UNIVERSITÉ DU QUÉBEC À MONTRÉAL

NETLANG : UN LANGAGE DE HAUT NIVEAU POUR LES ROUTEURS PROGRAMMABLES DANS LE CONTEXTE DES RÉSEAUX SDN

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

BOCHRA BOUGHZALA

JUILLET 2013

REMERCIEMENTS

Tout d'abord, je tiens à remercier mon directeur de recherche principal, le Professeur Omar Cherkaoui ainsi que mon codirecteur de recherche, le Professeur Etienne M. Gagnon, l'un autant que l'autre, d'avoir accepté de diriger mes travaux de recherche dans le cadre de ma maîtrise en informatique.

Je remercie Prof. Omar Cherkaoui de m'avoir apporté des idées particulièrement intéressantes, originales et innovantes, et de m'avoir placée sur des pistes enrichissantes. Je le remercie également pour les ressources qu'il a mises à ma disposition et qui étaient nécessaires à la réalisation de mon projet de maîtrise. Je le remercie également d'avoir suivi mon avancement et pour le temps qu'il m'a consacré ainsi que les directives qu'il m'a données afin de m'orienter. Je veux lui exprimer ma gratitude pour ses remarques pertinentes, la qualité de l'enseignement qu'il m'a offert en matière des réseaux haut-débits et des routeurs programmables, la disponibilité dont il a fait preuve et les efforts qu'il a fournis afin de m'expliquer aussi bien des concepts techniques que des notions théoriques.

Je remercie également Prof. Etienne Gagnon de m'avoir aidée à acquérir des connaissances nouvelles en matière de compilation et de machines virtuelles. Il m'a également été d'une grande aide pour m'initier à la recherche et me familiariser aux publications. Il m'a aussi donné son support pour avoir des méthodes scientifiques et une discipline de travail. Je le remercie également pour tout le temps et l'énergie qu'il a investis pour suivre mon avancement et me donner ses remarques pertinentes. Enfin, je tiens à lui exprimer ma profonde gratitude de m'avoir soutenue jusqu'au bout avec ses encouragements sincères.

Je tiens aussi à remercier Ericsson pour les ressources humaines ainsi que son support financier. Je remercie particulièrement Monsieur Yves Lemieux, le coordinateur du projet NetVirt dans lequel mon projet de maîtrise s'est déroulé, pour l'attention et le sens de l'écoute dont il a fait preuve, ainsi que ses efforts pour organiser et animer des rencontres permettant des échanges intéressants avec les gens d'Ericsson. Eric Dyke et Geoffrey Lefebvre sont parmi les personnes que je tiens particulièrement à remercier pour les remarques et les critiques constructives qu'ils m'ont données.

Je ne voudrais pas, non plus, oublier Madame Claudie Dufour Landry, l'assistante du Prof. Omar Cherkaoui, et le reste de l'équipe du laboratoire LTIR, aussi bien les étudiants de maîtrise que les stagiaires et les chercheurs postdoctoraux dont spécialement Madame Imen Limame pour les échanges enrichissants et les discussions fructueuses.

Mes remerciements sont également adressés à l'ensemble du corps enseignant ainsi qu'à tout le personnel de l'Université du Québec à Montréal (UQAM) et tous ceux qui ont participé de façon directe ou indirecte au bon déroulement de cette maîtrise.

Enfin, je présente mes remerciements aux membres du jury pour leur participation à l'évaluation de ce mémoire.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LEXIQUE

ACL (Access Control List) : liste de contrôle d'accès

Action Bucket : ensemble d'actions

Backbone Network : réseau fédérateur

Back-End : la partie bas niveau d'un compilateur

Best Matching : meilleure correspondance

Binder : un composant qui établit des liens

Binding : mise en lien

Bridging : un service qui permet de passer d'un réseau à un autre

Buffer : mémoire tampon

Candidate Configuration DataStore : base de données des configurations candidates

CPU (Central Processing Unit) : unité de traitement centrale

Checksum : empreinte qui permet de vérifier qu'un paquet est reçu sans erreurs

Cluster : un ensemble de micro-moteurs

Compound Element : élément composé

Configuration Data : données de configuration

Configuration Generator : générateur de configuration

CE Control Element : élément de contrôle

CP Control Plane : plan de contrôle ou plan de commande

Counters : compteurs

Customer Network : réseau client

Data Base : base de données

Data Path : chemin des données

DP (Data Plane) : plan de données

DPI (Deep Packet Inspection) : une application réseau qui vise à faire une analyse profonde du paquet, généralement appliquée pour des fins de sécurité informatique

Delete : supprimer

Direct Access Table : table à accès directe

DSL (Domain Specific Language) : langage à un domaine spécifique

Downstream : dans le sens descendant

Egress : à la sortie

Element : élément

Entries Generator : générateur d'entrées

ECC (Error Correcting Code) : code corresteur d'erreur

Field : champ dand l'entête du paquet

Find : trouver

Firewall : pare-feu

First Match : première correspondance

Flow : flux de paquets

Flow Classification Library : bibliothèque de classification de flow

Flow Entry : entrée pour un flux de paquets

Flow Table : table de flux, table à plusieurs entrées où chaque entrée est associé à un flux de paquets

Forwarding : transmission de paquets, un seul terme qui nous permet de désigner une commutation au niveau 2 ou un routage au niveau 3.

FE (Forwarding Element) : élément de transmission

FE (Forwarding Engines) : moteurs de transmission

Forwarding Entries : règles de transmission

Forwarding Table : table de transmission (table de commutation ou table de routage)

Forwarding Model : modèle de transmission

Framer : délimiteur de trames

Framing : mise en trames

Front-End : partie frontale d'un compilateur

Full Match : correspondance intégrale

General Purpose Processors : processeur à usage général

General Purpose Register : registre à usage général

Group Entry : entrée de groupes

Group Table : table de groupes

HAL (Hardware Abstraction Layer) : couche d'abstraction matérielle

HSL (Hardware Specific Layer) : couche dépendante du matériel

Header : en-tête de paquet, un champ ayant une valeur significative mais qui ne fait pas parti des données utiles du paquet

Ingress : à l'entrée du routeur

Insert : insérer

ITU (International Telecommunication Union) : union internationale de télécommunication

Line Card : carte de ligne, carte électronique

Loader : un chargeur

Local Memory : mémoire locale

LFB (Logical Functional Block) : bloc fonctionnel logique

LPM (Longest Prefix Match) : le préfixe de correspondance le plus long

Lookup : recherche de données dans une table

Loop Avoidance : prévention de boucle

MAC Learning : apprentissage des adresses MAC

Management Controller : contrôleur pour la gestion

Management Plane : plan de gestion

Mapping : mise correspondance

Masked Matching : correspondance avec des masques

Match : correspondance

Match Field : champ de correspondance

Match Lines : ligne d'égalité

Microcode Generator : générateur de code assembleur

Microengine : micro-moteur, l'équivalent d'un coeur dans un processeur multicoeurs

Mismatch : inégalité

Modify : modifier

Multicast : envoi de plusieurs copies d'un même paquet à plusieurs destinations

Multithreading : exploitation de plusieurs instances de petit processus s'exécutant de manière asynchrone et partagent les ressources du processus hôte

NE (Network Element) : élément réseau par exemple un routeur ou un commutateur

NFP (Network Flow Processor) : processeur des flux réseau

NIDS (Network Intrusion Detection System) : système de détection d'intrusion dans un réseau

NOS (Network Operating System) : système d'exploitation des réseaux

NP (Network Processor) : processeur réseau

NPsl (Network Processor Script Language) : un langage de script pour le processeur réseau

NPU (Network Processing Unit) : unité de traitement pour le réseau

Network Provider : fournisseur de réseau

Next Neighbor Register : registre pour communiquer avec le processeur adajacent

OpenFlow Port : port OpenFlow, point d'entrée et de sortie des paquets

OpenFlow Queue : file d'attente pour l'ordonnacement des paquets

OpenFlow Switch : équipement réseau qui peut aussi bien fonctionner au niveau 2 (couche liaison de données) qu'au niveau 3 (couche réseau) et 4 (couche transport)

ONF (Open Networking Foundation) fondation des réseaux ouverts

OSI (Open Systems Interconnection) : interconnexion des systèmes ouverts

Optional : optionnel

Packet Information Block : bloc contenant les informations sur le paquet

Packet Processing : traitement de paquets

PPP (Packet Processing Path) : un chemin de traitement de paquet

Parsing : analyse du contenu d'un paquet

Partial Matching : correspondance partielle

Path : chemin

Pop : jeter

Prioritizer : un composant qui instaure un système de priorité

Processing traitement

Pull : tirer

Rank : index

Record : un enregistrement

Running Configuration DataStore : base de données de la configuration en exécution

Push : pousser

Queueing : mise en file d'attente

Queue Manager : gestionnaire des files d'attente

Required : obligatoire

Ring : un type particulier de mémoire en anneau

Ring Manager : gestionnaire d'un type particulier de mémoire en anneau

Route Controller : contrôleur pour les routes

Routing : routage

Scope : portée

Search : recherche

Search Line : ligne de recherche

Searchset : une table de recherche

Search Structure Generator : générateur des structures des recherches

Secure Channel : canal sécurisé

SDN (Software Defined Networks) : réseaux définis de manière logicielle

Startup Configuration DataStore : base de données qui contient la configuration de l'équipement réseau

