (c.-à-d. des commandes ou des requêtes), on peut citer par exemple le message `clearTable` qui permet d'effacer la table de commutation d'un équipement OpenFlow ou la commande `InserRule` qui, elle, permet d'ajouter une nouvelle règle dans la table de commutation.

Ainsi, le paradigme de programmation que préconise Nettle est qu'un programme de

contrôle consistera en une fonction signal, qui aura comme paramètre d'entrée un signal qui transporte des valeurs de type `SwitchMessages` (première catégorie de messages) et retournera un signal de sortie qui transporte des valeurs de type `SwitchCommand` (deuxième catégorie de messages).

L'extrait ci-dessous représente un exemple de programme Nettle qui permet de vider la table d'un commutateur OpenFlow.

```
clearOnJoin = proc evt → do
 switchJoinEvt ← switchJoins ≺ evt
 returnA ≺ (cleartable switchJoinEvt)
```

Dans cet exemple, on a utilisé la primitive `switchJoins` afin de récupérer le flux d'évènements qui représente la connexion d'un nouveau commutateur au contrôleur, après pour chacun de ces événements `switchJoinEvt` on applique la primitive `clearTable` qui permet d'effacer toutes les entrées d'une table de commutation. Le résultat final est retourné par la suite sous la forme d'un nouveau signal.

Les fonctions signal peuvent être composées afin de créer des modules de contrôle plus élaborés. Si on reprend l'exemple précédent, vu que les tables des commutateurs ont été intégralement effacées, chaque nouveau paquet qui arrive au niveau d'un commutateur sera automatiquement redirigé vers le contrôleur. On peut ainsi définir une nouvelle fonction signal qui permet de prendre en charge les paquets entrants. L'extrait ci-dessous représente une des implémentations possibles de cette fonction.

```
floodPackets = proc evt → do
 packetInEvt ← packetIns ≺ evt
 returnA ≺ sendReceivedPacket flood packetInEvt
```

Dans l'exemple ci-dessus, on applique d'abord `PacketIns` afin de récupérer l'événement qui contient les paquets entrants vers le contrôleur, puis on retourne un signal qui instruit le commutateur de faire une diffusion de ce paquet, en utilisant notamment la primitive `flood`.

Maintenant que les deux fonctions signal sont définies, il ne reste plus qu'à les composer ensemble dans une nouvelle fonction. Le programme ci-dessous représente la fonction `clearAndFlood` qui permet de composer parallèlement les deux fonctions `clearOnJoin` et `floodPackets`.

```
clearAndFlood = proc evt → do
 clearCmd ← clearOnJoin ≺ evt
 floodCmd ← floodPackets ≺ evt
 returnA ≺ clearCmd ⊕ floodCmd
```

2.2.3 Procera

En 2012, [Voellmy et al., 2012] ont proposé le langage Procera, ce dernier est très similaire au langage Nettle, étant donné que lui aussi est embarqué dans le langage Haskell et utilise le paradigme FRP pour exprimer des politiques de contrôle réactives. Plus concrètement, Procera repose sur trois concepts clés : 1) l'utilisation du paradigme FRP pour décrire des politiques réactives 2) le support des concepts de *Windowing* et *Aggregation* qui sont des fonctions signal et des structures de données particulières et 3) le support d'un ensemble de

directives qui permettent d'appliquer des contraintes sur les flux qui circulent au sein d'un réseau.

2.2.3.1 Signal et fonctions signal

Comme pour Nettle, Procera est basé sur le paradigme FRP et permet aux administrateurs de définir des fonctions signal qui recevront un flux d'événements en entrée (un signal), appliqueront une transformation sur ce signal et retourneront en sortie un nouveau signal. L'extrait ci-dessous représente un exemple simple d'un programme Procera qui décrit une politique de contrôle d'accès statique qui autorise uniquement l'accès aux flux qui ont comme destination l'adresse IP 128.36.5.0/24 :

```
proc world → do
 returnA ≺
  λreq → if destIP req 'inSubnet' ipAddr 128 36 5 0 // 24
       then allow else deny
```

Par ailleurs, Procera permet aussi de capturer un large éventail d'évènements et pas seulement ceux qui proviennent des commutateurs OpenFlow, mais aussi ceux qui sont émis par les utilisateurs, administrateurs ou les capteurs présents dans l'infrastructure physique. Un exemple d'événements est `authEvents`, qui est un type qui représente tous les événements d'authentification. Ces derniers transportent une paire d'informations (*device*, *user*) qui renseigne sur l'équipement et l'utilisateur authentifié.

Procera intègre aussi un mécanisme de création fonctionnel d'événements (*event comprehension*) qui permet de filtrer, transformer ou de fusionner des événements. Par exemple l'extrait ci-dessous montre une instruction qui permet d'extraire, à partir d'un flux d'événements, tous les évènements qui transportent une valeur qui est un nombre premier, puis les incrémente de un.

```
[n+1 | n ← e, prime n]
```

2.2.3.2 Historique d'évènements

Les concepts de *Windowing and Aggregation* (fenêtrage et agrégation) peuvent être considérés comme les plus importants apports de Procera par rapport à Nettle. En effet, ces deux concepts représentent un ensemble de fonctions signal et de structure de données qui sont destinées à faciliter grandement la spécification de politiques réactives. La figure 6 représente un résumé de l'ensemble de ces primitives.

Concrètement, le concept de *Windowing* consiste en un groupe de primitives qui permettent de construire une fenêtre d'historique d'évènements. En particulier, on peut citer `since dt`, `limit size` et `limitBy attribute size` qui implémentent des fenêtres basées sur une date, un nombre et un attribut particulier.

Le deuxième groupe de primitives (*Aggregation*) inclut des fonctions signal qui prennent en entrée un historique d'événements, applique, éventuellement, un traitement sur ces évènements, puis retourne une structure de données. On peut citer par exemple la primitive `accumSet` dont la valeur de sortie est un signal qui transporte un ensemble qui contient des accumulations de toutes les valeurs de l'historique des événements reçus en entrée.

```
since dt                : génère une fenêtre historique datant depuis dt seconds.
limit size              : génère une fenêtre historique jusqu'à size événement.
limitBy attribute size  : génère une fenêtre avec size événement
                          dont la valeur est égale à attribute.
accumList               : accumule les événements présents dans une fenêtre
                          et les placent dans une liste.
accumSet                : accumule les événement présent dans une fenêtre
                          et les place dans un ensemble.
group op                : accumule un dictionnaire de clé-valeur et applique la
                          fonction op aux valeur qui ont la même clé.
add, remove, +          : ajoute, supprime et combine des événements.
```

FIGURE 6. Primitives de fenêtrage et d'agrégation

Le programme ci-dessous représente un exemple d'utilisation du concept de Windowing et Aggregation.

```
proc world → do
 recent ← since (daysAgo 5) ≺ add (usageEvents world)
 usageTable ← (group sum) ≺ recent
 returnA ≺ usageTable
```

Dans cet exemple, le programmeur commence d'abord par filtrer l'évènement `usageEvents`, qui donne des informations sur la consommation d'un utilisateur. La primitive `since` est utilisée avec le paramètre cinq jours (`daysAgo 5`) pour récupérer l'historique sur une période de cinq jours. Le résultat, qui est une fenêtre d'historique, est placé dans la variable `recent`. Puis, ce résultat est passé à la directive d'agrégation `group` qui utilise le paramètre `sum`, qui applique l'opération de somme sur l'ensemble des valeurs contenues dans cet historique pour obtenir à la fin une table qui associe à chaque utilisateur la somme de sa consommation. En dernier, le résultat est retourné sous la forme d'une variable (un signal) qui évolue dans le temps et dont la valeur dépend des évènements `usageEvents`.

2.2.3.3 Contraintes sur les flux

La dernière brique de Procera est un ensemble de primitives qui permettent d'appliquer des contraintes sur un flux particulier. En effet, en utilisant le paradigme FRP et les concepts de *Windowing* et *Aggregation*, un programmeur peut facilement modéliser les événements et les interactions d'un programme, puis leur associer un ensemble de contraintes (auxquelles seront soumis les flux de l'infrastructure physique). Ci-dessous sont listées les principales contraintes proposées par Procera :

- `allow` : autorise un paquet à travers le réseau.
- `deny` : bloque un paquet.
- `rate limit` : limite la bande passante à `limit`.
- `redirect host` : redirige un paquet vers un host particulier.

2.2.4 Frenetic

Apparu en 2010, Frenetic [Foster et al., 2010] est un langage dédié pour le contrôle d'architectures réseaux SDN. L'objectif du langage Frenetic est de fournir une abstraction de

haut niveau de la fonction de contrôle, permettant ainsi aux opérateurs de pouvoir préciser leurs besoins métiers tout en s'affranchissant des mécanismes de configuration de bas niveau, spécifiques à des constructeurs d'équipements réseaux et souvent sources d'erreurs.

Pour ce faire, Frenetic fournit une abstraction simple et intuitive des deux grandes fonctions qui constituent une boucle de contrôle, à savoir la surveillance de l'état du réseau et l'expression des politiques de gestion. Pour chacune de ces deux grandes fonctions, Frenetic fournit un sous-langage dédié. D'autre part, Frenetic utilise le standard OpenFlow pour accéder au plan de données des équipements réseaux en vue d'y installer les règles de commutation nécessaires à la mise en œuvre des opérations de surveillance et de contrôle du réseau.

2.2.4.1 Sous-langage de surveillance

Frenetic définit un langage de requêtes de haut niveau, avec une syntaxe très similaire à celle du langage SQL, qui permet de s'abonner à des flux d'informations sur l'état du réseau. Il permet aussi aux programmeurs de contrôler les informations qu'ils reçoivent à l'aide d'une collection d'opérateurs de classification, de filtrage, de transformation et d'agrégation des flux de paquets qui traversent le réseau. Dans ce qui suit, nous présentons brièvement les éléments syntaxiques qui peuvent constituer une requête sur l'état d'un réseau.

La syntaxe d'une requête du langage Frenetic est donnée dans la figure 7. Toutes les clauses qui composent une requête sont facultatives, excepté la clause `Select(a)`, qui permet d'agréger le résultat retourné par la requête selon la méthode `a` . Cette dernière définit le type du résultat qui doit être retourné, c'est-à-dire, soit les paquets eux-mêmes (`packets`), soit le nombre de paquets (`counts`) ou encore la somme des tailles des paquets (`bytes`). Il est aussi à noter que le symbole « * » est utilisé pour combiner les clauses d'une requête.

```
Queries     q ::= select(a)*
            Where(fp)*
            GroupBy([qh_1, ... , qh_n])*
            SplitWhen([qh_1, ... , qh_n])*
            Every(n)*
            Limit(n)
Aggregates  a ::= packets | counts | bytes
Headers     qh ::= inport | srcmac | dstmac | ethtype | vlan |
            srcip | dstip | protocol | srcport | dstport | switch
Pattern     sfp ::= true_fp() | qh_fp(n) |
            and_fp([fp_1, ... , fp_n]) |
            or_fp([fp_1, ... , fp_n]) |
            diff_fp(fp_1, fp_2) | not_fp(fp)
```

FIGURE 7. Requêtes de Frenetic

La clause `Where(fp)` permet de filtrer les résultats, ne retenant que les paquets répondant à un filtre `fp`. Les filtres peuvent être définis sur la base des champs d'en-tête des paquets (adresses MAC, adresses IP, etc.) et des informations de localisation (commutateur, port d'entrée). Des modèles de filtres plus complexes peuvent aussi être construits en utilisant des opérations sur les ensembles telles que : l'intersection (`and_fp`), l'union (`or_fp`), la différence (`diff_fp`) ou le complément (`not_fp`).

La clause `GroupBy([qh`$_1$ `, .... , qh`$_n$`])` permet de subdiviser l'ensemble des paquets demandés en sous-ensembles suivant la valeur des champs d'en-tête `qh`$_1$ jusqu'à `qh`$_n$.

La clause `SplitWhen([qh`$_1$ `, .... , qh`$_n$`])` est similaire à `GoupBy`. Cependant, `GroupBy` produit un sous-ensemble pour tous les paquets ayant des valeurs identiques sur les champs d'en-tête données (indépendamment de leur ordre d'arrivée), alors que `SpiltWhen` génère un nouveau sous ensemble à chaque fois que la valeur de l'un des champs d'en-tête change. Par exemple, supposons qu'une requête subdivise le trafic réseau selon l'adresse IP source et que les paquets avec une adresse IP source 10.0.0.1, 10.0.0.2 et 10.0.0.1 arrivent dans cet ordre. Dans ce cas, `SplitWhen` génère trois sous-ensembles (le premier et le troisième paquet sont placés dans des ensembles distincts, en raison de leur adresse IP différente de l'adresse du paquet intermédiaire). Si l'ordre d'arrivée était : 10.0.0.1, 10.0.0.1, 10.0.0.2, alors seulement deux sous-ensembles seraient générés.

Enfin, la clause `Every(n)` permet de regrouper les paquets qui arrivent dans la même fenêtre temporelle `n`. Quant à `Limit(n)` elle permet de limiter le nombre de paquets dans chaque sous-ensemble à `n`.

Frenetic définit donc un sous-langage de requête simple mais assez expressif qui permet aux programmeurs de spécifier ce qu'ils veulent surveiller à l'intérieur d'un réseau sans se soucier de comment cela est mis en œuvre. Le compilateur du langage permet notamment de traduire les besoins haut niveau de contrôle et de surveillance en un ensemble de commandes OpenFlow qui permettront d'installer sur les équipements réseaux concernés les règles de commutation correspondantes aux besoins exprimés.

2.2.4.2 Sous-langage de contrôle

Les programmeurs peuvent gérer leurs infrastructures physiques en utilisant la bibliothèque FRP de Frenetic (le sous-langage étant basé sur ce paradigme). L'opération de configuration de base consistera alors en la construction d'une règle de commutation en utilisant la primitive `Rule`, à laquelle on passe en paramètre un filtre et une liste d'actions à exécuter.

Le filtre utilisé pour construire une règle est similaire à celui utilisé pour la construction des requêtes de surveillance. Concernant les actions, Frenetic inclut les actions de base qui peuvent être réalisées par les commutateurs OpenFlow, entre autres : `forward(p)` qui permet de relayer, au sein d'un même commutateur, un paquet vers un port de sortie, `flood()` qui permet de diffuser un paquet sur tous les ports de sortie d'un commutateur, `controller()` qui ordonne au commutateur d'envoyer le paquet vers le contrôleur SDN et enfin `modify(h=v)` qui permet de modifier la valeur d'un champ d'en-tête `h` à `v`. La figure 8 représente un résumé des primitives de contrôle de base qui sont fournis avec Frenetic.

Par ailleurs, étant donné que le sous-langage de contrôle de Frenetic est basé sur le paradigme FRP, l'opération de configuration d'une infrastructure physique consistera en la génération d'un flux d'évènements.

Pour envoyer et recevoir des événements à partir de l'infrastructure physique, Frenetic implémente le concept de *Listeners*, ce dernier joue le rôle de consommateur d'évènements. Ainsi, `Send` est un listener qui permet d'envoyer un paquet vers un commutateur de

```
Events:
   Seconds:     nombre de seconds écoulées depuis le lancement du programme.
   SwitchJoin: identifiant du commutateur qui vient de rejoindre le réseau.
   SwitchExit: identifiant du commutateur qui vient de quitter le réseau.
   PortChange: triplet (sw, port, bool) qui renseigne sur l'état d'un port.
Listeners:
   Print:     consomme des informations textuelles et les affiches.
   Register: consomme une politique et l'installe sur l'infrastructure physique.
   Send:      consomme un triplet (commutateur, paquet, actions). Permet d'envoyer
              un paquet vers un commutateur et de l'instruire d'appliquer
              une liste d'actions sur le paquet.
Actions:
   forward:      commute un paquet vers un port de sortie.
   flood:        diffuse un paquet sur tous les ports d'un commutateur.
   controller:   redirige un paquet vers le controleur.
   modify(h,v): modifie la valeur du champ h dans l'entête d'un paquet à la valeur v.
```

FIGURE 8. Primitives de contrôle de Frenetic

l'infrastructure physique. Pour ce faire, `Send` consomme des évènements qui transportent un triplet qui contient le paquet à envoyer, le commutateur cible et la liste d'actions à appliquer sur le paquet par le commutateur. Le listener `Register`, quant à lui, consomme des événements de type politique (un dictionnaire qui associe un commutateur à une liste de règles) et se charge par la suite de les installer sur des commutateurs OpenFlow.

Par ailleurs, en plus des évènements de type politique, Frenetic prend aussi en charge des évènements qui proviennent de l'infrastructure physique comme `SwicthJoin` et `SwitchExit` qui indiquent respectivement qu'un commutateur a rejoint ou a quitté le réseau géré.

Un exemple simple d'un programme Frenetic est montré ci-dessous, où l'objectif est d'installer une politique de commutation sur les équipements OpenFlow qui viennent de rejoindre le réseau (L'exemple suppose que chaque commutateur dispose de deux ports).

```
rules = [Rule(inport_fp(1), [forward(2)]),
         Rule(inport_fp(2), [forward(1)])]
def repeater():
 (switchJoin() >>
 Lift(lambda switch:{switch:rules}) >>
 Register())
```

La politique qui est décrite dans le programme ci-dessus contient deux règles, la première capture tous les paquets qui arrivent sur le port numéro 1 et les commute sur le port 2, et inversement la deuxième règle capture tous les paquets qui arrivent sur le port 2 et les commute sur le port 1. La fonction `repeater` réceptionne le flux d'événements `SwitchJoin`, puis construit un dictionnaire qui associe les deux règles précédentes à chaque nouveau commutateur qui se connecte, puis passe le résultat au consommateur d'événements `Register` qui se charge d'installer cette politique sur les commutateurs OpenFlow.

2.2.5 NetCore

NetCore (*Network Core Programming Language*) [Monsanto et al., 2012] est un langage qui fait suite aux travaux menés sur Frenetic. L'apport principale de NetCore, par rapport à Frenetic, consiste en la proposition de nouveaux algorithmes de compilation qui permettent

de mieux gérer les interactions entre les commutateurs OpenFlow et le contrôleur SDN. Plus concrètement, NetCore essaye de traiter le plus possible de paquets au niveau des commutateurs OpenFlow et de ne remonter vers le contrôleur que les paquets qui sont expressément souhaités.

Pour ce faire, NetCore permet aux administrateurs de spécifier des règles de commutation qui utilisent des filtres incomplets (ou joker), sachant que la partie incomplète d'un filtre est générique et peut s'appliquer sur n'importe quelle valeur. La possibilité d'utiliser des filtres incomplets a pour conséquence directe que, d'une part NetCore aura besoin d'installer moins de règles sur les commutateurs (une règle avec un filtre incomplet peut couvrir plusieurs règles avec des filtres complètement spécifiés), et d'autre part, l'utilisation des filtres incomplets est un moyen pour traiter des flux dont on n'a pas forcement une connaissance précise.

2.2.5.1 Politiques de commutation de base

Dans NetCore, une politique de commutation consiste en la spécification d'un prédicat (un filtre) qui capture un flux de paquets, puis l'associer à un ensemble de localisations qui désignent les emplacements vers lesquels ces paquets doivent être transmis. L'extrait ci-dessous montre un exemple simple d'une politique NetCore qui indique que tous les paquets qui proviennent du sous-réseau "10.0.0.0/8" (filtre avec une partie générique), excepté ceux de l'hôte "10.0.0.1" (filtre complètement spécifié) et les flux web, doivent être transmis au commutateur numéro 1.

```
|| SrcAddr:10.0.0.0/8 \ (SrcAddr:10.0.0.1 ∪ DstPort:80) → switch{Switch 1}
```

Par ailleurs, on peut aussi remarquer dans l'exemple ci-dessus qu'un administrateur a la possibilité de construire des prédicats complexes en utilisant les opérateurs d'union (∪), d'intersection(∩), négation(¬) ou différence (\).

2.2.5.2 Politiques dynamiques

La partie dynamique du langage NetCore permet de spécifier des politiques dont le comportement dépend de l'état du réseau (c.-à-d. l'historique du trafic réseau). En effet, NetCore fournit un prédicat spécial appelé `inspect` qui interroge l'historique du trafic réseau. Pour en expliquer son utilisation, nous allons considérer le cas d'utilisation suivant, où l'objectif global est de spécifier une politique d'authentification. La figure 9 schématise le réseau physique sur lequel sera appliquée cette politique. Dans ce cas d'utilisation, les utilisateurs internes qui sont dans le réseau `N1` doivent d'abord s'authentifier auprès du serveur afin de pouvoir accéder à Internet (représenté par le réseau `N2`). On suppose aussi qu'un hôte est considéré comme authentifié si et seulement s'il a reçu un paquet qui provient du serveur d'authentification. En dernier, les hôtes du réseau `N2` n'ont pas le droit d'accéder au serveur d'authentification.

Le prédicat `inspect e f` prend toujours en entrée deux paramètres : un filtre `e`, qui est appliqué sur l'historique du trafic et une fonction booléenne `f`. Le filtre `e` génère un état Σ qui est une collection de statistiques, représentée abstraitement par un ensemble de couples

commutateur-paquet. Par la suite, la fonction `f` reçoit notamment cet état comme entrée afin de l'utiliser dans son processus de prise de décision.

```
(InPort:Network 1 ∩ inspect ps auth → {Network 2})
∪ (InPort:Network 1 ∩ ¬ (inspect ps auth) → {Server A})
∪ (InPort:Server A ∪ InPort:Network 2 → {Network 1})

ps = InPort:Server A
auth(Σ, s, p) = any(isAddr p) Σ
isAddr p (_ , p') = p.SrcAddr == p'.DstAddr
```

Dans l'exemple ci-dessus, un paquet `p` qui arrive du réseau `N1` sur le commutateur `S` est transmis au contrôleur qui l'évalue sur le prédicat `inspect ps auth`. Le filtre `ps` capture tous les paquets de l'historique qui proviennent du serveur d'authentification. Ce résultat est passé à la fonction `auth` qui teste si l'adresse IP source dy paquet actuel `p` est inclut dans cet ensemble. Si la fonction `auth` retourne une réponse positive, alors une règle de commutation est installée pour l'hôte en question, sinon le paquet sera rejeté.

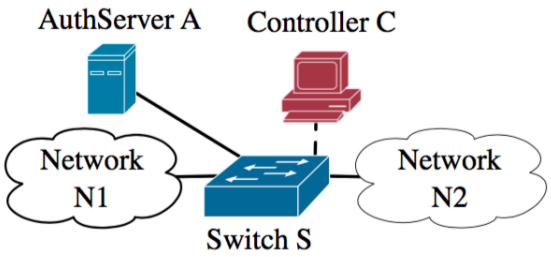

FIGURE 9. NetCore - Exemple topologie [Monsanto et al., 2012]

2.2.5.3 Compilation

Lors de la compilation des politiques de contrôle, l'environnement d'exécution de NetCore essaye de faire en sorte d'installer des règles qui remontent le moins possible de paquets vers le contrôleur SDN. Pour ce faire, en plus de permettre l'utilisation des filtres incomplets, l'environnement d'exécution de NetCore commence d'abord par compiler les politiques qui ne contiennent pas de prédicat de type `inspect`, puis il génère les règles de bas-niveau correspondantes et les installe sur les commutateurs. Pour les paquets qui ne peuvent pas être gérés par les commutateurs et doivent être transmis au contrôleur, ce qui inclus le cas du prédicat `inspect`, le compilateur identifie deux situations :

- La politique qui concerne le paquet est invariante. En d'autres termes, quand elle est appliquée sur ce paquet elle retournera toujours le même résultat d'actions pour les paquets de ce même flux, indépendamment du temps et de l'état du réseau.
- La politique qui concerne le paquet (et tous ceux qui appartiennent au même flux que lui) est volatile. En d'autres termes, les actions qui peuvent être appliquées sur les paquets d'un même flux peuvent changer dans le futur.

Dans le premier cas, le compilateur installe des règles qui prennent en charge les paquets qui sont semblables à celui qui vient d'être traité au niveau du contrôleur. Par exemple, dans le cas d'utilisation précédemment présenté, si un hôte est authentifié il va le rester tout le long de la période d'exécution, donc le compilateur va installer une règle qui se chargera de commuter tous les paquets de l'hôte. Dans le second cas, le compilateur ne peut pas se permettre d'installer des règles sur les commutateurs, étant donné que le prochain paquet pourrait être traité différemment. Ainsi, pour le second cas de figure, les paquets sur lesquels s'applique une politique volatile sont toujours envoyés vers le contrôleur SDN.

Par ailleurs, le compilateur a toujours besoin qu'un administrateur lui fournisse, manuellement, des informations sur l'invariance d'une fonction. Ces informations prennent la forme d'une fonction, écrite par l'administrateur, qui indique si une fonction est invariante ou non. Par exemple, dans le cas d'utilisation précédent, la fonction d'invariance `auth_inv` est triviale, comme cela est montré ci-dessous, étant donné que c'est une fonction qui retourne vrai si la fonction `auth` est vrai (un hôte reste toujours authentifié).

```
auth_inv(Σ, s, p) = auth(Σ, s, p)
```

2.2.6 Pyretic

Pyretic [Monsanto et al., 2013] est un langage qui fait suite aux travaux menés sur le langage NetCore. Le langage Pyretic intègre trois principales caractéristiques : 1) Les politiques de contrôle sont considérées comme des fonctions qui prennent en entrée un paquet avec une localisation (c.-à-d. un commutateur et un port d'entrée) et retournent en sortie un nouveau paquet avec une ou plusieurs localisations différentes, 2) Pyretic inclut des opérateurs de composition parallèle et séquentielle qui permettent de composer facilement les primitives de base du langage ainsi que les modules de contrôles existants, 3) Pyretic introduit aussi le concept *d'objets réseau* qui permettent de restreindre ce qu'un module de contrôle peut *voir* et *faire*.

2.2.6.1 Primitives de base

Pyretic définit trois principaux ensembles de primitives (figure 10) : les prédicats, les actions et les requêtes. Les actions sont des fonctions qui reçoivent en entrée un paquet associé à une localisation et retournent en sortie un paquet avec une ou plusieurs localisations différentes. On peut citer par exemple l'action `fwd(port)` qui commute un paquet, au sein d'un même équipement, d'un port d'entrée vers un port de sortie, ou l'action `flood()` qui prend un paquet en entrée et le diffuse par la suite sur tous les ports de sortie. La fonction `drop`, quant à elle, prend en entrée un paquet et retourne en sortie un ensemble vide, ce qui correspond à un rejet du paquet. L'action `modify(h, v)` est une action particulière, étant donné qu'elle ne change pas la localisation d'un paquet mais elle modifie la valeur de son champ d'en-tête `h` à `v`.

```
Actions:
   A ::= drop | fwd(port) | flood | push(h=v) | pop(h)
prédicats:
   P ::= all_packets | no_packets | match(h=v) | P & P | (P|P) | ~P
Requêtes:
   Q ::= packets(limit, [h]) | counts(every, [h])
Politiques:
   C ::= A | Q | P[C] | C+C | C>>C | if_(P, C, C)
```

FIGURE 10. Primitives du langage Pyretic

Prédicats En complément des actions, Pyretic fournit aussi un ensemble de prédicats qui permettent de capturer des paquets dans le réseau. En effet, les prédicats (ou filtres) sont essentiels pour la construction des politiques de contrôle. Plus concrètement, un prédicat appliqué sur un flux permet de retourner l'ensemble des paquets qui satisfont la condition du prédicat. Le prédicat le plus communément utilisé dans Pyretic est `match(h=v)` qui, s'il est appliqué sur un flux de paquets, va retourner uniquement les paquets dont la valeur de leur en-tête `h` est égale à `v`.

Opérateurs de composition Dans Pyretic toute primitive est considérée comme une politique. Néanmoins, pour disposer de réels modules de contrôle qui soient utilisables dans la pratique, les administrateurs doivent composer ces primitives en utilisant les opérateurs de composition définis par Pyretic. En effet, les opérateurs de composition sont des mécanismes centraux dans le processus de construction des politiques. Le premier est l'opérateur de composition parallèle (+), et il permet de donner l'illusion que deux politiques sont exécutées simultanément sur le même flux. Le deuxième opérateur (»), quant à lui, permet de composer d'une manière séquentielle deux ou plusieurs politiques de sorte à ce qu'une politique opère sur le résultat qui a été généré par la politique qui la précède dans la chaîne de composition séquentielle. Pyretic fournit aussi la primitive `if_(P, C1, C2)` qui permet d'appliquer l'action `C1` sur les paquets qui vérifient le prédicat `P` et la politique `C2` sur le reste des paquets.

2.2.6.2 Primitives dynamiques

Le dernier type de primitives de Pyretic sont les requêtes. Concrètement, les requêtes sont des politiques qui permettent de récupérer des informations qui proviennent de l'infrastructure physique et de les envoyer par la suite vers l'environnement de contrôle. Pyretic définit deux types de requêtes : `count` et `packet`, qui permettent respectivement, soit de remonter des statistiques sur les paquets traités au niveau de l'infrastructure physique, soit de remonter des paquets réseau entiers. L'administrateur peut spécifier des fonctions *callback* sur chacune de ces requêtes, afin d'enclencher l'exécution d'un traitement à l'arrivée d'une information. Aussi, la requête `packet(limit, [h])` nécessite la spécification de deux paramètres, le premier est `limit` qui fixe le nombre de paquets à remonter vers l'environnement de contrôle, quant au deuxième il représente une liste de champs d'en-tête qui, si spécifiée, sera utilisée par l'environnement d'exécution comme une clé afin d'établir une distinction entre l'ensemble des informations qui seront remontées. Concernant la

requête `count(every, [h])`, elle permet de remonter des statistiques sur le nombre de paquets et d'octets traités et cela suivant une fréquence qui a été préalablement fixée par l'administrateur en spécifiant le paramètre `every`.

L'extrait ci-dessous représente un exemple simple d'un programme Pyretic. L'objectif dans ce cas d'utilisation est de composer les deux fonctions (ou politiques) `flood` et `dpi`.

```
def printer(pkt):
 print pkt

def dpi():
 q = packet(None, [])
 q.when(printer)
 return q

def main():
 return match(srcip="1.2.3.4") >> (dpi() + flood())
```

Dans l'exemple ci-dessus, la fonction `dpi` commence d'abord par créer une requête `q` qui permet de remonter des paquets. La limite est positionnée à `None`, ce qui veut dire que tous les paquets doivent être remontés. Ensuite, la fonction `printer`, qui permet d'afficher un paquet, est enregistrée comme une fonction *callback*. Dans la fonction `main`, un filtre qui capture tous les paquets qui ont une adresse IP source "1.2.3.4" est défini. Les paquets qui sont retournés par ce filtre sont passés aux fonctions `dpi` et à l'action `flood` grâce à l'utilisation de l'opérateur de composition séquentielle. Ces deux fonctions seront exécutées en parallèle sur ce flux de paquets à l'aide de l'opérateur de composition parallèle.

2.2.6.3 Objet réseau et modèle d'abstraction d'un paquet

Afin d'augmenter la modularité des programmes de contrôle, Pyretic offre aux administrateurs la possibilité de spécifier leurs politiques de contrôle sur des topologies abstraites. Cela est principalement possible grâce à l'utilisation d'un modèle d'abstraction de paquet que propose Pyretic. En effet, ce modèle d'abstraction permet de voir un paquet comme un simple dictionnaire qui associe des noms d'en-tête à des valeurs. Ces en-têtes incluent la localisation du paquet (commutateur et port d'entrée), les en-têtes standard d'OpenFlow, mais surtout il permet aux administrateurs d'ajouter leurs propres en-têtes personnalisés, qui sont gérés par l'environnement d'exécution comme des en-têtes virtuels qui peuvent être associés à diverses structures de données. Par ailleurs, en plus de cette possibilité d'étendre la "largeur" d'un paquet, le modèle d'abstraction permet aussi d'étendre sa "hauteur" en étant capable d'empiler des valeurs sur un même champ d'en-tête, sachant qu'un module de contrôle a uniquement accès aux champs qui sont au sommet de la pile.

En se basant sur ce modèle d'abstraction, un administrateur est capable de créer des objets réseau abstraits afin de s'affranchir de la complexité de l'infrastructure physique et des détails de configuration qui ne sont pas pertinents ses politiques. Plus concrètement, un objet réseau consiste en une topologie abstraite, une politique de contrôle appliquée sur cette topologie abstraite et enfin des fonctions de mapping.

Une topologie abstraite (aussi appelée réseau dérivé) est une association un-à-plusieurs ou plusieurs-à-un entre des équipements virtuels et les équipements physiques. La figure

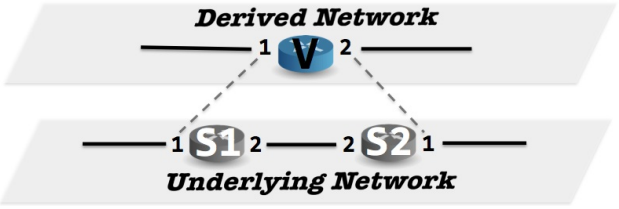

FIGURE 11. Topologies physique et abstraite - Pyretic [Monsanto et al., 2013]

11 illustre un exemple de création d'une topologie abstraite qui prend la forme d'un seul commutateur (ou *One Big Switch*) qui englobe en son sein tous les équipements présents dans une infrastructure physique sous-jacente. Le programme suivant représente la définition de la topologie virtuelle. Cette dernière consiste concrètement en un dictionnaire qui lie les composants du niveau virtuel aux équipements du niveau physique.

```
def bfs_vmap(topo):
 vswitch = 1
 vport = 1
 for (switch, port) in topo.egress_locations:
  vmap[(vswitch, vport)] = (switch, port)
  vport +=1
 return vmap
```

La prochaine étape dans l'implémentation de l'objet réseau consiste en la définition d'une fonction qui transforme les politiques écrites sur la topologie abstraite en politiques qui peuvent être installées sur l'infrastructure physique (fonctions de mapping). Pour ce faire, l'administrateur doit spécifier une politique de transformation. Cette dernière se base sur l'utilisation de trois politiques auxiliaires :

- Une politique *ingress* qui remonte les paquets du réseau sous-jacent vers le réseau dérivé. Cela est réalisé en empilant les identifiant des équipements et des ports virtuels sur les paquets.
- Une politique *egress* qui permet de faire redescendre les paquets du réseau dérivé vers le réseau sous-jacent. Cela est réalisé en dépilant les identifiant virtuels.
- Une politique *fabric* qui implémente les règles de commutation des équipements du réseau dérivé en utilisant les commutateurs et les liens du réseau sous-jacent.

Les extraits suivant illustrent ces trois politiques auxiliaires pour le cas d'utilisation du *One Big Switch*.

```
Ingress_policy = (match(switch=s1, inport=1)
   [push(vswitch=v, vinport=1)]
        | (match(switch=s2, inport=2)
   [push(vswitch=v, vinport=2)]
```

La politique `ingress_policy` permet simplement d'empiler les en-têtes virtuels qui font référence au commutateur virtuel `v`, qui englobera ainsi les deux commutateurs physiques `s1` et `s2`. Aussi, on peut noter que le concept d'extension d'un paquet en profondeur peut

permettre d'avoir plusieurs niveaux d'abstraction, ou chaque niveau correspondra à un empilement d'un en-tête virtuel.

```
egress_policy = match(vswitch=v)
[if_(match(switch=s1, voutport=1)
| match(switch=s2, voutport=2), pop(vswitch, vinport, voutport), passthrough)]
```

La politique `egress_policy`, quant à elle, est totalement symétrique à la politique `ingress_policy`, elle permet ainsi de dépiler les en-têtes virtuels qui font référence au commutateur virtuel `v`.

```
Fabric_policy = match(vswitch=V) [
(match(switch=s1, voutport=1)[fwd(1)]
| (match(switch=s1, voutport=2)[fwd(2)]
| (match(switch=s2, voutport=1)[fwd(2)]
| (match(switch=s2, voutport=2)[fwd(1)])]
```

Enfin, la politique `fabric_policy` permet de définir les règles de commutation entre ports adjacents du commutateur virtuel V en faisant référence aux commutateurs physiques qui le composent.