State Data : donnée d'état

Switch : commutateur, équipement réseau de niveau 2

Switch Fabric : commutateur qui envoie le trafic d'une carte de ligne à une autre

Tag : étiquette

Tagging : ajout d'étiquettes

TOP (Task Optimized Processor) : processeur à tâches spécialisés

Task Queue : file des tâches

TCAM (Ternary Content-addressable Memory) : mémoire addressable par le contenu

Thread : petit processus s'exécutant de manière asynchrone et partagent les ressources du processus hôte

Threading : utilisation de petit processus s'exécutant de manière asynchrone et partageant les ressources du processus hôte

TTL (Time-To-Live) : un champs dans un paquets IP qui indique le nombre de saut restant pour le paquet avant qu'il soit supprmé du réseau dans lequel il circule

Token : jeton

TM (Traffic Manager) : gestionnaire de trafic

Transceiver : émetteur-récepteur

Transfer Register : registre de transfert

Tunneling : un service qui permet de créer des tunnel en isolant un trafic allant d'un réseau à un autre

Upstream : dans le sens montant

Wildcards : des bits qui peuvent être à la fois 0 et 1

RÉSUMÉ

Développer des applications réseaux pour des routeurs programmables basés sur les *Network Processors* (NPs) implique l'utilisation de langages de bas-niveau et d'outils propriétaires fortement dépendants des architectures matérielles sous-jacentes. Le code source, généralement écrit en langage assembleur, n'est pas facile à écrire et cause des problèmes de maintenance. Les applications résultantes sont également difficiles à déboguer.

Dans ce mémoire nous proposons NETLANG, un nouveau langage de programmation de haut-niveau dédié aux NPs. De plus d'être un langage simple et élégant, de réduire les coûts de développement et de la maintenance, et d'améliorer la réutilisation du code, NETLANG a pour objectif essentiel de décrire le comportement des paquets dans un NP. NETLANG est un langage qui permet de développer des applications de traitement de paquets. Il établi deux niveaux. Le premier niveau du langage offre une abstraction et une description du routeur à travers un pipeline de tables OpenFlow et des règles de *forwarding* ayant l'aptitude d'être modifiées dynamiquement et donc de permettre de changer le comportement du routeur à la volée. La sémantique du langage est inspirée du protocole OpenFlow qui a permis d'exprimer les principales tâches de traitement de paquets telles que le *parsing*, le *lookup* et la modification. Le langage est bâti en respectant le modèle des *Software Defined Networks* (SDNs) qui définit un nouveau plan de séparation entre le *control plane* et le *data plane*. Le deuxième niveau de NETLANG est traduit en matériel et permet l'adaptabilité du langage à plusieurs plateformes. Des adaptateurs spécifiques à des plateformes différentes sont intégrés au compilateur de NETLANG et permettent de rendre le langage portable. En effet, nous avons utilisé deux environnements pour l'implémentation de NETLANG ; le NP4 d'EZchip caractérisé par sa structure de TOPs (*Task Optimized Processors*) en pipeline et le NFP-3240 de Netronome connu pour son parallélisme et l'exploitation du *multithreading*. La validation de NETLANG s'est basée sur un ensemble d'applications réseau ayant des complexités et des domaines différents.

A travers ce mémoire nous avons démontré qu'on est capable d'avoir aujourd'hui un langage pour les routeurs programmables. La sémantique d'OpenFlow, sur laquelle nous avons basé notre langage NETLANG, est suffisante et même pertinente en termes de description de comportement des paquets dans un NP.

Mots clés : langages à domaine spécifique, réseaux programmables, processeurs de réseau.

INTRODUCTION

Les routeurs programmables ont connu plusieurs évolutions. La première génération des routeurs supportait des applications qui s'exécutaient sur des *General Purpose Processors* (GPPs). Il était facile d'ajouter de nouvelles fonctionnalités à travers de simples mises à jour des applications et du système. Ces architectures étaient suffisantes pour les besoins en bande passante durant cette période. Avec l'accroissement exponentiel du trafic sur le réseau d'Internet, l'architecture des GPPs ne peut plus satisfaire les nouveaux besoins en termes de bande passante. Ainsi, les processeurs ASICs (*Application Specific Integrated Circuits*) ont été introduits pour remédier à ce problème. Ce qui nous amène aux routeurs de deuxième génération. Avec les processeurs ASICs, les opérations qui ont des besoins critiques en performance ont été séparées des autres opérations pour être réalisées par des processeurs dédiés. Bien que les processeurs ASICs aient une bonne performance, ils présentent l'inconvénient d'avoir un temps de commercialisation élevé (*time-to-market*) d'une part, et d'être rigide d'autre part, dans le sens où ils ne peuvent être utilisés que pour la fonctionnalité pour laquelle ils ont été conçus. Par ailleurs, les réseaux d'aujourd'hui accueillent de plus en plus de nouveaux protocoles qui sont de plus en plus complexes et sophistiqués tels que les protocoles IPv6, VoIP, IPsec, VPN, etc. Afin de répondre à ces nouvelles exigences, le besoin des opérateurs en solutions flexibles et performantes, ayant un temps de commercialisation (*time-to-market*) réduit et pouvant être déployées assez longtemps sur le marché (*time-in-market*), s'est accru et a conduit à la troisième génération de routeurs avec la conception d'un nouveau type de processeurs pour le traitement des paquets à haut-débit qui sont les *Network Processors* (NPs). "Un NP est un processeur programmable spécifique pour les applications réseaux" [CL07]. Le *Network Processing Forum* (NPF) a été fondé en 2001 dans le but d'accélérer le processus de standardisation des NPs. Avec l'établissement du NPF, un ensemble de caractéristiques communes aux NPs a été défini.

Cependant, les concepteurs de NPs ont adopté, chacun, leur propre architecture en proposant des outils spécifiques pour chaque type de plateforme. Certains vendeurs proposent des architecures exploitant le parallèlisme et utilisant des processeurs à jeu d'instructions réduit avec un support pour le *multithreading*. D'autres fabricants de NPs utilisent des architectures en *pipeline* basées sur des processeurs à jeu d'instructions spécialisé et hautement optimisé. Les

langages de programmations utilisés sont des langages propriétaires qui varient entre langages dans le style assembleur et langages dans le style C. La multiplicité des architectures des NPs et la diversité des outils et langages proposés pour la programmabilité des NPs ont été des facteurs qui ont joué contre l'évolution des NPs. Les paradigmes sont tellement divergents que cela implique un temps d'apprentissage important pour qu'un programmeur d'applications réseaux soit apte à écrire du code qui s'exécutera sur tel ou tel environnement. Les coûts de développement sont importants et les coûts de maintenance sont également élevés. De plus, le code produit n'est pas réutilisable et les applications sont difficiles à déboguer.

Nous nous sommes donc demandés s'il était possible d'avoir, aujourd'hui, un langage de programmation pour les NPs qui soit de haut-niveau et, surtout, qui soit assez générique en termes de description de traitements sur les paquets pour qu'il puisse être adapté à des plateformes matérielles différentes.

Nous nous sommes alors intéressés aux *Software Defined Networks* (SDNs), un nouveau paradigme qui encourage la programmabilité des réseaux et qui cherche à introduire de la flexibilité dans les routeurs d'aujourd'hui. SDN est une vision qui a été introduite par une implémentation concrète d'un protocole novateur appelé OpenFlow [MAB+08]. OpenFlow offre une abstraction de description du comportement des paquets dans un routeur et propose un mécanisme simple et original pour permettre la définition de divers types d'applications réseaux. En effet, OpenFlow utilise un *pipeline* de tables qui contiennent des règles. L'ensemble de ces règles forme la stratégies de *forwarding* qui sera appliquée sur les paquets.

En tenant compte des fonctionnalités et des aptitudes des NPs, et en utilisant le modèle d'OpenFlow, nous proposons NETLANG (*Network Programming Language*), un nouveau langage de programmation de haut-niveau dédié au développement des applications de traitement des paquets dans des environnements de NPs. A travers ce mémoire, nous allons évaluer si OpenFlow est suffisant en terme de description de comportement des paquets à l'intérieur d'un routeur programmable pour permettre la définition d'applications réseaux complètes.

Dans ce mémoire, nos principales contributions sont (1) L'étude des fonctionnalités internes des routeurs programmables et l'analyse des architectures des NPs qui a permis de voir les différences qui peuvent exister d'un vendeur à un autre. (2) La réalisation d'une étude sur l'état de l'art des réseaux SDNs et des solutions de programmabilité de réseaux qui ont été récemment proposées. (3) La définition d'un nouveau langage de programmation que nous

avons appelé NETLANG, un langage qui respecte les principes des réseaux SDNs et qui exploite et utilise les fonctionnalités des routeurs programmables. (4) L'implémentation du compilateur pour NETLANG afin de pouvoir le traduire vers le langage bas-niveau spécifiquement utilisé dans la plateforme cible. (5) La validation de l'expressivité et de l'adaptabilité du langage à travers deux plateformes de NPs distincts et des exemples d'applications réseaux pertinentes.