La dernière étape consiste à faire passer ces trois fonctions auxiliaires, plus la fonction qui sera exécutée sur la topologie virtuelle (un programme Pyretic à l'exemple de la fonction `dpi`), à une fonction de transformation, cette dernière se chargera de transformer cet ensemble de politiques en un ensemble de règles qui peuvent être installées sur l'infrastructure physique. L'extrait ci-dessous présente la fonction de transformation utilisée pour le cas d'utilisation du *One big switch*.

```
def virtualize(ingress_policy, egress_policy, fabric_policy, derived_policy):
 return if_(~match(vswitch=None), (ingress_policy >>
   Move(switch=vswitch, inport=vinport) >>
   derived_policy >>
   Move (vswitch=switch, vinport=inport, voutport=outport)), passthrough) >>
   fabric_policy >>
   egress_policy.
```

2.2.7 Splendid Isolation

Splendid Isolation [Gutz et al., 2012] est un langage de haut-niveau qui a été proposé afin de fournir aux administrateurs un outil qui permet d'assurer divers critères d'isolation entre les différents programmes de contrôle qui opèrent au-dessus d'une même architecture physique, ou entre les différents modules qui composent un programme de contrôle. En effet, l'objectif principal de Splendid Isolation est d'assurer l'isolation au niveau langage, à travers l'utilisation d'un compilateur et non plus en ayant recours à des mécanismes de bas-niveau tels que les VLAN ou à des équipements spécifiques tels que les pare-feu ou à des couches logicielles intermédiaires tel l'hyperviseur FlowVisor [Sherwood et al., 2009]. Pour ce faire, le langage s'appuie sur le concept de *Slice Abstraction* qui consiste à découper l'infrastructure physique en plusieurs *slices* et où chacun d'entre eux dépend d'un programme de contrôle différent.

Ainsi, dans le contexte de cette proposition, un *slice* (c.-à-d. tranche) réseau inclut trois principaux composants :

- Un graphe qui représente une vision restreinte de l'infrastructure physique. Ce dernier inclut des commutateurs semblables (avec des ports et des liens entre eux) à ceux présents dans l'infrastructure physique.
- Des informations de mapping qui permettent d'établir une association entre les nœuds du graphe et les équipements de l'infrastructure physique.
- Un ensemble de prédicats qui permettent de dire quels sont les paquets qui sont autorisés à entrer dans le slice.
- Chaque slice dispose d'un programme de contrôle dédié qui dicte sa politique de traitement des paquets qui se trouvent en son sein, ce programme peut être écrit avec un langage différent tel que NetCore.

Par ailleurs, il est à noter que splendid isolation n'autorise qu'une association un-à-un entre les nœuds du graphe et les commutateurs physiques, ainsi que pour les ports physiques. En d'autres termes, un nœud virtuel doit être associé à un unique nœud physique, et un port virtuel doit nécessairement être associé au plus à un seul port physique. Par contre, un nœud physique ou un port peut être associé à plusieurs slices, sachant que, si un paquet arrive au niveau d'un port qui est connecté à plusieurs slices, une copie de ce paquet est alors envoyée à chaque slice.

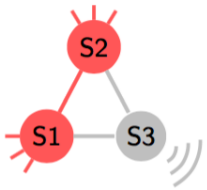

FIGURE 12. Extrait topologie pour l'exemple de Splendid Isolation

A titre d'exemple, considérons la figure 12 qui représente un fragment du réseau d'une université. Les hôtes connectés au commutateur S1 sont les utilisateurs de confiance à l'exemple du doyen ou des professeurs. Sur le commutateur S2 sont connectés les serveurs de l'université qui hébergent des informations sensibles. Quant aux utilisateurs non fiables, ils sont connectés au commutateur S3. La politique globale qu'on souhaite installer est la suivante : les hôtes qui sont connectés au commutateur S1 peuvent communiquer avec les serveurs connectés au commutateur S2. Les hôtes qui sont connectés au commutateur S3 peuvent uniquement communiquer avec les services web qui sont hébergés sur les serveurs connectés sur le commutateur S2. Enfin, les administrateurs peuvent envoyer des paquets afin de surveiller le fonctionnement des commutateurs et des liens, mais ils ne peuvent aucunement communiquer avec les hôtes qui sont connectés aux commutateurs.

En utilisant le concept de slice, il est très facile d'écrire un programme qui implémente correctement cette politique d'isolation. En effet, il suffit simplement de créer un slice pour chaque classe de trafic (Figure 13), puis de lancer un programme de contrôle distinct sur chaque slice. Ainsi, le slice rouge permet de prendre en charge le trafic échangé entre les hôtes

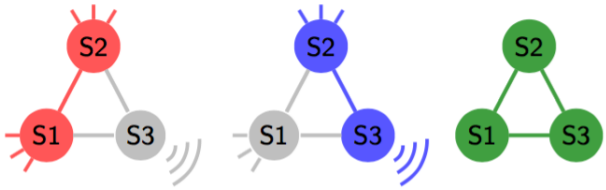

FIGURE 13. Rouge : trafic de confiance - Bleu : trafic non fiable
Vert : trafic des administrateurs

du commutateur S1 (utilisateurs de confiances) et les serveurs connectés au commutateur S2. Le slice bleu gère le trafic entre les hôtes du commutateur S3 et les serveurs du S2, sachant que dans la définition de ce slice des restrictions ont été imposées afin de restreindre les communications uniquement au flux web comme cela est montré dans l'extrait de programme ci-dessous. Le slice vert, quant à lui, permet de gérer le trafic entre les trois commutateurs, mais n'inclut pas les hôtes qui sont connectés à eux.

```
# Definition du slice BLEU
topo = nxtopo.NXTopo()
topo.add_switch(name="X", ports=[1,2,3,4])
topo.add_switch(name="Y", ports=[1,2,3,4])
topo.add_link(("X", 4),("Y", 4))
# mappings
s_map = {"X":"S2", "Y":"S3"}
p_map = identity_port_map(topo, s_map)
maps = (s_map, p_map)
# prédicats
preds = \ ([(p, headers("srcport", 80)) for p in topo.edge_ports("X")] +
          [(p, headers("dstport", 80)) for p in topo.edge_ports("Y")])
# constructeur du slice
slice = Slice(topo, phy_topo, maps, preds)
```

En dernier, il suffit d'écrire des programmes de commutation simple pour chaque slice. L'extrait ci-dessous montre un exemple de programme NetCore pour le slice bleu qui implémente une politique de diffusion globale entre les deux commutateurs qui appartiennent à ce slice, sans se soucier de violer la politique de sécurité globale.

```
(inport("X", [1, 2, 3]) | then | forward(4) +
(inport("X", [4]) | then | forward(1, 2, 3) +
(inport("Y", [1, 2, 3]) | then | forward(4) +
(inport("Y", [4]) | then | forward(1, 2, 3)
```

2.2.8 Merlin

Merlin [Soulé et al., 2014] est un langage pour la gestion de réseaux SDN dont la particularité est de permettre, non seulement la configuration des commutateurs SDN, mais aussi les possibles *middleboxes* (boites noires) et hôtes qui y résident. En effet, souvent un réseau SDN contient plusieurs équipements hétérogènes, en plus des commutateurs SDN, on peut citer l'exemple d'un centre de calcul qui peut éventuellement avoir recours à des machines hôtes d'extrémité pour effectuer des opérations de filtrage ou des *middleboxes* pour assurer des opérations telles que du cache ou de l'inspection en profondeur (DPI, *Deep Packet Inspection*). Ainsi, en utilisant le langage Merlin, l'administrateur dispose d'un langage

qui offre une gestion unifiée de tous ces équipements. Pour ce faire, Merlin partitionne la politique globale en plusieurs sous-politiques qui seront par la suite installées sur des équipements différents. Par ailleurs, il est à noter que la gestion de ces sous-politiques peut être déléguée à des *tenants* (tiers) différents, ces derniers ont par la suite la possibilité de les modifier afin qu'ils correspondent mieux à leurs besoins.

Merlin utilise deux principaux paradigmes de programmation : la logique des prédicats et les expressions régulières. En effet, un programme Merlin peut être vu comme une collection de déclaration (*statements*) qui spécifie la manière avec laquelle un sous-ensemble des flux d'un réseau est géré.

```
(ipSrc = 192.168.1.1 and ipDst = 192.168.1.2 and ipProto = 0x06) →
(.* nat .* dpi .* ) at max (100Mb/s)
```

L'extrait ci-dessus illustre bien les fonctionnalités principales de Merlin. Ce programme stipule que tout le trafic TCP entre les deux hôtes (192.168.1.1 et 192.168.1.2) doit d'abord passer par un nœud réseau qui fait de la traduction d'adresse réseau (`nat`), puis par un nœud qui assure une fonction d'inspection de paquets en profondeur (`dpi`), tout en assurant une bande passante maximale de 100Mb/s.

Ainsi, comme on peut le voir sur l'exemple précédent, chaque déclaration de Merlin est composée de deux parties. La première partie (qui est à gauche de la flèche) est un prédicat qui décrit un ensemble de paquets, alors que la partie à droite est une expression régulière qui spécifie le chemin à emprunter pour ce flux de paquets, d'éventuelles transformations à appliquer et aussi des possibles contraintes sur la bande passante.

2.2.8.1 Les prédicats

Merlin s'appuie sur un solide langage à prédicats afin de classifier les différents flux de paquets qui circulent au sein d'un réseau. Ainsi, un prédicat atomique est représenté sous la forme `f=v` et dénote tous les paquets avec un champ d'en-tête `f` a une valeur égale à `v`. Dans l'exemple précédent nous avons utilisé les prédicats `ipSrc` et `ipDst`. Merlin fournit des prédicats atomiques pour les champs d'en-tête de la plupart des protocoles communément utilisés, dont notamment : IP, TCP, UDP et même la charge utile (payload) d'un paquet. Par ailleurs, les prédicats peuvent aussi être construits en utilisant les expressions booléennes standards telles que la conjonction (and), disjonction (or) et la négation (!).

2.2.8.2 Les expressions régulières

En utilisant les expressions régulières, Merlin permet aux administrateurs de spécifier des chemins de données pour différentes classes de trafic réseau, qui sont définies en utilisant les prédicats. Néanmoins, au lieu de simplement s'appliquer sur des chaînes de caractères, les expressions régulières de Merlin s'appliquent sur une séquence d'emplacements réseau ainsi que sur les noms de fonctions de transformation qui modifient l'en-tête et le contenu des paquets circulant dans un réseau. Le compilateur de Merlin, quant à lui, est libre de choisir le chemin sous-jacent le plus approprié, sous condition que ce dernier respecte les contraintes imposées par la politique spécifiée. Par ailleurs, il est à noter qu'en plus de la contrainte `max` qui permet de limiter la bande passante maximale pour une classe de paquets,

Merlin fournit aussi la contrainte `min` qui permet d'assurer à une classe de paquet une bande passante minimale.

2.2.8.3 Compilation

Le compilateur de Merlin effectue deux principales tâches : la première est de transformer les politiques en un ensemble de problèmes de satisfaction de contraintes qui peuvent être par la suite résolues en ayant recours à la programmation linéaire, pour ainsi déterminer les chemins de données, le placement des fonctions de transformation et l'allocation de la bande passante. Pour ce faire, Merlin a besoin d'un fichier auxiliaire qui fournit des informations sur les noms de fonctions de transformation et leurs possibles déploiement sur un ou plusieurs emplacements dans le réseau. Les seules deux exigences qu'impose Merlin sur les fonctions de transformation sont : 1) la fonction doit prendre en paramètre d'entrée un seul paquet et générer en sortie zéro ou plusieurs paquets. 2) les fonctions de transformation peuvent uniquement accéder à leur état local, de cette manière le compilateur peut placer ces fonctions n'importe où dans le réseau sans se soucier du fait qu'elles peuvent éventuellement modifier l'état global de la politique.

La deuxième tâche consiste à générer des règles de bas niveau afin de configurer les équipements sous-jacent (commutateurs OpenFlow, hôtes et *middleboxes*). Plus concrètement, le compilateur de Merlin génère des messages OpenFlow afin de configurer les règles de commutation des équipement OpenFlow. Concernant les *middleboxes*, merlin se base sur le routeur programmable Click [Kohler et al., 2000] afin de générer des modules de contrôle. En dernier, pour configurer les hôtes, Merlin utilise des services linux standard tels que `iptables` ou `tc` pour configurer des fonctions de filtrages ou de limitation de bande passante.

2.2.9 FatTire

FatTire [Reitblatt et al., 2013] est un langage qui permet aux administrateurs de spécifier leurs politiques de commutation sous la forme d'un ensemble de chemins avec des contraintes de tolérance aux fautes explicitement indiquées. De façon similaire à Merlin, FatTire propose l'utilisation des expressions régulières afin de décrire les chemins de transmission ainsi que les possibles contraintes qui y sont imposées.

Traditionnellement, la reprise d'une erreur qui se produit dans l'infrastructure physique est réalisée au niveau du contrôleur SDN. En effet, les administrateurs doivent écrire en amont des algorithmes complexes qui permettent de générer des nouvelles règles dés que le contrôleur reçoit des événements qui indiquent qu'un commutateur ou un lien est hors service. Les auteurs de FatTire jugent que cette démarche est à la fois complexe et non optimale. En effet, lors de la réception d'un événement qui indique une erreur au niveau de l'infrastructure physique, le contrôleur enclenche une procédure de reprise sur erreur pour installer des nouvelles règles qui permettent d'emprunter le chemin de secours. Malheureusement, cette approche nécessite une quantité considérable de temps, de plus pendant cette phase de mise à jour des politiques de transmission, les paquets circulant encore sur le réseau peuvent être relayés sur différents équipements suivant une table de

commutation partielle et inconsistante, violant ainsi la politique globale de commutation et de sécurité.

Pour pallier cette problématique, FatTire repose sur l'utilisation d'une nouvelle version du standard OpenFlow (version 1.3). En effet, les versions les plus récentes d'OpenFlow proposent l'utilisation de règles conditionnelles dont les actions à exécuter dépendent de l'état local du commutateur. Ainsi, un nouveau type de table appelée *group table* a été introduit dans les commutateurs OpenFlow. Les entrées de cette nouvelle table contiennent des règles qui incluent une liste de seaux d'actions (*actions buckets*) à exécuter suivant un certain nombre de paramètres. Ces seaux d'actions permettent de définir plusieurs types de comportement pour une règle. En effet, à un instant donnée, un seul seau d'actions est actif pour une règle et si éventuellement une erreur se produit (un port est hors service), la règle de commutation change de seau, s'adaptant ainsi sans devoir attendre une mise à jour de la part du contrôleur.

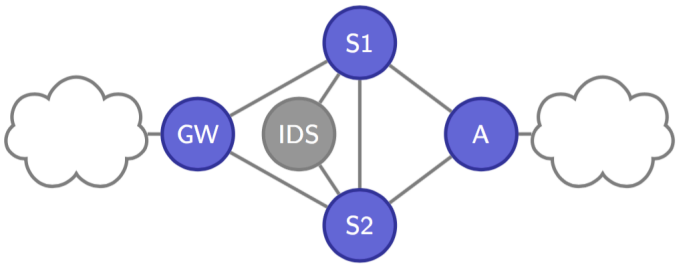

FIGURE 14. Exemple de topologie FatTire [Reitblatt et al., 2013]

Pour mieux illustrer l'utilisation de FatTire, les auteurs proposent le cas d'utilisation suivant. L'objectif est d'installer, sur la topologie qui est schématisée à la figure 14, la configuration qui respecte les contraintes suivantes :

- Le trafic SSH qui arrive au niveau de la passerelle (GW) doit être transmis au commutateur A.
- Le trafic SSH entrant doit passer par un système de détection d'intrusion (IDS).
- Le fonctionnement du réseau doit être assuré même dans le cas de la défaillance d'un lien.

Suivant ces contraintes de haut-niveau, un premier chemin a été sélectionné : [GW, S1, IDS, S2, A]. Cependant, pour assurer la contrainte de tolérance à la défaillance d'un lien, les chemins de secours suivant ont été proposés :

- le chemin [GW,S2,IDS,S2,A] si le lien (GW,S1) est défaillant.
- le chemin [GW,S1,S2,IDS,S2,A] si le lien (S1,IDS) est défaillant.
- le chemin [GW,S1,IDS,S1,A] si le lien (IDS,S2) est défaillant.
- le chemin [GW,S1,IDS,S2,S1,A] si le lien (S2,A) est défaillant.

Transcrire ces chemins en utilisant OpenFlow est une tâche fastidieuse et sujette à erreurs. La figure 15 affiche les tables de commutation et les tables `group` nécessaires pour implémenter les chemins décris précédemment.

GW Ruletable and Grouptable

Match	Instructions
tpDst = 22	Group 1

Group	Type	Actions
1	FF	⟨Fwd S1⟩, ⟨Fwd S2⟩

S1 Ruletable and Grouptable

Match	Instructions
inPort = GW, tpDst = 22	Group 1
inPort = IDS, tpDst = 22	Group 2
inPort = S2, tpDst = 22	Group 2

Group	Type	Actions
1	FF	⟨Fwd IDS⟩, ⟨Fwd S2⟩
2	FF	⟨Fwd A⟩

S2 Ruletable and Grouptable

Match	Instructions
inPort = IDS, tpDst = 22	Group 1
inPort = S1, tpDst = 22	Group 2
inPort = GW, tpDst = 22	Group 2

Group	Type	Actions
1	FF	⟨Fwd A⟩, ⟨Fwd S1⟩
2	FF	⟨Fwd IDS⟩

FIGURE 15. Tables des équipements dans l'exemple FatTire [Reitblatt et al., 2013]

Ainsi, sur chacun des équipements intermédiaires (GW, S1 et S2) on a deux tables qui sont installées. La première est la table de commutation, elle contient les règles qui permettent de capturer un flux de paquets, puis d'appliquer un ensemble d'actions. Ici, la partie actions contient juste un pointeur vers un groupe d'actions qui sont enregistrées dans la `group table`.

Par exemple, sur le commutateur GW, on retrouve une règle dans la table de commutation qui capture tous les flux SSH entrant (tpDst=22) et y applique le premier groupe d'actions. Ce dernier inclut deux seaux d'action, le premier commute les paquets vers le commutateur S1, mais si le lien vers le commutateur S1 est défaillant, alors le deuxième seau est appliqué et ce dernier inclut une action qui commute les paquets vers le commutateur S2.

Ainsi, en regroupant toutes ces tables on retrouve les quatre chemins alternatifs à mettre en œuvre en cas de défaillance d'un lien particulier. Néanmoins, il est clair que construire ces tables d'une manière manuelle en utilisant l'interface de bas-niveau d'un contrôleur SDN est une tâche complexe. Le langage FatTire permet la description de ces chemins principaux et alternatifs d'une manière très intuitive en utilisant les expressions régulières. L'exemple ci-dessous représente le programme nécessaire pour implémenter la politique décrite précédemment.

```
(tpDst = 22 ⇒ [★.IDS.★]) ⍎
(tpDst = 22 ⇒ [★] with 1 ) ⍎
(any  ⇒ [GW.★.A])
```

La première ligne du programme spécifie une règle qui indique que tout flux SSH entrant doit passer par un système de détection d'intrusions. La deuxième ligne utilise la primitive `with` pour indiquer que la politique d'acheminement doit être résiliante à la défaillance d'un lien. La troisième ligne représente la politique de routage et indique que tous les flux provenant de la passerelle (`GW`) doivent être acheminés vers le commutateur S1. Le programme complet consiste en l'intersection des ces trois sous-politiques pour donner une politique globale qui stipule que le trafic SSH entrant doit traverser un `IDS`, être résilient à une défaillance d'un lien et doit être acheminé de la passerelle jusqu'au commutateur A.

La figure 16 représente un résumé des primitives les plus importantes de FatTire, beaucoup de ces primitives ont été déjà proposées dans le cadre de travaux précédent tels que NetCore, mais le langage FatTire propose une méthode intuitive et performante pour gérer les contraintes de tolérances aux fautes.

```
Switch ID sw ∈ N
En-tête       sh::= dlSrc  --adresse MAC source
                  | dlDst  --adresse MAC destination
                  | ....
Prédicats     pr ::= any  --joker
                  | h =v  --filtre sur un en-tête
                  | not pr  --négation d'un prédicat
                  | pr1 and pr2 --intersection de deux prédicats
Politiques    pol::= pr ⇒ P with n
                  | pol1 ⊎ pol2
                  | pol1 ⊕ pol2
```

FIGURE 16. Primitives principales du langage FatTire

2.2.10 Maple

Maple [Voellmy et al., 2013] est un langage de programmation pour les réseaux SDN dont la principale caractéristique est de permettre aux administrateurs de spécifier leurs politiques de contrôle sous la forme d'un algorithme central et non plus sous la forme d'une suite de règles, avec des prédicats à appliquer sur les paquets. En effet, les auteurs de Maple affirment que les langages de programmation actuels (tels que Frenetic ou Pyretic) forcent les administrateurs à penser leurs politiques dans le cadre du langage utilisé, au lieu de les penser dans un cadre algorithmique. Par exemple, lors de la spécification d'une politique de contrôle, un administrateur doit explicitement indiquer les paquets qui doivent être traités au niveau du contrôleur et ceux qui doivent être traités au niveau des commutateurs. D'un autre côté, Maple permet aux administrateurs d'écrire un algorithme global de commutation, qui prendra la forme d'une fonction `f`, et qui sera par la suite appliquée, conceptuellement, sur tous les paquets qui entrent dans le réseau.

Le programme ci-dessous présente un exemple d'algorithme qui résume bien les fonctionnalités de base du langage Maple. La politique qui est décrite dans cet exemple inclut deux parties : la première est une politique de sécurité qui stipule que tous les flux SSH doivent être transmis sur un chemin sécurisé, pour le reste des flux le chemin par défaut est le plus court chemin. La deuxième politique, quant à elle, permet de garder une trace sur la

localisation de chaque hôte du réseau (port d'entrée sur un commutateur de bordure).

```
def f(pkt):
 srcSw = pkt.switch(); srcInp = pkt.inport();
 if locTable[pk.eth_src()] != (srcSw, srcInp):
  invalidateHost(pkt.eth_src())
  locTable[pkt.eth_src()] = (srcSw, srcInp)
 dstSw = lookupSwitch(pky.eth_dst())
 if pkt.tcp_dst_port() == 22:
  outcome.path = securePath(srcSw, dstSw)
 else:
  outcome.path = shortestPath(srcSw, dstSw)
 return outcome
```

Ainsi, la principale abstraction qui est offerte par Maple est de donner l'illusion que la fonctions `f` sera appliquée par le contrôleur SDN à tous les paquets qui entrent dans le réseau. Cela pourrait être réalisable en remontant tous les paquets vers le contrôleur, appliquer la fonction `f`, puis les renvoyer vers l'infrastructure physique tout en faisant en sorte qu'ils soient transmis suivant la décision prise (plus court chemin ou chemin sécurisé). Néanmoins, dans la pratique une telle approche n'est pas envisageable, considérant le goulot d'étranglement qui va se créer au niveau du contrôleur. En effet, remonter tous les paquets vers le contrôleur va créer d'une part un problème de bande passante, étant donné que tous les commutateurs vont envoyer tous leurs paquets vers un seul point central, et d'autre part, un problème de puissance de calcul se posera aussi au niveau du contrôleur, étant donné qu'il devra exécuter plusieurs fonctions, potentiellement complexes et coûteuses en temps, sur tous les paquets.

Pour répondre à ce problème de performance, Maple utilise deux principaux module : un optimiseur et un ordonnanceur multi-cœurs.

L'ordonnanceur multi-cœurs permet à Maple de supporter une exécution sur des machines qui peuvent embarquer jusqu'à 40 cœurs, offrant ainsi une première réponse à la problématique d'exécution des fonctions sur plusieurs paquets.

Le deuxième module l'optimiseur offre à Maple une abstraction d'un algorithme central s'exécutant sur tous les paquets qui entrent dans le réseau. Ce dernier réalise trois principales tâches : 1) maintient un cache d'exécution des fonctions, 2) construit les tables de commutation OpenFlow et 3) maintient les traces d'exécution et les tables de commutation à jour.

Maintenir un cache d'exécution : La première fonctionnalité de l'optimiseur de Maple est de détecter les parties réutilisables des fonctions potentiellement complexes. Pour ce faire, Maple enregistre l'ensemble des dépendances qui participent au processus de prise de décision d'une fonction. Ces données sont enregistrées par la suite dans des structures arborescentes appelées *tree trace*. A titre d'exemple, la figure 17 représente la structure où ont été enregistrées trois traces d'exécution d'un programme qui fait passer tous les flux, à part les flux SSH entrants, par le chemin le plus court (politique 2 de l'exemple précédant).

Dans la première trace de cet exemple, le programme commence par tester si le port TCP destination est égal à 22 (SSH). Si le résultat retourné est négatif (branche de droite), il lit ensuite l'adresse mac destination (le résultat est égal à 4) et l'adresse source (le résultat est

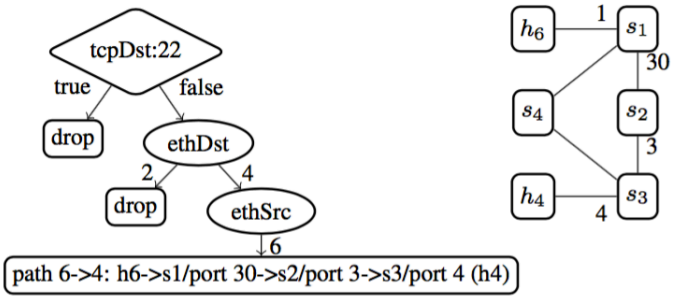

FIGURE 17. Exemple d'un arbre de traces [Voellmy et al., 2013]

égal à 6) et à la fin il retourne un résultat correspond à la transmission du paquet suivant le plus court chemin. Les deux autres traces restantes correspondent à un rejet de paquet soit pour une adresse MAC destination inconnue (égale à 2), soit pour un flux SSH. Ainsi, l'utilité de cette structure est de pouvoir réutiliser le résultat d'exécution sur des paquets semblables à ceux déjà traités.

Construire les tables de commutation : La deuxième fonctionnalité de l'optimiseur de Maple est d'essayer de pousser le plus possible de traitements à effectuer sur les paquets vers les commutateurs, afin d'alléger la charge sur le contrôleur. Pour ce faire, l'optimiseur maintient aussi des structures de type *tree trace* pour chaque commutateur présent dans l'infrastructure physique. Ces structures sont très semblables à celles utilisées pour les fonctions, à l'exception des feuilles qui dans cette structure représentent uniquement les actions de commutation et de modification qui sont réalisées par le commutateur en question. A titre d'exemple, le commutateur `S1` a la même structure que celle présentée dans la figure 17, si ce n'est que la feuille de la branche qui est à l'éxtreme droite contient une seule action qui est de faire sortir le paquet par le port 30 afin d'atteindre le commutateur `S2` (au lieu de tout le chemin présent dans l'arbre de la fonction). Ainsi, à partir de ces structure, Maple est capable de générer des règles OpenFlow afin de construire les tables de commutations de chaque commutateur.

Maintenir les tables de commutation : La dernière fonctionnalité implémentée par l'optimiseur de Maple est de s'assurer que ces différentes structures ainsi que les tables des commutateurs restent à jour et consistantes. En effet, les décisions prises par les fonctions ne dépendent pas uniquement des en-têtes des paquets, mais dépendent aussi d'autres variables telles que la topologie du réseau (qui peut éventuellement évoluer en cours d'exécution) ou la configuration des commutateurs. Ainsi, Maple fournit une API qui permet de spécifier des clauses de sélection afin d'invalider ces structures, sachant que ces clauses peuvent être utilisées soit pas l'environnement d'exécution de Maple ou des programmes qui ont été implémentés par les administrateurs. Il est à noter que cette partie n'a pas été détaillée par les auteurs [Voellmy et al., 2013].

2.2.11 NetKat

NetKat [Anderson et al., 2014] est un langage de programmation SDN dont la principale particularité est d'être bâti sur une base mathématique solide appelée KAT (*Kleene Algebra with Tests*), à la différence de nombreux langages existants dont la conception a été plus poussée par des besoins applicatifs et les capacités des équipements réseaux existants. Par ce choix de conception, les auteurs de NetKat ont souhaité proposer un modèle formel pour les langages de programmation réseau qui permettra de : 1) identifier les primitives et les constructions essentielles pour un langage de programmation réseau, 2) fournir des lignes directrices pour l'ajout et le support de nouvelles fonctionnalités et 3) unifier le raisonnement à propos des commutateurs, la topologie et le comportement de bout-en-bout d'un programme de contrôle.

NetKat représente, d'une manière abstraite, la structure globale d'un réseau sous la forme d'un automate qui transporte un paquet d'un nœud à un autre en utilisant les liens présents dans la topologie. D'un point de vue langage, cette abstraction est décrite en utilisant des expressions régulières où un chemin est représenté par une concaténation de transitions (p · q ...), alors qu'un ensemble de chemins est représenté comme une union de chemins.

Par ailleurs, NetKat se base sur l'utilisation de l'algèbre de Boole afin de décrire les fonctionnalités d'un commutateur, qui fondamentalement peuvent être résumées à l'utilisation d'un prédicat afin de capturer un paquet, puis d'appliquer des actions sur ce paquet afin de le modifier ou le relayer vers un port de sortie.

Ainsi, le modèle formel que préconise NetKat consiste en l'utilisation de l'algèbre de Kleene afin de représenter et analyser la structure globale d'un réseau et l'algèbre de Boole afin de décrire les fonctionnalités des commutateurs. Pour ce faire, NetKat se base sur des travaux existants qui ont permis d'unir ces deux bases mathématiques en une seule nommée KAT (*Kleene algebra with tests*).

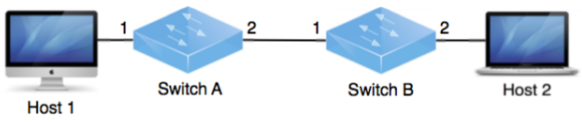

FIGURE 18. Exemple topologie NetKat [Anderson et al., 2014]

Une politique NetKat peut être considérée comme une fonction qui prend en entrée un paquet et retourne en sortie un ensemble de paquets (zéro, un ou plusieurs). Un paquet est, quant à lui, représenté comme un enregistrement avec des champs pour désigner les en-têtes standard tels que l'adresse source (`src`), l'adresse destination (`dst`) ou le protocole utilisé (`type`). Un paquet inclut aussi les deux champs commutateur (`sw`) et port (`pt`) qui permettent d'identifier la localisation courante du paquet au sein de la topologie physique.

Pour les actions atomiques, NetKat en distingue deux : les prédicat ou filtre (`f=n`) et l'action de modification (`f` ← `n`). Un filtre (`f=n`) prend toujours en entrée un paquet `pk` et retourne le singleton {`pk`} si le champs `f` de `pk` est égale à `n`, sinon un ensemble vide est retourné { }, ce qui

correspond à un rejet du paquet. L'action de modification `f ← n` prend en entrée un paquet `pk` et retourne en sortie le singleton {`pk'`} ou `pk'` correspond au paquet `pk` avec le champs `f` mis à jour avec la valeur `n`. Ainsi, en changeant par exemple les champs de localisation d'un paquet, l'action de modification permet de le commuter vers la nouvelle destination (nouvelles valeurs des champs `sw` et `pt`).

Nous allons considérer le cas d'utilisation suivant (extrait de [Anderson et al., 2014]) afin de bien présenter les fonctionnalités de base du langage NetKat. La figure 18 décrit une topologie constituée de deux commutateurs `A` et `B` et de deux hôtes 1 et 2. L'objectif global est d'implémenter une politique d'acheminement pour tous les flux entre les deux hôtes, excepté les flux SSH. L'extrait ci-dessous présente une politique NetKat qui assure la commutation des paquets entre les deux hôtes.

```
P ≜ (dst = H1 · pt ⟵ 1) + (dst = H2 · pt ⟵ 2)
```

La politique qui est décrite dans le programme ci-dessus consiste réellement en la composition parallèle de deux sous-politiques. La première met à jour le champ de localisation `pt` à 1 pour tous les paquets qui ont comme destination l'hôte 1, quant à la deuxième elle met à jour le champ port à 2 pour tous les paquets à destination de l'hôte 2.

La deuxième étape maintenant est d'intégrer la politique de contrôle d'accès rejetant tous les flux SSH. Cela est réalisé par la composition séquentielle d'un filtre qui capture tous les paquets sauf les paquets SSH avec la politique énoncée dans l'extrait précédent.

```
P_AC ≜ ¬ (type = SSH) · P
```

La force de NetKat est que, en partant d'une politique comme celle affichée ci-dessus, il est capable de répondre à des questions telles que :

- Est-ce que les paquets non SSH sont relayés ?
- Est-ce que les paquets SSH sont effectivement bloqués ?
- Est-ce que les politiques PA et PAC sont équivalente ?

Pour pouvoir répondre à ce type de questions sur le comportement de bout-en-bout d'un programme de contrôle, l'administrateur doit aussi décrire la topologie sur laquelle vont s'appliquer ces politiques. Décrire une topologie avec NetKat consiste concrètement à modéliser chaque lien de cette topologie sous la forme d'une politique, la topologie correspondra alors à l'union de toutes ces sous-politiques. Pour modéliser un lien, un prédicat est utilisé pour représenter l'une des extrémités d'un lien, ce dernier est composé par la suite avec une action de modification des champs `sw` et `pt` pour qu'ils correspondent à l'autre extrémité du lien, capturant ainsi l'effet de transfert d'un paquet sur un lien. L'extrait ci-dessous présente la modélisation de la topologie de la figure 18.

```
t = (sw = A · pt = 2 · sw ← B · pt ← 1) +
    (sw = B · pt = 1 · sw ← A · pt ← 2) +
    (sw = A · pt = 1) +
    (sw = B · pt = 2)
```

L'acheminement d'un paquet via les commutateurs et la topologie peut être exprimé par la combinaison des politiques précédentes : $P_{AC} \cdot t \cdot P_{AC}$, ou même d'une manière plus générale

en utilisant l'étoile de Kleene (P_{AC}·t)*, car un paquet peut faire plusieurs cycles dans une topologie, surtout si cette dernière contient des boucles.

Étant donné que NetKat satisfait tous les axiomes de KAT, les réponses aux questions précédentes se résumeront alors à la résolution d'un problème d'équivalence de politiques (p≡q signifie que les deux politiques retournent le même ensemble de paquets). Ainsi, pour prouver que le réseau ne commute aucun paquet SSH n'est transmis du port 1 du commutateur `A` vers le port 2 du commutateur `B` , il faut juste prouver l'équivalence suivante, sachant que 0 est une politique qui filtre tous les paquets :

$$\left(\begin{array}{c} type = SSH \cdot sw = A \cdot pt = 1 \cdot \\ (P_{AC} \cdot t) * \cdot \\ sw = B \cdot pt = 2 \end{array} \right) \equiv 0$$

2.3 Discussion

Après cette description détaillée des langages de programmation réseau existants, nous allons, dans cette section, discuter de leurs motivations et principaux apports. Par ailleurs, il est important de noter que cette discussion n'a pas pour objectif de déterminer quel est le meilleur langage d'entre eux, mais plutôt de proposer une grille de lecture qui permettra de mieux situer les problématiques ainsi que les principaux apports de chacun.

En effet, de par la nature et le rôle que jouent les *Northbound API*, nous pensons qu'il n'est pas pertinent d'essayer de statuer que telle approche ou telle fonctionnalité est meilleure ou plus intéressante que les autres, surtout considérant leurs différents contextes d'utilisation et les constantes évolution de leurs exigences [Robert Sherwood, 2013, Isabelle GUIS, 2012, Brent Salisbury, 2012, Greg Ferro, 2012, Ivan Pepelnjak, 2012, Sally Johnson, 2012, Rivka Gewirtz Little, 2013]. Pour étayer plus notre propos, nous pourrions faire l'analogie avec les systèmes d'exploitation machine ou mobile tels Windows ou MAC OS, qui ne proposent aucune API standard et ceux malgré leur longue existence. De plus, leur écosystème applicatif se porte très bien et est en constante évolution. Ainsi, la même chose s'applique aux *Northbound API* dans le contexte SDN où ces interfaces représentent la brique qui permettra aux opérateurs et aux administrateurs de créer le plus de valeur ajoutée. En effet, c'est en les utilisant qu'un administrateur pourra faire évoluer, rapidement et efficacement, le comportement de son réseau afin d'atteindre au mieux ses objectifs métiers.

Ainsi, nous avons vu que plusieurs langages de programmation ont été proposés dans un cadre SDN, le tableau 1 résume brièvement ces langages ainsi que leurs principaux apports. Un point commun entre tous ces langages est que tous proposent des abstractions pour le standard OpenFlow. Pour ce faire, chaque langage utilise un paradigme de programmation différent. On peut aussi noter que le paradigme déclaratif, sous ses diverses formes (logique, fonctionnel, etc.), est le plus prédominant, cela est notamment dû à la nature déclarative des règles OpenFlow qui suivent une logique de type `filtre-action`.