Ce mémoire est constitué de cinq chapitres. Le premier chapitre a pour objectif de présenter les routeurs programmables généralement et les NPs spécifiquement. Nous présenterons l'architecture typique d'un routeur de la nouvelle génération et nous expliquerons les fonctionnalités de base d'un NP. Nous expliquerons, également, les opérations principales de traitement de paquets et nous donnerons, à la fin, deux exemples de NPs différents.

Ensuite, dans le deuxième chapitre, nous allons présenter les différentes solutions de programmabilité des réseaux dont principalement SDNs et OpenFlow. Nous présenterons également d'autres langages et outils de programmations qui ont été proposés pour les réseaux.

Le troisième chapitre porte sur la description de NETLANG. Dans de ce chapitre, nous allons voir comment NETLANG exploite la sémantique d'OpenFlow pour pouvoir définir un routeur à trois niveaux : le *control plane*, le *data plane* et le *management plane*. Ce chapitre, présentera les fonctionnalités de NETLANG, la structure d'un programme NETLANG ainsi que ses principaux types de données et primitives. Nous expliquerons également l'implémentation de NETLANG et nous présenterons l'architecture de son compilateur.

Le quatrième chapitre a pour objectif de mettre l'accent sur l'adaptabilité de NETLANG pour différentes plateformes matérielles. Dans ce chapitre, nous allons présenter les spécificités de chaque adaptateur qui sont strictement liées à l'environnement cible. Nous expliquerons le fonctionnement de l'adaptateur pour le NP d'EZchip, ensuite nous présenterons l'adaptateur pour Netronome.

Avant de donner la conclusion générale de ce mémoire, nous présenterons, dans un cinquième chapitre, des exemples d'applications réseaux exprimées avec NETLANG, ce qui nous permettra de valider le langage et de donner un résumé de nos résultats.

CHAPITRE I

LES ROUTEURS PROGRAMMABLES

1.1 Introduction

L'objectif de ce premier chapitre est d'expliquer la dynamique des routeurs programmables, généralement, et des *Network Processors*, particulièrement. Suite à ce chapitre, nous saurons cerner les opérations principales de traitement de paquets ainsi que les stratégies logicielles et les ressources matérielles qui y interviennent. Afin de mieux comprendre les plateformes de traitement de paquets, nous donnerons, à la fin de ce chapitre, deux exemples d'architectures de NPs différents que nous utiliserons, justement, pour l'implémentation de NETLANG.

1.2 Les routeurs programmables

Pour expliquer le fonctionnement des routeurs programmables, nous allons d'abord présenter leur architecture et expliquer le rôle de chacune de leurs composantes. Nous présenterons ensuite les principales tâches qui sont réalisées dans le processus de traitement de paquets.

1.2.1 Architecture d'un routeur programmable

L'architecture typique d'un routeur programmable contient principalement plusieurs *line cards* qui sont connectées à travers un *switch fabric* (voir Figure 1.1) [CL07]. Ces *line cards* sont en effet des cartes électroniques qui présentent les points d'entrée et de sortie des paquets. Ils fournissent l'interface entre le port physique du routeur et le *switch fabric*. Le routeur contient également un contrôleur de route (*Route Controller*) et un contrôleur de gestion (*Management Controller*). Ces deux contrôleurs sont également connectés au reste des composants du routeur à travers le *switch fabric*.

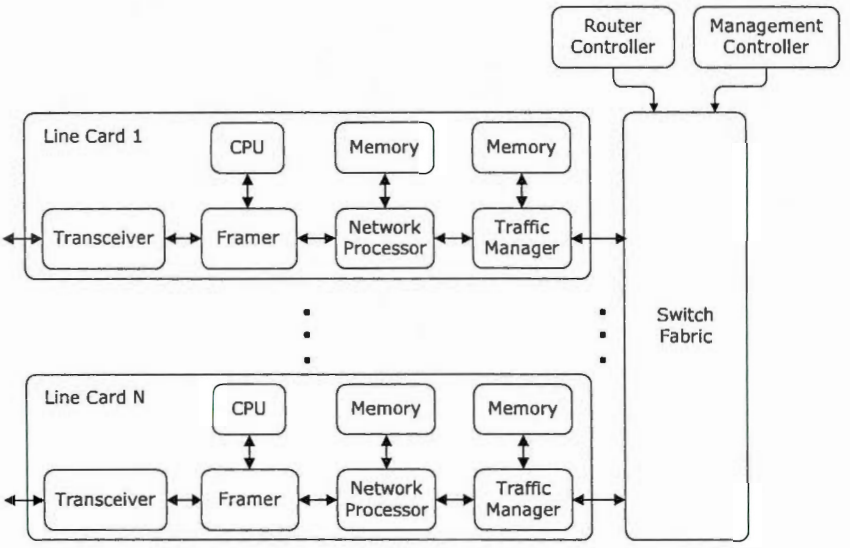

Figure 1.1 Architecture typique d'un routeur programmable [CL07]

Un *line card* est constitué généralement des composants suivants. (1) Un *Transceiver* responsable de la conversion des signaux comme, par exemple, la conversion d'un signal optique en un signal électrique ou la conversion d'un signal série en un signal parallèle. (2) Un *Framer* qui gère la synchronisation et la délimitation des trames, la gestion de la parité et la détection d'alarme. (3) Un *Network Processor* qui assure des opérations de recherche dans les tables de routage par exemple, l'analyse des paquets, la classification et la modification des paquets. Le NP réalise ces opérations en utilisant des mémoires internes ou des mémoires externes comme par exemple la SRAM (*Static Random Access Memory*) ou la DRAM (*Dynamic Random Access Memory*). Le NP peut également nécessiter une TCAM (*Ternary Content Addressable Memory*) ou des coprocesseurs spécialisés pour effectuer une classification des paquets plus raffinée. On peut avoir plus d'un seul NP sur un même *line card*. (4) Un *Traffic Manager* (TM) qui permet de répondre aux besoins de chaque connexion et classe de service. Le TM assure plusieurs fonctions de contrôle de flux des paquets, telles que le contrôle d'accès, la gestion des *buffers*, l'ordonnancement des paquets. Quand les *buffers* atteignent un certain seuil, c'est le TM qui assure la gestion des *buffers* pleins en supprimant des paquets tout en respectant leurs niveaux de priorité. (5) Un *Central Processing Unit* appelé aussi *Host* qui assure des fonctions nécessaires au traitement de paquets telles que la mise à jour des tables de *forwarding*. Le CPU ne fait pas parti du chemin de traitement sur lequel le maximum de la bande passante du trafic réseau se déplace entre les interfaces physiques et le *switch fabric* [CL07].

1.2.2 Fonctions d'un routeur programmable

Un routeur offre un ensemble de fonctionnalités qui peuvent être classées en deux catégories : les fonctions de *data plane* et les fonctions de *control plane*. Avant d'expliquer la différences entre ces deux types de fonctions, il est important de faire la distinction entre la notion de tâches, la notion de chemins et la notion de directions (voir Figure 1.2). En effet, le traitement de paquets dans un routeur peut suivre l'un des deux chemins suivants (1) le *data path* qui est le chemin rapide et qui englobe toutes les fonctions de *data plane* ou (2) le *control path* qui est le chemin lent et qui traite les tâches de *control plane*. De plus de ces deux chemins que le paquet peut emprunter, il peut suivre l'une des deux directions suivantes (1) la direction *ingress* pour les paquets en provenance du réseau et (2) la direction *egress* pour les paquets sortant de l'équipement. Les tâches de traitement de paquets sont principalement [Gil08]:

- La mise en trame (*framing*)
- L'analyse et l'extraction des champs d'entête du paquet (*parsing*)
- La classification des paquets selon les valeurs des champs d'entête
- Le *lookup* dans les tables de recherche
- La transmission des paquets (*forwarding*)
- La compression/décompression du contenu du paquet
- Le cryptage (chiffrement)/décryptage du paquet
- La gestion du trafic (*traffic management* et *queueing*)

1.2.2.1 Fonctions de *data plane*

Les fonctions de *data plane* sont effectuées sur chaque paquet qui passe par le routeur. Elles sont assurées par le NP. Par exemple, si on considère une application de routage IP, quand un paquet arrive, son adresse IP destination est d'abord appliquée au masque de sous-réseau par une opération 'ET' logique et l'adresse résultante est ensuite utilisée pour le *lookup* dans la table de routage qui fait correspondre à chaque adresse IP destination, un numéro de port de sortie. Une stratégie de recherche appelée *Longest Prefix Match* (LPM) est utilisée pour trouver le port de sortie. Dans certaines applications, les paquets sont classifiés en se basant

sur une séquences de bits qui comprend les adresses IP source et destination, les numéros de port source et destination de la couche transport et le type du protocole. Cette classification est généralement appelée la classification 5-uplets (*5-tuple classification*). En se basant sur le résultat de la classification, les paquets peuvent être soit rejetés (dans une application *firewall*) ou traités à différents niveaux de priorités. Ensuite, la valeur du champ TTL (*Time-To-Live*) est décrémentée et un nouveau *Checksum* est recalculé [CL07].