Langage	Paradigme de programmation	Points clés
FML	logique, réactif	Règles avec une logique si-alors, deux mécanismes de résolution de conflits.
Nettle	fonctionnel, réactif	Basé sur le paradigme FRP, un programme est une fonction signal.
Frenetic	fonctionnel, réactif, requête	Deux sous-langages : surveillance et contrôle, règles avec une logique filtre-actions
Procera	fonctionnel, réactif	Concepts de fenêtrage et d'agrégation, création fonctionnelle d'évènements.
NetCore	fonctionnel	Fait suite aux travaux Frenetic, nouveaux algorithmes de compilation, primitives pour consulter l'état du contrôleur.
Pyretic	impératif	Opérateurs de composition séquentielle et parallèle, virtualisation de réseau.
Splendid Isolation	descriptif	Isolation de trafic, virtualisation de réseau.
Merlin	expression régulière	Gestion unifiée de tous les équipements (commutateur, hôte, boite noir) d'un réseau.
FatTire	expression régulière	Utilise les expressions régulières pour exprimer des chemin ainsi que les contraintes de tolérance aux fautes.
Mapple	imperatif	Algorithme de compilation, exécution parallèle, abstraction d'un algorithme centralisé.
NetKat	fonctionnel	Respecte la base mathématique KAT, permet la vérification des politiques.

TABLE 1. Synthèse des langages de programmation réseau

2.3.1 Une API OpenFlow de plus haut niveau

Les touts premiers langages comme FML, Nettle et Procera sont des langages fonctionnels et réactifs. Ainsi, les politiques et les programmes de contrôle qui sont écrits avec ces langages prendront la forme d'une fonction, ou une composition de fonctions, qui va recevoir un flux d'évènements OpenFlow (ou évènements extérieurs telle l'authentification d'un hôte), éventuellement appliquer des transformations sur ces évènements, puis retourner comme résultat un nouveau flux d'évènements qui correspondra souvent à des actions de type `allow` ou `deny` qui serviront ensuite à configurer les tables de commutation des équipements réseaux.

Ainsi, cette démarche permet de masquer une grande partie de la complexité des interactions entre le programme de contrôle et l'infrastructure physique, notamment le fait de construire les différents types de messages OpenFlow ou de devoir considérer à chaque fois le type de l'API et du contrôleur sous-jacent.

2.3.2 Composition et modularité

D'autres langages SDN comme Frenetic, NetCore et Pyretic ont été conçus dans l'objectif d'exprimer plus efficacement les règles de commutation des paquets, notamment en se basant sur des prédicats et des primitives qui permettent de spécifier des filtres incomplets et d'envoyer des requêtes afin de récupérer des statistiques sur l'état du contrôleur. Les règles de contrôle de ces langages suivent principalement la logique `filtre→actions`, où `filtre` capture un ensemble de paquets et `actions` représente la liste d'actions à appliquer sur ces

paquets.

Par ailleurs, ces langages offrent plusieurs mécanismes pour résoudre les problèmes d'intersection entre les règles de différents modules de contrôle, principalement à travers l'introduction des opérateurs de composition séquentielle et parallèle (opérateurs intégrés par exemple dans le langage Pyretic et que nous réutiliserons dans notre contribution). Ces opérateurs permettent, d'une part, de composer plus facilement les diverses primitives d'un langage afin d'obtenir des politiques plus élaborées, et d'autre part, permettent aussi de composer des modules de contrôle existants, améliorant ainsi la réutilisabilité des programmes.

2.3.3 Fonctionnalités avancées

D'autres fonctionnalités plus avancées sont fournies par des langages tels que FatTire qui permet aux administrateurs d'exprimer leurs chemins réseau tout en positionnant des exigences de tolérance aux fautes. Par la suite, c'est la responsabilité de l'environnement d'exécution de FatTire d'installer des chemins de secours pour répondre à ces exigences en se basant notamment sur de nouveaux types de tables de commutation introduites dans les versions récentes du standard OpenFlow.

Le langage Merlin a été, quant à lui, une des premières tentatives pour avoir un environnement de gestion unifié (dans un contexte SDN) pour tous les équipements d'un réseau tels que, les commutateurs OpenFlow, les machines hôtes et les différentes *middleboxes* qu'on peut retrouver. Pour ce faire, Merlin génère un code spécifique pour chaque type d'équipement, tout en proposant aussi une possible délégation du contrôle d'une partie de ces équipements à un tier (*tenant*) extérieur.

Le langage Mapple introduit une abstraction qui permet aux administrateurs de voir leur programme de contrôle comme un seul algorithme central qui va être appliqué sur tous les paquets qui entreront dans leur réseau. Ainsi, un administrateur n'a plus besoin de penser quel paquet doit remonter au contrôleur et quels sont ceux qui doivent être traités au niveau des commutateurs, tout cela est fait automatiquement par le compilateur de Maple qui optimise au mieux cette répartition afin de préserver les performances du réseau.

En dernier, le langage NetKat, dont les primitives respectent la base mathématique *KAT*, permet aux administrateurs de vérifier leurs programmes de contrôle. En effet, NetKat est capable de répondre formellement à des questions telles que : les flux SSH sont-ils autorisés entre l'hôte `A` et un serveur web `S`.

2.3.4 Virtualisation de réseau

Des langages comme Pyretic et Splendid Isolation, fournissent un support pour la virtualisation de réseau. Ainsi, en utilisant ces langages, un administrateur peut définir ses propres topologies virtuelles, suivant ses propres objectifs de haut-niveau, puis d'appliquer des politiques sur ces topologies abstraites. L'avantage de cette démarche est qu'elle facilite significativement la configuration d'un réseau, étant donné que l'administrateur ne va considérer que les informations qui sont les plus pertinentes pour sa politique de configuration globale. Encore plus important, cette démarche permet d'augmenter

significativement la réutilisabilité des programmes vu que les politiques sont spécifiées sur des topologies virtuelles et peuvent être par la suite utilisées sur des infrastructure physiques différentes.

De plus, Splendid Isolation permet de définir des *slices* (c.-à.d., des tranches) de réseau avec un mapping un-à-un entre équipements virtuels et physiques, et où chaque slice pourra avoir son propre programme de contrôle, assurant ainsi une isolation stricte entre ces différents modules de contrôle. Pyretic, quant à lui, fournit un modèle de paquets qui permet de définir des topologies virtuelles avec des possibilités de mapping très flexibles. L'administrateur pourra, par la suite, appliquer ses politiques de contrôle sur cette topologie virtuelle et cela sera la responsabilité de l'hyperviseur d'installer une configuration physique qui est sémantiquement équivalente à la politique virtuelle.

2.4 Bilan

L'intérêt qu'a suscité le paradigme SDN a fait que, en l'espace de quelques années, beaucoup de travaux ont été menés au sein de la communauté de gestion de réseaux afin de proposer des *Northbound API* métiers et modernes, principalement sous la forme de langage de programmation haut niveau. De par leurs contributions, chacun de ces travaux a permis de faire évoluer les exigences et le type de fonctionnalités que nous attendons d'une *Northbound API*. Néanmoins, en dépit de ces nombreux travaux, aucun consensus n'a pu être dégagé quant à la nature et au rôle précis de la *Northbound API*, notamment son niveau d'abstraction et le type de fonctionnalités qu'elle doit proposer. Ainsi, l'objectif premier de ce chapitre a été de présenter une synthèse de ces travaux et une analyse de leurs principales contributions à la problématique de conception et d'implémentation d'une *Northbound API*.

A la fin de cette étude, nous avons constaté que l'approche de virtualisation de réseau devient de plus en plus incontournable pour une *Northbound API*, étant donné les gains considérables que cela peut apporter, en termes d'expressivité, de modularité et de flexibilité, pour les langages de contrôle. En effet, disposer d'une vision abstraite de l'infrastructure physique, avec seulement les détails les plus pertinents pour les politiques de contrôle de haut niveau, permettrait de simplifier grandement leur expression.

Parmi les langages présentés dans ce chapitre, les plus récents d'entre eux (notamment Pyretic, Merlin et Splendid Isolation) ont eu recours à la virtualisation de réseau afin d'assurer diverses propriétés telles que la montée en abstraction ou le partage des ressources. Pour ce faire, ces derniers se sont basés sur des modèles d'abstraction afin de cacher la nature complexe et dynamique de l'infrastructure physique. Toutefois, la plupart de ces travaux manipulent des abstractions dans un objectif bien précis : par exemple, Splendid Isolation dans le but d'isoler des programmes de contrôle ou encore Merlin pour résoudre des problèmes de satisfaction de contraintes sur des chemins réseau. Il en résulte des interfaces nord qui sont orientées vers ces spécificités. Dans notre cas, nous ciblons des politiques de contrôle réseau métiers de haut niveau, non dédiées à un objectif ou domaine particulier. Ainsi, les modèles d'abstraction utilisés par ces travaux ne nous paraissent pas appropriés dans notre contexte. Dans le chapitre suivant, nous reviendrons plus en détails

sur ces modèles d'abstraction, leurs avantages et inconvénients, puis nous détaillerons notre proposition de modèle d'abstraction sur lequel notre langage de programmation réseau est bâti.

Deuxième partie

Le langage AirNet

Chapitre 3
Le modèle d'abstraction Edge-Fabric

Ce chapitre a pour objectif de présenter le nouveau modèle d'abstraction que nous préconisons pour les langages de programmation SDN supportant la virtualisation de réseau. Dans un premier temps nous présentons les exigences que nous avons identifiées pour un plan de contrôle SDN se présentant sous la forme d'un langage de programmation réseau. Puis, nous introduirons l'approche de virtualisation de réseau. Dans un second temps, nous présenterons les modèles d'abstraction qui ont été utilisés par les langages de programmation existants. Enfin, nous détaillerons notre modèle d'abstraction qui est basé sur l'idée clé de séparation entre les fonctions de cœur et de bordure de réseau.

3.1 Exigences pour un plan de contrôle SDN modulaire et flexible

Tout le paradigme SDN est basé sur l'idée clé de séparer le plan de contrôle du plan de données et de le placer dans un point logiquement centralisé. C'est notamment à travers cette séparation que le paradigme SDN est parvenu à introduire de la modularité et de la flexibilité dans les modules de contrôle, comparé aux infrastructures traditionnelles. Ainsi, pour être en cohérence avec les fondements du paradigme SDN, un plan de contrôle, exposant ses services sous la forme d'un langage de programmation réseau, devrait satisfaire les trois grandes exigences suivantes :

- **Expressivité** : le langage doit être orienté métier, tant par sa syntaxe que par son modèle de programmation. Plus concrètement, le langage doit permettre aux administrateurs de décrire, le plus simplement possible, des services réseaux et leurs interactions (le comportement du réseau), plutôt que de spécifier comment ces services vont être mis en œuvre sur les équipements physiques (leur implémentation concrète).
- **Modularité et réutilisation** : les administrateurs réseaux doivent être en mesure de mettre en œuvre leurs fonctions réseau en tant que modules séparés qui peuvent être, d'une part, facilement composés pour construire des programmes de contrôle élaborés, et d'autre part, réutilisés sur différentes infrastructures physiques.
- **Flexibilité** : le langage doit permettre de spécifier des politiques de contrôle qui répondent à la diversité des fonctionnalités actuelles (par exemple, le routage, le contrôle d'accès ou la surveillance), et dans divers contextes d'utilisation (par exemple, les réseaux de campus, centres de données ou réseaux d'opérateurs). De plus, le langage

ne doit pas imposer des restrictions trop fortes pour être en mesure de répondre, dans la mesure du possible, aux évolutions futures.

3.2 Virtualisation de réseau

La virtualisation de réseau peut aider à réaliser les exigences précédemment présentées. En effet, la virtualisation de réseau est une approche qui propose de découpler les fonctionnalités du réseau de l'infrastructure physique. Ainsi, un environnement réseau supporte la virtualisation de réseau s'il permet la définition et la coexistence de réseaux virtuels, où chacun d'entre eux inclut plusieurs composants et liens virtuels qui partagent les ressources de l'infrastructure physique sous-jacente [Chowdhury and Boutaba, 2009, Chowdhury and Boutaba, 2010].

Actuellement, il existe plusieurs motivations à virtualiser des ressources, nous en citons ci-dessous les principales [Jain and Paul, 2013] :

- Partage : quand une ressource est trop "grande" pour un seul utilisateur, sa virtualisation permet de la partager entre plusieurs utilisateurs, optimisant ainsi son utilisation.
- Agrégation : si une ressource est trop petite, la virtualisation peut permettre de regrouper plusieurs de ces petites ressources en une seule, qui pourra être par la suite allouée à un utilisateur.
- Isolation : les utilisateurs de ressources ne devraient pas être capable de surveiller ou d'interférer dans les ressources virtuelles des autres utilisateurs, même s'ils partagent la même ressource physique.
- Réservation dynamique : les ressources virtuelles sont beaucoup plus simples à réserver, à déplacer et à détruire.
- Facilité de gestion : à travers des abstractions, la virtualisation peut significativement simplifier la gestion d'une infrastructure physique en n'exposant que les informations qui sont essentielles pour l'administrateur.

Parmi ces motivations, faciliter la gestion d'une infrastructure physique en est peut-être la plus importante [Casado et al., 2010], surtout dans ce contexte de *Northbound API* où la virtualisation réseau peut apporter des gains considérables quant à la facilité de spécification, modularité et réutilisabilité des programmes de contrôle réseau. En effet, penser et écrire des modules ou des programmes de contrôle sera une tâche beaucoup plus simple et moins sujette aux erreurs, étant donné que les administrateurs spécifieront leurs politiques sur des visions abstraites (topologies virtuelles) de l'infrastructure physique. Ces topologies virtuelles n'exposeront alors que les informations les plus pertinentes et essentielles pour les politiques de contrôle des administrateurs. Idéalement, ces visions abstraites devraient aussi introduire des frontières logiques entre les différents types de politiques afin d'éviter tout couplage fort entre elles, améliorant ainsi la modularité des programmes de contrôle. Enfin, étant donné que les politiques sont écrites au-dessus de topologies virtuelles, elles seront alors complètement détachées de l'infrastructure physique, leur permettant ainsi, d'une part,

d'être toujours valides en cas de mise à jour ou changement au niveau de l'infrastructure physique sous-jacente, et d'autre part, d'être réutilisées dans d'autres contextes et sur différents réseaux.

Néanmoins, pour exploiter au mieux la virtualisation de réseau et obtenir le maximum de gain en matière de faciliter d'utilisation, de modularité et de réutilisabilité, le **modèle d'abstraction** utilisé pour définir des vues virtuelles doit être adapté à ce contexte de *Northbound API*. En effet, le modèle d'abstraction représente un enjeu majeur dans un approche basée sur la virtualisation de réseaux [Casado et al., 2010], étant donné que ses propriétés intrinsèques (facilité d'utilisation, modularité, réutilisation, flexibilité) se répercuteront, in fine, sur les propriétés de la *Northbound API*.

Dans ce qui suit, nous présentons les différents modèles d'abstraction qui ont été utilisés par les langages de programmation SDN existants. Puis, nous présenterons notre modèle d'abstraction qui est la pièce centrale du langage AirNet que nous proposons.

3.3 Modèles d'abstraction existants

On retrouve dans la littérature deux principaux modèles d'abstraction : le modèle *OBS* (*One Big Switch*) [Keller and Rexford, 2010], qui à notre connaissance, a inspiré des langages (notamment Merlin [Soulé et al., 2014] et Maple [Voellmy et al., 2013]) mais n'a pas été utilisé comme modèle d'abstraction à part entière et le modèle *Overlay Network* (réseau superposé) [Casado et al., 2010] qui a été notamment utilisé par les langages Pyretic [Monsanto et al., 2013] et Splendid Isolation [Gutz et al., 2012]. Dans ce qui suit nous présentons chacun de ces deux modèles.

3.3.1 *One Big Switch*

Le modèle *OBS* ou *Platform as a service* [Keller and Rexford, 2010] est un modèle très puissant qui permet d'abstraire toute la topologie physique en un seul et unique commutateur logique sur lequel tous les hôtes sont connectés, comme cela est illustré à la figure 19 *b*. L'avantage de cette approche est qu'elle permet aux administrateurs de se concentrer pleinement sur la définition de leurs services réseaux complexes tels que l'équilibrage de charge et le contrôle d'accès. En effet, avec ce modèle, toute la complexité du réseau est cachée, et la problématique de transport dans le réseau, à travers les différents nœuds intermédiaires, devient une simple problématique de commutation au sein d'un même composant logique.

Néanmoins, cette approche présente à notre avis deux principaux inconvénients. Le premier inconvénient, d'ordre logiciel, est le fait d'obliger les administrateurs à toujours mettre leurs services ou fonctions réseaux au sein d'un même conteneur logique, ce qui va rendre leurs tests et déboggage plus difficile. En effet, étant donné que ce commutateur logique englobe toute l'infrastructure physique et que tous les modules de contrôle sont exécutés en son sein, si un problème ou un dysfonctionnement est détecté, alors il sera difficile d'identifier les services et/ou les commutateurs physiques impliqués. De plus, cette approche peut diminuer significativement la lisibilité des programmes de contrôle, étant

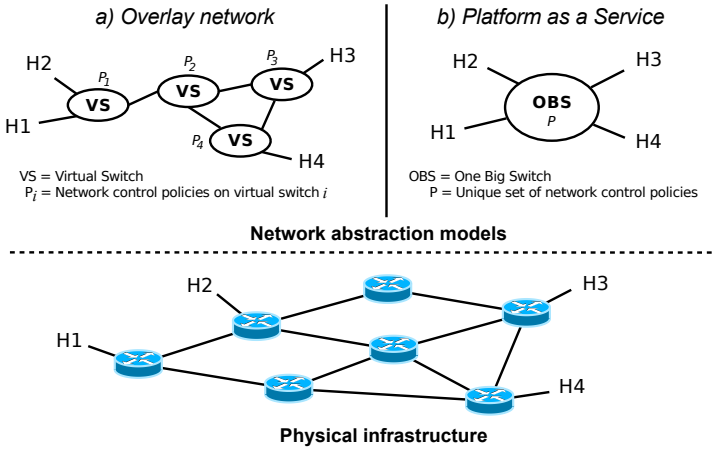

FIGURE 19. a) Le modèle *One Big Switch* | b) *Le modèle overlay network*

donné qu'aucune marge de conception n'est possible, et cela plus particulièrement dans le cas de larges réseaux qui incluent de nombreux services et utilisateurs.

Le deuxième inconvénient, le plus préjudiciable à notre avis, est que ce modèle d'abstraction est très peu flexible et ne permet pas de prendre en considération des possibles contraintes d'ordre physique si cela s'avèrait nécessaire ou souhaité. En effet, lors de la configuration d'un réseau, un administrateur peut être dans une position où il ne **souhaite pas** ou ne **peux pas** masquer certaines contraintes ou particularités de l'infrastructure physique comme le souhait de vouloir gérer de manières différentes deux groupes d'équipements physiques à cause de leur emplacement, performance ou suivant le type de flux qu'ils acheminent. Ainsi, l'utilisation du modèle OBS suppose que l'administrateur souhaite tout le temps entièrement masquer toutes les caractéristiques de l'infrastructure physique, ce qui n'est pas toujours le cas.

En somme, le modèle d'abstraction OBS est un modèle d'abstraction très fort et peut-être même le plus haut niveau d'abstraction qu'on puisse obtenir. Ce dernier permet aux administrateurs de se concentrer sur leurs services les plus complexes et abstrait la fonctionnalité de transport du réseau physique en un seul commutateur logique. Cependant, ce modèle est trop contraignant, surtout dans le contexte où il va être utilisé comme base pour un langage de programmation réseau, dont une des exigences est d'être flexible et utilisable dans différents contextes.

3.3.2 *Overlay network*

Un deuxième modèle d'abstraction bien connu dans notre communauté est le modèle *Overlay network* [Casado et al., 2010] qui consiste à superposer au-dessus d'une même infrastructure physique partagée un ou plusieurs réseaux virtuels composés de commutateurs logiques qui sont assez semblables à ceux présents dans l'infrastructure physique. En effet, ces derniers exposent des tables de commutation et des ports virtuels qui sont connectés entre eux par le bais de liens virtuels. La plus grande force de ce modèle

d'abstraction est sa flexibilité, étant donné qu'un administrateur a la possibilité de construire divers types de topologies virtuelles qui correspondent à ces objectifs de haut niveau ou aux contraintes physiques qu'il souhaite prendre en considération, puis d'associer chaque équipement virtuel à un ou plusieurs équipements physiques.

Toutefois, ce modèle, à l'inverse du modèle OBS, ne fait aucune distinction entre les politiques de transport et les services réseaux plus complexes, bien que ces deux deux types de politiques résolvent deux problématiques différentes [Casado et al., 2012]. Cette approche a pour conséquence que la définition des services et des fonctions complexes devient une opération plus difficile étant donné que l'administrateur doit considérer des problématiques de transport dans le code de ses fonctions réseau. De plus, cette approche diminue la modularité des programmes de contrôle, étant donné que les politiques de transport et les services réseau sont fortement couplés. A titre d'exemple, si un service de filtrage applicatif est déplacé sur le réseau virtuel alors cela nécessitera de mettre à jour les commutateurs virtuels afin de rediriger les flux vers la nouvelle localisation du service.

En définitive, le modèle *overlay network* est une approche très flexible, cependant elle n'est pas la plus optimale car l'administrateur doit aussi s'occuper de la gestion du réseau virtuel lui même au lieu de se concentrer pleinement sur la définition et la programmation des services réseau plus riches.

3.4 Le modèle Edge-Fabric comme plan de contrôle virtuel

Afin de répondre au mieux aux exigences d'un plan de contrôle virtuel, nous nous sommes basés sur un concept bien connu dans la communauté réseau qui est de faire une claire distinction entre les équipements de cœur et de bordure du réseau, comme c'est notamment le cas dans les architectures MPLS [Ghein, 2006]. Cette idée a été également évoquée par [Casado et al., 2012] afin de faire évoluer les architectures SDN physiques actuelles vers un modèle plus flexible. La nouveauté de notre proposition est que nous souhaitons faire remonter cette idée de conception des infrastructures du plan physique au plan virtuel. En effet, au meilleur de notre connaissance, aucun langage ou outil n'a encore intégré ce concept au niveau du plan virtuel ou même au niveau d'un plan de contrôle SDN.

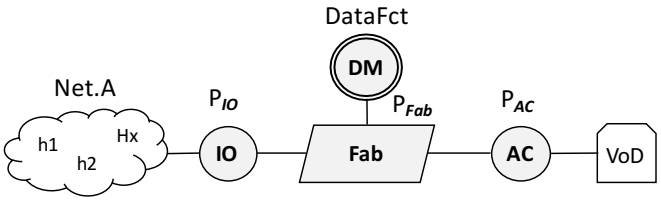

FIGURE 20. Le modèle Edge-Fabric

Le modèle d'abstraction Edge-Fabric [1] que nous proposons s'appuie sur quatre principaux types de composants, comme cela est schématisé à la figure 20 :

1. Dans la suite du document, nous utilisons la terminologie anglo-saxonne pour designer le modèle Edge-Fabric ainsi que ses composants (edges, data machines, etc.)

- ***Edges*** : composants virtuels qui supportent l'exécution des fonctions et des services réseaux complexes du **plan de contrôle**.
- ***Data machines*** : représentent des *edges* spécialisés qui effectuent des opérations complexes sur les paquets au niveau du **plan de données**.
- ***Fabrics*** : composants virtuels plus restreints qui se chargent principalement des problématiques liées au transport des paquets.
- ***Hosts and Networks*** : représentent les sources et destinations des paquets.

Ainsi, ce modèle permet, d'une part, une séparation claire entre les politiques de transport (ainsi que leurs exigences) et les services réseau plus complexes tels que le contrôle d'accès ou l'équilibrage de charge. D'autre part, la possibilité d'utiliser plusieurs équipements virtuels permet de répondre, si c'est nécessaire, aux différentes contraintes de l'infrastructure physique susceptibles de survenir. Dans ce qui suit nous présentons plus en détails chacun de ces composants virtuels et dans le chapitre suivant les primitives du langage AirNet qui permettent de les déclarer et de les configurer.

3.4.1 Edges

Dans notre modèle d'abstraction, les edges représentent des composants virtuels polyvalents disposés à la périphérie du réseau virtuel. De ce fait, ils seront, conceptuellement, connectés aux hôtes qui représentent les sources et destinations des paquets. Ainsi, les edges ont deux principales fonctionnalités :

- Interface hôte-réseau : interface par laquelle les hôtes informent le réseau de leurs besoins.
- Execution des fonctions réseaux : les edges offrent un support logique pour l'installation des services réseaux à valeur ajoutée, soit à l'entrée ou à la sortie de ce dernier.

Le rôle de l'interface hôte-réseau consiste principalement à identifier les différents flux qui entrent dans le réseau ainsi que leurs exigences. Cela est notamment réalisé en se basant sur les champs d'en-tête des paquets (source, destination, ToS, etc.). Concrètement, un edge examine les en-têtes des paquets entrant, puis attache à ces paquets une étiquette (label) qui sera par la suite utilisé pour l'acheminement au sein du cœur du réseau (la fabric). Ainsi, un edge doit fournir des primitives qui permettent, d'une part, de capturer et de discriminer les différents flux de paquets entrants et sortants et, d'autre part, de leur attribuer des étiquettes pour le transport à l'intérieur du réseau.

La deuxième fonctionnalité des edges est le support des fonctions réseaux complexes. En effet, le modèle Edge-Fabric suppose de cantonner l'intelligence du réseau à la périphérie et de garder son cœur simple, comme c'est déjà le cas dans beaucoup de cas d'utilisation réel [Casado et al., 2012]. Considérant cela, les edges devraient exposer un jeu d'instructions riches, qui permet notamment aux administrateurs de spécifier leurs diverses fonctions réseaux telles que le contrôle d'accès ou l'inspection en profondeur des paquets. Dans le contexte de notre proposition, ces fonctions sont divisées en deux principaux types :

- Statiques : fonctions qui seront installées et exécutées au niveau des commutateurs physique présents dans le plan de données, et vers lesquels un edge est associé.
- Dynamiques : fonctions qui seront exécutées au niveau du contrôleur SDN (plan de contrôle).

Enfin, les edges étant des composants logiques, ils peuvent dans la réalité *mapper* vers un ou plusieurs commutateurs physique de bordure, et de la même manière un commutateur physique peut être associé à un ou plusieurs edges logiques, comme cela est montré dans la figure 21. Cette flexibilité est primordiale de sorte à être capable de considérer plusieurs contextes d'utilisation, divers choix de conception et aussi de pouvoir répondre à diverses contraintes de topologie physique comme par exemple que le fait de vouloir modéliser tous les points d'accès sans fils par un edge et les accès filaires par un autre.

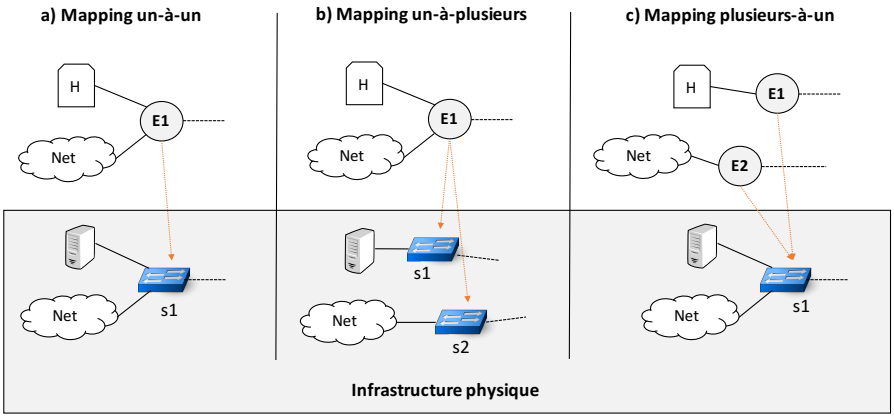

FIGURE 21. Possibilités de mapping pour les edges

3.4.2 Data machines

Les *data machines* sont des edges spéciaux dont la seule particularité est que leurs fonctions sont toutes exécutées au niveau du plan de données. En effet, les data machines dans le plan virtuel sont semblables aux *middleboxes* qu'on retrouve communément dans les réseaux au niveau physique. Chaque data machine peut embarquer une ou plusieurs fonctions qui sont capables d'effectuer un traitement complexe sur les données que transportent un paquet tels que de la compression, du chiffrement ou du transcodage. Une date machine, comme les edges, peut aussi *mapper* vers une ou plusieurs machines physiques réelles.

Un autre aspect important des data machine est que, à l'inverse des edges classiques, elles ne participent aucunement au processus de prise de décision et de contrôle. En effet, ces machines opèrent exclusivement au niveau du plan de données, elles reçoivent en entrée un paquet et retourne en sortie un ou plusieurs paquets.

3.4.3 Fabrics

Les fabrics [2] représentent des composants virtuels qui se chargent exclusivement de la logique de transport entre les edges et les data machines. Concrètement, une fabric représente une collection d'équipements de commutation physique de cœur de réseau dont le premier objectif est le transport des paquets d'un point à un autre.

Pour identifier un flux de paquets, une fabric se base exclusivement sur un label qui a été préalablement inséré par un edge, permettant ainsi de séparer les deux interfaces "hôte-réseau" et "réseau-réseau". Cette séparation est une des clés de modularité de ce modèle. En effet, un administrateur peut mettre à jour les fonctions qui sont appliquées sur un flux au niveau d'un edge et tant qu'il ne change pas le label qui leur a été attribué, la politique de transport au niveau de la fabric ne sera pas impactée. Inversement, un administrateur a aussi la possibilité d'effectuer des mises à jour dans les politiques de transport sans que cela impact ou nécessite un changement au niveau des politiques installées sur les edges.

Ainsi, les fabrics exposent un jeu d'instructions plus restreint comparé à celui des edges. Ces instructions incluent principalement des primitives qui permettent de transporter les flux d'un edge à un autre suivant des contraintes de QoS, mais aussi des primitives qui permettent de faire passer un flux par des data machines puis de le récupérer et de l'acheminer vers sa destination finale.

Communément, un réseau virtuel contiendra une seule fabric qui représentera ses capacités de transport. Néanmoins, un administrateur a aussi la possibilité d'utiliser plusieurs fabrics afin de prendre en considération des contraintes physiques ou des objectifs de contrôle de haut niveau, sachant que deux fabrics peuvent mapper vers un groupe de commutateurs commun ou distinct. En effet, l'utilisation des fabrics peut être pensée sous trois principales approches :

- Une seule fabric (Figure 22 a) : approche similaire à celle du modèle *One Big Switch* par le fait qu'elle permet d'abstraire toutes les capacités de transport du réseau au sein d'une seule et unique fabric.
- Plusieurs fabrics en parallèle (Figure 22 b) : approche qui permet de distinguer plusieurs possibilités de transport au sein d'un réseau. Ce souhait peut être d'ordre conceptuel (politiques de haut niveau) ou en rapport à des contraintes physiques qu'un administrateur souhaite faire remonter au niveau virtuel.
- Une séquence de fabrics (Figure 22 c) : approche qui permet d'expliciter un chemin de transport qui est constitué de plusieurs tronçons différents. Cette différence peut-être la aussi d'ordre physique ou simplement conceptuel.

Enfin, il est à noter que, comme pour les edges, un administrateur a le choix entre plusieurs possibilités de mapping, notamment : un-à-un, un-à-plusieurs ou plusieurs-à-un. La seule contrainte qui existe est que les équipements physiques doivent être des équipements de cœur de réseau.

2. Comme indiqué précédemment, dans la suite du document nous écrivons *fabric* et non *fabrique* pour rester syntaxiquement cohérent avec le modèle *Edge-Fabric*

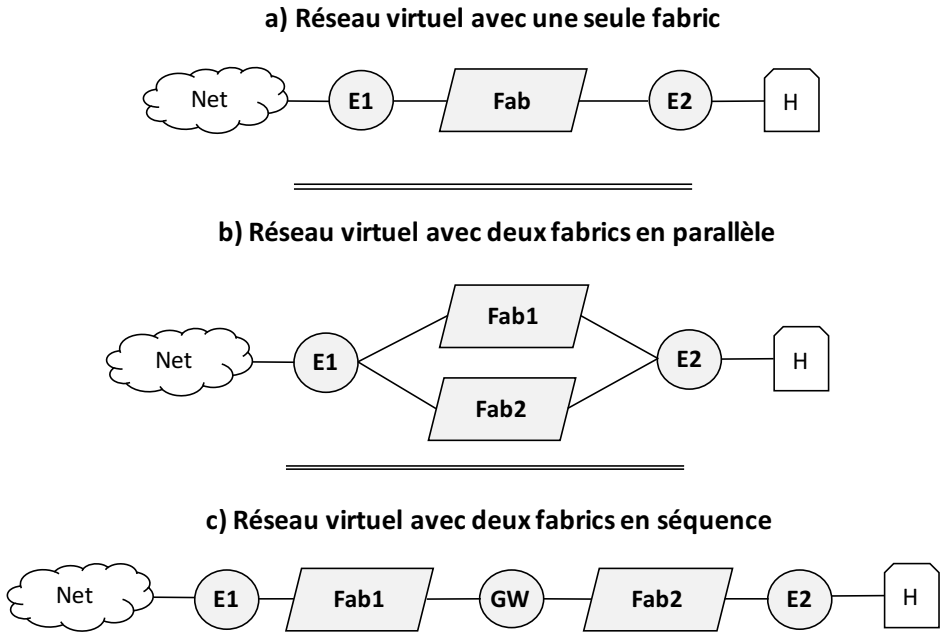

FIGURE 22. Trois principales approches d'utilisation des fabrics

3.4.4 Hosts et Networks

Les *hosts* (hôtes) et les *networks* (réseaux) sont des composants qui permettent de représenter les sources et destinations des paquets. Un host représente une seule machine munie d'une adresse logique et physique alors qu'un network représente un sous-réseau logique composé d'une collection de machines. Notez que seuls des noms symboliques sont manipulés et non des adresses réseaux. Ces abstractions masquent donc divers détails techniques de bas niveau spécifiques aux machines hôtes, permettant ainsi aux administrateurs de manipuler, au sein de leurs programmes, des identifiants qui sont significatifs à leurs objectifs de haut niveau en lieu et place des traditionnelles adresses IP et MAC.

3.5 Conclusion

Dans ce chapitre nous avons présenté le modèle Edge-Fabric qui, à l'inverse du modèle *overlay network*, introduit une frontière logique entre les politiques de transport et les fonctions réseaux les plus complexes. Cette distinction permet, d'une part, aux administrateurs de se concentrer sur la définition de leurs fonctions réseaux, qui seront par la suite installées sur les edges, et d'autre part, à ces deux types de politiques différents d'évoluer séparément et d'être réutilisables dans d'autres contextes d'utilisations.

Par ailleurs, le modèle Edge-Fabric répond aussi au manque de flexibilité du modèle OBS. En effet, le fait de pouvoir utiliser plusieurs edges et plusieurs fabrics permet, d'une part, de répondre à différents contextes d'utilisations, et d'autre part, de pouvoir considérer

les possibles contraintes physiques que l'administrateur ne souhaite pas ou ne peux pas abstraire. En revanche, si aucune contrainte particulière n'est à considérer, l'administrateur peut opter pour une seule fabric qui va abstraire toutes les capacités de transport de son réseau physique.

Dans le chapitre suivant, nous présentons le langage AirNet qui a été entièrement construit autour du modèle Edge-Fabric. Ainsi, ses primitives permettent de configurer les équipements virtuels de ce modèle. Quant à son modèle de programmation, il est conforme à la logique du modèle Edge-Fabric, en d'autres termes les traitements complexes sont effectués aux bordures du réseau (sur les edges) afin de laisser le cœur du réseau simple (les fabrics), s'occupant essentiellement des problématiques de transports.

Chapitre 4
AirNet : langage et modèle de programmation

Dans ce chapitre, nous présentons la syntaxe et la sémantique des primitives du langage AirNet ainsi que son modèle de programmation. D'abord, nous allons décrire les primitives qui permettent de construire des topologies virtuelles ainsi que les différentes possibilités de conception envisageables. Puis, nous présenterons les primitives de base du langage AirNet, ou en d'autres termes, les jeux d'instructions des composants logiques edges et fabrics. Enfin, nous détaillerons les primitives avancées (fonctions réseaux) qui permettent aux administrateurs d'écrire des politiques dynamiques afin de construire des programmes de contrôle plus élaborés et réalistes.

4.1 Définition de la topologie virtuelle

AirNet a un modèle de programmation qui nécessite qu'un administrateur doit spécifier une topologie virtuelle, suivant le modèle Edge-Fabric, qui correspond à ses objectifs de haut niveau ou aux contraintes physiques qu'il souhaite prendre en considération. Puis, il doit spécifier les modules de contrôle qui vont s'exécuter sur cette topologie virtuelle. Cette dernière est complètement détachée de l'infrastructure physique. Cela signifie qu'elle ne contient aucune référence vers les équipements physiques, notamment les commutateurs OpenFlow.

Pour un administrateur, définir une topologie abstraite est une opération simple qui consiste, principalement, à spécifier d'une manière déclarative les différents composants logiques (`edge`, `fabric`, `host` et `network`) qu'inclut une topologie virtuelle, ainsi que les liens qui les rattachent et cela en utilisant les instructions qui sont présentées dans la figure 23.

```
VTopology() crée un objet topologie virtuelle.
addEdge("id", ports) ajoute un edge à une topologie virtuelle.
addFabric("id", ports) ajoute une fabric à une topologie virtuelle.
addDataMachine("id", ports) ajoute une data machine à une topologie virtuelle.
addHost("id", ports) ajoute un hote à une topologie virtuelle.
addNetwork("id", ports) ajoute un réseau à une topologie virtuelle.
addLink(("id1", port1), ("id2", port2)) ajoute un lien virtuel entre deux composants.
```

FIGURE 23. Primitives AirNet pour construire une topologie virtuelle

Ainsi, la définition d'une topologie virtuelle correspondra à une fonction qui inclut une instruction de déclaration par composant virtuel. Pour chaque composant, l'administrateur

doit spécifier l'identifiant et le nombre de ports qu'il contient. Puis dans un second temps, l'administrateur doit relier tous ces équipements en utilisant la primitive `addLink`, en indiquant quels sont les équipements et les ports qui doivent être reliés. Par ailleurs, lors de la phase de conception, un certain nombre de règles doivent être respectées, nous les énumérons ci-dessous :

- La topologie minimale qu'un administrateur peut définir consiste en une fabric et deux edges qui lui sont reliés.
- Une topologie virtuelle peut contenir une ou plusieurs fabrics.
- Les utilisateurs finaux (host et network) sont toujours connectés à des edges.
- Les edges et les data machines ont toujours, au moins, un port qui est connecté à une fabric.

Ces choix de conception ont été motivés, principalement, par le souhait de maintenir une séparation claire et précise entre les interfaces **Hôte-Réseau** (utilisateurs-edge) et **Réseau-Réseau** (edge-fabric). Néanmoins, nous laissons aussi beaucoup de liberté pour les administrateurs dans leurs choix de conception afin qu'ils puissent s'adapter aux différents contextes d'utilisation possibles. Par exemple, on peut facilement imaginer deux topologies virtuelles différentes pour une même topologie physique, comme cela est montré dans les figures 24 et 25.

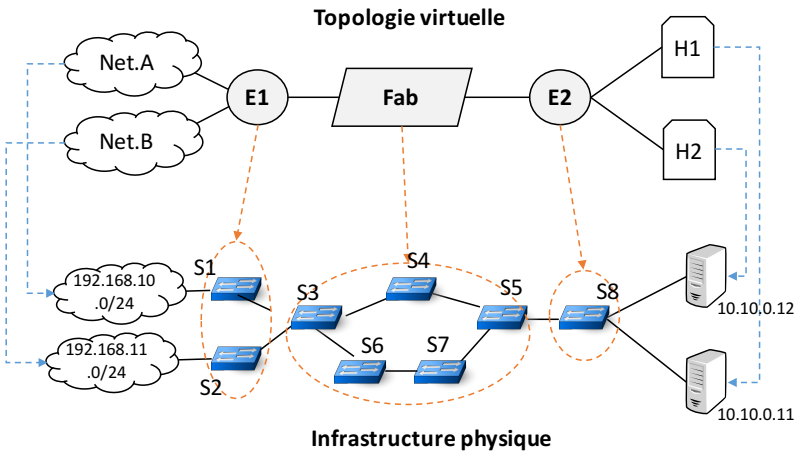

FIGURE 24. Topologie virtuelle avec une seule fabric

La figure 24 montre une première possibilité de conception qui consiste à utiliser une seule fabric pour représenter tous les équipements de cœur de réseau et deux edges pour connecter les hôtes et les réseaux d'utilisateurs. On peut aussi constater, sur cette même figure 24, que pour le mapping des edges, on a un mapping **un-à-un** pour l'edge `E2` et un mapping **un-à-plusieurs** pour l'edge `E1`. L'extrait ci-dessous représente le programme AirNet qui permet d'obtenir cette topologie virtuelle. Ce dernier correspond globalement, comme il a été mentionné auparavant, à une instruction par composant virtuel.

```
def create_topology()
 topology = VTopology()
 topology.addEdge("E1", 3)
 topology.addEdge("E2", 3)
 topology.addFabric("Fab", 2)
 topology.addHost("H1")
 topology.addHost("H2")
 topology.addNetwork("Net.A")
 topology.addNetwork("Net.B")
 topology.addLink(("E1",1),("Net.A",0))
 topology.addLink(("E1",2),("Net.B",0))
 topology.addLink(("E1",3),("Fab",1))
 topology.addLink(("E2",1),("H1",0))
 topology.addLink(("E2",2),("H2",0))
 topology.addLink(("E2",3),(Fab",2))
 return topology
```

La figure 25, quant à elle, représente une deuxième possibilité de conception où ici on a utilisé deux fabrics et trois edges. Dans ce cas, l'utilisation de deux fabrics peut être motivée, par exemple, par le souhait de vouloir expliciter directement au niveau de la topologie virtuelle deux chemins différents, c'est-à-dire le chemin `[S3, s4, s5]` et le chemin `[S3, S6, S7, s5]`. Par ailleurs, notons aussi qu'il est possible d'avoir un mapping **plusieurs-à-un** comme c'est le cas pour les deux edges `E2` et `E3` qui mappent vers le même équipement physique `S8`.

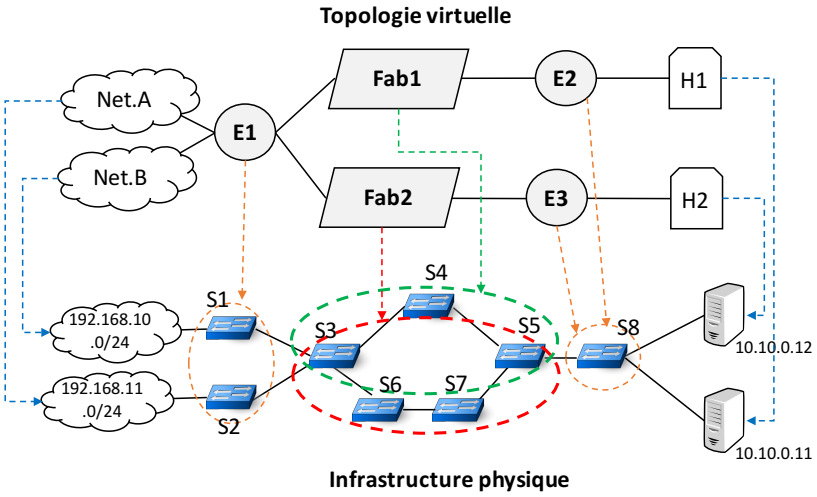

FIGURE 25. Topologie virtuelle avec deux fabrics

4.2 Primitives du langage AirNet

AirNet fournit deux principaux types de politiques : les politiques statiques et les politiques dynamique. Dans cette section nous allons commencer par présenter les politiques statiques qui, comme leur nom l'indique, sont des politiques qui, une fois installées, elles ont toujours le même comportement et retournent toujours le même résultat pour les paquets

d'un même flux.

Chaque politique de contrôle peut être modélisée, conceptuellement, comme une fonction qui prend en entrée un paquet et retourne en sortie un ensemble de paquets. Cette fonction décrit comment un commutateur devrait se comporter à la réception d'un paquet. Par exemple, si une fonction retourne un ensemble vide cela est équivalent à un rejet du paquet. Par contre si la fonction retourne un paquet identique, mais avec une localisation différente, cela est équivalent alors à une opération de commutation.

Par ailleurs, il est à noter que les politiques statiques peuvent être elles aussi divisées en deux principaux groupes, suivant si elles sont installées sur des edges ou sur des fabrics. Dans ce qui suit, nous allons d'abord commencer par présenter les politiques des edges, puis nous détaillerons celles des fabrics.

4.2.1 Primitives d'un Edge

Comme cela a été mentionné précédemment, les edges sont des équipements logiques de traitement de paquets positionnés à la périphérie d'un réseau virtuel. Les politiques qui sont installées sur les edges suivent une logique de type `prédicat -> actions`, ou le prédicat représente un filtre qui est appliqué sur l'ensemble des paquets entrant dans un edge, alors que `actions` représente une liste d'actions à appliquer, en parallèle ou en séquence [1], sur l'ensemble des paquets retournés. La figure 26 décrit l'ensemble des primitives que fournit un edge, alors que la figure 27 représente leur modélisation fonctionnelle. Sur cette dernière, on peut voir l'ensemble résultat (partie droite) qui est retourné après l'application d'une instruction sur un paquet `pk` (partie gauche).

```
Prédicat e ::= all_packets          identité, retourne tous les paquets
             | match(Edge=e, h=v)   filtre
             | e1 + e2              Disjonction
             | e1 >> e2             Conjonction
             | ! e                  Négation
Politiques p ::=  e                 Filtre
               |  modify(h=v)       Modification
               |  forward(next_dst) Commutation
               |  tag(label)       Attribution d'un label
               |  drop              Rejet
               |  p1 + p2          Composition parallele
               |  p1  >>  p2       Composition séquentielle
```

FIGURE 26. Primitives d'un Edge

4.2.1.1 Les filtres

La primitive `match(h=v)` représente le filtre qui sera communément utilisé dans les programmes AirNet. Quand un filtre est appliqué sur un ensemble de paquets, il permet de retourner tous les paquets dont la valeur de leur champ d'en-tête `h` est égal à `v`. Le filtre `all_packet`, quant à lui, retourne tous les paquets qui passent par un edge donné.

1. Les opérateurs de composition seront détaillés dans la section 4.3.

```
⟦all_packet⟧pk = {pk}
⟦drop⟧pk = { }
⟦match(h=v)⟧pk = {pk} si pk.h == v, sinon { }
⟦modify(h=v)⟧pk = {pk[h=v]}
⟦forward(dst)⟧pk = {pk[loc=dst]}
⟦tag(label)⟧pk = {pk[lb=label]}
⟦p + q⟧pk = ⟦p⟧pk + ⟦q⟧pk
⟦p ≫ q⟧pk = q(p(pk))
```

FIGURE 27. Vision fonctionnelle des primitives d'un Edge

Notre implémentation actuelle du langage AirNet suppose de remonter les informations suivantes concernant les paquets, informations qui peuvent notamment être utilisées pour construire des filtres :

- `src` : représente les paquets qui proviennent d'un hôte ou d'un réseau déclaré dans la topologie virtuelle (sous la forme d'un nom symbolique).
- `dst` : représente les paquets à destination d'un hôte ou d'un réseau déclaré dans la topologie virtuelle (sous la forme d'un nom symbolique).
- `nw_src` : représente l'adresse réseau source d'une machine ou d'une collection de machines. Cette adresse sera communément une adresse IP.
- `nw_dst` : représente l'adresse réseau destination d'une machine ou d'une collection de machines. Cette adresse sera communément une adresse IP.
- `dl_src` : représente l'adresse liaison source, communément une adresse MAC.
- `dl_dst` : représente l'adresse liaison destination, communément une adresse MAC.
- `tp_src` : représente le port transport source.
- `tp_dst` : représente le port transport destination.
- `nw_proto` : indique le protocole de communication spécifié dans l'en-tête de niveau trois (ICMP, UDP, TCP, etc.).

Comme exemple concret, nous allons considérer l'extrait de programme ci-dessous qui représente un filtre qui permet de capturer tous les paquets web (port TCP destination 80) qui proviennent de l'adresse IP `1.2.3.4` et qui ont comme destination l'adresse IP `10.10.10.10` :

```
match(Edge="E1", nw_src="1.2.3.4", nw_dst="10.10.10.10", nw_proto=TCP, tp_dst=80)
```

Les filtres d'AirNet sont aussi munis des opérations d'ensemble d'intersection (`»`), d'union (`+`) et de complément (`!`). La liste d'exemples ci-dessous donne un bon aperçu sur les opérations ensemblistes des filtres d'AirNet :

`match(Edge="E1", nw_src="1.2.3.4") + match(Edge="E1", nw_src="4.3.2.1")` : retourne l'ensemble des paquets qui proviennent de l'adresse IP source `"1.2.3.4"` ou `"4.3.2.1"`.

`match(Edge="E1", nw_src="1.2.3.4") » match(Edge="E1", tp_dst=80)` : retourne l'ensemble des paquets web (`tp_dst=80`) qui proviennent de l'adresse IP source `"1.2.3.4"`.

`!match(Edge="E1", nw_src="1.2.3.4")` : retourne l'ensemble des paquets qui ne proviennent pas de l'adresse IP source `"1.2.3.4"`.

`match(Edge="E1", nw_src="1.2.3.4") » match(Edge="E1", nw_src="4.3.2.1")` : retourne un ensemble vide car aucune intersection n'existe entre les paquets qui proviennent de l'adresse IP source `"1.2.3.4"` et de l'adresse source `"4.3.2.1"`.

4.2.1.2 Les actions

Pour construire une politique de contrôle, un administrateur a besoin d'appliquer des actions sur les paquets qui sont retournés par les filtres. Comme on peut le voir sur la figure 26, AirNet inclut les actions (`forward`, `modify` et `drop`) qu'on retrouve communément embarquées dans les équipements réseaux, en plus de l'action `tag(label)` qui permet d'attribuer une étiquette à un flux de paquets.

Dans ce qui suit, nous expliquons plus en détails chacune de ces primitives. Pour ce faire, nous allons considérer un exemple d'utilisation que nous allons appliquer sur la topologie virtuelle qui est schématisée à la figure 28. Cette dernière inclut deux réseaux (`Net.A` et `Net.B`), deux hôtes qui jouent le rôle de serveurs web (`WS1` et `WS2`), deux edges (`IO` et `AC`) et enfin une fabric (`Fab`). L'objectif global est d'implémenter une politique de répartition de charge entre les deux serveurs `WS1` et `WS2`. Par ailleurs, nous supposons aussi que les utilisateurs des deux réseaux `Net.A` et `Net.B` ont uniquement connaissance de la présence d'un seul serveur avec une adresse publique, alors que les deux serveurs `WS1` et `WS2` sont configurés avec des adresses IP privées.

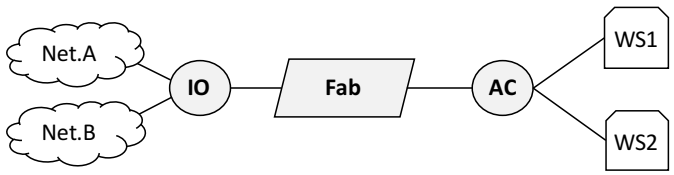

FIGURE 28. Topologie virtuelle avec deux serveurs web

Commuter un paquet : L'action `forward(dst)` est l'action de base du langage AirNet, dans la mesure où elle permet de commuter un paquet, au sein du même edge, d'un port d'entrée vers un port de sortie, en direction d'une destination `dst` (destination directement connecté à l'edge). L'extrait de programme ci-dessous donne un exemple d'une politique de relayage qui est appliquée sur l'edge `IO` de la topologie 28. Cette politique contient deux règles qui permettent de commuter les flux de paquets, respectivement, vers les réseaux `Net.A` et `Net.B`.

```
e1 = match(edge="IO", dst="Net.A") >> forward("Net.A")
e2 = match(edge="IO", dst="Net.B") >> forward("Net.B")
```

Attribuer une étiquette à un paquet : Une des fonctionnalités principales des edges est d'identifier les flux de paquets entrants et leurs besoins, puis de leur attribuer une étiquette pour le transport au sein d'une fabric. Cette étiquette est primordiale dans le schéma de communication que nous proposons, étant donné que c'est l'unique information qui sera utilisée par la fabric afin d'identifier les différents flux qui y entrent. A cet égard, AirNet

inclut la primitive `tag(label)` qui permet d'attribuer une étiquette `label` à un paquet entrant, comme cela est montré dans l'extrait [2] de programme ci-dessous :

```
match(edge="IO", tp_dst=HTTP) >> tag("inWebFlows") >> forward("Fab")
```

La règle qui est montrée ci-dessus capture tous les flux web (`tp_dst=HTTP`) entrants, leur attribue le label `inWebFlows`, puis les commute vers la fabric `Fab`. Cette règle est construite d'abord en spécifiant un filtre qui capture tous les paquets qui nous intéressent, puis de composer séquentiellement le résultat retourné par ce filtre avec la primitive `tag` afin de leur attribuer le label `inWebFlow`. Enfin, ce résultat est passé à la primitive `forward` (en ayant une nouvelle fois recours à la composition séquentielle) qui va se charger de commuter tous les paquets résultants vers leur destination appropriée, dans ce cas c'est la fabric `Fab`

Rejeter un paquet : En plus de l'action `forward` qui permet de commuter un paquet vers un port de sortie, AirNet inclut aussi l'action `drop` qui permet de rejeter un paquet. Cette action est importante vu qu'elle contrôle l'accès au réseau. Comme exemple, on pourrait imaginer une règle qui rejetterait tous les flux SSH entrants (règle `e4`). Cette dernière serait composée avec les précédentes règles déjà définis afin de former une fonction pour l'edge `IO` qui lui permet de gérer les entrées/sorties vers le réseau.

```
def inputOutput()
 e1 = match(edge="IO", tp_dst=HTTP) >> tag("inWebFlows") >> forward("Fab")
 e2 = match(edge="IO", dst="Net.A") >> forward("Net.A")
 e3 = match(edge="IO", dst="Net.B") >> forward("Net.B")
 e4 = match(edge="IO", tp_dst=SSH) >> drop
 return e1 + e2 + e3 + e4
```

Par ailleurs, il est à noter, qu'AirNet permet de manipuler des identifiants symboliques, en lieu et place des détails de bas niveau. En effet, on peut voir dans les précédents extraits que pour commuter un paquet, un administrateur a besoin uniquement de communiquer l'identifiant de la destination et non pas le numéro de port qui permet de l'atteindre.

Modifier l'en-tête d'un paquet : L'action `modify(h=v)` modifie la valeur de l'en-tête `h` d'un paquet à `v`, sachant que les champs sur lesquels pourra s'appliquer l'action `modify` sont les mêmes que ceux utilisés pour le filtre `match(h=v)`. Pour illustrer l'utilisation de l'action `modify`, considérons l'exemple ci-dessous où l'objectif est d'installer une politique qui répartit la charge entre les deux serveurs `WS1` et `WS2` sur l'edge `AC`, sachant que la politique de répartition consiste à rediriger tous les flux du réseau `Net.A` vers le serveur `WS1` et ceux du réseau `Net.B` vers le serveur `WS2`.

```
def loadBalancer()
 e1 = match(edge="AC", src="Net.A", nw_dst=PUBLIC_IP, tp_dst=HTTP) >>
       modify(nw_dst=WS1_IP, dl_dst=WS1_MAC) >> forward("WS1")
 e2 = match(edge="AC", src="Net.B", nw_dst=PUBLIC_IP, tp_dst=HTTP) >>
       modify(nw_dst=WS2_IP, dl_dst=WS2_MAC) >> forward("WS2")
 e3 = match(edge="AC", tp_src=HTTP) >> tag("outWebFlows") >> forward("Fab")
 return e1 + e2 + e3
```

2. Notez que dans cet exemple et dans la suite nous filtrons uniquement sur le numéro de port pour alléger la notation.

Ainsi, on peut voir que la première règle `e1` capture, dans un premier temps, tous les flux web qui proviennent du réseau `Net.A` et qui ont comme destination l'adresse publique du serveur web. Puis, l'action `modify` est appliquée sur tous ces paquets afin de modifier leurs champs d'en-têtes d'adresse IP destination (`nw_dst`) et d'adresse MAC destination (`dl_dst`) par les valeurs privées du serveur web `WS1`. La règle `e2` effectue les mêmes opérations, mais cette fois-ci entre les utilisateurs du réseau `Net.B` et le serveur web `WS2`. La règle `e3`, quant à elle, prend en charge tous les flux web sortant en leur attribuant notamment le label `outWebFlows`, puis en les commutant vers la fabric `Fab`. Enfin, ces trois règles sont composées parallèlement pour être, par la suite, installées sur l'edge `AC`.

A ce stade, nous avons uniquement décrit, sur l'exemple précédent, les politiques qui seront installées sur les deux edges `IO` et `AC`. Dans ce qui suit, nous allons présenter les politiques qui peuvent être installées sur les fabrics et qui permettent d'implémenter un service de transport.

4.2.2 Primitives d'une Fabric

Les fabrics sont dés composants virtuels qui se chargent exclusivement des problématiques liées au transport des paquets à l'intérieur du réseau. Pour identifier ces paquets, une fabric se base uniquement sur l'étiquette qui a été insérée préalablement par un edge. En effet, c'est à travers cette étiquette qu'un opérateur exprime le type de service de transport qu'il souhaite apporter à un flux de paquet. Ainsi, les fabrics exposent trois principales primitives, comme cela est montré dans la figure 29, qui permettent de transporter des flux de paquets d'un edge à un autre, suivant des contraintes de QoS et potentiellement au travers d'une chaîne de services (des data machines).

```
Prédicat e ::= catch([edge], fabric, flow)      Retourne un flux
             | e1 + e2                          Disjonction
             | e1 >> e2                         Conjonction
             | ! e                              Négation
Politiques p ::= e     Prédicat
             |   carry(edge, [contraintes QoS])   Transporte un flux vers un edge
             |   via(dataMachine)  Fait passer un flux par un noeud intermédiaire
             |   p1 + p2         Composition parallele
             |   p1 >> p2        Composition séquentielle
```

FIGURE 29. Primitives d'une Fabric

La première primitive `catch([src], fabric, flow)` permet de capturer un flux entrant dans une fabric en se basant sur le label (`flow`) du flux. Par ailleurs, une fabric peut aussi se baser sur l'edge émetteur (`src`) afin d'identifier un flux. La spécification de l'edge émetteur permet à l'administrateur d'utiliser un même label pour plusieurs flux qui proviennent d'edges différents. La primitive `carry`, quant à elle, permet de transporter un flux de paquets d'un edge à un autre tout en offrant la possibilité de spécifier des paramètres de qualité de service tels que : la bande passante minimum à assurer ou le délai maximal toléré. Il est à noter que la primitive `via` sera présentée dans les sections suivantes qui traitent des fonctions de données et des data machines.

Comme exemple, nous allons considérer l'extrait de programme ci-dessous qui représente la politique de transport utilisée pour notre cas d'étude de répartition de charge.

```
def transport()
 t1 = catch(fabric="Fab", flow="inWebFlows") >> carry("AC")
 t2 = catch(fabric="Fab", flow="outWebFlows") >> carry("IO")
 return t1 + t2
```

La fonction de transport qui est décrite ci-dessus inclut deux règles : la première capture tous les paquets en direction des serveurs web (`flow="inWebFlows"`) et les achemine vers l'edge `AC`, alors que la deuxième permet de faire l'inverse en acheminant tous les flux web sortant de l'edge `AC` vers l'edge `IO`.

Par ailleurs, de la même manière que pour les filtres des edges, les primitives `catch` peuvent être composées afin d'appliquer une même politique de transport à plusieurs flux différents, comme cela est montré dans l'exemple ci-dessous qui décrit une politique transport de l'edge `IO` vers l'edge `AC` pour les deux flux `InWebFlows` et `InSSHFlows` (On suppose ici que les flux SSH sont autorisés).

```
def transport()
(catch(fabric="Fab", src="IO", flow="inWebFlows") +
 catch(fabric="Fab", src="IO", flow="inSSHFlows") ) >> carry("AC")
```

4.3 Opérateurs de composition

Comme vu précédemment, AirNet intègre deux opérateurs de composition : séquentiel et parallèle. Ces deux opérateurs permettent, dans un premier temps, de composer les primitives de chaque équipement virtuel afin d'obtenir des règles de contrôle puis, dans un second temps, de composer ces règles en modules de contrôle indépendants. En effet, dans les exemples précédents on a souvent eu recours à une opération de composition séquentielle entre un filtre et une liste d'actions afin d'obtenir une règle de contrôle. Par la suite, nous avons composé parallèlement plusieurs de ces règles entre elles afin de former des modules de contrôle à l'exemple des fonctions `loadBalancer` ou `inputOutput`. Dans ce qui suit, nous allons présenter plus en détails chacun de ces deux opérateurs, notamment les différents schémas de composition possibles.

4.3.1 Opération de composition séquentielle

L'opérateur de composition séquentielle (`»`) permet d'appliquer une politique (filtre, action ou une composition) sur le résultat retourné par une autre politique qui la précède dans la chaîne de composition. Ainsi, P_1 » P_2 signifiera que la politique P_2 sera appliquée sur le résultat (ensemble de paquets) retourné par la politique P_1. Aussi, l'ordre d'apparition des politiques a une incidence forte dans le cas d'une composition séquentielle et, de ce fait, les règles suivantes sont à prendre en considération :

- La composition séquentielle de deux filtres correspond à une opération d'intersection entre les deux ensembles de paquets qui sont retournés par les deux filtres.
- Un composant virtuel ne peut pas appliquer une action sur un paquet qu'il n'a pas encore réceptionné, qu'il a déjà commuté ou rejeté. En d'autres termes, il n'est pas

permis pour les edges de composer séquentiellement des actions de type `forward` et `drop`. De même, il est impossible pour une fabric de composer des actions de type `carry`.

- Une politique (filtre ou action) ne retournera aucun résultat si elle est appliquée sur un ensemble vide.
- Si plusieurs actions de type `tag` sont spécifiées, le résultat final (l'ensemble de paquets) aura pour étiquette celle de la dernière action `tag` qui est dans la chaine de composition séquentielle.
- Il est tout à fait possible d'appliquer une suite d'actions `modify`, cela correspondra à des mises à jour successives qui seront appliquées à chaque fois sur l'ensemble de paquets retourné.

4.3.2 Opération de composition parallèle

L'opérateur de composition parallèle (+) permet de donner l'illusion que deux ou plusieurs actions sont exécutées au même moment sur un même ensemble de paquets. Plus précisément, P_1 + P_2 signifie que les deux politique P_1 et P_2 seront exécutées mais que l'ordre d'application des actions est supposé n'avoir aucune incidence.

L'utilité de l'opérateur de composition parallèle est de permettre aux administrateurs de composer les primitives du langage AirNet ou leurs modules de contrôle sans avoir à se soucier des problématiques d'intersection qui peuvent exister entre eux. En effet, comme nous avons pu le voir, les actions sont toujours exécutées sur des ensembles de paquets retournés par des filtres. Cependant, des intersections peuvent exister entre les différents filtres définis par l'administrateur. Comme exemple concret, on peut considérer l'extrait de programme ci-dessous qui contient deux règles, une pour le relayage des paquets web (`e1`) et la deuxième (`e2`) pour le relayage des paquets provenant de l'adresse IP `1.2.3.4` en direction du serveurs web 2 (`dst="WS2"`).

```
def exemple_composition():
 e1 = match(Edge="IO", nw_src="1.2.3.4") >> forward("WS1")
 e2 = match(Edge="IO", tp_dst=80, dst=WS2) >> modify(dst="WS1") >>forward("WS1")
 return e1 + e2
```

Dans cet exemple, il existe une intersection entre les deux règles `e1` et `e2` correspondant au flux web provenant de l'adresse IP `1.2.3.4` et en direction du serveur web 2. Le fait d'utiliser l'opérateur "+" est ainsi un moyen de gérer toutes les éventuelles intersections.

Par ailleurs, comme pour l'opérateur de composition séquentielle, il existe des règles d'utilisation pour l'opérateur de composition parallèle que nous énumérons ci-dessous :

- La composition parallèle de deux filtres correspond à une opération d'union des deux ensembles de paquets qui sont retournés par les deux filtres.
- Dans le cas d'une composition parallèle entre une action `forward` et une action `drop`, l'action `drop` est plus prioritaire.
- Plusieurs actions `forward` ne peuvent être composées en parallèle, de même pour les actions `tag`.

- Si plusieurs actions `modify` sont spécifiées, elles seront toutes appliquées, mais l'ordre de leur application n'est pas garantie.

Par ailleurs, il est important de souligner que l'opérateur de composition parallèle permet de résoudre les problématiques d'intersection entre les règles et non pas les conflits. En effet, s'il y a un conflit sémantique (par exemple, commutation vers des destinations différentes) entre deux règles, alors qu'une intersection existe entre eux, AirNet ne pourra pas prendre une décision et va seulement afficher un message d'erreur de compilation pour prévenir l'administrateur. En effet, nous considérons les conflits sémantiques entre règles comme un problème de haut niveau qui doit être géré par l'administrateur lui même au moment de la spécification de ses politiques.

4.4 Fonctions réseaux

Jusqu'à ce point, nous avons uniquement présenté le mode de fonctionnement statique d'AirNet. Ce dernier permet de construire des politiques de contrôle qui seront déployées au niveau du plan de données sur des équipements réseau. En utilisant ce mode de fonctionnement, un administrateur est capable de modéliser une multitude de cas d'utilisation allant d'une simple politique de routage jusqu'au contrôle d'accès. Néanmoins, ce mode de fonctionnement a des limites qui sont principalement dues au fait qu'il manipule, in fine, uniquement les instructions autorisées dans la table de flux du commutateur. Ainsi, une utilisation exclusive d'une approche statique suppose que :

- l'administrateur a une connaissance précise (sources et destinations, protocoles de transport et applicatif, etc.) de tous les flux présents et futurs qui vont passer par son réseau, ainsi que des services qui devraient leur être fournis tout au long de leur transport.
- les opérations des commutateurs OpenFlow couvrent tous les besoins de l'administrateur. Ainsi, en utilisant les actions `forward`, `drop` et `modify` l'administrateur devrait être capable de modéliser ses différents services réseaux, que ce soit du simple transport ou des services plus riches et complexes.
- l'administrateur n'a pas besoin d'accéder à l'état et à l'historique du réseau afin de spécifier ses politiques, ni à des informations externes telles qu'une base de données d'utilisateurs.

Malheureusement, ces suppositions sont rarement satisfaites dans la réalité, principalement à cause de la nature complexe et dynamique d'une infrastructure physique. En effet, dans beaucoup de cas, un administrateur n'a pas la connaissance précise de tous les flux qui passent ou qui passeront par son réseau. Par exemple, dans le cas où des nouveaux types de flux apparaissent, une politique statique ne sera pas capable de les prendre en charge. De plus, les capacités des commutateurs OpenFlow ne permettent pas d'implémenter tous les services réseaux dont un administrateur pourrait avoir besoin à l'exemple d'une décision de relayage basée sur l'inspection de paquets en profondeur ou encore une opération de transcodage de flux audio. Enfin, un administrateur a toujours

besoin d'avoir accès à des informations concernant l'état et l'utilisation des ressources de son réseau tel le taux d'utilisation d'un lien. En effet, ces informations sont primordiales, notamment pour des services tels que l'équilibrage de charge.

Pour répondre à ces besoins, AirNet inclut des primitives spécifiques qui permettent, d'une part, de capturer ce côté dynamique des infrastructures SDN et, d'autre part, de pouvoir appliquer des opérations complexes sur les données que transportent les flux de paquets. Ces primitives sont divisées en deux principaux groupes : les fonctions de contrôle dynamique et les fonctions de données. Dans ce qui suit, nous allons commencer par la présentation des fonctions de contrôle dynamique, puis nous passerons aux fonctions de données.

4.4.1 Fonctions de contrôle dynamique

Les fonctions de contrôle dynamique implémentent un processus de prise de décision en cours d'exécution (*at runtime*). En effet, ces fonctions permettent aux administrateurs de remonter des informations à partir de l'infrastructure physique, de les analyser, puis de générer des nouvelles politiques qui seront installées sur l'infrastructure physique. Par ailleurs, il est à noter que les fonctions de contrôle dynamique sont exécutées au niveau de l'hyperviseur d'AirNet (qui est lui même exécuté au-dessus d'un contrôleur SDN) et non pas au niveau du plan de données. Dans le chapitre 5 nous allons revenir plus en détails sur l'exécution et l'implémentation de ces fonctions de contrôle dynamique.

Pour mettre en œuvre ce concept, AirNet fournit une primitive sous la forme d'une annotation (patron de conception «décorateur»). Cette dernière transforme une simple fonction en une politique dynamique qui peut être composée et installée sur les contrôleurs SDN. La figure 30 expose les deux structures possibles pour un décorateur de fonction dynamique, suivant si l'administrateur souhaite remonter des paquets réseau complets (`data="packet"`) ou uniquement des statistiques (`data="stat"`) sur un flux.

```
@DynamicControlFct(data="packet", limit=int, split=[h])
@DynamicControlFct(data="stat", every=int, split=[h])
```

FIGURE 30. Décorateur d'une fonction de contrôle dynamique

4.4.1.1 Collecte des paquets

Dans le cas où l'administrateur souhaite remonter un paquet réseau complet, il doit aussi spécifier deux autres paramètres. Le premier est `limit` et il permet d'indiquer le nombre de paquets qu'on souhaite remonter vers la fonction réseau. Le deuxième, quant à lui, est `split` et il permet de discriminer les différents paquets qui sont envoyés vers la fonction réseau. Le paramètre `split` représente concrètement une liste de champs d'en-tête (par exemple `split=["nw_src","tp_dst"]`) qui est utilisée par l'hyperviseur d'AirNet comme une clé afin d'identifier les sous-flux de paquets sur lesquels le paramètre `limit` s'applique. Si la valeur du paramètre `split` est positionnée à `None`, cela signifie que le paramètre `limit` comptabilisera tous les paquets qui seront remontés sans aucune discrimination entre eux. De même, si le

paramètre `limit` est positionné à `None`, cela équivaut à dire que tous les paquets du flux doivent passer par cette fonction et qu'aucune limite n'est fixée.

Afin de mieux appréhender le concept de fonction dynamique, nous allons considérer un exemple qui consiste en la définition d'un service de contrôle d'accès dynamique. Pour ce faire, nous allons réutiliser la topologie virtuelle illustrée à la figure 28.

```
@DynamicControlFct(data="packet", limit=1, split=["nw_src"])
def authenticate(packet):
    ip = packet.find('ipv4')
    hostIP = ip.srcip.toStr()
    if whitelist.has_key(hostIP):
        fwd_policy = (match(edge='IO', nw_src=hostIP, tp_dst=80) >>
                      tag("authFlows") >> forward("Fab"))
        return fwd_policy
    else:
        reject_policy = (match(edge='IO', nw_src=hostIP) >> drop())
        return reject_policy
```

L'extrait ci-dessus représente la définition d'une fonction de contrôle dynamique, sachant que chaque définition inclut trois parties : le décorateur, le traitement effectué à l'intérieur de la fonction et en dernier la nouvelle politique qui est retournée. Ainsi, pour la partie décorateur, nous avons spécifié ces trois paramètres :

- `data='packet'` : on souhaite que l'hyperviseur remonte des paquets réseau entiers et non pas des statistiques.
- `limit=1` : on est intéressé uniquement par le premier paquet de chaque flux.
- `split=["nw_src"]` : on veut que l'hyperviseur applique une discrimination entre les paquets remontés en se basant sur leur adresse IP source.

En conséquence, le décorateur ci-dessus signifie que la fonction `authenticate` va être appelée pour le premier paquet provenant de chaque nouvelle adresse IP source. Par la suite, la fonction `authenticate` teste si l'adresse source de ce paquet est autorisée ou non (elle appartient à une liste blanche ou non). Si oui, une nouvelle politique de relayage est renvoyée pour cette nouvelle source (`fwd_policy`), sinon, une politique qui va bloquer les paquets de cette source est renvoyée (`reject_policy`).

La définition de la fonction dynamique étant terminée, on peut maintenant la composer avec d'autres politiques d'AirNet, comme cela est montré dans l'extrait de programme ci-dessous où on souhaite contrôler l'accès de tous les clients qui veulent accéder au serveur web :

```
match(edge='IO', nw_dst=PUBLIC_IP, tp_dst=HTTP) >> authenticate()
```

4.4.1.2 Collecte des statistiques

Les fonctions de contrôle dynamique permettent aussi de remonter des statistiques sur les flux traités. Pour ce faire, l'administrateur doit positionner la valeur du paramètre du décorateur `data` à `stat`. Par ailleurs, dans le cas des statistiques il n'existe pas de paramètre `limit`, celui-là étant remplacé par un paramètre `every` qui permet de définir la période à laquelle l'hyperviseur doit récupérer ces statistiques. Ces statistiques contiennent

actuellement des informations sur le nombre et la taille des paquets traités par une règle particulière.