1.2.2.2 Fonctions de *control plane*

Les fonctions de *control plane* se font moins souvent que les fonctions de *data plane*, elles sont effectuées relativement rarement et elles sont généralement assurées par le CPU du Host [CL07]. Ces fonctions comprennent la configuration du système, la gestion et l'échange d'information sur les table de *forwarding*. Un contrôleur échange des informations sur la topologie du réseau avec les autres routeurs et construit une table de routage en se basant sur un protocole de routage, tel que le protocole RIP (*Routing Information Protocol*) [Hed88], le protocole OSPF (*Open Shortest Path Bridging*) [Moy09] ou le protocole BGP (*Border Gateway Protocol*) [RL09]. Le contrôleur peut également créer lui même la table de *forwarding* en utilisant une technique appelée *learning* [CL07]. Comme les fonctions de *control plane* ne sont pas effectuées à chaque arrivée de paquet, elles n'ont donc pas de contraintes de vitesses et sont généralement implémentées en logiciel.

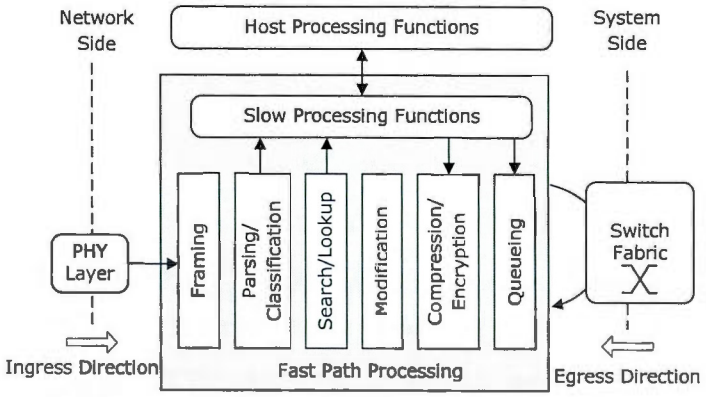

Figure 1.2 Framework de traitement de paquets [Gil08]

La figure 1.2 représente un framework de traitement de paquets dans un routeur programmable. Les paquets arrivent sur le module de la couche physique, dans la direction *ingress*, en provenance du réseau. Ensuite, ils passent soit par le *slow path* (*control plane*) où ils sont

redirigés vers le *Host* pour un traitement de niveau supérieur comme par exemple la mise à jour des tables de routage du NP, soit par le *fast path* en passant à travers les fonctions de recherche et de modification dans le NP. Les paquets sont ensuite transmis au *switch fabric* ou au réseau pour être expédiés dans la direction *egress*.

1.2.3 Le processus de traitement de paquets

Le processus de traitement de paquets commence par l'arrivée du paquet et sa mise en trame (*framing*) immédiate [Gil08]. La fonction de *framing* permet de s'assurer que le paquet est reçu correctement. Dans la direction *egress*, la fonction de *framing* est la dernière tâche réalisée. Ensuite, une classification du paquet est réalisée en se basant sur une analyse plus ou moins complexe de son entête [Gil08]. La tâche d'analyser et de classifier le paquet signifie tout simplement que le routeur doit comprendre le contenu du paquet, reconnaitre le type de paquet et ensuite il doit le classifier selon les besoins de l'application. En général, les fonctions de classification requièrent des opérations de recherche. La dernière tâche que le NP exécute est la modification du paquet, ce qui comprend la suppression du paquet si nécessaire, la duplication du paquet (pour une application de *multicast* par exemple), et la modification de son entête et/ou de son contenu. Enfin, la transmission du paquet peut impliquer d'autres fonctions de *queueing*, de priorisation et de gestion de trafic pour s'assurer que le récepteur du paquet va le recevoir selon le modèle de trafic attendu. Les tâches de *queueing* et de *traffic management* peuvent s'appliquer à l'intérieur ou à l'extérieur du NP dépendamment de l'architecture du vendeur [Gil08]. Les tâches de compression et de cryptage de paquet sont des tâches optionnelles que le paquet peut subir et, généralement, elles sont effectuées à l'extérieur du NP.

1.3 Les *Network Processors*

1.3.1 Caractéristiques des NPs

Un NP est typiquement utilisé dans un routeur programmable de troisième génération. Les NPs sont des processeurs multi-coeurs dont chaque coeur est appelé élément de traitement ou *Processing Element* (PE). Un NP possède 3 interfaces (1) une interface physique par laquelle les paquets vont arriver, (2) une interface vers le *switch fabric* ou un port physique vers lequel les paquets vont être dirigés vers la sortie (3) et enfin une interface avec le *Host* [Gil08].

- Architecture et jeu d'instruction. La plupart des NPs adoptent l'architecture RISC (*Reduced Instruction Set Computing*). Certains vendeurs introduisent des optimisations dans leur jeu d'instruction pour réduire le nombre de cycles horloge requis pour une opération donnée, comme par exemple le décodage d'une adresse IP. Toutefois, un jeu d'instructions optimisé est une arme à double tranchant. En effet, comme chaque vendeur développe son propre jeu d'instruction, le développement d'une application réseau devient un point critique surtout avec le manque de support [CL07].

- Les unités fonctionnelles et les coprocesseurs. Les unités fonctionnelles spécialisées sont fournies à l'intérieur du NP pour les opérations les plus communes des réseaux [CL07]. Ce sont des opérations faciles à implémenter matériellement mais qui peuvent entraîner des erreurs si elles sont codées manuellement, comme le calcul du *Cyclic Redundancy Check* (CRC) et du *Checksum*. Une unité fonctionnelle est utilisée pour une opération simple. Elle produit un résultat qui ne peut pas être partagé à travers plusieurs PEs. Un coprocesseur est une autre forme de composant qui est aussi très utilisé. Contrairement aux unités fonctionnelles, les coprocesseurs effectuent souvent des tâches complexes et peuvent être accédés par plusieurs PEs [CL07]. La plupart des NPs sont dotés de coprocesseurs externes de classification, de *trafic management* ou de sécurité (*Deep Packet Inspection*, cryptage et décryptage) [CL07].

- Le parallélisme. Le traitement de paquet est caractérisé intrinsèquement par un parallélisme des données qui peut être exploité [CL07]. Le parallélisme existe à deux niveaux (1) entre les paquets et (2) à l'intérieur des paquets. Si on considère un flux de paquet, la plus grande partie du traitement est effectuée par paquet où chaque paquet va suivre un traitement similaire mais indépendant [CL07]. C'est la caractéristique principale du parallélisme entre les paquets. De plus, à l'intérieur de la suite des tâches à appliquer sur le paquet, généralement certaines tâches sont indépendantes. C'est le parallélisme à l'intérieur du paquet.

- Le support logiciel. L'avantage des NPs dépend fortement du support en termes de logiciel pour fournir un environnement convivial qui facilite la programmation [CL07]. Ceci réduit le temps de développement et accélère le temps de commercialisation. Le processus est plus difficile que les autres produits de communication à cause de l'architecture parallèle des NPs, l'utilisation d'applications non standardisées, des jeux d'instructions spécialisés et propriétaires [CL07].

1.3.2 Fonctionnalités des NPs

Certains NPs doivent effectuer des opérations à usage général dans le cas où ils ne sont pas connectés à un processeur *Host*. Cependant les tâches typiques d'un NP sont le *parsing*, le *lookup*, la classification, la modification, le *queueing* et le *traffic management* [Gil08].

1.3.2.1 Le *parsing*

Chaque paquet entrant doit aller à travers une analyse pour que le NP examine et comprenne quel type de paquet c'est et quels sont ses besoins [Gil08]. Ensuite, il doit être classifié et traité selon son type et le traitement requis. Le *parsing* présente ainsi la première action qui s'applique sur le contenu du paquet. Le *parsing* peut être très simple, trivial et inaperçu durant le traitement du paquet, comme il peut être une réelle tâche complexe. La tâche de *parsing* peut être réalisée par un PE dédié [Gil08]. Ainsi, la fonction de *parsing* consiste à identifier les champs pertinents des paquets entrants selon leurs emplacements et extraire les valeurs de ces champs afin de continuer l'opération de *parsing* ou utiliser ces valeurs pour la classification. Un exemple simple de *parsing* dans un paquet IPv4 serait d'extraire l'adresse IP de destination, ce qui est assez simple puisqu'il s'agit d'un champ de taille fixe dans l'entête IP et toujours au même emplacement.

1.3.2.2 Le *lookup*

Les opérations de recherche sont assurées soit par un processus logiciel, une circuiterie matérielle ou une combinaison des deux conçue autant qu'une unité fonctionnelle sur le NP. Cette fonction consiste à retourner une valeur appelée résultat, lorsqu'une valeur appelée clé de recherche se présente [Gil08].