A titre d'exemple, nous allons considérer la politique suivante dont l'objectif est de surveiller la consommation de données des utilisateurs du réseau `Net.A`. Pour ce faire, nous utilisons une fonction dynamique appelée `checkDataCap` qui vérifie que chaque hôte ne peut télécharger (réceptionner) au maximum que 10 méga octets toutes les trente secondes (seuil spécifié dans le programme ci-dessous par la variable `data_threshold`). S'il dépasse ce quota, ses communications seront bloquées pour les prochaines trente secondes, puis il sera à nouveau autorisé à communiquer. Les détails d'implémentation de cette fonction sont donnés dans l'extrait de programme ci-dessous :

```
@DynamicControlFct(data="stat", every=30, split=["nw_dst"])
def checkDataCap(stat):
    if stat.nw_dst in hosts_permission.keys():
        data_amount = stat.byte_count - hosts_data_amount[stat.nw_dst]
        hosts_data_amount[stat.nw_dst] = stat.byte_count
        if hosts_permission[stat.nw_dst] == "allow":
            if data_amount > data_threshold:
                hosts_permission[stat.nw_dst] = "deny"
                return (match(edge="IO", nw_dst=stat.nw_dst) >> drop)
        else:
            hosts_permission[stat.nw_dst] = "allow"
            return (match(edge="IO", nw_dst=stat.nw_dst) >> forward("Net.A"))
    else:
     hosts_data_amount[stat.nw_dst] = stat.byte_count
        if stat.byte_count > data_threshold:
            hosts_permission[stat.nw_dst] = "deny"
            return (match(edge="IO", nw_dst=stat.nw_dst) >> drop)
        else:
            hosts_permission[stat.nw_dst] ="allow"
```

Considérant les paramètres qui ont été passés au décorateur de la fonction `checkDataCap`, cette dernière va recevoir toutes les trente secondes des statistiques pour chaque destination distincte du flux de paquets qui est redirigé vers cette fonction (`every=30` et `split=["nw_dst"]`). Dans un premier temps, la fonction `checkDataCap` teste si la destination est déjà identifiée (`stat.nw_dst in hosts_permission.keys()`), si non, elle initialise ses paramètres de consommation, si oui, elle commence à tester si la consommation de la source dépasse la limite fixée ou non. Dans le cas ou l'hôte a dépassée la limite fixée, une politique qui rejette (`drop`) les communications de l'hôte en question est installée, sinon aucune modification n'est effectuée sur la politique de contrôle globale.

Afin de remonter uniquement des statistiques sur la quantité de données que consomme les utilisateurs du réseau `Net.A`, nous composons cette fonction dynamique avec un filtre AirNet, comme cela est montré dans l'extrait ci-dessous. Aussi, nous avons composé, de manière parallèle, une action de commutation (`forward`) avec la fonction dynamique afin d'assurer le relayage des paquets tout en implémentant une fonction de surveillance de consommation pour tous les hôtes du réseau `Net.A`.

```
match(edge=IO, dst="Net.A")  >> (checkDataCap() + forward("Net.A"))
```

4.4.2 Fonctions de données

Les fonctions de contrôle dynamique retournent toujours à la fin de leur exécution une nouvelle politique qui sera installée sur l'infrastructure physique. AirNet inclut aussi un autre type de fonction réseau qui permet l'inspection et la modification en profondeur des paquets sans générer une nouvelle politique de contrôle. Dans ce qui suit, nous allons présenter les deux approches par lesquelles AirNet autorise l'application d'un traitement complexe sur les données transportées par un paquet réseau.

4.4.2.1 Fonctions de données au sein du plan de contrôle

Suivant le même patron de conception que pour les fonctions de contrôle, AirNet inclut un décorateur qui permet de transformer une fonction existante en une fonction de données. La différence principale entre ces deux fonctions réseau est que les fonctions de contrôle dynamique retournent toujours une politique à installer sur l'infrastructure physique, alors que les fonctions de données retournent toujours un paquet, après lui avoir éventuellement appliqué un traitement. La figure 31 ci-dessous donne la structure d'un décorateur pour définir une fonction de données.

```
@DataFct(limit=int, split=[h])
```

FIGURE 31. Décorateur d'une fonction données

Comme on peut le voir sur la figure 31, la structure de ce décorateur est assez semblable à celle du décorateur pour les fonctions de contrôle si ce n'est qu'il est inutile de spécifier le type de données à remonter (statistique ou paquet) car dans ce cas, il sera toujours de type "paquet". Comme exemple d'utilisation, nous allons considérer l'extrait de programme ci-dessous qui décrit une fonction de compression.

```
from tools import cpr
@DataFct(limit=None, split=None)
def compress(packet):
 cpr(packet)
 return packet
```

Dans cet exemple, la fonction `compress` reçoit toujours en entrée un paquet, applique un traitement sur ce paquet, puis retourne le paquet modifié comme résultat. En effet, nous avons défini une simple fonction (`cpr`) de test pour simuler une compression de la charge utile du paquet, mais l'administrateur a la liberté d'intégrer ses propres fonctions, tant que ces dernières retournent à la fin un paquet.

Par ailleurs, comme pour les fonctions de contrôle dynamique, les fonctions de données doivent être elles aussi composées avec des filtres afin de recevoir des paquets à partir de l'infrastructure physique, comme cela est montré dans l'extrait de programme ci-dessous.

```
match(edge=IO, nw_dst=PUBLIC_IP) >> tag("inWebFlows") >>
                                    compress() >>  forward("Fab")
```

En somme, avec les fonctions de données, un administrateur a la possibilité d'implémenter, **au niveau des contrôleurs SDN**, des traitements complexes sur les données

que transportent les paquets, traitements qui ne peuvent être exécutés sur les commutateurs OpenFlow en utilisant leur jeu d'instructions.

4.4.2.2 Fonctions de données au sein du plan de données

Aujourd'hui, beaucoup d'entreprise ont recours à des *middleboxes* afin d'implémenter leurs services réseaux complexes et cela dans différents contextes d'utilisation (LAN, WAN, centre de calculs, etc.). En effet, un des principaux avantages des *middleboxes* est qu'elles peuvent être implémentées sur différents types d'équipements : matériels dédiés, serveurs dédiés ou même sur des machines virtuelles. Par ailleurs, il est important de noter que ces *middleboxes* ne sont pas des serveurs d'applications d'extrémité, étant donné qu'elles ciblent des fonctionnalités sur les flux réseaux (en amont de la destination finale).

L'apparition du paradigme SDN a permis de déployer ces *middleboxes* dans des emplacements arbitraires et dans certains cas, ces fonctions peuvent être implémentées sur les équipements OpenFlow eux-mêmes ; comme on l'a vu précédemment avec les exemples d'équilibrage de charge ou du contrôle d'accès. Néanmoins, le jeu d'instructions (`match`, `forward`, `modify`, etc.) qu'expose les commutateurs OpenFlow est restreint et ne permet pas d'implémenter toutes les fonctionnalités nécessaires telles que l'inspection et la modification des paquets en profondeur. Tout cela fait qu'il existe toujours de nombreuse *middleboxes* dans le réseau qui hébergent des fonctions complexes sur les données (filtrage applicatif, cache, chiffrement, transcodage, etc.) [Gember et al., 2012].

Par ailleurs, déployer toutes les fonctions de données au niveau du contrôleur SDN peut potentiellement causer des problèmes de passage à l'échelle et de performance. En effet, le contrôleur SDN, particulièrement s'il est totalement centralisé, peut devenir un goulot d'étranglement dans le cas de large réseaux où plusieurs fonctions de données doivent être appliquées sur des flux de données volumineux. Ainsi, l'utilisation des *middleboxes* peut être une solution pour décharger le contrôleur d'une partie des traitements qui doivent être effectués sur les données transportées par les paquets réseau.

Afin d'offrir une solution à ces problématiques, AirNet inclut la primitive `via` qui est implémentée au niveau des fabrics et qui permet de spécifier qu'un flux de paquets doit passer par une ou plusieurs *middleboxes*, facilitant ainsi le chaînage et le déploiement de nouveaux services réseaux. Dans AirNet, nous faisons référence à ces boites noires par le terme de `data machines`. En effet, dans le modèle de communication que nous proposons, à travers AirNet et son modèle d'abstraction, les data machines sont des edges spécialisés qui implémentent des fonctions complexes sur les données que transportent les paquets. Aussi, nous avons fait les suppositions suivantes concernant les data machines :

- Les fonctions qu'embarquent les data machines reçoivent toujours en entrée un paquet et retournent en sortie un ou plusieurs paquets.
- Les opérations effectuées par les data machines n'ont aucune incidence sur les politiques de contrôle. En d'autres termes, ces fonctions peuvent uniquement avoir accès et modifier l'état local d'une data machine.
- Les fonctions des data machine n'ont pas le droit de modifier les en-têtes des paquets.

Cette contrainte est motivée, d'une part, par le fait que c'est une opération qui peut être parfaitement réalisée par les commutateurs OpenFlow standard et, d'autre part, modifier les en-têtes peut avoir par la suite une incidence sur la manière avec laquelle les paquets sont transportés et le type de services qui leurs sont fournis, modifiant ainsi la politique globale de contrôle.

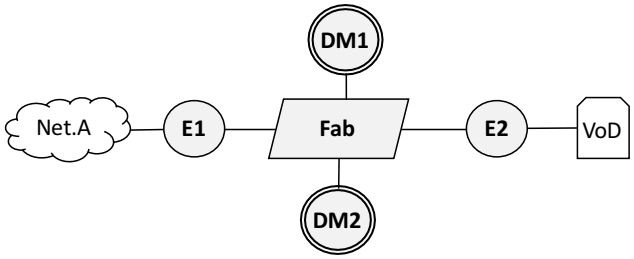

FIGURE 32. Topologie virtuelle avec deux data machines

Plus concrètement, nous allons considérer l'extrait de programme suivant qui résume bien l'utilisation des data machines au sein d'une topologie virtuelle. Cette dernière étant illustrée à la figure 32.

```
def transport_via_dataMachines():
 t1 = catch(fabric="Fab", src="E1", flow="inFlows") >>
        via("dm1") >> via("dm2") >>
        carry(dst="E2")
 t1 = catch(fabric="Fab", src="E2", flow="outFlows") >> carry(dst="E1")
 return t1 + t2
```

Ainsi, comme on peut le voir sur l'exemple ci-dessus, la primitive `via`, qui est implémentée au sein de la fabric, permet de rediriger un flux de paquets vers une ou plusieurs data machines qui sont présentes dans le réseau, dans ce cas les data machines `dm1` et `dm2`. Ainsi, en utilisant l'opérateur de composition séquentielle, un administrateur est capable de spécifier une chaîne de services, avec une relation d'ordre stricte, par laquelle un flux doit passer. L'hyperviseur d'AirNet se chargera par la suite d'installer les chemins nécessaires pour amener un flux vers une data machine, puis le récupérer pour le transporter vers un edge destination ou une autre data machine.

4.5 Conclusion

Dans ce chapitre nous avons présenté le langage AirNet dont la particularité est d'être basé sur le modèle d'abstraction Edge-Fabric. Cette caractéristique permet à AirNet de faire une claire distinction entre les politiques de transport et les services réseaux plus complexes. AirNet a un modèle de programmation qui consiste à d'abord spécifier une topologie virtuelle qui correspond à des besoins de conception de haut niveau ou à des contraintes physiques existantes, puis de spécifier les politiques de contrôle qui vont s'exécuter au-dessus de cette topologie. Ces politiques sont par ailleurs divisées en quatre principaux types : politiques de

transport, politiques de contrôle statiques, politiques de contrôle dynamiques et politiques sur des données.

Dans le chapitre suivant, nous allons présenter la mise on œuvre du langage AirNet, ou plus concrètement comment ces quatre types de politiques sont implémentés au sein de l'hyperviseur et quels sont les algorithmes utilisés pour assurer leur déploiement automatique sur l'infrastructure physique.

Chapitre 5
L'hyperviseur d'AirNet

AirNet a été implémenté en tant que langage dédié, embarqué dans Python. AirNet dispose d'un système d'exécution qui permet d'assurer la composition et l'installation des politiques virtuelles sur l'infrastructure physique. Actuellement, le système d'exécution d'AirNet, que nous nommons *Hyperviseur d'AirNet*, est au stade de prototype. Néanmoins, il a été testé avec succès sur de nombreux cas d'utilisations (dont certains sont présentés dans ce manuscrit).

Dans ce qui suit, nous allons d'abord présenter l'architecture générale de l'hyperviseur ainsi que les différents modules qui le composent. Puis nous présenterons les deux modes de fonctionnement de cet hyperviseur (mode proactif et mode réactif) ainsi que les grandes lignes des algorithmes qu'ils implémentent.

5.1 Architecture générale

La figure 33 présente une vue globale de l'architecture de l'hyperviseur. Nous pouvons voir que cette architecture inclut quatre principaux niveaux : *i*) le programme de contrôle défini par l'administrateur qui inclut la topologie virtuelle, les politiques de contrôle et le module de mapping *ii*) l'hyperviseur en lui-même, *iii*) le contrôleur SDN sur lequel est exécuté l'hyperviseur et *iv)* l'infrastructure physique qui comprend les équipements d'extrémité, les *middleboxes* et les commutateurs OpenFlow.

Dans les sections suivantes, nous allons nous intéresser de plus près aux différents modules qui composent l'hyperviseur. Ce dernier inclut notamment six modules essentiels, tous codés en Python :

- **infrastructure** : centralise toutes les informations concernant l'infrastructure physique.
- **language** : contient les classes et les structures de données qui implémentent les primitives du langage AirNet.
- **classifier** : implémente des structures logiques qui stockent les règles de contrôle selon leur ordre de priorité.
- **proxy ARP** : résout les problématiques liées au protocole ARP.
- **client** : module d'intégration qui permet à l'hyperviseur de communiquer avec le contrôleur SDN.

- **runtime** : implémente les algorithmes de compilation et de mapping.

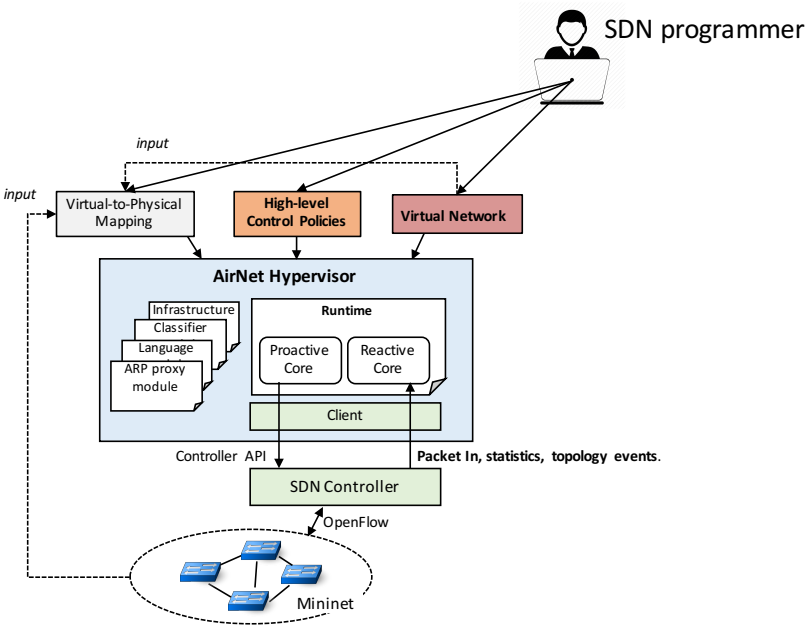

FIGURE 33. Architecture générale de l'hyperviseur d'AirNet

5.2 Module "infrastructure"

Le module *infrastructure* est un composant majeur de l'hyperviseur, étant donné que sa tâche est de maintenir une vision globale, cohérente et à jour de la topologie physique. Dans un premier temps, le module infrastructure envoie une série de requêtes au contrôleur SDN afin de récupérer des informations sur tous les commutateurs, hôtes et liens qui constituent la topologie physique. Les données récupérées comprennent, principalement, des informations sur la configuration des commutateurs (nombre de ports, identifiant), des hôtes (ports, adresses IP et MAC) et des liens réseaux (extrémités, capacités).

Le module infrastructure utilise toutes ces données pour construire un graphe qui embarque des algorithmes de parcours et de recherche. Cette structure sera par la suite utilisée par tous les autres modules de l'hyperviseur afin d'accéder à une vue globale sur la topologie physique. Par exemple, comme nous le verrons plus loin, le module `Proxy_ARP` utilise cette vue globale pour trouver les adresses MAC nécessaires à son service de résolution d'adresse IP.

Par ailleurs, afin de maintenir, durant toute la phase d'exécution, une vision cohérente de la topologie physique, le module infrastructure s'abonne à un ensemble d'évènements qui sont émis par le contrôleur SDN, dont notamment les suivants :

- `SwitchJoin` : Évènement qui indique qu'un nouveau commutateur OpenFlow s'est connecté au contrôleur SDN. En d'autres termes, un nouveau commutateur a rejoint la topologie physique.

- `SwitchLeave` : Évènement qui indique que le contrôleur SDN a perdu la connexion avec un commutateur OpenFlow.
- `LinkEvent` : Évènement par rapport à l'état d'un lien. Cet évènement peut correspondre à la détection d'un nouveau lien ou à la disparition d'un lien existant.
- `HostEvent` : Évènement par rapport à l'état d'un hôte (une machine terminale). Cet évènement peut indiquer la détection d'un nouveau hôte, la disparition ou le changement d'emplacement d'un hôte existant (n'est plus connecté au même commutateur).

Ainsi, à la réception d'un de ces évènements, le module infrastructure, dans un premier temps, exécute des fonctions qui permettent de mettre à jour la topologie logique qu'il maintient. Puis, dans un second temps, il envoie une notification vers le module *runtime* afin que ce dernier puisse ajuster les règles de contrôle installées de sorte qu'elles soient cohérentes avec la nouvelle configuration de l'infrastructure physique.

5.3 Module "mapping"

En plus de la définition de la topologie virtuelle, l'administrateur doit aussi fournir à l'hyperviseur un module de mapping entre les composants virtuels et les équipements physiques. L'hyperviseur utilise ce module afin de générer une configuration pour l'infrastructure physique qui soit sémantiquement équivalente à la politique virtuelle. D'autre part, le fait d'utiliser un module de mapping indépendant permet aux administrateurs de réutiliser leurs topologies virtuelles et leurs politiques de contrôle sur différentes infrastructures physiques sans nécessiter des modifications en dehors des informations de mapping.

Nous avons implémenté le module de mapping comme un ensemble de dictionnaires liant chaque unité virtuelle (edge, fabric, data machine, host et network) à des éléments de l'architecture physique. L'hyperviseur supporte plusieurs schémas de mapping entre équipements physiques et composants virtuels : *1)* un-à-un ; *2)* un-à-plusieurs ; *3)* plusieurs-à-un ; *4)* plusieurs-à-plusieurs.

Les dictionnaires du module de mapping sont construits en utilisant les primitives d'AirNet qui sont fournies à cet égard, ces dernières sont présentées dans la figure 34.

```
addEdgeMap("id_edge", [liste commutateurs physiques])
addFabricMap("id_fabric", [liste commutateurs physiques])
addDataMachineMap("id_dm", dl_addr | nw_addr)
addHostMap("id_dm", dl_addr | nw_addr)
addNetworkMap("id_net",  nw_addr/prefix)
```

FIGURE 34. Primitives de mapping

Ainsi, associer un edge à un ensemble de commutateurs physiques ou un hôte à une adresse IP sont des exemples de règles de mapping, comme on peut le voir sur l'extrait de programme ci-dessous qui représente le module de mapping utilisé entre les topologies physique et virtuelle de la figure 24 (chapitre 4).

```
class Mymapping(Mapping):
    def __init__(self):
        Mapping.__init__(self)
        self.addEdgeMap("E1", "s1", "s2")
        self.addEdgeMap("E2", "s8")
        self.addFabricMap("Fab", "s3", "s4", "s5" ,"s6" ,"s7")
        self.addHostMap("H1", "10.10.0.11")
        self.addHostMap("H2", "10.10.0.12")
        self.addNetworkMap("Net.A", "192.168.10.0/24")
        self.addNetworkMap("Net.B", "192.168.11.0/24")

def main():
    return Mymapping()
```

5.4 Modules "language" et "classifier"

Le module *language* contient toutes les classes de base qui implémentent les primitives et les opérateurs du langage AirNet. En incluant ce module dans un programme Python, un administrateur aura accès aussi bien aux primitives qui permettent de construire des topologies virtuelles (`addHost`, `addLink`, etc.), qu'aux primitives qui permettent de spécifier des politiques de contrôles (`match`, `forward`, etc.).

Le module *language* inclut deux classes de base : *EdgePolicy* de laquelle héritent toutes les primitives des composants virtuels de type edge et *FabricPolicy* de laquelle héritent toutes les primitives des composants virtuels de type fabric. C'est notamment au niveau de ces deux classes que sont surchargés les deux opérateurs de composition séquentielle et parallèle. En effet, les classes du module langage permettent d'assurer un premier niveau de compilation des politiques que nous appelons *composition virtuelle*. Cette étape consiste à d'abord composer toutes les politiques d'un programme AirNet pour obtenir à la fin deux politiques de composition : une pour toutes les politiques de type edge et une deuxième pour les politiques de type fabric. Par la suite, ces deux politiques de composition seront compilées en plusieurs règles. Ces dernières étant des structures intermédiaires (`Rule`) qui représentent, avec un haut niveau d'abstraction, des règles de contrôle basées sur des flux, similaires à des règles OpenFlow. En effet, comme cela est montré ci-dessous, chaque règle inclut un filtre, un label et un ensemble d'actions à exécuter sur les paquets retournés par le filtre.

```
Rule(filtre, label, set([actions]))
```

Par la suite, chacune de ces règles est insérée, individuellement, dans un conteneur logique ordonné appelé *classifier*. Puis, ces conteneurs sont fusionnés deux à deux d'une manière cumulative, produisant à la fin un seul classifier avec toutes les règles de contrôle ordonnées par priorité. L'objectif de cette fusion cumulative est de résoudre les problématiques d'intersection qui peuvent exister entre les règles. En effet, à chaque fois que deux classifiers sont composés, les règles de ces deux classifiers sont comparées entre elles. Si une intersection existe entre deux règles de deux classifiers différents, une troisième est créée et est insérée dans le classifier résultant avec une priorité plus haute que les deux précédentes. La nouvelle règle créée aura un filtre qui correspondra à l'intersection des filtres des règles originales, plus une liste d'actions qui sera le résultat de la fusion des ensembles d'actions

des deux règles en intersection.

Pour mieux appréhender ces problématiques d'intersection entre les règles, nous allons considérer l'exemple ci-dessous où l'objectif est, d'une part, de commuter tous les paquets d'une source particulière (`IP addr = 1.2.3.4`) vers un hôte `H1` et, d'autre part, de faire passer tous les flux `SSH` par une fonction dynamique d'inspection en profondeur `dpi`.

```
def exempleIntersec()
 e1 = match(edge='AC', nw_src='1.2.3.4') >> tag("H1Flows") >> forward('H1')
 e2 = match(edge='AC', tp_dst=SSH) >> dpi()
 return e1 + e2
```

Ainsi, dans l'exemple ci-dessus, une intersection existe entre les règles `e1` et `e2` pour les flux de paquets SSH qui proviennent de l'adresse IP source `'1.2.3.4'`. En effet, les deux règles `e1` et `e2` sont applicables à ce flux. De ce fait, à l'arrivée d'un paquet de ce flux particulier, le commutateur sera dans l'incapacité de choisir laquelle des deux règles à appliquer. Pour répondre à ce problème d'intersection, l'hyperviseur d'AirNet construit une troisième règle d'une manière totalement transparente pour l'administrateur. Cette règle a un filtre qui correspond à l'intersection des filtres des deux règles `e1` et `e2`; et une liste d'actions qui consiste en la fusion des deux listes d'actions des deux règles de départ. Au final, comme exposé ci-dessous, l'hyperviseur va générer un classifier qui contient quatre règles classées par ordre de priorité (de la plus prioritaire à la moins prioritaire). La dernière règle correspond à une règle de sécurité par défaut qui est inserée par l'hyperviseur pour gérer tous les flux de paquets qui ne correspondent à aucune règle spécifiée par l'administrateur.

```
r1 = Rule(match(edge='AC', nw_src='1.2.3.4', tp_dst=SSH),
                "H1Flows", ([forward('H1'), dpi()]))
r2 = Rule(match(edge='AC', nw_src='1.2.3.4'), "H1Flows", ([forward('H1')]))
r3 = Rule(match(edge='AC', tp_dst=SSH), identity, ([dpi()]))
r4 = Rule(identity, identity, ([drop]))
```

5.5 Module "Proxy ARP"

L'hyperviseur d'AirNet dispose d'un module *Proxy_APR* qui permet de répondre aux requêtes ARP des hôtes du réseau. Pour ce faire, le module proxy ARP installe, à l'initialisation, sur tous les commutateurs de bordure des règles qui redirigent tous les paquets de type ARP vers l'hyperviseur d'AirNet.

A la réception d'une requête ARP, le module proxy ARP effectue une opération de résolution d'adresse IP afin de récupérer l'adresse MAC recherchée. Pour ce faire, le proxy ARP sollicite le module infrastructure, étant donné que ce dernier détient toutes les informations concernant la configuration des hôtes du réseau. Par la suite, le module proxy ARP construit un paquet de type *ARP reply* et le transmet au commutateur de bordure qui est connecté à l'hôte qui a envoyé la requête ARP.

Outre son service de résolution d'adresses, l'utilisation d'un proxy ARP a pour avantage de diminuer le nombre de diffusions dans le réseau (les requêtes ARP ne dépassant pas les premiers équipements de bordure). Nous profitons ici clairement des intérêts du paradigme SDN, notamment de la vision globale sur la topologie physique et de la configuration des équipements que le contrôleur détient.

5.6 Les clients POX et RYU

L'hyperviseur d'AirNet dispose d'un module d'intégration (module *client*) au travers duquel passent toutes ses communications avec le contrôleur SDN. La motivation principale derrière l'implémentation de ce module est de faire en sorte que l'hyperviseur soit le plus agnostique possible par rapport aux différents types et versions de contrôleurs existants. En effet, dans le cas où l'on souhaiterait intégrer un nouveau contrôleur, il suffirait alors seulement de modifier le module client afin de prendre en considération les spécificités de la nouvelle infrastructure sous-jacente.

Ainsi, la fonction principale du module client est de jouer le rôle de passerelle (un *Nexus*) entre l'hyperviseur et le contrôleur SDN. Pour ce faire, un client doit fournir deux principaux services, que nous citons ci-dessous :

- Remonter tous les évènements, statistiques et paquets dont l'hyperviseur a besoin.
- Construire à partir de structures intermédiaires (`Rule`) des règles OpenFlow et les transmettre au contrôleur cible, en s'appuyant sur l'API spécifique de ce dernier.

Comme preuve de concept, nous avons développé deux clients : un pour le contrôleur POX [POX, 2013] et un deuxième pour le contrôleur Ryu [Nippon Telegraph and Telephone Corporation, 2012]. POX est un contrôleur open source basé sur NOX [Gude et al., 2008] mais entièrement écrit en Python. Il a été principalement développé pour le monde académique, ce qui fait de lui un excellent outil de test et de prototypage. La figure 35 montre l'interfaçage de l'hyperviseur avec le contrôleur POX.

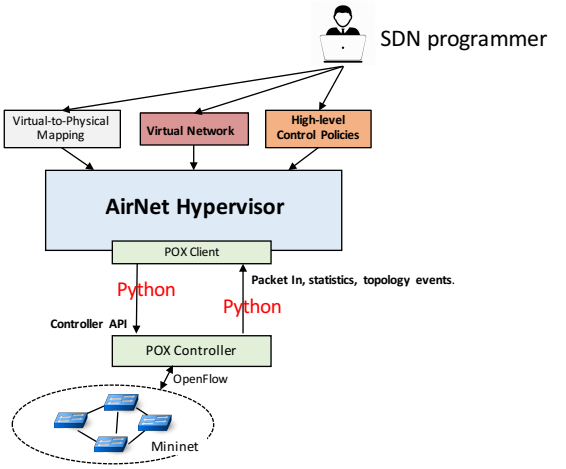

FIGURE 35. Architecture AirNet avec un client POX

Sur cette figure 35, il est important de constater que POX fournit uniquement une API Python pour s'interfacer. Cette dernière fournit un grand nombre des services que l'on attend d'un contrôleur SDN, notamment la découverte de la topologie et la configuration du plan de données. Cependant, elle présente une limite qui est qu'elle ne propose pas les appels de

fonctions à distance, ou du moins ce n'est pas implémenté d'une manière native. Même s'ils peuvent s'exécuter sur des fils d'exécution (*thread*) différents, l'hyperviseur et le contrôleur restent embarqués dans le même processus.

Ainsi, nous avons fait le choix de développer un deuxième client pour un contrôleur différent, et cela avec un double objectif :

- S'appuyer sur des mécanismes d'interactions réparties afin de dissocier les processus "contrôleur" et "hyperviseur", et ainsi avoir une approche plus modulaire.
- Montrer la portabilité de l'hyperviseur en l'exécutant sur un contrôleur différent.

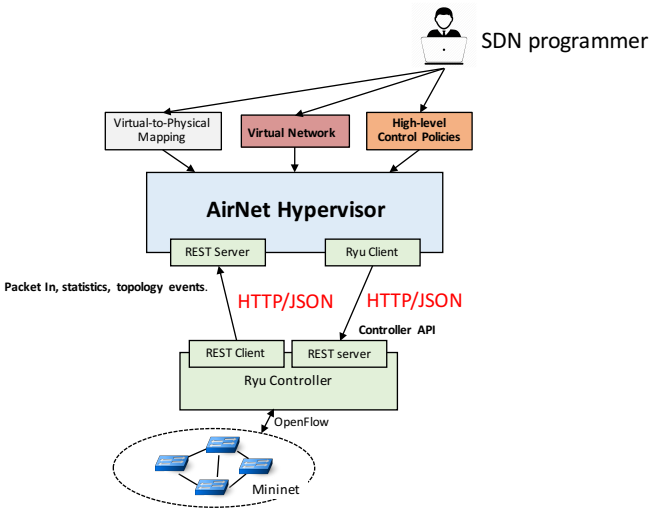

FIGURE 36. Architecture AirNet avec un client Ryu

La figure 36 illustre l'architecture de l'hyperviseur avec ce nouveau client. Nous pouvons voir dans cette figure que notre choix de contrôleur s'est porté sur Ryu. Ce dernier est lui aussi, comme POX, un contrôleur open source écrit en Python. Cependant, d'une part, c'est un contrôleur utilisé dans des contextes industriels opérationnels et, d'autre part, il offre beaucoup plus de fonctionnalités telles que le support des dernières versions du protocole OpenFlow (de la version 1.1 jusqu'à la version 1.5) ou l'intégration d'une API d'appels à distance REST, ce qui dans notre cas nous intéresse précisément. Les API REST suivent un modèle client/serveur et où aucun état ne doit être conservé sur le serveur entre chaque requête d'un client. Ces deux caractéristiques fondamentales d'une architecture REST nous permettent ainsi de séparer complètement AirNet et Ryu, facilitant ainsi la réutilisation de ce client sur d'autres contrôleurs qui disposent d'une API REST. Le passage à un autre contrôleur REST nécessiterait alors quelques mises à jour comme l'identifiant et la représentation des ressources.

5.7 Module "runtime"

Le module *runtime* est la pièce centrale de l'hyperviseur d'AirNet, étant donné que c'est au niveau de ce module que seront transformées les politiques virtuelles en règles physiques. Le

module *runtime* prend en entrée les résultats produits par les autres modules de l'hyperviseur (infrastructure, language et classifier), applique les algorithmes de compilation, puis génère un ensemble de règles physiques pour les équipements du plan de données. Ces règles sont passées par la suite à son client pour être transformées en règles OpenFlow et installées sur les commutateurs physiques.

Le module *runtime* inclut deux grandes parties (dites *core*) qui supportent l'exécution des deux principaux mode de fonctionnement d'un programme AirNet, à savoir le mode proactif et le mode réactif :

- Le *proactive core* (partie proactive) permet l'installation et l'initialisation des politiques statiques sur le plan de données.
- Le *reactive core* (partie réactive) traite des questions relatives à l'exécution des fonctions réseaux (contrôle et données) et les changements qui peuvent survenir dans la topologie physique.

Dans ce qui suit, nous présentons plus en détails le fonctionnement de chacun de ces deux modules, ainsi que les algorithmes qu'ils embarquent.

5.7.1 Proactive core

Le *proactive core* permet l'installation des politiques statiques dans le plan de données. Pour ce faire, ces politiques doivent passer par deux principales étapes de transformation :

- Composition virtuelle : les politiques sont composées entre elles afin de résoudre tous problèmes d'intersection.
- Mapping physique : les politiques de haut niveau sont transformées en règles physiques de bas niveau.

Dans ce qui suit, nous expliquons chacune de ces étapes de transformation. Pour ce faire, nous allons considérer l'exemple suivant qui décrit une simple politique d'acheminement de données. Cet exemple est exécuté sur les topologies physique et virtuelle qui sont illustrées à la figure 37.

```
i1 = match(edge="E1", tp_dst=80) >> tag("WebFlows") >>  forward("Fab")
t1 = catch(fabric="Fab", src="E1", flow="WebFlows")  >>  carry("E2")
i2 = match(edge="E2", dst="H1") >> forward("H1")
i3 = match(edge="E2", dst="H2") >> forward("H2")
```

5.7.1.1 Composition virtuelle

La composition virtuelle est une phase dont l'exécution est complètement indépendante de l'infrastructure physique. En effet, comme cela a été présenté auparavant, l'objectif principale de la phase de composition virtuelle est la résolution des intersections pouvant exister entre les différentes politiques de contrôle virtuelles. Dans cette première phase, le proactive core fait principalement appel aux fonctions définies dans les modules *language* et *classifier* afin de transformer les politiques de contrôle en un ensemble de règles intermédiaires et de les stocker, par ordre de priorité, dans un conteneur logique ordonné appelé *classifier* (cf. section 5.4).

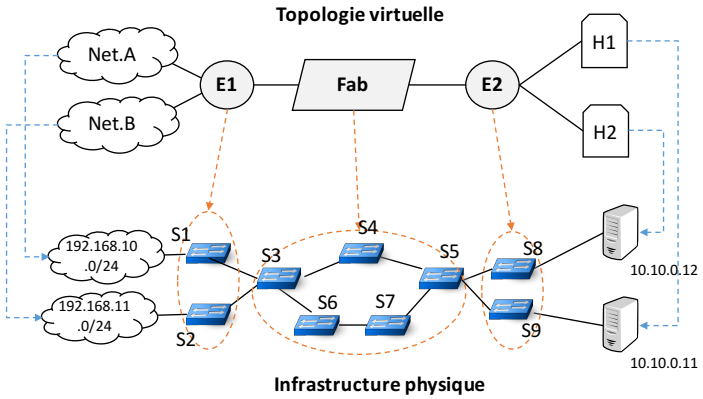

FIGURE 37. Implémentation : exemple de topologies physique et virtuelle

5.7.1.2 Mapping physique

La seconde phase consiste à compiler les règles stockées dans le classifier, en fonction de l'infrastructure physique cible et des instructions de mapping qui ont été définies par l'administrateur. Ainsi, à l'inverse de la phase de composition virtuelle, le résultat produit par la phase de mapping physique est très dépendant de la taille et de la complexité de l'infrastructure physique. Nous décrivons ci-dessous l'algorithme utilisé pour cette phase.

Pour chaque règle intermédiaire d'un edge, le *proactive core* va récupérer les commutateurs physiques associés à cet edge, puis la transforme en une ou plusieurs règles physiques. Cette transformation inclut par exemple le remplacement des identifiants symboliques par des paramètres physiques de bas niveau (adresses réseaux, port de sortie, etc.). Considérant cela, la première règle (`i1`) de l'exemple précédent sera transformée en deux règles physiques qui seront par la suite installées sur les deux commutateurs `s1` et `s2`, étant donné que l'edge `E1` mappe vers ces deux commutateurs physiques.