- Support de recherche. Le support le plus simple est une mémoire qui retourne une valeur lorsque son adresse se présente. En effet, une mémoire retourne la valeur contenue dans l'adresse qui est reçue comme clé de recherche. L'opération inverse est possible avec une mémoire CAM (*Content Addressable Memory*) qui lorsque une valeur se présente, elle retourne son adresse. La CAM est un support matériel qui permet d'effectuer une opération de recherche en un seul cycle horloge. Une mémoire CAM compare simultanément la clé de recherche avec toutes les entrées qui y sont stockées. Bien que les mémoires CAMs soient rapides, efficaces et flexibles, elles présentent l'inconvénient d'avoir d'importantes

consommations d'énergie à cause du nombre important des circuits activés en parallèle. Elles sont aussi très coûteuses. Il existe deux types de mémoire CAM : la BCAM (*Binary Content Addressable Memory*) et la TCAM (*Ternary Content Addressable Memory*). La BCAM stocke et compare des valeurs binaires, c'est à dire des '0' et des '1' alors que la TCAM supporte un type additionnel de bit qui est le *don't care bit* appelé aussi *wildcard*.

- La TCAM est composée de cellules organisées selon la forme d'une matrice dont les lignes sont appelées des *match lines* et les colonnes sont appelées des *search lines* (voir Figure 1.3). Les cellules sont disposés horizontalement pour constituer les entrées de la TCAM où chaque bit est une cellule qui contient un espace de stockage ainsi qu'un circuit de comparaison. Lors d'une opération de recherche dans la TCAM, la clé de recherche est diffusée sur toutes les cellules de la CAM. Chaque mot stocké est comparé au mot recherché. La ligne de *match* indique si le mot stocké est identique au mot recherché, cas dans lequel la ligne de *match* est activée. C'est ce qu'on appelle un *hit*. Si le mot stocké est différent de la clé alors la ligne de *match* est désactivée. Il s'agit d'un *mismatch*. Toutes les lignes de *match* actives sont ensuite appliquées à un encodeur qui va produire l'adresse du mot recherché dans la table.

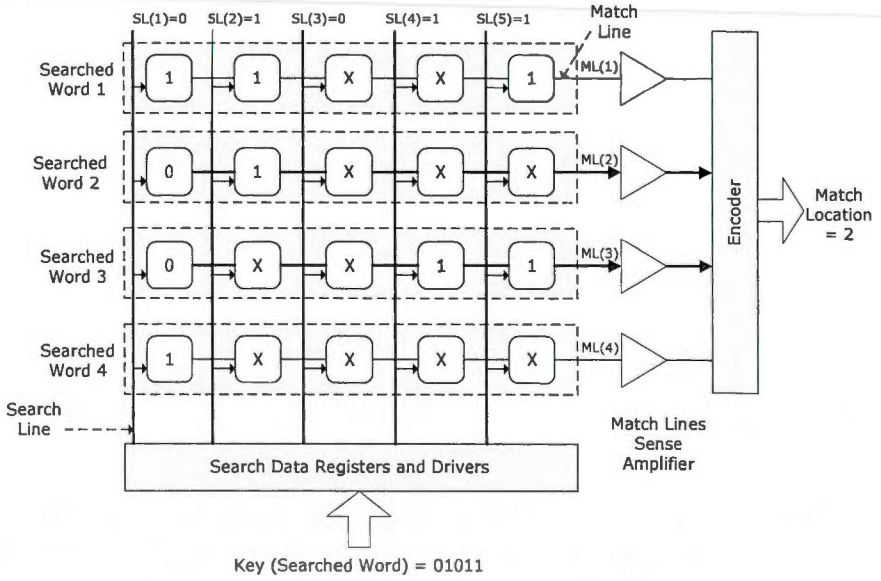

Figure 1.3 Structure de la TCAM [Gil08]

Les *wildcards* utilisés dans la TCAM permettent que, lors d'une opération de recherche, la ligne de *match* ne soit pas affectée par les cellules qui contiennent ces wildcards,

indépendamment de la valeur du bit recherché. Cette caractéristique est intéressante pour les recherches masquées où on utilise des masques avec des bits non spécifiés. Il est possible de trouver plus d'un *match* entre la clé recherchée et les entrées de la TCAM. En effet, le fait de pouvoir masquer des bits crée plus de *hits*. Dans ce cas, au lieu d'un simple encodeur, c'est un encodeur à priorité qui est utilisé. Le *prioritizer* sélectionne l'entrée qui a la plus grande priorité, c'est à dire la valeur d'adresse la plus petite. Cette stratégie est appelée *First Matching*. D'autres mécanismes de priorisation utilisent des critères comme le *Longest Prefix Matching*, le nombre total des bits actifs dans la *match line* ou le nombre des bits qui sont actifs et en plus consécutifs dans la *match line*.

- Stratégies de recherche. Les principales techniques de recherche sont le *Full Matching*, le *Best Matching*, le *Pattern Matching* et la classification (voir Tableau 1.1). Le *Full Matching* signifie que la totalité de la clé de recherche doit correspondre à la donnée dans la mémoire, soit au complet et dans ce cas il s'agit d'un *Exact Match*, soit en partie et dans ce cas la stratégie de reccherche est appelée classification. Dans la classification, certains champs de la donnée doivent correspondre à la totalité de la clé de recherche. La recherche partielle signifie qu'une partie de la clé de recherche doit correspondre soit à la totalité de la donnée et dans ce cas, la technique de recherche est dite *Pattern Matching* ou à une partie de la donnée pour avoir un *Best Matching*.

		Donnée dans la table de recherche	
		Complette	Partielle
Clé de recherche	Complette	Exact match	Classification
	Partielle	Pattern match	Best Match

Tableau 1.1 Stratégies de Recherche [Gil08]

- Opérations sur les supports de recherche. Les principales opérations de manipulation des données dans une structure de recherche sont (1) Le *lookup*. C'est l'opération la plus critique en termes de temps. Elle consiste à envoyer une requête contenant la clé de recherche pour avoir la valeur associée à la clé donnée. (2) L'insertion. C'est l'ajout d'une entrée à l'ensemble des éléments déjà maintenus dans la table de recherche. (3) La suppression. Cette opération consiste à effacer une entrée de la table de recherche. (4) La modification. Cette opération permet de changer la valeur d'une entrée de la table de

recherche.

- Structures de recherche. Les structures de recherche les plus utilisés sont les tables de recherche dont, essentiellement, les tables à accès direct (*Direct Access Tables*) et les tables de hachage (*Hash Tables*). Les tables à accès direct sont de simples tableaux d'éléments de valeurs appelés entrées. Chaque entrée est identifiée et adressée par une clé. Ces structures sont les plus adaptées pour faire de l'*Exact Match* vu que toutes les opérations de mise à jour ont une complexité O(1). En effet, la valeur recherchée est disponible avec une seule tentative d'accès. L'inconvénient de ce type de structure est qu'il demande un espace mémoire de la taille de la plage des valeurs possibles de la clé. Par exemple, pour une clé qui représente une adresse IPv4, la plage de valeurs possibles est de 2^{32} bits, donc une table de taille 2^{32} entrées est nécessaire, indépendamment du nombre d'entrées qui sera réellement utilisé. Les tables de hachage offrent une complexité O(1) tout en ayant une taille de table assez compacte. En effet, les *Hash Tables* sont comme les *Direct Access Tables*, sauf qu'une fonction de hachage est appliquée à toutes les clés des différentes entrées.

  En plus des tables de recherche, il y a aussi les structures de recherche en arbres qui sont le plus souvent utilisées pour le *lookup* d'adresses IP.

1.3.2.3 La classification

La classification consiste à catégoriser les paquets en flux dans lesquels ils seront traités de façon similaire. Ces flux sont définis par des règles et l'ensemble de ces règles est appelé classificateur ou *classifier*. La base des règles contient plusieurs entrées. Chacune des règles est composée d'une paire: une description spécifique pour la règle et une action appropriée qui lui est associée. Ces règles spécifiques sont comparées avec les paquets entrants et la meilleure correspondance détermine l'action à prendre pour le paquet. Très souvent, l'action consiste à faire un marquage sur le paquet avec une notation spécifique et c'est le processus suivant qui prend l'action appropriée en se basant sur cette notation. La classification de paquets peut être basée sur plusieurs champs du paquet. Les clés de recherche générées à partir des champs du paquet sont utilisées pour chercher la règle de classification qui sera utilisée pour la prise de décisions et d'actions. Cette base de règles consiste, généralement, en une table de recherche qui supporte l'*Exact Matching* d'une clé à taille et à format fixes, ou le *Partial Matching* comme le *Best Matching*, le *Longest Prefix Matching*. Une simple classification qui est basée, par exemple,

sur le type ou le protocole du paquet peut être réalisée avec une table à accès direct. Quand la classification utilise plusieurs champs ou quand la clé est de grande taille on utilise, généralement, une table de hachage. Quand la classification est quelconque, qu'elle demande des algorithmes de classification complexes ou qu'elle utilise en partie les adresses IP, alors, c'est du *Partial Matching* basé sur des arbres de recherche qui est utilisé pour classifier les paquets.

1.3.2.4 La modification des paquets

Dans toutes les applications de traitement de paquets, en dehors des applications de simple *forwarding* de paquets, la modification de paquet est finalement l'objectif même du traitement [Gil08]. Cependant, des modifications sont parfois requises dans des applications de *forwarding*, par exemple la modification d'un entête IP peut impliquer le changement de la valeur du champ Time-To-Live (TTL), ainsi que les adresses IP source et destination. Ainsi, le recalcul de la nouvelle valeur du champ de *checksum* sur l'entête peut être aussi requis. La modification de paquet comprend les opérations suivantes. (1) Changer le contenu du paquet. Généralement, les changements concernent l'entête du paquet mais peuvent aussi impliquer les données utiles du paquet dans le cas de certains traitements de compression et de cryptage. (2) Supprimer une partie du contenu du paquet ou de son entête comme dans le cas d'une décapsulation. (3) Ajouter de l'information additionnelle au paquet, comme pour le cas d'une encapsulation. (4) Supprimer le paquet en entier du système. (5) Dupliquer le paquet et en créer plusieurs copies, comme dans le cas d'une application de *multicast* [Gil08].

1.3.2.5 Le *queueing* et le *traffic Management*

Une fois que le paquet est traité, la tâche finale reste de décider comment l'expédier ou, plus précisément, comment le passer au récepteur, c'est-à-dire l'équipement ou le lien de communication suivant qui est sur son chemin vers sa destination finale [Gil08]. Comme les étapes précédentes dans le NP ont déterminé le port de sortie, la priorité ainsi que d'autres paramètres, le processus de gestion de trafic est responsable de transmettre le paquet à la file appropriée selon un ordonnancement dicté par les conditions du récepteur et les paramètres du paquet [Gil08]. Ce processus permet d'effectuer des mesures sur le paquet et de l'expédier en respectant un modèle de transmission. Le processus de gestion de trafic est assez compliqué pour qu'il soit, dans certains cas, implémenté à l'extérieur du NP dans un coprocesseur dédié [Gil08]. Cependant, il y a quelques constructeurs qui intègrent la tâche de gestion de trafic à l'intérieur

du NP, ce qui permet d'économiser le coût de l'implémentation et, surtout, de permettre des services plus rapides pour les paquets [Gil08]. Dans certains cas, les paquets entrants passent directement dans le gestionnaire de trafic pour des services de mesures, de *queueing* et de priorisation qui sont déterminés par le port d'entrée du paquet ou d'autres paramètres. Ceci afin que le paquet soit traité dans le NP selon un schéma prédéfini.

1.3.3 Architectures des NPs

Dans cette section, nous allons présenter l'architecture de deux processeurs de réseaux distincts : le NP4 d'EZchip et le NFP-3240 de Netronome. Dans cette description, nous allons voir que les NPs peuvent avoir des composantes différentes malgrès qu'ils fournissent les mêmes fonctionnalités. Ces deux architectures seront utilisées dans la validation de NETLANG qui est décrite dans le chapitre 5.

1.3.3.1 Architecture du NP4 d'EZchip

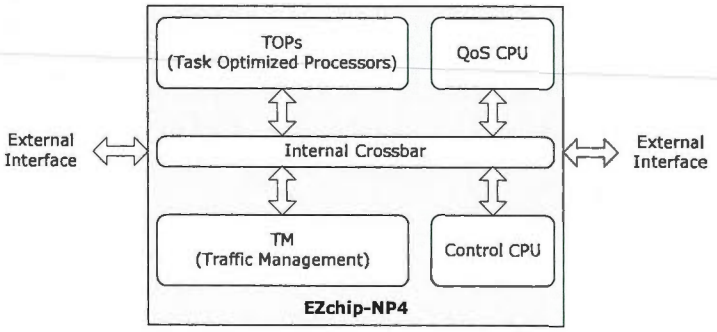

Figure 1.4 Architecture du NP4 d'EZchip [Tec11]

Le NP4 d'EZchip est un NP très flexible et programmable qui délivre un débit égal à 100 Gigabit [ezc99]. EZchip-NP4 offre la vitesse des processeurs ASIC combinée à la flexibilité des processeurs programmables. A travers sa programmabilité, le NP4 peut supporter une variété d'applications allant de la commutation au niveau 2 aux encapsulations VLAN, MPLS, VPLS et routage IPv4/IPv6 avec un support de QoS. Le traitement de paquet est flexible dans le sens où l'analyse du paquet peut être effectuée sur n'importe quel champ dans n'importe quel emplacement du paquet. Plusieurs options de *lookup* dans les tables de recherche sont fournies avec des clés et des résultats de recherche de grande taille. Les flux de paquets peuvent être classifiés sur la base de n'importe quelle combinaison des informations extraites du paquet. L'entête et

même le contenu du paquet peuvent être modifiés et les paquets peuvent être répliqués.

Le NP4 est un processeur qui intègre dans une même puce 5 types de blocs fonctionnels (1) les TOPs (*Task Optimized Processors*) sont des éléments programmables qui assurent le traitement de paquets et les opérations de *lookups*, (2) le TM (*Traffic Manager*) est une composante configurable qui permet de mettre en place un contrôle avancé sur les flux de paquets, (3) le CPU de contrôle, (4) le CPU de QoS (5) et un bloc de commutation interne qui permet de déterminer le flux de données entre ces 4 blocs fonctionnels et, aussi, avec l'interface d'entrée et l'interface de sortie du NP (voir Figure 1.4).

La technologie des TOPs consiste à intégrer plusieurs processeurs dans une architecture en pipeline. Chaque TOP est optimisé pour réaliser une tâche spécifique. Le NP4 contient 5 types de TOPs : TOPparse, TOPsearch, TOPresolve, TOPlearn et TOPmodifiy qui sont utilisés pour effectuer les tâches essentielles de traitement de paquets c'est-à-dire le *parsing*, la classification, le *lookup* et la modification. Un TOP programmable est associé à chacune de ces tâches et réalise ses opérations de manière optimisée. Chaque type de processeur possède une architecture unique avec un jeu d'instructions dédié. La performance des TOPs est amplifiée par une architecture super-scalaire dans laquelle plusieurs instances d'un même TOP fonctionnent en parallèle dans chaque étage du pipeline. Les TOPs du NP4 sont totalement programmables selon un modèle de programmation basé sur une seule image. Ce modèle n'expose pas la programmation parallèle et le *multithreading*. L'assignation des TOPs aux trames entrantes ainsi que le maintien de l'ordre des trames sont implémentés matériellement et de façon transparente par rapport au programmeur.

Le NP4 supporte 3 types de tables de *lookup* : les tables à accès direct, les tables de hachage et les arbres de recherches. Chacune des structures est définie de manière flexible et peut être utilisée par des applications diverses. Les tables peuvent être utilisées pour la transmission et le routage, la classification des flux de paquets, le contrôle d'accès, etc. De nombreuses tables de chaque type peuvent être définies et stockées dans la mémoire interne ou externe du NP. La taille de la clé de recherche ainsi que la taille du résultat de recherche et le nombre d'entrées dans les tables sont, tous, des paramètres programmables que l'utilisateur peut définir pour chaque table. Des algorithmes de recherche brevetés permettent de faire des *lookups* de manière optimisée. Tous les algorithmes de recherches pour les tables de hachage et les structures en arbre sont implémentés dans le matériel pour assurer la performance et la simplicité. Les résultats de recherche dans une table de hachage sont déterministes et indépendants de la taille de cette

table. Les arbres de recherche, tout comme les tables de hachage, peuvent contenir des millions d'entrées et les recherches qui y sont appliquées se font à un coût constant. La taille d'une clé peut aller jusqu'à 72 octets ce qui permet d'avoir des tables avec des entrées de grande taille comme dans le cas du flux IPv6 à 5-uplets. Les résultats de recherche peuvent aller jusqu'à 96 octets.

Une interface optionnelle est disponible pour une TCAM externe. Cette option est particulièrement utile pour effectuer des opérations de *lookup* à travers de grandes tables de manière rapide et en utilisant les *wildcards* qui sont utiles dans le cas des ACL (*Access Control List*) par exemple. Les opérations de recherche dans la TCAM peuvent être effectuées en parallèle avec les *lookups* réalisés par les moteurs intégrés du TOPsearch.

1.3.3.2 Architecture du NFP-3240 de Netronome

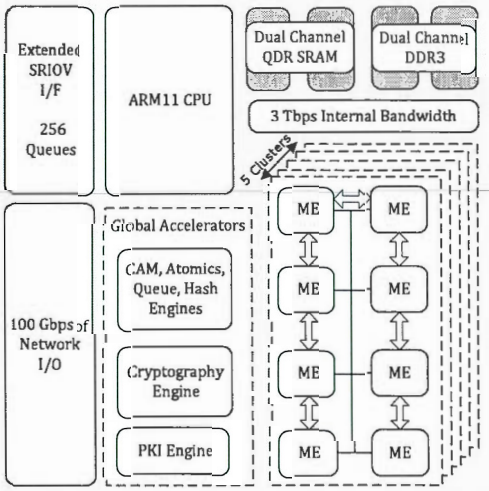

Figure 1.5 Architecture du NFP-3240 de Netronome [Com10]

Le NFP-3240 (*Network Flow Processor*) de Netronome est un NP programmable de troisième génération qui fournit la puissance et la flexibilité d'un traitement parallèle à hautes performances délivrant un débit allant jusqu'à 40 Gbps [net10]. Le NFP-3240 est spécialisé dans les classifications complexes des flux de paquets, le traitement et la transmission des paquets.