En ce qui concerne les règles de la fabric, l'algorithme est plus complexe, vu que le *proactive core* ne peut pas se baser uniquement sur la table de commutation d'une fabric. En effet, si nous reprenons notre exemple, la politique de transport `t1` indique que tous les flux web (`label == WebFlows`) qui proviennent de l'edge `E1` doivent être acheminés vers l'edge `E2`. Cependant, l'edge `E2` mappe vers deux commutateurs physiques, à savoir `s8` et `s9`. Ainsi, nous avons besoin de diviser les flux qui sont transportés vers un edge de sortie en plusieurs sous-flux en fonction de leur destination physique finale. En d'autres termes, nous devons acheminer vers les commutateurs de bordure uniquement les flux qui sont destinés à des hôtes terminaux qui sont directement connectés à eux. Pour ce faire, le proactive core commence par effectuer diverses opérations d'intersection entre les règles de bordure qui envoient des flux à l'intérieur du réseau (`ingress rules`) et les règles qui distribuent les paquets à la sortie du réseau (`egress rules`), puis installe des chemins pour les flux d'intersection obtenus.

Considérant cela, on peut constater que l'exemple de programme précédent inclut une

unique règle *ingress* (`i1`) et deux règles *egress* (`i2` et `i3`). L'intersection de ces règles va ainsi générer deux sous-flux, pour lesquels l'hyperviseur va installer les deux chemins suivants :

- `tp_dst=80 et dst=H1` : les paquets qui correspondent à ce filtre vont suivre le chemin `[s3, s4, s5, s9]`
- `tp_dst=80 et dst=H2` : les paquets qui correspondent à ce filtre vont suivre le chemin `[s3, s4, s5, s8]`

Concernant l'algorithme de parcours qui est utilisé au sein de la fabric pour sélectionner le meilleur chemin à suivre, la version actuelle de l'hyperviseur utilise l'algorithme de Dijkstra pour trouver le plus court chemin. C'est pour cette raison qu'au sein de la fabric le chemin `[s3, s4, s5]` est préféré au chemin `[s3, s6, s7, s5]`. Il est par ailleurs à noter que d'autres algorithmes peuvent être facilement rajoutés, la seule contrainte qu'impose l'hyperviseur est que cet algorithme doit prendre en entrée un objet `graph` qui est généré par le module infrastructure, un départ et une arrivée, puis de retourner à la fin un chemin qui correspond à une liste ordonnée de commutateurs.

5.7.2 Reactive core

Le *reactive core* permet de gérer toutes les interactions avec l'infrastructure physique pendant la phase d'exécution. Il assure deux principales fonctions : la gestion du cycle de vie des fonctions réseaux et l'adaptation des règles de contrôle en cas de changement au niveau de l'infrastructure physique.

5.7.2.1 Fonctions réseau au sein du plan de contrôle

Dans cette section, nous présentons, à travers un ensemble d'algorithmes écrits en Python, l'implémentation de la gestion des fonctions réseaux au sein de l'hyperviseur AirNet. Pour ce faire, nous décrivons les trois phases principales du cycle de vie d'une fonction réseau, à savoir : *i*) l'initialisation au moment du déploiement, *ii*) la collecte des paquets et des statistiques, *iii*) le paramètre `limit` atteint.

Première phase : initialisation La première phase a deux objectifs. Le premier consiste à installer les règles de commutation qui permettront d'envoyer des paquets et des statistiques vers le contrôleur. Le second objectif est de créer des structures de données nommées `buckets` (seaux) qui stockeront les données reçues par le contrôleur.

Comme nous pouvons le voir sur l'algorithme de la figure 38, à la compilation d'un programme AirNet, le module `runtime` commence par parcourir la liste (`net_function_rules`) qui contient toutes les règles qui incluent une ou plusieurs fonctions réseaux. Pour chacune de ces règles, une structure `bucket` est créée et lui est associée. Par la suite, cette structure va recevoir toutes les données relatives à cette règle. La deuxième étape de cette phase est l'installation d'une règle (`ctrl_rule`) qui permettra de remonter les paquets souhaités à partir de l'infrastructure physique. Pour ce faire, une nouvelle règle est créée à partir de la règle dynamique, en copiant notamment son filtre et son label. Pour les actions, elles sont toutes copiées sauf les actions de type fonction réseau et les actions de commutation. En effet, la

```
def init_net_functions():
 for net_rule in net_function_rules:
  create_bucket(net_rule)
  actions = set()
  for act in net_rule.actions:
   if (not isinstance(act, NetworkFunction)) and
      (not isinstance(act, forward)):
    actions.add(act)
  actions.add(forward('controller'))
  ctrl_rule = Rule(net_rule.match, net_rule.tag, actions)
  enforce_rule(ctrl_rule)
```

FIGURE 38. Algorithme première phase : initialisation

règle qui sera installée contiendra une seule action de commutation qui relaie les paquets vers l'hyperviseur.

Par ailleurs, dans le cas de statistiques, les étapes de la première phase sont identiques, si ce n'est qu'aucune règle de commutation vers le contrôleur SDN n'est installée. En effet, pour les statistiques, en plus de la structure `bucket`, un temporisateur (`Timer`) est initialisé avec la valeur du paramètre `every` de chaque règle. A chaque cycle, ce temporisateur va envoyer des messages OpenFlow (des requêtes) afin de récupérer des statistiques sur le nombre de paquets et d'octets traités par la règle installée. Une fois ces statistiques collectées, elles sont redirigées vers le `bucket` approprié.

Deuxième phase : collecte des données Après la phase d'initialisation, à chaque fois qu'un paquet (ou une statistique) est remonté vers le contrôleur, l'hyperviseur d'AirNet exécute l'algorithme qui est décrit dans la figure 39. Il commence par récupérer le `bucket` approprié dans lequel le paquet sera stocké. Pour ce faire, le `reactive core` teste si le filtre associé au `bucket` recouvre l'en-tête du paquet ou non. Si oui, le `bucket` va appeler la fonction réseau qui lui est associée et lui passer en paramètre le paquet. Dans le cas d'une fonction de données, le résultat retourné est un paquet modifié qui sera réinjecté dans le réseau et transporté à sa destination finale suivant les politiques existantes. Dans le cas d'une fonction de contrôle dynamique, le résultat consistera, dans un premier temps, en une politique qui sera compilée puis installée sur l'infrastructure physique, puis dans un second temps, le paquet original sera réinjecté dans le réseau et transporté suivant la nouvelle politique générée.

Si le paramètre `limit` est spécifié dans le décorateur de la fonction réseau, le `bucket` comptera alors le nombre de paquets entrants et appellera la fonction appropriée lorsque cette limite est atteinte (troisième phase, section suivante). Notons que si le paramètre `split` est également défini, alors il sera nécessaire d'avoir un compteur de paquets pour chaque micro-flux, qui est obtenu en appliquant l'argument `split` sur l'en-tête du paquet.

Troisième phase : limite atteinte Quand la limite d'un flux est atteinte, cela signifie que la fonction réseau ne doit plus être appliquée. Pour ce faire, le *reactive core* applique l'algorithme qui est illustré dans la figure 40. Il commence par chercher la règle qui est associée au flux, puis applique sur elle la fonction `remove_net_function` qui supprime la fonction réseau de

```
def process_packet_in(switch_id, packet_match, packet):
 bucket = get_bucket_covering(packet_match)
 bucket.apply_network_function(switch_id, packet)
 if bucket.split is None:
  bucket.nb_packets += 1
  if bucket.nb_packets == bucket.limit:
   flow_limit_reached(bucket.match)
 else:
  micro_flow = get_micro_flow(packet)
  try:
   bucket.nb_packets[micro_flow] += 1
  except KeyError:
   bucket.nb_packets[micro_flow] = 1
  if bucket.nb_packets[micro_flow] == limit:
   micro_flow_limit_reached(micro_flow)
```

FIGURE 39. Algorithme deuxième phase : collecte des paquets

cette règle, tout en laissant les autres actions de base. La prochaine étape consiste à générer des nouveaux classifiers et de les comparer avec l'ancienne configuration installée. Cela est effectué par la fonction `get_diff_lists` qui retourne à la fin de son exécution un dictionnaire de différences qui contient trois listes :

- `to_add` : contient l'ensemble des règles qui doivent être ajoutées.
- `to_delete` : contient les règles qui doivent être supprimées.
- `to_modify` : contient les règles qui doivent être modifiées, avec les modifications à appliquer.

Par la suite, ces listes sont transmises au client du module *runtime* qui se chargera d'appliquer ces modifications en installant, supprimant ou en modifiant des règles sur les tables de flux des commutateurs du plan de données.

```
def flow_limit_reached(flow):
 for rule in edge_policies.rules:
  if rule.match == flow:
   remove_net_function(rule)
 new_classifiers = to_physical_rules(edge_policies)
 updates = get_diff_lists(old_classifiers, new_classifiers)
 install_diff_lists(updates)
```

FIGURE 40. Algorithme troisième phase : limite atteinte

Par ailleurs, il est possible que la limite soit atteinte, mais uniquement pour un micro-flux et non pas pour tous les flux qui sont destinés à la fonction réseau. En effet, dans le cas où le paramètre `split` est spécifié, un compteur distinct est associé à chaque micro-flux obtenu en appliquant le paramètre `split` sur les en-têtes des paquets entrants. Dans ce cas, si le compteur d'un micro-flux atteint la limite fixée (qui est la même pour tous les micro-flux), l'hyperviseur exécute alors l'algorithme qui est décrit dans la figure 41 qui, au contraire de l'algorithme précédant, ne supprime pas la règle qui permet de rediriger des paquets vers la fonction réseau, mais au lieu de cela, il installe une nouvelle règle spécifique pour le micro

flux, qui ne contient pas la fonction réseau et qui est plus prioritaire que la règle originale. Ainsi, les paquets de ce micro-flux ne seront plus redirigés vers la fonction réseau, mais les paquets des autres micro-flux qui n'ont pas encore atteint leur limite le seront toujours.

```
def micro_flow_limit_reached(micro_flow):
 for rule in edge_policies.rules:
  if rule.match.covers(micro_flow):
   new_rule = copy.deepcopy(rule)
   new_rule.match = micro_flow
   remove_net_function(new_rule)
   enforce_rule(new_rule)
```

FIGURE 41. Algorithme troisième phase : limite d'un micro-flux atteinte

5.7.2.2 Fonctions de données au sein du plan de données

La gestion des `data machine` est un peu différente des fonctions réseaux qui sont exécutées au niveau du contrôleur SDN. En effet, dans ce cas, aucun paquet n'est remonté vers l'hyperviseur et par conséquent aucun `bucket` n'est créé. Pour implémenter des fonctions de données nous avons utilisé le routeur *Click* [Kohler et al., 2000]. Ce dernier fournit une architecture logicielle qui permet de construire des routeurs flexibles et programmables. En effet, un routeur click est composé de plusieurs modules appelés *elements*. Chaque *element* représente une unité de traitement d'un routeur qui implémente une fonction d'un routeur telle que la classification des paquets, des files d'attentes et de l'ordonnancement. Ainsi, pour implémenter un routeur, l'administrateur ne fait que choisir les éléments à utiliser et définit les connexions entre eux.

Ainsi, la fonction principale de l'hyperviseur est d'installer, pour chaque action `via`, un premier chemin qui transporte les paquets vers la data machine, puis d'installer un deuxième chemin qui les récupére à la sortie et les acheminer vers leur destination finale ou vers une autre data machine (plusieurs actions `via`), comme cela est résumé dans l'algorithme qui est illustré dans la figure 42.

```
for rule in via_rules:
 dm = rule.via_list.pop()
 for node in path:
  if node in dm.switches(dm):
   install_rule_to_dm(rule, dm)
   install_rule_from_dm(rule, dm, path)
   dm = rule.via_list.pop()
  else:
   install_next_hop_rule(rule, node, path)
```

FIGURE 42. Algorithme des data machines

Dans cet algorithme, pour chaque règle qui contient une action de type `via`, l'hyperviseur commence par extraire la première data machine par laquelle le flux de cette règle doit passer, puis il déploie cette règle tout le long du chemin. Pour chaque nœud du chemin, si le nœud correspond au commutateur connecté à la data machine, l'hyperviseur installe alors deux

règles : une qui envoie le flux vers la data machine et une autre qui le réceptionne à la sortie pour l'envoyer vers le nœud suivant. Dans le cas contraire (le nœud n'est pas un commutateur connecté à la data machine), l'hyperviseur installe uniquement une règle qui relaie le flux vers le nœud suivant dans le chemin.

5.7.2.3 Evénements de l'infrastructure physique

La deuxième fonctionnalité du *reactive core* est de gérer les changements de topologie qui peuvent survenir au niveau de l'infrastructure physique tels que la découverte d'un nouveau chemin ou la déconnexion d'un lien d'un commutateur. En effet, dès qu'un évènement de type topologie est reçu par le client de l'hyperviseur, ce dernier le transmet immédiatement au module `infrastructure` afin qu'il puisse mettre à jour les informations dont il dispose. Le module `infrastructure` se charge ensuite de notifier le module `runtime` qu'il y a eu un changement au niveau de l'infrastructure physique et de lui communiquer un nouveau objet `graph` qui modélise la nouvelle topologie.

Par la suite, le *reactive core* du module `runtime` actionne une procedure d'adaptation à la nouvelle infrastructure. Cette procedure contient trois grandes étapes :

- Recalculer les chemins : cette phase consiste à exécuter une nouvelle fois la phase de mapping physique afin de générer des nouveaux classifiers qui soient consistant avec la nouvelle infrastructure physique. Par ailleurs, il est à noter que puisque la phase de composition virtuelle est complètement dissociée de l'infrastructure physique, elle ne sera pas exécutée une nouvelle fois.
- Générer les dictionnaires de différences : après la génération des nouveaux classifier, le *reactive core* effectue une comparaison entre les nouveaux classifiers et les anciens afin d'identifier les potentielles différences entre eux. Cette étape donnera comme résultat un dictionnaire qui contient les règles à ajouter, à supprimer ou à modifier sur chaque commutateur physique, comme dans le cas de la troisième phase des fonctions réseaux (limite atteinte).
- Reconfigurer l'infrastructure physique : si des différences sont détectées, le *reactive core* les communique à son client afin que ce dernier puisse les mettre en œuvre sur l'infrastructure physique en générant les règles OpenFlow correspondantes.

5.8 Conclusion

Dans ce chapitre nous avons présenté la conception et l'implémentation de l'hyperviseur du langage AirNet. La fonction principale de cet hyperviseur est d'assurer la composition des politiques virtuelles et leur installation sur les commutateurs OpenFlow. Pour ce faire, l'hyperviseur utilise plusieurs modules qui lui permettent, d'une part, d'interagir avec l'infrastructure physique (*client*, *infrastructure*, *Proxy_ARP*) et, d'autre part, d'assurer la composition et le mapping des politiques (*language*, *classifier*, *runtime*).

Tout le développement de l'hyperviseur a été fait en Python et l'ensemble représente environ 5000 lignes de code. Ce prototype a été testé avec succès sur plusieurs cas d'utilisation (équilibrage de charge, authentification dynamique, limitation de bande

passante, reconfiguration suite à l'arrêt d'un commutateur, etc.). Dans le chapitre suivant, nous déroulons un scénario complet de la spécification des politiques de contrôle et du réseau virtuel jusqu'aux tests sur plusieurs infrastructures physiques exécutées dans un émulateur de réseaux.

Troisième partie

Mise en pratique et expérimentations d'AirNet

Chapitre 6
Cas d'étude général : conception et exécution d'un programme AirNet

Après avoir présenté le modèle d'abstraction Edge-Fabric et notre langage de programmation AirNet, dans ce chapitre nous présentons une mise en pratique de ce dernier à travers le déroulement d'un cas d'utilisation du début jusqu'à la fin pour illustrer les différentes étapes de la spécification jusqu'à l'exécution d'un programme AirNet.

Nous commençons par détailler le scénario que nous avons retenu pour illustrer l'utilisation du langage AirNet. Puis nous présentons les trois modules qui constituent ce cas d'étude, à savoir : la topologie virtuelle, le programme de contrôle et enfin les informations de mapping. En dernier, nous présentons les résultats d'exécution du programme sur deux topologies différentes afin d'en montrer sa réutilisabilité.

6.1 Description détaillée du scénario

Le contexte général de ce cas d'étude consiste en la configuration d'un réseau d'entreprise qui héberge des services applicatifs (serveurs web et bases de données) et qui offre des services réseau d'authentification et de répartition de charge.

L'infrastructure physique qui représente le réseau de l'entreprise est illustrée à la figure 43. Cette dernière comprend douze commutateurs OpenFlow, quatre d'entres eux sont des commutateurs de bordure (`s1`, `s2`, `s11` et `s12`), les autres sont des commutateurs de cœur de réseau. L'infrastructure physique inclut aussi deux réseaux d'utilisateurs : `Internal users` et `External users` qui représentent respectivement les employées de l'entreprise et les utilisateurs qui lui sont externes (les invités par exemple). Enfin, le réseau inclut quatre serveurs, deux pour assurer les services web de l'entreprise et les deux restants pour héberger des bases de données internes à l'entreprise. Par ailleurs, nous supposons que l'état initial des commutateurs qui constituent ce réseau est vierge, en d'autres termes les tables de flux des commutateurs OpenFlow sont vides.

Ainsi, les objectifs de gestion de réseaux de ce cas d'étude sont les suivants :

- Les serveurs web, qui hébergent le portail web de l'entreprise, sont accessibles à tous les utilisateurs internes et externes.
- Les serveurs qui hébergent les bases de données sont uniquement accessibles aux employées de l'entreprise qui disposent d'une autorisation. Ces employés sont

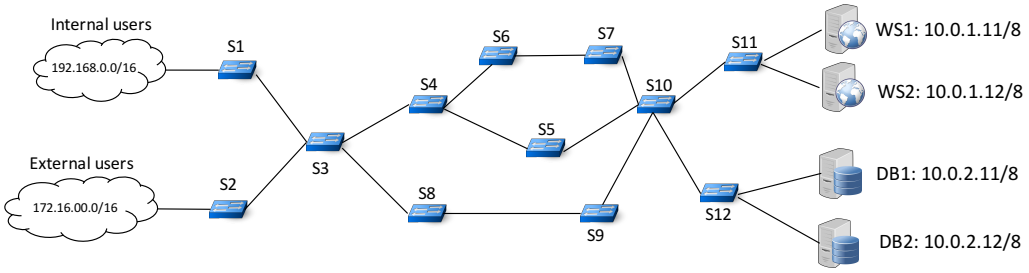

FIGURE 43. Topologie physique du cas d'etude

obligatoirement situés sur le réseau *Internal users*

- Pour assurer une bonne qualité de service, l'entreprise souhaite mettre en place une politique de répartition de charge entre ses serveurs (web et bases de données). La mise en place d'une telle politique suppose que les serveurs privés sont accessibles via une adresse publique

6.2 Conception de la topologie virtuelle

La première étape de spécification d'un programme AirNet est la définition de la topologie virtuelle. Cette étape se résume en des choix de conception (par exemple, nombre d'edges et de fabrics) qui sont guidés principalement par les politiques de contrôle à mettre en place ou des contraintes pouvant exister dans l'infrastructure physique.

Dans le contexte de ce cas d'étude, nous avons opté pour la topologie virtuelle qui est illustrée à la figure 44. Cette dernière inclut trois edges (`AC1`, `AC2` et `IO`), deux fabrics (`Fab1` et `Fab2`), deux réseaux (`Int.Net` et `Ext.Net`) et enfin quatre hôtes (`WS1`, `WS2`, `DB1` et `DB2`). L'extrait de programme illustré à la figure 45 représente le code AirNet qui instancie cette topologie virtuelle.

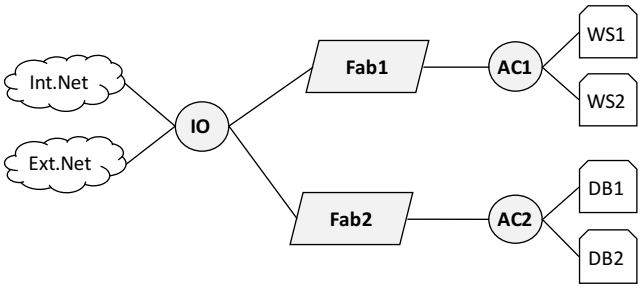

FIGURE 44. Topologie virtuelle du cas d'étude

Les deux réseaux virtuels `Int.Net` et `Ext.Net` représentent respectivement les utilisateurs internes et externes de l'entreprise. De même, les hôtes `WS1`, `WS2`, `DB1` et `DB2` représentent les serveurs de l'entreprise. Ces composants virtuels permettent d'abstraire les détails de bas niveau (adresses réseau et liaison) des serveurs et des réseaux d'utilisateurs d'extrémités.

L'edge `IO` joue le rôle d'interface `host-network` entre les utilisateurs et le réseau. Nous avons choisi d'utiliser un seul edge comme point d'entrée au réseau afin de regrouper toutes les politiques de contrôle d'accès au niveau d'un seul composant virtuel. Néanmoins, il aurait été possible d'utiliser deux edges, un pour connecter le réseau des utilisateurs externes et le deuxième pour connecter les employés de l'entreprise.

La topologie virtuelle dispose aussi de deux fabrics (`FAB1` et `FAB2`). Ce choix de conception a pour objectif de distinguer, conceptuellement, deux chemins de transport suivant la destination (serveur web ou base de données). Là aussi, une seule fabric aurait été aussi envisageable.

Enfin, la topologie virtuelle contient deux edges de distribution `AC1` et `AC2` pour connecter les serveurs privés de l'entreprise. C'est notamment sur ces edges que seront installées les fonctions de répartition de charge que nous présentons dans la section suivante.

```
def virtual_network():
    topologie = VTopology()
    topologie.addEdge(IO,4)
    topologie.addEdge(AC1,3)
    topologie.addEdge(AC2,3)
    topologie.addFabric(FAB1,2)
    topologie.addFabric(FAB2,2)
    topologie.addHost(WS1)
    topologie.addHost(WS2)
    topologie.addHost(DB1)
    topologie.addHost(DB2)
    topologie.addNetwork(IntNet)
    topologie.addNetwork(ExtNet)
    topologie.addLink((IO,1), (IntNet,0))
    topologie.addLink((IO,2), (ExtNet,0))
    topologie.addLink((IO,3), (FAB1,1))
    topologie.addLink((IO,4), (FAB2,1))
    topologie.addLink((AC1,1), (WS1,0))
    topologie.addLink((AC1,2), (WS2,0))
    topologie.addLink((AC1,3), (FAB1,2))
    topologie.addLink((AC2,1), (DB1,0))
    topologie.addLink((AC2,2), (DB2,0))
    topologie.addLink((AC2,3), (FAB2,2))
    return topologie
```

FIGURE 45. Code AirNet de création de la topologie virtuelle

6.3 Politiques de contrôle

Après avoir défini la topologie virtuelle, la prochaine étape est la spécification des politiques de contrôle. Considérant les objectifs de haut niveau de ce cas d'étude, nous avons besoin de définir les politiques suivantes :

- Une politique d'authentification pour gérer l'accès aux serveurs qui hébergent les bases de données des clients. Cette dernière sera naturellement installée à l'entrée du réseau (edge `IO`) pour ne laisser passer que les flux autorisés.
- Une politique de répartition de charge entre les serveurs privés de l'entreprise. Cette dernière inclut deux fonctions de répartition, une pour les serveurs web et une

deuxième pour les serveurs de base de données. Ces fonctions doivent par ailleurs être installées à proximité des serveurs privés, en d'autres termes sur les edges `AC1` et `AC2`.

- Une politique d'entrées/sorties statique pour relayer les paquets vers les hôtes et les réseaux d'extrémités.
- Deux politiques de transport pour configurer les deux fabrics du réseau.

Authentification Dans ce cas d'étude, nous supposons un système d'authentification et d'autorisation relativement simple basé sur l'utilisation d'une liste d'adresses IP source autorisées à accéder aux serveurs de base de données. Cette liste est appelée liste blanche ou *white list*. Implémenter cette politique de contrôle d'accès en AirNet se fait à travers l'utilisation d'une fonction de contrôle dynamique. Cette dernière permettrait de vérifier pour chaque nouveau utilisateur si son adresse IP source est dans la liste blanche ou non. Le programme ci-dessous (figure 46) représente le code AirNet de cette fonction d'authentification.

```
@DynamicControlFct(data="packet", limit=1, split=["nw_src"])
def authenticate(packet):
    ip = packet.find('ipv4')
    host_ip_src = ip.srcip.toStr()

    if host_ip_src in whitelist:
        new_policy = (match(edge=IO, nw_src=host_ip_src, nw_dst=PUBLIC_DB_IP) >>
                      tag(in_trusted_flows) >> forward(FAB2))
    else:
        new_policy = (match(edge=IO, nw_src=host_ip_src, nw_dst=PUBLIC_DB_IP) >> drop)
    return new_policy
```

FIGURE 46. Fonction de contrôle dynamique d'authentification

La fonction `authenticate` dispose d'un décorateur qui lui permet de toujours recevoir le premier paquet (`limit=1`) de chaque nouvelle source (`split=["nw_src"]`) qui se manifeste. Par la suite, `authenticate` extrait la partie IP de chaque paquet, puis teste si l'adresse IP source de ce paquet appartient à la liste blanche établie par l'administrateur ou non. Si oui, une politique de transmission est installée pour cette source particulière. Cette politique de transmission consiste à étiqueter tous ces flux comme des flux de confiance (`tag(in_trusted_flows)`) et de les envoyer par la suite vers la deuxième fabric du réseau. En revanche, si l'adresse IP source du paquet n'est pas dans la liste blanche, une règle qui bloque tous les paquets provenant de cette source est installée.

L'avantage d'utiliser une fonction de contrôle dynamique en lieu et place d'une approche statique est que l'administrateur n'a pas besoin de définir une politique spécifique pour chaque entrée de la liste blanche, mais uniquement une seule politique pour tous les utilisateurs du réseau interne. De plus, dans le cas d'une installation dynamique, la liste blanche peut être externalisée sous une forme quelconque (base de données, service applicatif, etc.) et chaque mise à jour (ajout ou retrait d'adresses) sera prise en compte automatiquement par la fonction dynamique.

La fonction d'authentification étant définie, il reste maintenant à l'associer à des filtres afin de lui rediriger les flux de paquets des utilisateurs internes, comme cela est montré dans la politique `i2` de l'extrait de programme exposé à la figure 47 qui représente les politiques de l'edge `IO`.

```
def IO_policy():
    i1 = match(edge=IO, nw_dst=PUBLIC_WS_IP, tp_dst=HTTP) >> tag(in_web_flows) >>
                    forward(FAB1)
    i2 = match(edge=IO, src=IntNet, nw_dst=PUBLIC_DB_IP) >> authenticate()
    i3 = match(edge=IO, dst=IntNet)  >> forward(IntNet)
    i4 = match(edge=IO, dst=ExtNet)  >> forward(ExtNet)
    return i1 + i2 + i3 + i4
```

FIGURE 47. Politiques installées sur l'edge IO

Politiques d'entrées/sorties La fonction `IO_policy` inclut quatre politiques : la première politique (`i1`) redirige tous les paquets web qui ont comme destination l'adresse IP publique du serveur web vers la première fabric. Cette dernière se chargera par la suite de les transporter vers l'edge qui est connecté aux serveurs web privés. La deuxième politique, comme nous venons de le voir, capture tous les paquets qui proviennent des utilisateurs internes et qui ont comme destination le serveur public qui héberge les bases de données, pour les faire passer par la fonction `authenticate` afin de vérifier s'ils sont autorisés à accéder à la base de données ou non. Les deux politiques `i3` et `i4`, quant à elles, désignent des politiques de distribution standards qui permettent de relayer les paquets respectivement vers les utilisateurs internes et externes du réseau. En dernier, ces quatre politiques sont composées ensemble pour former une politique globale de contrôle d'accès qui sera installée sur l'edge `IO`.

```
@DynamicControlFct(data="packet", split=["nw_src"], limit=1)
def dynamic_LB(packet, args):

    ip = packet.find('ipv4')
    host = ip.srcip.toStr()

    if dynamic_LB.token == 1:
        newPolicy = (match(edge=args.Edge, nw_src=host, nw_dst=args.PUBLIC_IP) >>
                     modify(nw_dst=args.PRIVATE_IP1) >>
                     modify(dl_dst=args.PRIVATE_MAC1) >>
                     forward(args.DST1))
        dynamic_LB.token = 2
    else:
        newPolicy = (match(edge=args.Edge, nw_src=host, nw_dst=args.PUBLIC_IP) >>
                     modify(nw_dst=args.PRIVATE_IP2) >>
                     modify(dl_dst=args.PRIVATE_MAC2) >>
                     forward(args.DST2))
        dynamic_LB.token = 1
    return newPolicy
dynamic_LB.token = 1
```

FIGURE 48. Fonction de contrôle dynamique de répartition de charge

Répartition de charge Dans ce cas d'étude, nous avons besoin de deux politiques de répartition de charge, une pour les serveurs web et une deuxième pour les serveurs qui hébergent les bases de données. Ici aussi, nous allons utiliser des fonctions de contrôle dynamiques. Néanmoins, nous définissons une seule fonction dynamique que nous réutiliserons sur les deux edges `AC1` et `AC2`. L'extrait de programme illustré à la figure 48 représente la définition de la fonction de contrôle dynamique `dynamic_LB` qui implémente un service de répartition de charge en tourniquet (*round-robin*).

Comme pour la fonction `authenticate`, la fonction `dynamic_LB` reçoit chaque premier paquet d'une nouvelle source, plus une structure de données (`args`) qui contient les informations concernant les serveurs privés de l'entreprise. Comme on peut le voir sur l'extrait de programme ci-dessus, la fonction `dynamic_LB` utilise un jeton (`token`) pour prendre sa décision de relayer les paquets soit sur le premier, soit le deuxième serveur privé. Relayer les paquets vers les serveurs privés implique de changer les adresses destination de niveau deux (`dl_dst`) et trois (`nw_dst`). De plus, dès qu'une décision est prise, la fonction met à jour la valeur du jeton de sorte à ce que le prochain nouveau flux soit relayé vers un serveur privé différent.

La fonction `dynamic_LB` est composée par la suite avec des filtres afin de rediriger vers elle les paquets appropriés. Par ailleurs, il est à noter que les réponses des serveurs privés doivent elles aussi être modifiées afin de remplacer les adresses privées (IP et MAC) par les adresses publiques visibles pour les utilisateurs d'extrémités, comme cela est montré dans l'extrait de programme (figure 49) ci-dessous.

```
def AC1_policy():
    i1 = match(edge=AC1, nw_dst=PUBLIC_WS_IP) >> dynamic_LB(args=WS_Infos)
    i2 = match(edge=AC1, src=WS1) >>
                    modify(nw_src=PUBLIC_WS_IP) >>
                    modify(dl_src=PUBLIC_WS_MAC) >>
                    tag(out_web_flows) >>
                    forward(FAB1)
    i3 = match(edge=AC1, src=WS2) >>
                    modify(nw_src=PUBLIC_WS_IP) >>
                    modify(dl_src=PUBLIC_WS_MAC) >>
                    tag(out_web_flows) >>
                    forward(FAB1)
    return i1 + i2 + i3

def AC2_policy():
    i1 = match(edge=AC2, nw_dst=PUBLIC_DB_IP) >> dynamic_LB(args=DB_Infos)
    i2 = match(edge=AC2, src=DB1) >>
                    modify(nw_src=PUBLIC_DB_IP) >>
                    modify(dl_src=PUBLIC_DB_MAC) >>
                    tag(out_trusted_flows) >>
                    forward(FAB2)
    i3 = match(edge=AC2, src=DB2) >>
                    modify(nw_src=PUBLIC_DB_IP) >>
                    modify(dl_src=PUBLIC_DB_MAC) >>
                    tag(out_trusted_flows) >>
                    forward(FAB2)
    return i1 + i2 + i3
```

FIGURE 49. Politiques installées sur les edges AC1 et AC2

Les fonctions `AC1_policy` et `AC2_policy` assurent les mêmes fonctionnalités, excepté que la première est installée sur l'edge `AC1` et assure la répartition de charge entre les serveurs `WS1` et `WS2`, alors que la deuxième est installée sur l'edge `AC2` et assure la répartition entre les serveurs `DB1` et `DB2`. Chacune de ces fonctions contient trois règles, la première `i1` permet de capturer les flux en direction de l'adresse publique du serveur, puis les redirige vers la fonction de contrôle dynamique `dynamic_LB` tout en lui passant une structure de données qui contient des informations sur les serveurs privés (edge, destination, adresses réseau et liaison). Les règles `i2` et `i3`, quant à elles, assurent la modification des réponses des serveurs en remplaçant les adresses privés (MAC et IP) par les adresses publiques.

Transport Les politiques des edges étant spécifiées, la prochaine étape est de définir les politiques de transport à installer sur les deux fabrics. Concrètement, cela consiste à spécifier des règles de transport pour les étiquettes qui ont été préalablement insérées par les différents edges. Dans notre exemple, ces politiques de transport sont triviales comme cela est montré dans l'extrait (figure 50) de programme ci-dessous :

```
def fabric_policies():
    t1 = catch(fabric=FAB1, src=IO, flow=in_web_flows) >> carry(dst=AC1)
    t2 = catch(fabric=FAB1, src=AC1, flow=out_web_flows) >> carry(dst=IO)
    t3 = catch(fabric=FAB2, src=IO, flow=in_trusted_flows) >> carry(dst=AC2)
    t4 = catch(fabric=FAB2, src=AC2, flow=out_trusted_flows) >> carry(dst=IO)
    return t1 + t2 + t3 + t4
```

FIGURE 50. Politiques de transport du cas d'étude

La fonction `fabric_policies` contient quatre règles, deux pour chaque fabric. Chacune de ces règles permet d'acheminer les flux d'un edge vers un autre dans un sens, comme la règle `t1` qui configure la fabric `Fab1` afin de transporter les flux de paquets web entrants (`in_web_flows`) de l'edge `IO` vers l'edge `AC1`. En dernier, toutes ces politiques de transport sont composées ensemble d'une manière parallèle et retournées sous la forme d'une seule.

6.4 Informations de mapping

La dernière étape dans le processus de construction d'un programme AirNet est la spécification du module de mapping. Ce module contient les informations qui permettent d'associer chaque composant virtuel à un ou plusieurs équipements de l'infrastructure physique. Pour ce cas d'étude, nous avons utilisé le module de mapping qui est listé à la figure 51.

Ce schéma de mapping est associé à la topologie physique illustrée à la figure 43. Ainsi, les edges de la topologie virtuelle sont forcément associés aux commutateurs physiques de bordure de réseau. On retrouve ici deux schémas de mapping différents : *i*) un-à-un pour les edges `AC1` et `AC2` qui sont associés aux commutateurs physiques `s11` et `s12`, et *ii*) un-à-plusieurs pour l'edge `IO` qui est associé aux commutateurs `s1` et `s2`.

```
class Mymapping(Mapping):
    def __init__(self):
        Mapping.__init__(self)
        self.addEdgeMap(IO, "s1", "s2")
        self.addEdgeMap(AC1, "s11")
        self.addEdgeMap(AC2, "s12")
        self.addFabricMap(FAB1, "s3", "s4", "s5", "s6", "s7", "s10")
        self.addFabricMap(FAB2, "s3", "s8", "s9", "s10")
        self.addHostMap(WS1, "10.0.1.11")
        self.addHostMap(WS2, "10.0.1.12")
        self.addHostMap(DB1, "10.0.2.11")
        self.addHostMap(DB2, "10.0.2.12")
        self.addNetworkMap(IntNet, "192.168.0.0/16")
        self.addNetworkMap(ExtNet, "172.16.0.0/16")

def main():
    return Mymapping()
```

FIGURE 51. Premier scénario de mapping du cas d'étude

Concernant les fabrics, nous avons choisi un mapping qui nous permet de distinguer deux chemins physiques différents. Ainsi, la première fabric (`FAB1`) est associée aux groupe de commutateurs {`s3, s4, s5, s6, s7, s10`} et permet d'accéder aux serveurs web privés, alors que la deuxième fabric est associée au groupe de commutateurs {`s3, s8, s9, s10`} qui eux permettent d'atteindre les serveurs qui hébergent les bases de données.

6.5 Expérimentation

La spécification du programme AirNet étant terminée, nous passons maintenant à son expérimentation, qui inclut les étapes suivantes :

- Instancier la topologie physique en utilisant l'émulateur *Mininet*.
- Lancer la phase de compilation proactive (installation des politiques statiques).
- Tester les politiques statiques.
- Tester l'installation des politiques dynamiques (phase réactive).
- Tester la reconfiguration dynamique du réseau suite à une défaillance dans le réseau physique (*LinkDown*, *LinkUp*).
- Tester la portabilité du programme de contrôle par sa réutilisation sur une infrastructure physique différente.