Dans l'architecture du NFP-3240 (voir Figure 4.3), un processeur ARM est utilisé pour initialiser et gérer le NFP, traiter les exceptions et effectuer les tâches de *control plane*. Une interface appelée MSF (*Media Switch Fabric*) est utilisée pour connecter le NFP au dispositif

de la couche physique (PHY) ou au *switch fabric*. Chacune des interfaces de transmission et de réception peut être configurée séparément pour être utilisée comme une interface de type SPI4.2 (*System Packet Interface*), pour les ports physiques, ou bien une interface de type CSIX-L1, pour le *switch fabric*. Les ports de transmission et de réception sont unidirectionnels et indépendants. Chaque port détient une horloge, un signal de contrôle, un signal de parité et des signaux de données. Le *buffer Rbuf* est une mémoire RAM qui stocke les paquets reçus. Il sauvegarde les données reçues dans des sous blocs qui sont par la suite accédés par les *Micro-Engines* (MEs) ou le *Host* pour la lecture de l'information reçue. Le *buffer Tbuf* est une mémoire RAM qui contient les données à transmettre. Tous les éléments dans une partition du *TBuf* sont transmis dans l'ordre. Le NFP-3240 est composé de 40 *micro-engines* de type RISC à 32 bits qui sont utilisés dans la plupart des opérations de traitement de paquets. Les MEs sont répartis sur 5 *clusters* de 8 *micro-engines* et ils ont accès à toutes les ressources partagées comme la SRAM, la DRAM et les interfaces MSF. Le NFP-3240 est, en réalité, compatible avec la famille des IXP-28xx d'Intel avec une amélioration des performances. L'IXP-28xx comporte 16 *micro-engines*. Les MEs fournissent un support pour le *multithreading*. Vu la différence entre la vitesse du processeur et la vitesse des mémoires externes, l'exécution d'un *thread* va souvent se bloquer en attendant que les opérations mémoire soient complétées. Le fait qu'il y ait du *multithreading* permet d'avoir l'entrelacement des opérations, mais il y a souvent un *thread* au moins qui est prêt pour s'exécuter pendant que les autres *threads* sont bloqués. En plus des différents types de mémoires, les MEs ont aussi des blocs de traitement spéciaux pour faire des tâches dédiées comme le CRC (*Cyclic Redundancy Check*). Le NFP-3240 fournit également des *rings* qui sont des mémoires qui peuvent être utilisés pour la communication entre les *micro-engines*.

1.4 Résumé

Les NPs sont les principales composantes d'un routeur programmable. Ils permettent d'implémenter des applications complètes de traitement de paquet. Ces applications sont généralement constituées d'un ensemble de tâches de *parsing*, de *lookup*, et de modification de paquets. Le traitement de paquet dans un NP implique l'accès à d'autres ressources du routeur dont, principalement, les mémoires qui contiennent les structures de recherche. Les opérations d'un routeur programmable sont classées en opérations de *data plane* et opérations de *control plane*. Le NP est impliqué dans le *data path* alors que le *Host* assure les fonctions de *control path*.

CHAPITRE II

LA PROGRAMMABILITÉ DES RÉSEAUX

2.1 Introduction

Dans ce chapitre, nous allons présenter, premièrement, les *Software Defined Networks* (SDNs), une nouvelle approche qui facilite la programmabilité des réseaux et qui a fait émerger de nouveaux protocoles tels que ForCES et OpenFlow. Nous expliquerons les principes du standard ForCES et nous examinerons particulièrement les caractéristiques du protocole OpenFlow pour montrer sa généricité et sa richesse en termes d'expressivité pour la description du comportement des paquets dans un routeur programmable. Ensuite, nous allons présenter un ensemble d'outils et langages de programmation qui ont été proposés comme solutions pour les réseaux programmables et aussi pour les réseaux OpenFlow.

2.2 Les réseaux SDNs

SDN est une proposition d'un ensemble de principes pour la conception des architectures de réseaux [She13]. SDN n'est pas une évolution technologique dans le sens classique du terme, mais une façon d'organiser les fonctionnalités du réseau.

Une introduction basique aux réseaux SDNs serait de parler des racines de SDN parce que ses racines sont profondes et larges. Tout d'abord, le terme SDN a été inventé par des professeurs de l'université Stanford, en Californie. Plus spécifiquement, ce terme a été introduit pour la première fois par Martin Casado. Bien qu'il soit l'inventeur de ce terme, Martin Casado n'était pas le seul à travailler sur SDN, Nick McKeown et Scott Shenker font partie des personnes les plus cités dans ce domaine.

SDN n'est pas apparu d'un seul travail isolé. C'est le résultat des efforts de plusieurs

gens, travaillant sur plusieurs problématiques qui touchent l'architecture des réseaux. Parmis ces travaux, il y a l'investigation sur la séparation entre le *control plane* et le *data plane*, la gestion des réseaux sans fil, en plus des efforts pour apprivoiser les réseaux de centre de données (*Data Center Networks*), ainsi que des projets académiques pour réorganiser la gestion des réseaux. Il s'agit de travaux qui n'étaient pas exactement sur SDN mais qui étaient étroitement liés. Ce large mouvement intellectuel, de gens travaillant sur des problématiques reliées, a été dirigé par certains besoins. Les problèmes qui liaient tous ces travaux sont les difficultés qu'on recontre dans les réseaux informatiques ; ils sont difficiles à gérer, ils sont difficiles à faire évoluer et contrairement aux autres domaines de l'informatique comme les systèmes d'exploitation, la conception des logiciels ou des bases de données, la conception des réseaux n'est pas fondée sur des principes formels.

SDN est un effort pour construire des principes pour aider les gens à penser à propos de cette matière. SDN considère le réseau comme un système et propose de redéfinir ses composantes. Alors, pour bâtir un système qui fonctionne correctement, il faut briser le problème en des morceaux solvables. L'objectif de SDN est de bâtir un réseau qui marche et qui évolue. Pour cela, il utilise la modularité basée sur l'abstraction [She13]. En fait, si on ne peut pas gérer, faire évoluer et comprendre un système, alors, il se peut que les abstractions ne sont pas bonnes [She13]. SDN propose alors les abstractions qu'on a besoin pour les réseaux.

Selon SDN, on doit réaliser que le réseau a deux différents plans, deux différents problèmes qui doivent être résolus. Le premier plan est le plan de données ou *data plane*, et le deuxième plan est le plan de contrôle ou *control plane* [She13]. Dans le plan de données, quand un paquet arrive pour un routeur, et que ce paquet a un entête , le routeur analyse cet entête, regarde sa table de routage et décide ce qu'il doit faire de ce paquet (s'il doit le rejeter ou s'il doit le transmettre sur son port numéro 3 ou son port numéro 4). Ainsi, le *data plane* est le traitement appliqué sur le paquet. C'est un traitement fréquent, rapide et complètement local. En général, les fonctions de *data plane* sont implémentées en matériel.

Le *control plane* est le traitement qui permet de mettre en place les tables de routage des différents routeurs dans un réseau donné. La définition des tables de routages peut être faite à travers des protocoles distribués ou à travers des configurations manuelles. La fréquence à laquelle ces tables sont mises à jour est beaucoup moins importante que la fréquence à laquelle les paquets arrivent. Par contre, c'est un traitement qui peut prendre plusieurs heures et même des jours lorsque les configurations sont manuelles. Le *control plane* est intrinsèquement

distant puisque l'information doit venir de l'extérieur (un routeur ne peut pas comprendre de lui-même comment il doit envoyer le paquet). Les fonctions de *control plane* sont implémentées généralement en logiciel.

Le *data plane* et le *control plane* sont deux différents problèmes, l'un est local (dans le routeur), fréquent et très rapide et l'autre n'est pas local, est moins fréquent et prend plus de temps pour être implémenté. Ce sont deux différents problèmes qui ont des abstractions différentes. Pour le *data plane*, il existe déjà un ensemble d'abstraction très connu. C'est la définition des 7 couches du modèle OSI (*Open Systems Interconnection*) qui a été approuvé initialement par l'ITU (*International Telecommunication Union*) comme l'un des fondements même des réseaux [Zim80]. Les abstractions du *data plane* qui se font à travers les couches Ethernet, IP, TCP et application, sont utilisées dans le but de réduire la complexité. Cependant, les interfaces sont plutôt mal implémentées et violent les principes même de la modularité [KWGT12]. De plus, le *control plane* ne fournit aucune abstraction ce qui renforce la complexité [KWGT12]. Enfin, un bloc de *management plane* est nécessaire pour bâtir et valider des configurations bien élaborées.