L'objectif de ces tests est de valider expérimentalement les différentes fonctionnalités offertes par l'hyperviseur AirNet. Un volet mesure de performances sera traité dans le chapitre suivant.

6.5.1 L'émulateur Mininet

SDN étant une approche relativement récente, les équipements implémentant des protocoles SDN tel que OpenFlow ne sont pas nombreux, et représentent un investissement

conséquent. Heureusement il existe une solution logicielle permettant de reproduire le comportement de ces commutateurs. En effet, Mininet [Min, 2010, Lantz et al., 2010] est un projet open source (écrit en python) dont le but est d'émuler un réseau d'hôtes, de commutateurs OpenFlow et de contrôleurs SDN qui exécutent des vrais noyaux systèmes et cela sur une machine unique qu'elle soit physique, virtuelle ou sur le cloud.

De plus, l'installation et l'utilisation de Mininet sont relativement simples, le rendant de facto indispensable pour tout travail de recherche sur SDN. Même s'il embarque un contrôleur, le fait qu'il puisse être utilisé avec n'importe quel autre contrôleur le rend encore plus intéressant.

Pour expérimenter AirNet (incluant ce cas d'étude), nous avons utilisé la version 2.2.1 de Mininet que nous exécutons sur une machine virtuelle. L'hyperviseur d'AirNet est, quant à lui, lancé sur la même machine physique qui héberge la machine virtuelle de Mininet. La communication entre ces deux composants est assurée par le canal OpenFlow qui connecte l'émulateur Mininet et le contrôleur POX au-dessus duquel l'hyperviseur est installé.

Ainsi, la première étape de l'expérimentation de ce cas d'étude consiste à lancer Mininet afin d'instancier un réseau virtuel qui corresponde à la topologie illustrée à la figure 43. Il est à noter que dans la suite de ce chapitre nous allons toujours faire référence à ce réseau virtuel créé par Mininet comme étant l'infrastructure physique.

6.5.2 Phase proactive

Durant cette première phase, l'hyperviseur d'AirNet compile les politiques virtuelles et génère par la suite des règles OpenFlow qui sont installées sur les commutateurs à travers le contrôleur POX. Pour ce cas d'étude, l'hyperviseur a généré et installé un total de quarante quatre (44) règles OpenFlow en 149 millisecondes. Les entrées OpenFlow qui ont été insérées dans les tables de flux des commutateurs peuvent être divisées en trois groupes, suivant leur rôle :

- Gestion des paquets ARP.
- Commutation des paquets entre les équipements OpenFlow.
- Relayage des paquets vers le contrôleur POX (fonctions réseau).

Le premier groupe de règles permet de rediriger toutes les requêtes ARP vers le contrôleur. L'hyperviseur d'AirNet installe une règle sur chaque équipement OpenFlow qui est associé à un edge virtuel, ce qui fait un nombre total de quatre règles, une sur chaque commutateur de bordure (`s1, s2, s11` et `s12`).

Pour assurer la transmission des paquets entre les hôtes (les serveurs) et les réseaux d'extrémités (utilisateurs internes et externes), l'hyperviseur installe des règles de commutation le long de plusieurs chemins, suivant les informations mapping et les plus courts chemins qu'il a identifiés. Par exemple, pour les flux de paquets web, l'hyperviseur installe les règles OpenFlow suivantes :

- Bordure du réseau : sur les commutateurs `s1` et `s2`, l'hyperviseur installe les règles de distribution classiques qui permettent d'envoyer et de recevoir des flux web en provenance des hôtes d'extrémités (utilisateurs internes et externes).

- Cœur du réseau : pour accéder aux serveurs web privés, les flux de paquets doivent passer par la première fabric. Ainsi, l'hyperviseur installe des règles de commutation sur le plus court chemin qu'inclut cette dernière et qui correspond au chemin (`s3, s4, s5, s10`).

Un exemple concrèt de ces règles est donné à la figure 52 qui représente une capture d'écran du contenu de la table de flux du commutateur `S11` (celui qui est connecté aux serveurs web privés) après l'exécution de la phase proactive. Cette table de flux contient les cinq règles OpenFlow suivantes :

- `nw_src=10.0.1.12` : modifie les flux de paquets qui proviennent du premier serveur web privé de sorte à remplacer les adresses privées (MAC et IP) par des adresses publiques.
- `nw_src=10.0.1.11` : modifie les flux de paquets qui proviennent du deuxième serveur web privé de sorte à remplacer les adresses privées (MAC et IP) par des adresses publiques.
- `nw_dst=10.0.0.50` : les flux de paquets à destination de l'adresse IP publique du serveur web sont redirigés vers le contrôleur (fonction de répartition de charge).
- Tous les flux IP : une règle de sécurité par défaut qui rejette tous les paquets qui ne correspondent à aucune des règles de la table de flux. Il est à noter que pour éviter toute intersection avec les flux des autres règles de la table, l'hyperviseur attribue toujours à cette règle de sécurité la priorité la plus basse.
- `arp` : tous les paquets de type ARP sont redirigés vers le contrôleur SDN (module `Proxy_ARP`).

```
*** s11 ----------------------------------------------------------------------
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=11.167s, table=0, n_packets=0, n_bytes=0, idle_age=11, priority=2
,ip,nw_src=10.0.1.12 actions=mod_nw_src:10.0.0.50,mod_dl_src:00:26:55:42:9a:62,output:3
 cookie=0x0, duration=11.167s, table=0, n_packets=0, n_bytes=0, idle_age=11, priority=3
,ip,nw_src=10.0.1.11 actions=mod_nw_src:10.0.0.50,mod_dl_src:00:26:55:42:9a:62,output:3
 cookie=0x0, duration=11.167s, table=0, n_packets=0, n_bytes=0, idle_age=11, priority=4
,ip,nw_dst=10.0.0.50 actions=CONTROLLER:65535
 cookie=0x0, duration=11.167s, table=0, n_packets=0, n_bytes=0, idle_age=11, priority=1
,ip actions=drop
 cookie=0x0, duration=11.198s, table=0, n_packets=0, n_bytes=0, idle_age=11, arp action
s=CONTROLLER:65535
```

FIGURE 52. Table de flux du commutateur s11 (phase proactive)

6.5.3 Phase réactive

Pour distribuer les flux vers les serveurs privés, le premier paquet de chaque nouvelle source doit passer par la fonction d'équilibrage de charge. Pour ce faire, l'hyperviseur installe sur les commutateurs `s11` et `s12` une règle qui redirige vers le contrôleur (sur lequel l'hyperviseur est installé) tous les flux en direction d'une adresse IP publique spécifique. A la fin de l'exécution de la fonction de répartition de charge, une nouvelle politique est générée, puis compilée et installée sur les commutateurs physiques.

```
*** s11 ----------------------------------------------------------------------
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=5.882s, table=0, n_packets=4, n_bytes=379, idle_age=5, priority=5,ip,nw_src=192.168
.0.11,nw_dst=10.0.0.50 actions=mod_nw_dst:10.0.1.11,mod_dl_dst:00:00:00:00:11:00,output:1
 cookie=0x0, duration=39.527s, table=0, n_packets=0, n_bytes=0, idle_age=39, priority=2,ip,nw_src=10.0.1.
12 actions=mod_nw_src:10.0.0.50,mod_dl_src:00:26:55:42:9a:62,output:3
 cookie=0x0, duration=39.528s, table=0, n_packets=5, n_bytes=548, idle_age=5, priority=3,ip,nw_src=10.0.1
.11 actions=mod_nw_src:10.0.0.50,mod_dl_src:00:26:55:42:9a:62,output:3
 cookie=0x0, duration=39.528s, table=0, n_packets=1, n_bytes=74, idle_age=5, priority=4,ip,nw_dst=10.0.0.
50 actions=CONTROLLER:65535
 cookie=0x0, duration=39.527s, table=0, n_packets=0, n_bytes=0, idle_age=39, priority=1,ip actions=drop
 cookie=0x0, duration=39.556s, table=0, n_packets=1, n_bytes=42, idle_age=5, arp actions=CONTROLLER:65535
```

FIGURE 53. Table de flux du commutateur s11 (phase reactive)

La figure 53 illustre la nouvelle table de flux du commutateur `s11` avec notamment la nouvelle règle qui a été ajoutée suite à l'exécution de la fonction de répartition de charge (la règle encadrée en rouge). La figure 54, quant à elle, représente le résultat de l'envoi d'une requête web (`wget`) d'un hôte interne vers l'adresse IP publique du serveur web (`10.0.0.50`). Sur la figure 53, nous pouvons constater que la nouvelle règle installée est, d'une part, spécifique à l'hôte interne (`nw_src=192.168.0.11, nw_dst=10.0.0.50`) et, d'autre part, plus prioritaire (`priority=5`) que la règle qui redirige les flux web vers le contrôleur SDN (`priority=4`), permettant ainsi d'envoyer directement les flux web de cet hôte vers le premier serveur privé (`actions=mod_nw_dst:10.0.1.11, mod_dl_dst:00:00:00:00:11:00, output:1`).

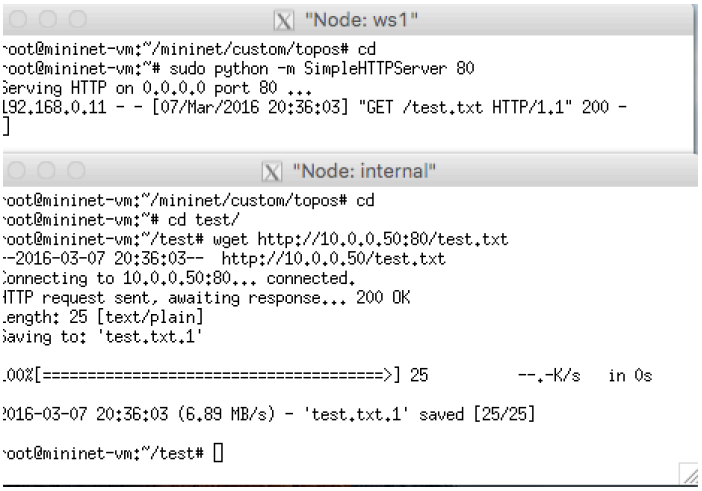

FIGURE 54. Résultat d'envoi d'une requête d'un hôte interne vers le serveur web public

Concernant les serveurs de base de données, avant d'arriver à la fonction de répartition de charge, les flux qui proviennent du réseau des utilisateurs internes doivent passer par la fonction d'authentification. Si cette dernière retourne une politique qui autorise un utilisateur, la politique en question est alors compilée et donnera cinq nouvelles règles OpenFlow qui seront installées sur le commutateur de bordure `s1` et les commutateurs de la deuxième fabric (`s3, s8, s9, s10`). La figure 55 illustre le contenu de la table de flux du commutateur `s1` après l'envoi d'un `ping` d'un utilisateur du réseau interne vers l'adresse IP publique du serveur de base de données. Sur cette figure, l'entrée encadrée en rouge

représente la règle qui a été installée après authentification de la source `192.168.0.11` et qui autorise maintenant la commutation des paquets de cette source vers le commutateur `s3` au lieu de les rediriger vers le contrôleur SDN (règle encadrée en vert). Là encore, la priorité respective de ces deux règles résout les problèmes d'intersection entre filtres.

```
mininet> sh ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=174.935s, table=0, n_packets=4, n_bytes=392, idle_age=170, priority=5,ip,nw_src
=192.168.0.11,nw_dst=10.0.0.60 actions=output:2
 cookie=0x0, duration=223.296s, table=0, n_packets=10, n_bytes=1038, idle_age=170, priority=2,ip,nw_d
st=192.168.0.0/16 actions=output:1
 cookie=0x0, duration=223.296s, table=0, n_packets=1, n_bytes=98, idle_age=174, priority=3,ip,nw_src=
192.168.0.0/16,nw_dst=10.0.0.60 actions=CONTROLLER:65535
 cookie=0x0, duration=223.296s, table=0, n_packets=0, n_bytes=0, idle_age=223, priority=1,ip actions=
drop
 cookie=0x0, duration=223.321s, table=0, n_packets=2, n_bytes=84, idle_age=174, arp actions=CONTROLLE
R:65535
 cookie=0x0, duration=223.296s, table=0, n_packets=5, n_bytes=453, idle_age=195, priority=4,tcp,nw_ds
t=10.0.0.50,tp_dst=80 actions=output:2
```

FIGURE 55. Table de flux du commutateur S1 (phase réactive)

Par la suite, la fonction de répartition de charge qui est sur le commutateur `s12` installera une règle OpenFlow par flux pour rediriger les paquets vers l'un des deux serveurs de base de données privés. La figure 56 représente une capture d'écran de la commande `ping` que nous avons effectuée à partir d'un hôte du réseau interne vers l'adresse IP publique (`10.0.0.60`) du serveur de base de données.

```
mininet> internal ping 10.0.0.60
PING 10.0.0.60 (10.0.0.60) 56(84) bytes of data.
64 bytes from 10.0.0.60: icmp_seq=1 ttl=64 time=125 ms
64 bytes from 10.0.0.60: icmp_seq=2 ttl=64 time=0.477 ms
64 bytes from 10.0.0.60: icmp_seq=3 ttl=64 time=0.067 ms
64 bytes from 10.0.0.60: icmp_seq=4 ttl=64 time=0.073 ms
64 bytes from 10.0.0.60: icmp_seq=5 ttl=64 time=0.067 ms
^C
--- 10.0.0.60 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4000ms
rtt min/avg/max/mdev = 0.067/25.185/125.244/50.029 ms
mininet>
```

FIGURE 56. Test de communication entre un utilisateur interne et un serveur de base de données

6.5.4 Evénements de l'infrastructure

Pour tester la capacité de l'hyperviseur à réagir face à une défaillance dans l'infrastructure physique, nous avons coupé le lien mininet qui relie le commutateur `s4` et le commutateur `s5`. A la réception d'un événement de type `LinkDown`, l'hyperviseur enclenche une procédure de reconfiguration des chemins de la première fabric, qui consiste à exécuter les étapes suivantes :

- Recalculer les chemins afin de trouver un nouveau plus court chemin.
- Réexécuter la phase de mapping physique afin de générer de nouveaux *classifiers* pour les commutateurs physiques. Notez que dans le cas d'une mise à jour de l'infrastructure, l'hyperviseur ne réexécute pas la phase de composition virtuelle.

- Générer les dictionnaires de différences qui contiennent les règles à ajouter, à supprimer ou a modifier sur les commutateurs de l'infrastructure physique.
- Communiquer les modifications au client de l'hyperviseur afin qu'il puisse les mettre en œuvre sur l'infrastructure physique.

Le résultat est un basculement de tous les flux qui passaient par le chemin (`s4`, `s5`, `s10`) vers le chemin (`s4`, `s6`, `s7`, `s10`). Concrètement, l'hyperviseur a supprimé trois règles existantes sur le commutateur `s5`, installé six nouvelles sur les commutateurs `s6` et `s7` et en à modifié aussi trois existantes sur les commutateurs `s4` et `s10`.

Dans un deuxième temps, on réactive le lien entre les deux commutateurs `s4` et `s5`, A la réception d'un événement de type `LinkUP`, l'hyperviseur recalcule les chemins existants et identifie un nouveau plus court chemin. L'hyperviseur enclenche alors une nouvelle procédure d'adaptation qui consiste à rebasculer les flux sur le chemin original (`s4`, `s5`, `s10`). Concrètement, cette adaptation consiste à faire l'inverse des opération de l'adaptation précédente. Ainsi, six règles sont supprimées, trois nouvelles installées et trois existantes modifiées.

6.5.5 Réutilisabilité

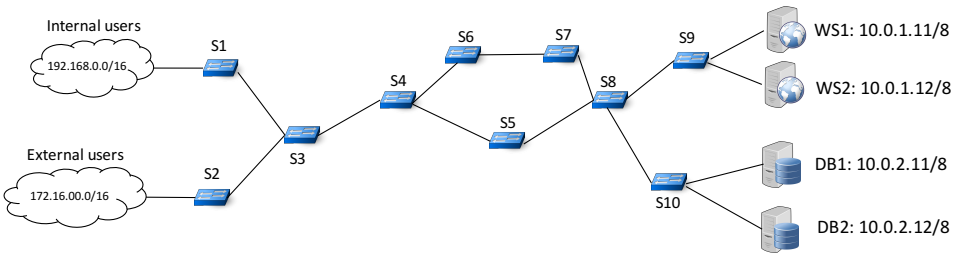

FIGURE 57. Cas d'étude : infrastructure physique 2

Pour montrer la réutilisabilité des programmes AirNet et la pertinence d'une approche basée sur la virtualisation de réseau, nous avons réutilisé la même topologie virtuelle et le même programme de contrôle sur une topologie physique différente (figure 57). La seule modification que nous avons dû apporter concerne le nouveau module de mapping qui est donné dans l'extrait de programme ci-dessous (figure 58) :

Avec ces nouvelles règles de mapping, nous avons associé les deux fabrics aux mêmes équipements physiques. Aussi, nous avons exécuté les mêmes étapes de tests sur cette nouvelle topologie et aucune erreur ou comportement imprévu n'a été détecté. Outre la réutilisabilité et la portabilité du programme AirNet, cet exemple met en évidence la flexibilité de mapping des fabrics qui peuvent être liées à des commutateurs physiques de diverses manières suivant les besoins de l'administrateur.

6.6 Bilan

Démontrer l'utilité d'une *Northbound API* est un exercice complexe, étant donné la multitude des critères et des contextes d'utilisation qui peuvent être considérés. Néanmoins,

```
class Mymapping(Mapping):
    def __init__(self):
        Mapping.__init__(self)
        self.addEdgeMap(IO, "s1", "s2")
        self.addEdgeMap(AC1, "s9")
        self.addEdgeMap(AC2, "s10")
        self.addFabricMap(FAB1, "s3", "s4", "s5", "s6", "s7", "s8")
        self.addFabricMap(FAB2, "s3", "s4", "s5", "s6", "s7", "s8")
        self.addHostMap(WS1, "10.0.1.11")
        self.addHostMap(WS2, "10.0.1.12")
        self.addHostMap(DB1, "10.0.2.11")
        self.addHostMap(DB2, "10.0.2.12")
        self.addNetworkMap(IntNet, "192.168.0.0/16")
        self.addNetworkMap(ExtNet, "172.16.0.0/16")

def main():
    return Mymapping()
```

FIGURE 58. Cas d'étude : module de mapping topologie 2

dans ce chapitre nous avons essayé de souligner la simplicité d'utilisation du langage AirNet à travers le déroulement d'un cas d'étude en partant de la phase d'étude, jusqu'aux résultats d'exécution. De plus, ce chapitre montre clairement l'utilité d'une approche basée sur la virtualisation de réseau, surtout en considérant la grande différence qui existe entre le nombre de politiques spécifiées par l'administrateur et le nombre réel de règles OpenFlow installées par l'hyperviseur sur les commutateurs physiques. Le tableau 2 résume les chiffres obtenus pour chaque phase d'exécution de ce cas d'étude et où chaque ligne doit être interprétée de la façon suivante :

- ***Policies*** : nombre de politiques virtuelles spécifiées dans le programme de contrôle.
- ***init.phase*** : nombre de règles physiques installées à la fin de la phase proactive (phase d'initialisation de l'infrastructure).
- ***Link s4-s5 down*** : adaptations effectuées après une défaillance sur un lien de l'infrastructure physique. Il est à noter que ce test a été effectué avant la phase reactive.
- ***Host–>WS*** : nombre de règles ajoutées après l'envoi d'une requête http (de l'hôte `192.168.0.11`) vers l'adresse publique du serveur web.
- ***Host–>DB*** : nombre de règles ajoutées après l'envoi d'un ping (de l'hôte `192.168.0.11`) vers l'adresse publique du serveur de base de données.

Ces chiffres attestent clairement de la différence importante existant entre le nombre de politiques spécifiées par l'administrateur et le nombre de règles réellement installées par l'hyperviseur. Pour un seul utilisateur (*i.e.*, un seul flux source), un facteur de trois existe entre le nombre de politiques pour la phase proactive, et un facteur de quatre pour la phase réactive (accès aux serveurs de base de données). Ce nombre augmente bien sûr linéairement en fonction du nombre de flux sources différents pour la fonction de contrôle dynamique.

Un autre apport, ce dernier étant plutôt qualitatif, est relatif à l'usage d'AirNet : l'administrateur manipule des politiques qui sont écrites dans un langage métier, avec des

TABLE 2. Nombre de politiques virtuelles et règles physiques pour le cas d'étude

Virtual device	**IO**		**Fabric 1 et 2**								**AC1**	**AC2**	Total
Policies	4		4								3	3	14
Physical switch	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	Total
Rules (init. phase)	5	5	5	4	4	1	1	2	2	5	5	5	44
Rules (link s4-s5 down)	–	–	–	mod 1	-3	+3	+3	–	–	mod 2	–	–	47
Rules (IntHost –> WS)	–	–	+1	+1	+1	–	–	–	–	+1	+1	–	49
Rules (IntHost –> DB)	+1	+1	+1	–	–	–	–	+1	+1	+1	–	+1	55

paramètres sous la forme de nom symbolique en lieu et en place des paramètres de bas niveau des règles OpenFlow (filtre, numéro de port de sortie, priorité, etc.)

Une autre caractéristique importante est que l'hyperviseur d'AirNet cache toute la nature complexe et dynamique de l'infrastructure. Ainsi, en cas d'une défaillance d'un lien, c'est l'hyperviseur qui se charge de trouver un nouveau chemin pour les flux de paquets, si ce dernier existe.

Dernier point, mais non le moindre, les politiques d'AirNet étant spécifiées au-dessus de topologies virtuelles, elles peuvent être réutilisées sur différentes topologies physiques, comme cela a été montré dans ce chapitre, sachant que la seule modification qui doit être apportée par l'administrateur est de modifier le module de mapping qui contient les différentes informations sur les associations entre composants virtuels et équipements physiques.

Dans le chapitre suivant, nous aborderons un deuxième volet d'évaluation de l'hyperviseur d'AirNet qui consiste en un ensemble de tests et de mesures de performances afin de montrer que le coût de cette simplification reste parfaitement acceptable.

Chapitre 7
Mesures de performances

Pour évaluer le surcoût de l'utilisation d'AirNet, nous avons mené plusieurs tests de performances. La première partie de ce chapitre commence par présenter l'environnement d'évaluation que nous avons utilisé pour mener ces tests. Puis, dans un second temps, nous détaillons les différents tests et résultats obtenus. Ces derniers étant organisés en deux grandes sections, suivant leur type : *i*) mesure du coût de la compilation des politiques AirNet et *ii*) tests de performance des fonctions réseau.

7.1 Environment d'évalutation

Le paradigme SDN étant une approche relativement récente, concevoir un environnement de tests réel, qui inclut des commutateurs OpenFlow et un contrôleur SDN embarqué dans un serveur dédié est une démarche coûteuse. Heureusement, il existe des outils libres qui, combinés, permettent d'émuler un environnement SDN de tests. La figure 59 illustre l'architecture de notre banc d'essais, qui inclut les quatre parties principales suivantes :

- La machine hôte qui représente la machine physique sur laquelle est exécutée l'hyperviseur AirNet et qui héberge les différentes machines virtuelles.
- L'hyperviseur AirNet qui est exécuté sur le contrôleur POX. Le lien entre l'hyperviseur et POX se fait au travers d'une API Python.
- La machine virtuelle Mininet qui permet d'émuler un réseau virtuel d'hôtes et de commutateurs OpenFlow. La VM Mininet dispose de deux connexions, une vers la machine hôte pour communiquer avec le contrôleur POX et une deuxième vers la machine virtuelle Click.
- La machine virtuelle Click qui permet d'instancier un service de type routeur programmable. La machine virtuelle Click dispose d'une connexion vers l'un des commutateurs OpenFlow de la machine virtuelle Mininet.

Dans le contexte de cet environnement de tests, la rôle de l'administrateur consiste à définir les modules suivants :

- **Pour la VM Click** : définir un fichier de configuration qui décrit le comportement du routeur programmable.

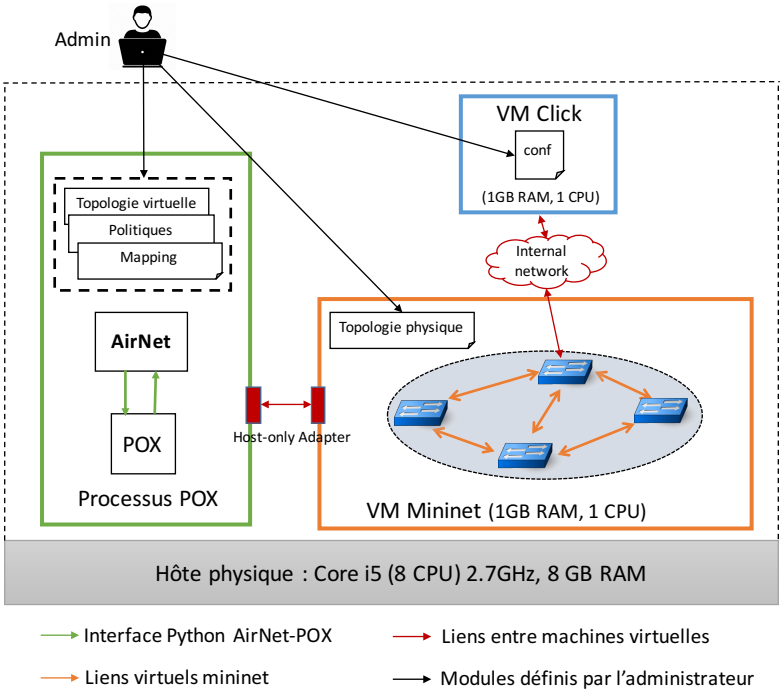

FIGURE 59. Architecture de l'environnement d'évaluation

- **Pour la VM Mininet** : définir un fichier de configuration qui décrit la topologie physique à émuler. C'est notamment au niveau de ce fichier qu'est définie la connexion d'un commutateur OpenFlow avec la VM Click.
- **Pour l'hyperviseur** : définir la topologie virtuelle, les politiques AirNet qui seront exécutées au-dessus de cette dernière et enfin, le module de mapping entre la topologie virtuelle et l'infrastructure physique émulée.

Concernant l'ensemble des tests que nous avons mené, il est important de noter que les résultats présentés représentent une **valeur moyenne de dix essais** que nous avons réalisés pour chaque test.

7.2 Composition virtuelle et mapping physique

Ce premier groupe de tests a été mené afin d'investiguer le coût des phases de composition virtuelle et de mapping physique, qui composent le processus de compilation des politiques virtuelles. Le tableau 3 donne un premier aperçu sur les résultats d'exécution de différents cas d'étude et où chacun d'entre eux implémente une fonctionnalité spécifique d'AirNet. Notez que la figure 60 illustre la topologie physique sur laquelle tous les cas d'étude du tableau 3 ont été testés, et qui sont les suivants :

- *twoFab* : cas d'étude simple qui intègre deux fabrics et qui n'utilise aucune fonction réseau.

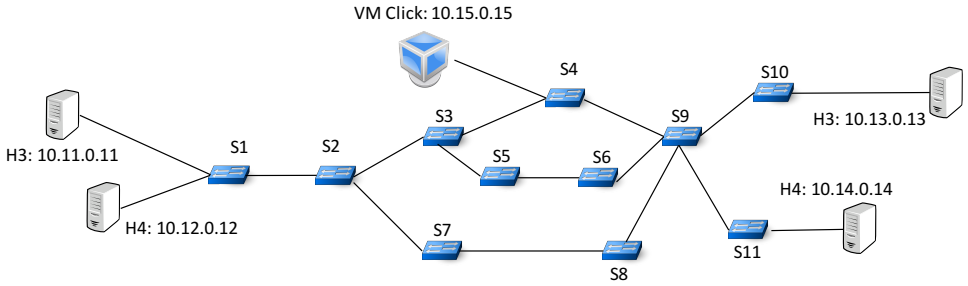

FIGURE 60. Topologie physique pour les tests de performances

TABLE 3. Nombre de politiques et de règles pour chaque cas d'étude

Use case	***Virtual policies***	***Physical rules generated***	***Composition time (ms)***	***Physical mapping time (ms)***
twoFab	12	42	101	20
dynLB	9	31	94	16
bwCap	6	27	89	11
dataFct (control plane)	6	25	95	14
dataFct (data plane)	6	26	99	13

- *dynLB* : cas d'étude de répartition de charge qui utilise une fonction de contrôle dynamique pour remonter le premier paquet de chaque flux.
- *bwCap* : cas d'étude qui surveille le débit maximum des hôtes d'un réseau, notamment en utilisant une fonction de contrôle dynamique qui remonte des statistiques pour ses prises de décision.
- *dataFct (control plane)* : cas d'étude qui définit un chemin de données qui passe par une fonction *data* qui est exécutée au niveau du plan de contrôle (sur l'hyperviseur).
- *dataFct (data plane)* : cas d'étude qui définit un chemin de données qui passe par une fonction *data* qui est exécutée au niveau du plan de données à travers l'utilisation d'un routeur programmable Click.

A partir des résultats obtenus et listés dans le tableau 3, nous pouvons confirmer que le nombre de règles OpenFlow généré par l'hyperviseur est fortement dépendant du nombre de politiques virtuelles d'un programme de contrôle. En effet, chaque politique virtuelle est transformée en une ou plusieurs règles physiques. Ainsi, plus un programme de contrôle contient des politiques de haut niveau, plus l'hyperviseur générera de règles OpenFlow. Par exemple pour le cas d'étude *twoFab*, qui inclut 12 politiques, l'hyperviseur a généré 42 règles physiques, alors que pour le cas d'étude *dynLB* qui lui n'inclut que 9 politiques, l'hyperviseur n'a généré que 31 règles physiques, sachant que ces deux cas d'étude ont été exécutés sur la même topologie physique.

Concernant les temps de composition virtuelle et de mapping physique, le tableau 3

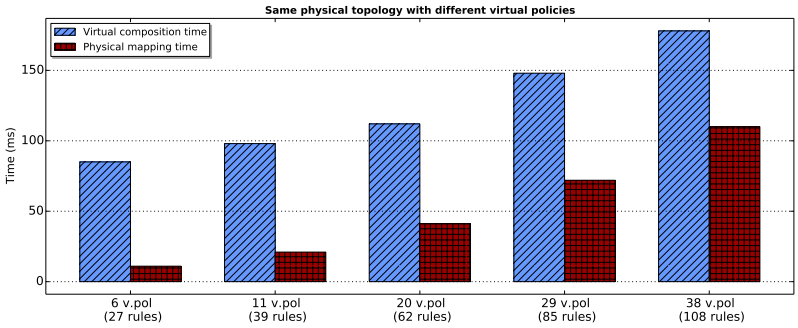

FIGURE 61. Temps de compilation en fonction du nombre de politiques

permet d'avoir une première indication quant au coût de ces deux phases de compilation. En effet, nous pouvons constater que pour tous les cas d'étude, ces temps restent largement acceptables et se mesurent en centaines de millisecondes. Néanmoins, pour avoir des statistiques plus précises qui renseignent sur les facteurs qui impactent le plus ces deux phases de compilation, nous avons effectué les tests suivants :

- *Impact du nombre de politiques virtuelles* (figure 61) : nous exécutons plusieurs cas d'utilisation sur la même topologie physique. Ces cas d'utilisation incluent des politiques de contrôle statiques, mais en nombres différents.
- *Impact de la taille de l'infrastructure physique* (figure 62) : nous exécutons le même cas d'utilisation sur plusieurs topologies physiques de tailles différentes.

7.2.1 Impact du nombre de politiques virtuelles

La figure 61 affiche l'histogramme qui représente les temps de compilation (axe des ordonnées) par rapport au nombre de politiques virtuelles (axe des abscisses). Dans cet histogramme, la classe bleue représente le temps de la phase composition virtuelle, alors que la classe rouge représente le temps de la phase de mapping physique.

Ainsi, nous pouvons constater que le nombre de politiques virtuelles a effectivement un impact direct sur les deux phases de compilation. Concernant la phase de composition virtuelle, les résultats obtenus sont prévisibles, étant donné que plus un programme de contrôle contient de politiques virtuelles, plus l'hyperviseur aura besoin d'effectuer des compositions entre ces dernières afin de résoudre les problèmes d'intersection (cf. section 5.4).

D'autre part, comme cela a été présenté dans la section 5.7.1.2, pour installer les règles des fabrics, l'hyperviseur a besoin d'effectuer des intersections entre les règles *ingress* et les règles *egress*, puis de calculer des chemins à travers les commutateurs de chaque fabric. Ainsi, plus un programme de contrôle contient de politiques, plus l'hyperviseur aura besoin d'effectuer des intersections, de calculer des chemins et d'installer des règles OpenFlow sur les commutateurs physiques (nombre de règles indiqué entre parenthèse sur l'axe des abscisses), menant ainsi à des temps de mapping physique plus importants.

7.2.2 Impact de la taille de l'infrastructure physique

Les résultats d'exécution du deuxième groupe de tests sont montrés à la figure 62. Avec cet histogramme nous investiguons l'impact de la taille de la topologie physique (axe des abscisses) sur les temps de compilation (axe des ordonnées). Au contraire du nombre de politiques virtuelles, nous pouvons constater que la phase de composition virtuelle (classe bleue) est complètement indépendante de la taille et de la complexité de l'infrastructure physique. En effet, lors de l'exécution de cette phase, aucun traitement n'est dépendant des informations qui proviennent de l'infrastructure physique. Ainsi, nous pouvons clairement voir que le coût de la phase de composition virtuelle reste stable à 100 millisecondes malgré l'évolution conséquente du nombre de commutateurs de l'infrastructure physique qui est passé de 7 à 127 commutateurs.

À l'inverse, les résultats qui concernent la phase de mapping physique (classe rouge) montrent que la taille de l'infrastructure physique a un impact important. En effet, plus la topologie physique est grande et complexe, plus l'hyperviseur aura besoin de parcourir de chemins afin d'en trouver les plus courts, puis d'installer les règles OpenFlow sur tous les commutateurs des chemins selectionnés (pour 127 commutateurs, l'hyperviseur a installé 347 règles OpenFlow), menant inexorablement à des temps de mapping physique plus importants.

Par ailleurs, il est important de souligner le fait que l'indépendance de la phase de composition virtuelle vis-à-vis de l'infrastructure physique est particulièrement intéressante pendant la phase d'exécution réactive (fonctions de contrôle dynamiques ou mise à jour de l'infrastructure physique). En effet, dans ce cas, l'hyperviseur n'a pas besoin de réexécuter la phase de composition virtuelle, mais réutilise ses résultats et réexecute uniquement la phase de mapping physique.

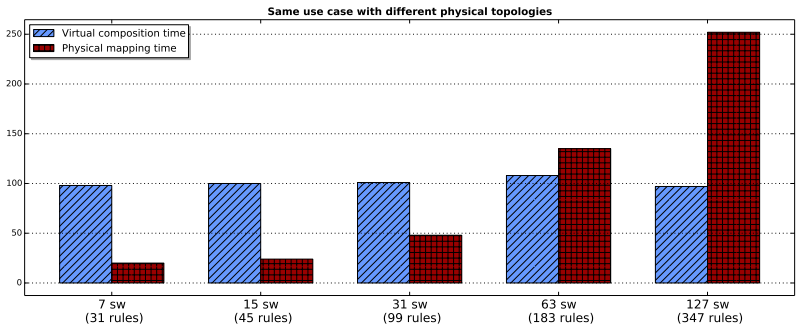

FIGURE 62. Temps de compilation en fonction du nombre de commutateurs physiques

7.3 Performances des fonctions réseau

L'objectif de cette section est de mesurer les délais de bout en bout par rapport à l'utilisation ou non d'une fonction réseau. La figure 63 illustre les quatre solutions que nous testons, à savoir :

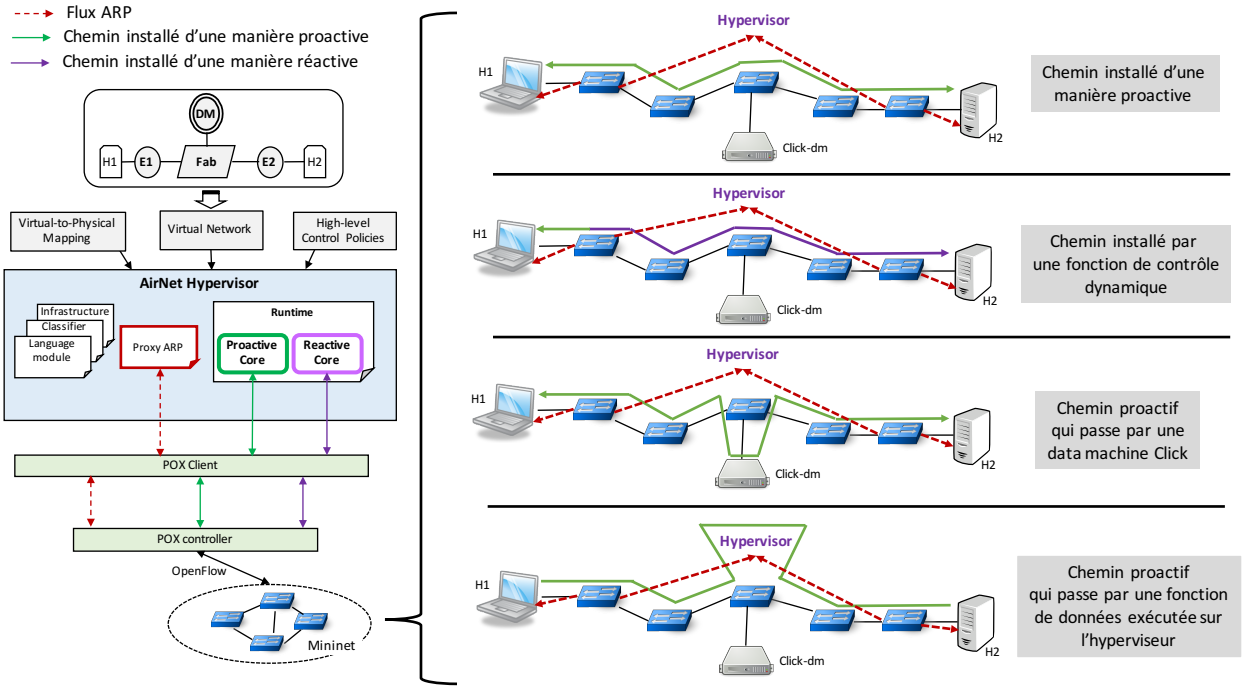

FIGURE 63. Types de chemin testés pour mesurer les délais de bout en bout

- Un chemin est installé d'une manière proactive et ne passe par aucune fonction *data*, qu'elle soit au niveau du plan de contrôle ou du plan de données.
- Un chemin est installé d'une manière réactive (suite à l'exécution d'une fonction de contrôle dynamique) qui, lui aussi, ne passe par aucune fonction *data*.
- Un chemin est installé d'une manière proactive et passe par une *data machine*.
- Un chemin est installé d'une manière proactive et passe par une fonction *data* qui est exécutée au niveau de l'hyperviseur (plan de contrôle).

La figure 64 représente les résultats des délais (temps RTT) entre deux hôtes pour chacun de ces quatre chemins, sachant que ces chemins passent tous par les mêmes commutateurs qui sont au nombre de six.

Dans les quatre types de chemin, la mesure sur le premier paquet est plus importante que les paquets suivants. Cela s'explique par la résolution des requêtes ARP qui sont redirigés vers le proxy ARP de l'hyperviseur. Cette redirection ainsi que la réponse du Proxy ARP prend environ 20 millisecondes dans tous nos tests.

Concernant le chemin installé d'une manière réactive par la fonction de contrôle dynamique (courbe en rouge), tout son coût est concentré sur le premier paquet. En effet, outre le traitement ARP, ce délai supplémentaire est la conséquence de la redirection du premier paquet vers l'hyperviseur, l'exécution de la fonction de contrôle dynamique, l'installation du chemin et, en dernier la réinjection du paquet dans l'infrastructure physique. Tout ce processus prend environ 80 millisecondes dans nos tests. Une fois le premier paquet

passé, les délais chutent et deviennent négligeables comme ceux du chemin installé d'une manière proactive (courbe en bleu).

Pour ce qui est de l'impact des fonctions de données, nous distinguons deux cas : le premier concerne les fonctions *data* qui sont exécutées au niveau du plan de contrôle sur l'hyperviseur et le deuxième, celles qui sont exécutées sur les data machines au niveau du plan de données. La particularité de ces deux types de chemin, en comparaison avec les chemins passant par une fonction de contrôle dynamique, est qu'une fonction *data* est censée être appliquée sur tous les paquets du flux et non pas seulement sur le premier. Cette spécificité implique un impact plus important sur les délais de bout en bout, particulièrement pour les fonctions *data* qui sont exécutées au niveau du plan de contrôle.

En effet, sur la figure 64, nous pouvons constater que les délais du chemin qui passe par ce type de fonction (courbe jaune) sont les plus importants (plus de 20 millisecondes). Ce délai significatif est dû au fait que chaque paquet doit être redirigé vers le contrôleur, puis vers l'hyperviseur qui exécute la fonction *data* et enfin suivre le chemin inverse vers le plan de données pour être acheminé vers l'hôte destination. En revanche, le chemin qui passe par une data machine (une fonction *data* installée au niveau du plan de données) présente de bien meilleures performances (courbe verte), presque négligeables dans notre environnement de tests (moins d'une milliseconde). Cela est explicable par le fait que tous les paquets restent dans le plan de données et n'ont pas besoin de passer par plusieurs couches logicielles afin d'atteindre la fonction *data*.

Notez que pour la réalisation des tests sur les fonctions *data*, nous avons utilisé de simples fonctions de redirection qui reçoivent un paquet sur une interface d'entrée, puis le redirige vers une interface de sortie, sans traitement sur le paquet en lui-même. La principale motivation de cette démarche est que nous voulions exclure tout facteur externe qui pourrait altérer les mesures de performances, étant donné qu'implémenter exactement la même fonction dans deux environnements différents [1] fausserait les résultats.

Un cas, que nous ne présentons pas ici, est l'installation réactive d'un chemin contenant une fonction *data*. Les résultats que nous avons obtenus sont similaires, pour le premier paquet, aux tests de la fonction de contrôle dynamique (courbe rouge) ; et pour les autres paquets, aux tests de la fonction *data* installée proactivement (courbes verte et jaune).

7.4 Bilan

Afin de fournir une validation expérimentale de notre prototype, nous avons réalisé un ensemble de tests et de mesures de performance que nous avons présentés dans ce dernier chapitre. Ces mesures concernent, d'une part, le coût supplémentaire qui est induit par les deux phases de compilation (composition virtuelle et mapping physique) et, d'autre part, l'évaluation du coût des fonctions réseau AirNet.

Ainsi, avec le premier jeu de tests, nous avons pu montrer que le coût supplémentaire induit par cette nouvelle couche d'abstraction reste parfaitement acceptable. En effet,

1. Les fonctions sur l'hyperviseur sont implémentées en utilisant Python, alors que celles sur Click sont implémentées en C++

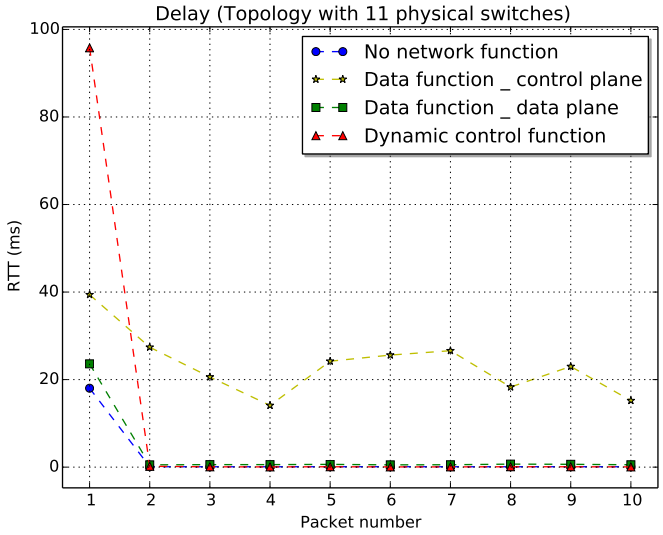

FIGURE 64. Délai de bout en bout en fonction du type de chemin

malgré la variation du nombre de politiques et l'augmentation du nombre de commutateurs physiques dans l'infrastructure, les temps de compilation sont restés, au total, inférieurs à 400 millisecondes. Dans le même temps, le nombre de règles physiques générées a dépassé les 300. Ces mesures sont encourageantes, d'autant plus si nous les confrontons aux gains apportés par l'utilisation d'AirNet en termes de simplicité et de temps de configuration pour l'administrateur.

Enfin, à travers ces premiers tests, nous avons aussi montré la capacité de l'hyperviseur d'AirNet à pouvoir gérer la configuration d'un nombre important de commutateurs OpenFlow, offrant ainsi une première étape à la validation du passage à l'échelle de l'hyperviseur. La prochaine étape sera de tester l'hyperviseur sur un nombre de commutateurs encore plus important, puis de le tester sur une infrastructure réelle afin d'obtenir des mesures qui soient les plus réalistes possibles.

Dans un second temps, nous avons détaillé les mesures de performance du coût d'exécution des fonctions réseau. Nous avons pu constater que le fait d'utiliser une fonction de contrôle dynamique pour installer des chemins de de bout en bout ne présente pas un fort impact sur les performances, étant donné que tout son coût est concentré sur le premier paquet qui est utilisé par la fonction de contrôle dynamique pour sa prise de décision.

Dans le cas des fonctions de données, le coût d'exécution est plus conséquent, particulièrement pour les fonction de données qui sont exécutées au niveau du plan de contrôle. Les performances de ces dernières dépendent fortement de l'architecture SDN mise en place, notamment le type du contrôleur SDN utilisé (distribué ou centralisé, supportant le parallélisme ou non). Les performances des fonctions exécutées dans le plan de données dépendent, quant à elles, de la capacité de traitement du service assuré par la data machine.

Conclusion générale

Rappel de la problématique

Disposer d'une *Northbound API* expressive, modulaire et flexible est une condition essentielle si nous souhaitons réaliser les objectifs premiers du paradigme SDN, à savoir rendre les architectures réseaux actuelles plus programmables afin de pallier leur complexité et rigidité architecturale. Les travaux de cette thèse s'inscrivaient dans ce contexte bien précis de proposition d'une nouvelle *Northbound API*, qui exposerait ses services sous la forme d'un langage de programmation métier moderne.

A partir d'un état de l'art sur le paradigme SDN et des langages de programmation réseau, nous avons identifié la virtualisation de réseau comme une propriété clé, qui nous permettrait d'introduire plus de modularité, de réutilisabilité et de flexibilité dans une *Northbound API*, déverrouillant ainsi les capacités d'innovation au sein même du réseau. Néanmoins, pour exploiter au mieux la virtualisation de réseau, dans ce contexte de *Northbound API*, les défis majeurs suivants doivent être résolus :

- Disposer d'un modèle d'abstraction pertinent pour cacher la nature complexe et dynamique de l'infrastructure physique. Ce modèle doit notamment assurer les propriétés d'expressivité, modularité et flexibilité des programmes de contrôle.
- Définir une approche, qui soit cohérente avec le modèle d'abstraction et, qui permette de spécifier les politiques virtuelles qui décrivent les services réseaux et leur orchestration suivant les objectifs de haut niveau de l'administrateur.
- Développer le système d'exécution nécessaire pour la composition des politiques virtuelles et leur transposition sur l'infrastructure physique.

Synthèse des contributions

Cette partie résume les quatre principales contributions apportées dans cette thèse.

Modèle d'abstraction Edge-Fabric : en partant de l'état de l'art que nous avons réalisé sur les langages de programmation réseau, nous avons constaté que les plus récents d'entre eux ont eu recours à la virtualisation de réseau afin de mieux assurer diverses propriétés telles que la montée en abstraction ou le partage des ressources. Néanmoins, nous avons aussi constaté que ces travaux n'accordaient pas assez d'importance au choix du modèle d'abstraction et de son contexte d'utilisation bien précis qui est la spécification de politiques métiers de haut

niveau. Ainsi, nous avons porté notre attention sur la proposition d'un modèle d'abstraction qui soit adéquat avec nos exigences de départ. Cette réflexion a donné lieu à la proposition du modèle Edge-Fabric, dont l'idée clé est d'introduire une claire séparation entre les politiques de transport et les services réseaux plus complexes.

Le modèle Edge-Fabric étant un modèle très connu de conception d'infrastructure physique, l'originalité de ce travail réside donc dans sa réutilisation au niveau du plan de contrôle virtualisé afin d'assurer les exigences d'expressivité, modularité et flexibilité d'une *Northbound API*. En effet, de par sa popularité, la prise en main du modèle et la spécification des politiques de contrôle sera une tâche naturelle pour les administrateurs réseaux. De plus, la séparation que maintient le modèle d'abstraction entre les différents types de politiques que peuvent contenir les programmes de contrôle, assure une meilleure lisibilité et une plus grande modularité de ces derniers. Enfin, le fait de pouvoir utiliser plusieurs composants virtuels, de types différents, permettra aux administrateurs de répondre aux différentes exigences actuelles, qu'elles soient conceptuelles (politiques de haut niveau) ou en rapport à des contraintes physiques qu'un administrateur ne souhaite pas ou ne peut pas cacher.

Le langage AirNet : Une fois le modèle d'abstraction Edge-Fabric défini, nous nous sommes alors intéressés au langage construit autour de ce dernier. Pour ce faire, nous avons, dans un premier temps, réutilisé plusieurs idées proposées par des langages existants (principalement Pyretic) que nous avons par la suite adaptées afin qu'elles soient conformes à la philosophie du modèle Edge-Fabric. Puis, dans un second temps, nous avons introduit de nouveaux concepts propres à AirNet tels que les primitives de transport ou les fonctions réseau (contrôle et données). Pour rester en cohérence avec notre vision d'un langage de programmation de haut niveau supportant de la virtualisation de réseau, nous avons construit toute la phase de conception des primitives du langage AirNet et de son modèle de programmation en veillant à respecter les exigences suivantes :

- L'utilisation de la virtualisation doit être dans une optique de simplifier la gestion d'un réseau SDN.
- Assurer une distinction claire entre les différents type de politiques d'un programme de contrôle.
- Les politiques de contrôle de haut niveau doivent être totalement découplées de l'infrastructure physique, contrairement à des travaux existants (*e.g.*, Pyretic, Merlin, Splendid Isolation), qui eux, n'assurent pas une stricte séparation.

Ainsi, AirNet est un langage de programmation de haut niveau dont le modèle de programmation consiste à définir une topologie virtuelle selon les exigences de l'administrateur, qu'elles soient conceptuelles ou physiques, puis de spécifier les politiques de contrôle virtuelles qui seront exécutées sur cette dernière. Ces politiques, sont divisées en quatre principaux types de services, suivant la nature des problèmes ciblés : politique de transport, fonction de contrôle statique, fonction de contrôle dynamique et fonction *data*. Par la suite, l'administrateur pourra installer ses services réseau sur les différents composants virtuels de sa topologie, selon une logique qui consiste à placer les services réseaux à valeur

ajoutée sur les composants virtuels de bordure (edges et data machines) et à utiliser ceux du cœur de réseaux (fabrics) pour orchestrer le passage des différents flux de paquets par ses services. Cette orchestration se fait sans devoir se soucier des détails de bas niveau, mais simplement en indiquant les nom symboliques des différents services ainsi que leur enchaînement logique.

L'hyperviseur d'AirNet : une autre contribution importante de ce travail de thèse est la conception et l'implémentation d'un système d'exécution pour le langage AirNet, que nous appelons *AirNet hypervisor*. Ce dernier répond principalement au troisième défi d'une approche basée sur la virtualisation, à savoir disposer d'un système d'exécution pour assurer, d'une part, la composition des politiques virtuelles et, d'autre part, l'installation des règles OpenFlow sur les commutateurs physiques. Pour ce faire, l'hyperviseur fournit plusieurs modules, indépendants, qui implémentent les principales fonctionnalités suivantes :

- *language* et *classifier* : implémentent les primitives du langage et permettent la composition des politiques virtuelles, indépendamment de la structure et de la complexité de l'infrastructure physique.
- *proactive core* : transforme les politiques virtuelles de haut-niveau en règles physiques de bas niveau, puis initie leur installation sur les commutateurs OpenFlow à travers le module client de l'hyperviseur.
- *reactive core* : assure la gestion de l'infrastructure physique pendant la phase d'exécution (*at runtime*) du programme de contrôle, particulièrement l'exécution des fonctions dynamiques et l'adaptation des règles physiques après un changement dans l'infrastructure physique.
- clients *POX* et *RYU*, *infrastructure* et *Proxy ARP* : modules de supports qui fournissent des fonctionnalités supplémentaires telles que la découverte de l'infrastructure physique ou la résolution des requêtes ARP.

Ainsi, pour configurer une infrastructure SDN, l'administrateur n'aura besoin de fournir en entrée à l'hyperviseur que les trois modules suivants : *i*) une topologie virtuelle pour abstraire tous les détails, non pertinents, de l'infrastructure physique, *ii*) les politiques de contrôle virtuelles à installer sur cette topologie et, enfin, *iii*) un module de mapping qui permet d'associer les composants de la topologie virtuelle aux équipements de l'infrastructure physique.

Prototypage et tests : Comme validation expérimentale du langage AirNet et de son hyperviseur, nous avons mené plusieurs tests de fonctionnement et des mesures de performances. Le chapitre 6 décrit ainsi un cas d'utilisation du début jusqu'à la fin pour illustrer les différentes étapes de la spécification jusqu'à l'exécution d'un programme AirNet. A travers ce cas d'étude, nous avons pu voir la pertinence du modèle Edge-Fabric et l'utilité d'une approche basée sur la virtualisation de réseau. En effet, les frontières logiques qu'assure le modèle Edge-Fabric ont permis d'introduire plus de lisibilité et de modularité dans les

politiques de contrôle et, par conséquent, de faciliter la possibilité de leur réutilisation. D'autre part, nous avons aussi constaté que la virtualisation de réseau a permis de déléguer une grande partie de la complexité de configuration d'une infrastructure physique vers l'hyperviseur, preuve en est la grande différence entre le nombre de politiques virtuelles spécifiées et le nombre réel de règles physiques installées par l'hyperviseur.

Afin d'investiguer plus sur le coût d'exécution de l'hyperviseur, nous avons mené des mesures, présentées dans le chapitre 7, sur les différents facteurs qui peuvent impacter les performances de ce dernier. Le premier jeu de tests a montré que, malgré la variation du nombre de politiques et l'augmentation du nombre de commutateurs physiques dans l'infrastructure, les temps de compilation sont parfaitement acceptables et restent inférieurs à 400 millisecondes. De plus, nous avons pu aussi vérifier que la phase de composition virtuelle des politiques est complètement indépendante de la taille et de la complexité de l'infrastructure physique. Le second jeu de tests, quant à lui, nous a permis de valider nos hypothèses concernant le coût d'exécution des fonctions réseau, à savoir que : *i*) le coût d'une fonction de contrôle dynamique est minime, étant donné que ce dernier est concentré sur les premiers paquets qui sont utilisés par la fonction pour sa prise de décision, *ii*) le coût d'exécution des fonctions *data* est plus important que celui des fonctions de contrôle dynamique, particulièrement pour celles qui sont exécutées au niveau du plan de contrôle, cela en raison des diverses couches logicielles que doivent traverser tous les paquets d'un flux.

Perspectives

Pour terminer, cette section discute de quelques questions ouvertes et perspectives pour ces travaux.

Modèle formel : La spécification de la syntaxe et de la sémantique du langage AirNet a été dirigée avant tout par des besoins applicatifs et non pas par un objectif de s'appuyer sur un modèle ou une base mathématique. Une première perspective consiste à définir un modèle formel pour le langage AirNet qui permettrait ainsi de vérifier la sémantique d'un programme de contrôle dès la phase de compilation.

Une première piste que nous envisageons est d'utiliser la base mathématique KAT (*Kleene algebras with tests*) [Kozen, 1997] afin de formaliser, d'une part, la définition d'une topologie virtuelle et, d'autre part, la construction et la composition des politiques de contrôle.

Vérification des règles physiques : AirNet étant un langage basé sur la virtualisation de réseau qui utilise un hyperviseur afin d'assurer la transposition des politiques virtuelles sur les commutateurs physiques, un outil de vérification de la configuration déployée est ainsi nécessaire pour vérifier que les règles installées sur le réseau physique correspondent bien aux politiques de haut niveau spécifiées par l'administrateur.

La solution que nous envisageons est de développer un outil de génération de tests automatiques. Ce dernier prendrait en entrée les politiques de haut niveau spécifiées par l'administrateur et générerait par la suite plusieurs jeux de paquets de tests qu'il

ferait transiter par le réseau. Cet outil se chargerait, dans un premier temps, de remonter automatiquement des statistiques sur les règles qui ont traité ces paquets de tests et les hôtes qui les ont réceptionnés, puis dans un deuxième temps, de vérifier que les résultats obtenus correspondent bien aux objectifs de haut niveau de l'administrateur [Zeng et al., 2012, Lebrun et al., 2014].

Déploiment des data machines : Dans l'état actuel du prototype de l'hyperviseur, nous supposons que les fonctions *data* sont déjà déployées sur les data machines. Une amélioration de cette approche serait de faire en sorte que l'hyperviseur soit capable de déployer les fonctions *data* sur des data machines vierges, suivant des contraintes de placement telles que la capacité de calculs des machines ou l'efficacité énergétique. Cette amélioration de l'hyperviseur ne nécessiterait aucune modification au niveau du modèle d'abstraction ou du langage en lui-même. En effet, cette perspective s'inscrit uniquement dans le cadre des algorithmes de mapping physique et pourra s'appuyer sur les nombreux travaux de recherche existants dans le domaine des problèmes d'optimisation [Soulé et al., 2014].

Support des nouvelles versions du standard OpenFlow : La version actuelle de l'hyperviseur se base le standard OpenFlow 1.0. Dans la suite, nous souhaitons faire en sorte que l'hyperviseur puisse supporter des versions plus récentes du protocole OpenFlow, particulièrement la version 1.3 qui permet l'utilisation de plusieurs tables de flux au sein d'un même commutateur. Cette mise à jour nous permettrait de simplifier grandement plusieurs de nos algorithmes tels que la récolte des statistiques ou la composition des politiques. Un premier pas a été fait dans ce sens par le développement d'un module d'intégration pour le contrôleur Ryu. Ce dernier supportant l'utilisation des versions les plus récentes du standard OpenFlow.

Tests de passage à l'échelle : Le chapitre 7 a présenté une première étape de tests de performances que nous avons réalisés afin de mesurer le coût de cette nouvelle couche logicielle d'abstraction. Une perspective est de réaliser des tests plus poussés concernant les capacités de l'hyperviseur à passer à l'échelle, notamment en utilisant des programmes de contrôle comprenant un plus grand nombre de politiques et des infrastructure de plus grande taille (actuellement le nombre maximum de commutateurs que nous avons utilisé est de 127). Une autre perspective liée aux tests de l'hyperviseur est d'essayer de déployer l'hyperviseur d'AirNet sur un réseau SDN réel et non pas sur un réseau émulé, afin d'obtenir des mesures qui soient les plus réalistes possibles.

Bibliographie

[802, 2010] (2010). IEEE Standard for Local and metropolitan area networks–Port-Based Network Access Control. *IEEE Std 802.1X-2010 (Revision of IEEE Std 802.1X-2004)*, pages 1–205.

[Min, 2010] (2010). Mininet Overview. http://mininet.org/overview/. Dernier accès : Juillet 2016.

[Ope, 2012] (2012). OpenFlow Switch Specication Version 1.3.0.

[POX, 2013] (2013). About POX. http://www.noxrepo.org/pox/about-pox/. Dernier accès : Juin 2016.

[Anderson et al., 2014] Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., and Walker, D. (2014). NetKAT : Semantic Foundations for Networks. *SIGPLAN Not.*, 49(1) :113–126.

[Benson et al., 2009] Benson, T., Akella, A., and Maltz, D. (2009). Unraveling the Complexity of Network Management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA. USENIX Association.

[Berde et al., 2014] Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., and Parulkar, G. (2014). ONOS : Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA. ACM.

[Bhattacharjee et al., 1997] Bhattacharjee, S., Calvert, K. L., and Zegura, E. W. (1997). *An Architecture for Active Networking*, pages 265–279. Springer US, Boston, MA.

[Brent Salisbury, 2012] Brent Salisbury (2012). The Northbound API- A Big Little Problem. http://networkstatic.net/the-northbound-api-2/. Dernier accès : Juin 2016.

[Caesar et al., 2005] Caesar, M., Caldwell, D., Feamster, N., Rexford, J., Shaikh, A., and van der Merwe, J. (2005). Design and Implementation of a Routing Control Platform. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 15–28, Berkeley, CA, USA. USENIX Association.

[Casado et al., 2010] Casado, M., Koponen, T., Ramanathan, R., and Shenker, S. (2010). Virtualizing the Network Forwarding Plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 8 :1–8 :6, New York, NY, USA. ACM.

[Casado et al., 2012] Casado, M., Koponen, T., Shenker, S., and Tootoonchian, A. (2012). Fabric : A Retrospective on Evolving SDN. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 85–90, New York, NY, USA. ACM.

[Chowdhury and Boutaba, 2009] Chowdhury, N. M. K. and Boutaba, R. (2009). Network virtualization : state of the art and research challenges. *IEEE Communications magazine*, 47(7) :20–26.

[Chowdhury and Boutaba, 2010] Chowdhury, N. M. K. and Boutaba, R. (2010). A survey of network virtualization. *Computer Networks*, 54(5) :862–876.

[Doria et al., 2010] Doria, A., Salim, J. H., Haas, R., Khosravi, H., Wang, W., Dong, L., Gopal, R., and Halpern, J. (2010). Forwarding and Control Element Separation (ForCES) Protocol Specification. RFC 5810.

[Elliott and Hudak, 1997] Elliott, C. and Hudak, P. (1997). Functional reactive animation. In *International Conference on Functional Programming*.

[Erickson, 2013] Erickson, D. (2013). The Beacon Openflow Controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA. ACM.

[Feamster et al., 2013] Feamster, N., Rexford, J., and Zegura, E. (2013). The Road to SDN. *Queue*, 11(12) :20 :20–20 :40.

[Foster et al., 2010] Foster, N., Freedman, M. J., Harrison, R., Rexford, J., Meola, M. L., and Walker, D. (2010). Frenetic : A High-level Language for OpenFlow Networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 6 :1–6 :6, New York, NY, USA. ACM.

[Gember et al., 2012] Gember, A., Prabhu, P., Ghadiyali, Z., and Akella, A. (2012). Toward Software-defined Middlebox Networking. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 7–12, New York, NY, USA. ACM.

[Ghein, 2006] Ghein, L. D. (2006). *MPLS Fundamentals*. Cisco Press, 1st edition.

[Ghodsi et al., 2011] Ghodsi, A., Shenker, S., Koponen, T., Singla, A., Raghavan, B., and Wilcox, J. (2011). Intelligent Design Enables Architectural Evolution. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 3 :1–3 :6, New York, NY, USA. ACM.

[Greg Ferro, 2012] Greg Ferro (2012). Northbound API, Southbound API, East/North – LAN Navigation in an OpenFlow World and an SDN Compass. `http://etherealmind.com/northbound-api-southbound-api-eastnorth-lan-navigation-in-an-openflow-world-and-an-sdn-compass/`. Dernier accès : Juin 2016.

[Gude et al., 2008] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. (2008). NOX : Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38(3) :105–110.

[Gutz et al., 2012] Gutz, S., Story, A., Schlesinger, C., and Foster, N. (2012). Splendid Isolation : A Slice Abstraction for Software-defined Networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 79–84, New York, NY, USA. ACM.

[Hinrichs et al., 2009] Hinrichs, T. L., Gude, N. S., Casado, M., Mitchell, J. C., and Shenker, S. (2009). Practical Declarative Network Management. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 1–10, New York, NY, USA. ACM.

[Huang et al., 2011] Huang, S. S., Green, T. J., and Loo, B. T. (2011). Datalog and Emerging Applications : An Interactive Tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1213–1216, New York, NY, USA. ACM.

[Isabelle GUIS, 2012] Isabelle GUIS (2012). The SDN Gold Rush To The Northbound API. https://www.sdxcentral.com/articles/contributed/the-sdn-gold-rush-to-the-northbound-api/2012/11/. Dernier accès : Juin 2016.

[Ivan Pepelnjak, 2012] Ivan Pepelnjak (2012). SDN Controller northbound API is the crucial missing piece. http://blog.ipspace.net/2012/09/sdn-controller-northbound-api-is.html. Dernier accès : Juin 2016.

[Jain and Paul, 2013] Jain, R. and Paul, S. (2013). Network virtualization and software defined networking for cloud computing : a survey. *IEEE Communications Magazine*, 51(11) :24–31.

[Jain et al., 2013] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., and Vahdat, A. (2013). B4 : Experience with a Globally-deployed Software Defined Wan. *SIGCOMM Comput. Commun. Rev.*, 43(4) :3–14.

[Keller and Rexford, 2010] Keller, E. and Rexford, J. (2010). The "Platform As a Service" Model for Networking. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, pages 4–4, Berkeley, CA, USA. USENIX Association.

[Kohler et al., 2000] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. (2000). The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3) :263–297.

[Koponen et al., 2010] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., and Shenker, S. (2010). Onix : A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 351–364, Berkeley, CA, USA. USENIX Association.

[Kozen, 1997] Kozen, D. (1997). Kleene Algebra with Tests. *ACM Trans. Program. Lang. Syst.*, 19(3) :427–443.

[Kreutz et al., 2014] Kreutz, D., Ramos, F. M. V., Verissimo, P., Esteve Rothenberg, C., Azodolmolky, S., and Uhlig, S. (2014). Software-Defined Networking : A Comprehensive Survey. *ArXiv e-prints*.

[Lantz et al., 2010] Lantz, B., Heller, B., and McKeown, N. (2010). A Network in a Laptop : Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19 :1–19 :6, New York, NY, USA. ACM.

[Lebrun et al., 2014] Lebrun, D., Vissicchio, S., and Bonaventure, O. (2014). Towards test-driven software defined networking. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–9.

[McKeown et al., 2008] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). OpenFlow : Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2) :69–74.

[Monsanto et al., 2012] Monsanto, C., Foster, N., Harrison, R., and Walker, D. (2012). A Compiler and Run-time System for Network Programming Languages. *SIGPLAN Not.*, 47(1) :217–230.

[Monsanto et al., 2013] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. (2013). Composing Software-defined Networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA. USENIX Association.

[Nippon Telegraph and Telephone Corporation, 2012] Nippon Telegraph and Telephone Corporation (2012). RYU network operating system. http://osrg.github.com/ryu/. Dernier accès : Juin 2016.

[ONF, 2012] ONF (2012). Software-Defined Networking : The New Norm for Networks. https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

[Pfaff and Davie, 2013] Pfaff, B. and Davie, B. (2013). The Open vSwitch Database Management Protocol. RFC 7047.

[Project Floodlight, 2012] Project Floodlight (2012). Floodlight. http://www.projectfloodlight.org/floodlight/. Dernier accès : Juin 2016.

[Raghavan et al., 2012] Raghavan, B., Casado, M., Koponen, T., Ratnasamy, S., Ghodsi, A., and Shenker, S. (2012). Software-defined Internet Architecture : Decoupling Architecture from Infrastructure. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 43–48, New York, NY, USA. ACM.

[Reitblatt et al., 2013] Reitblatt, M., Canini, M., Guha, A., and Foster, N. (2013). FatTire : Declarative Fault Tolerance for Software-defined Networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 109–114, New York, NY, USA. ACM.

[Rivka Gewirtz Little, 2013] Rivka Gewirtz Little (2013). ONF to standardize northbound API for SDN applications ? http://searchsdn.techtarget.com/news/2240206604/ONF-to-standardize-northbound-API-for-SDN-applications. Dernier accès : Juin 2016.

[Robert Sherwood, 2013] Robert Sherwood (2013). Clarifying the role of software-defined networking northbound APIs. http://www.networkworld.com/article/2165901/lan-wan/clarifying-the-role-of-software-defined-networking-northbound-apis.html. Dernier accès : Juin 2016.

[Sally Johnson, 2012] Sally Johnson (2012). A primer on northbound APIs : Their role in a software-defined network. http://searchsdn.techtarget.com/feature/A-primer-on-northbound-APIs-Their-role-in-a-software-defined-network. Dernier accès : Juin 2016.

[Scott Shenker, 2011] Scott Shenker (2011). The Future of Networking, and the Past of Protocols. https://www.youtube.com/watch?v=YHeyuD89n1Y. Dernier accès : Juin 2016.

[Sherwood et al., 2009] Sherwood, R., Gibb, G., kiong Yap, K., Casado, M., Mckeown, N., and Parulkar, G. (2009). FlowVisor : A Network Virtualization Layer. Technical report.

[Shin et al., 2014] Shin, S., Song, Y., Lee, T., Lee, S., Chung, J., Porras, P., Yegneswaran, V., Noh, J., and Kang, B. B. (2014). Rosemary : A Robust, Secure, and High-performance Network Operating System. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 78–89, New York, NY, USA. ACM.

[Song, 2013] Song, H. (2013). 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. ACM.

[Soulé et al., 2014] Soulé, R., Basu, S., Marandi, P. J., Pedone, F., Kleinberg, R., Sirer, E. G., and Foster, N. (2014). Merlin : A Language for Provisioning Network Resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 213–226, New York, NY, USA. ACM.

[Tennenhouse et al., 1997] Tennenhouse, D. L., Smith, J. M., Sincoskie, W. D., Wetherall, D. J., and Minden, G. J. (1997). A survey of active network research. *IEEE Communications Magazine*, 35(1) :80–86.

[Tootoonchian et al., 2012] Tootoonchian, A., Gorbunov, S., Ganjali, Y., Casado, M., and Sherwood, R. (2012). On Controller Performance in Software-defined Networks. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pages 10–10, Berkeley, CA, USA. USENIX Association.

[Voellmy et al., 2010] Voellmy, A., Agarwal, A., and Hudak, P. (2010). Nettle : Functional Reactive Programming for OpenFlow Networks. Technical Report YALEU/DCS/RR-1431, Yale University.

[Voellmy et al., 2012] Voellmy, A., Kim, H., and Feamster, N. (2012). Procera : A Language for High-level Reactive Network Control. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 43–48, New York, NY, USA. ACM.

[Voellmy et al., 2013] Voellmy, A., Wang, J., Yang, Y. R., Ford, B., and Hudak, P. (2013). Maple : Simplifying SDN Programming Using Algorithmic Policies. *SIGCOMM Comput. Commun. Rev.*, 43(4) :87–98.

[Wetherall et al., 1998] Wetherall, D. J., Guttag, J. V., and Tennenhouse, D. L. (1998). Ants : a toolkit for building and dynamically deploying network protocols. In *Open Architectures and Network Programming, 1998 IEEE*, pages 117–129.

[Zeng et al., 2012] Zeng, H., Kazemian, P., Varghese, G., and McKeown, N. (2012). Automatic Test Packet Generation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 241–252, New York, NY, USA. ACM.

Publications

- ***AirNet : the Edge-Fabric model as a virtual control plane*** (regular paper). Messaoud Aouadj, Emmanuel Lavinal, Thierry Desprats, Michelle Sibilla. SWFAN (workshop INFOCOM 2016), San Francisco, USA, 11/04/2016, IEEE, p. 743-748, avril 2016 (**Best Paper Award**).
- ***Composing data and control functions to ease virtual network programmability*** (regular paper). Messaoud Aouadj, Emmanuel Lavinal, Thierry Desprats, Michelle Sibilla. IEEE/IFIP Network Operations and Management Symposium (NOMS 2016) : Mini-Conference, Istanbul, Turkey, 25/04/2016-29/04/2016, IEEE, p. 461-467, avril 2016.
- ***Demo : AirNet in action*** (short paper). Messaoud Aouadj, Emmanuel Lavinal, Thierry Desprats, Michelle Sibilla.IEEE/IFIP Network Operations and Management Symposium (NOMS 2016) : Demonstration Session Paper, Istanbul, Turkey, 25/04/2016-29/04/2016, IEEE, p. 997-998, avril 2016.
- ***Towards a Virtualization-based Control Language for SDN Platforms*** (short paper). Messaoud Aouadj, Emmanuel Lavinal, Thierry Desprats, Michelle Sibilla. International Conference on Network and Service Management (CNSM 2014), Rio de Janeiro, Brazil, 17/11/2014- 21/11/2014, IFIP, p. 324-327, novembre 2014.
- ***Towards a Modular and Flexible SDN Control Language*** (regular paper). Messaoud Aouadj, Emmanuel Lavinal, Thierry Desprats, Michelle Sibilla. Global Information Infrastructure and Networking Symposium (GIIS 2014), Montreal, Canada, 15/09/2014-18/09/2014.