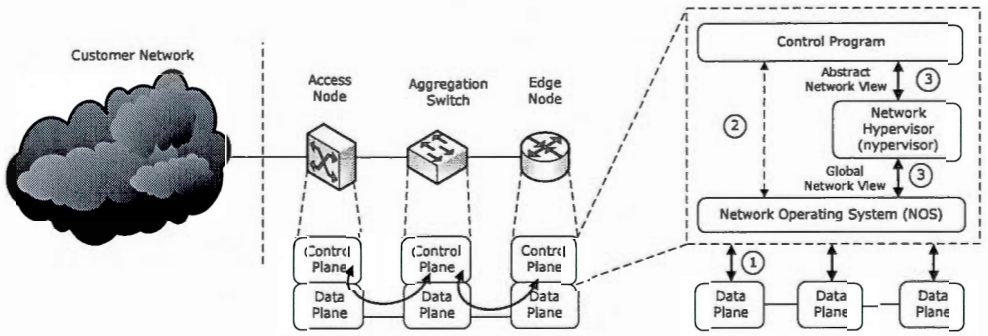

Figure 2.1 Comparaison entre les réseaux actuels et les réseaux SDNs [KWGT12]

La figure 2.1 montre la différence entre les réseaux traditionnels et les réseaux SDNs. Dans le cas général, lorsqu'un client veut accèder à Internet, il doit passer par 3 domaines de réseaux. Le premier domaine rencontré est le réseau d'accès, ensuite le réseau d'aggrégation et enfin le réseau *edge*. Dans le cas traditionnel, le *control plane* et le *data plane* sont incorporés dans la même boite. Ce qui fait que les routeurs peuvent aussi bien faire des fonctions de *data plane*, comme le traitement du paquet au niveau 2 ou 3, que des fonctions de *control plane*, comme la mise à jour des tables de routage [KWGT12]. Ceci est valable pour les 3 différents domaines de réseaux (le réseau accès, le réseau d'aggrégation et le réseau *edge*).

La partie extrêmement à droite de la figure 2.1, montre comment le concept SDN s'est

intéressé au réseau *edge* où il a séparé le *control plane* du *data plane*. La figure montre aussi que le *data plane* est resté local dans le routeur, alors que le *control plane* a été placé à l'extérieur du routeur. Selon la figure, SDN a ajouté 3 nouvelles séparations (réprésentées par les chiffres 1, 2 et 3 dans la figure) entre les couches du réseau afin d'introduire de nouvelles abstractions et dépasser ainsi les limitations causées par la complexité du modèle actuel [RFRW11].

Le premier niveau dans SDN introduit une séparation entre le *data plane* et le *control plane*, et déplace ainsi toutes les fonctions de *control plane* vers un nouveau composant appelé *Network Operating System* (NOS). Le deuxième niveau de séparation est entre le NOS et les applications de contrôle. Cette séparation permet de fournir une vue globale du réseau. Cette vue est ensuite raffinée pour donner une vue restreinte mais tout autant significative. Le troisième niveau de séparation introduit un hyperviseur pour le réseau appelé le *nypervisor*, qui se situe entre le NOS et les applications de contrôle. Les vues du réseau sont des vues globales entre le NOS et le *nypervisor*. Alors qu'entre le *nypervisor* et les applications de contrôle, ce sont des vues abstraites qui sont utilisées.

Avec SDN, il devient possible de traiter divers problèmes pertinents dans le domaine des réseaux tel que le problème d'absence de principes de conception pour les applications réseaux. Le principe fondamental du concept SDN reste la séparation entre le *control plane* et le *data plane* [KWGT12]. Les niveaux d'abstraction élevée du concept SDN présentent un principe essentiel qui peut être utilisé pour améliorer l'efficacité et réduire la complexité du réseau d'un point de vue conception et gestion. Les réseaux SDNs ont réduit significativement les obstacles face à l'innovation. SDN a aussi enrichit les travaux académiques et facilité l'entrée au marché à plusieurs vendeurs. Ce concept a restitué le *control plane* et a introduit de la flexibilité au réseau [KWGT12]. Le concept SDN a seulement défini une architecture générale et essaie de développer une vision alternative du *data plane* basée sur la redéfinition des couches. En effet, chaque couche n'est plus supposée être totalement isolée des autres couches et agir séparément sur le *parsing*, le *matching*, le *lookup* et le *processing*. Le concept SDN s'applique à l'implémentation d'OpenFlow qui fournit un *matching* générique au *data plane* allant de la couche 2 à la couche 4, une recherche d'actions suivant ce *matching* et un traitement en pipeline intégrant différentes couches [KWGT12].

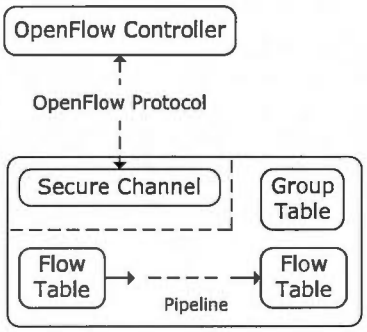

Figure 2.2 Une *Switch* OpenFlow [off11]

2.2.1 OpenFlow

OpenFlow est une implémentation d'une partie spécifique du concept SDN, une partie qui gère la communication entre le NOS et les noeuds du réseau. En d'autres termes, OpenFlow décrit le protocole sur l'interface entre le *control plane* et le *data plane* ou plus spécifiquement entre le contrôleur du réseau et le *forwarding* dans le *data plane* (voir Figure 2.2). OpenFlow régit l'échange de l'information entre le contrôleur et le routeur. Jusqu'à présent, OpenFlow a été principalement évalué par des académiciens. Cependant, en se basant sur le fait qu'OpenFlow est aussi un concept prometteur pour les opérateurs industriels de réseau et des centres de données, la fondation ONF (*Open Networking Foundation*) a été créée [onf]. Il s'agit d'une organisation qui évalue les capacités opérationnelles d'OpenFlow pour le développement de grands réseaux et des centres de données basés sur OpenFlow. La description de l'interface du protocole OpenFlow est fournie dans la spécification d'OpenFlow v1.1.0 [off11]. Un contrôleur OpenFlow peut ajouter, modifier, et supprimer des entrées dans la table de *forwarding* du routeur. D'un autre côté, si le routeur ne trouve pas d'entrée valide (une règle de transmission indiquant comment traiter le paquet) pour un nouveau paquet entrant, dans la table de *forwarding*, le paquet est transmis au contrôleur en utilisant le protocole OpenFlow. Le contrôleur analyse le paquet et décide du traitement qu'il aura sur ce paquet ainsi que tous les paquets ayant des caractéristiques similaires (des *flows*). La logique de décision est implémentée et effectuée par les applications de contrôle du contrôleur OpenFlow. Le paquet ainsi que la nouvelle décision de *forwarding* sont renvoyés au routeur. Suite à la réponse du contrôleur, le routeur met à jour sa table de *forwarding* en y ajoutant la règle envoyée par le contôleur.

Un routeur OpenFlow se compose d'une ou plusieurs *flow tables* en pipeline et d'une

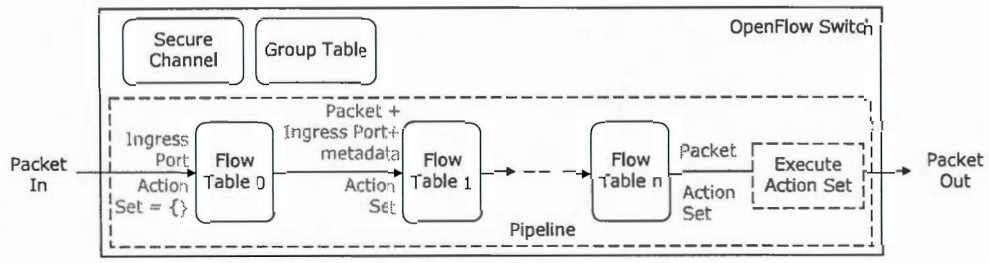

Figure 2.3 Le pipeline des Tables dans OpenFlow [off11]

group table (voir Figure 2.3). L'ensemble de ces tables forment le support pour le *lookup* et le *forwarding* des paquets. Le routeur OpenFlow dispose également d'un *secure channel* pour l'échange avec le contrôleur externe. Chaque *flow table* du routeur contient un ensemble de *flow entries* qui présentent les règles de *forwarding* des paquets.

Une *flow entry* est composée (1) d'un ensemble de *match fields* (champs de correspondance) qui définissent le modèle du flux de paquets à travers l'instanciation des champs d'entête allant de la couche Ethernet à la couche Transport (2) des compteurs sur les paquets (3) et des instructions qui indiquent les décisions à prendre à travers le pipeline ainsi que les actions à appliquer sur le paquet qui correspond à l'entrée en question.

Le processus de *matching* commence à partir de la Table 0 qui est la première table du pipeline et peut se poursuivre sur des tables additionnelles. Le paquet est appliqué aux *flow entries* dans un ordre de priorité et c'est la première *flow entry* qui sera utilisée pour traiter le paquet. Si une correspondance est trouvée, alors les instructions associées à la *flow entry* en question, seront exécutées. Si aucun correspondance n'est trouvée, la suite du processus dépendra de la configuration par défaut du routeur. Trois cas de figure sont possibles ; le paquet peut être envoyé au contrôleur, supprimé, ou passé à la *flow table* suivante.

Il y a 15 types de champs de correspondances supportés dans la version 1.1.0 d'OpenFlow : port d'entrée, métadonnées, adresse MAC source, adresse MAC destination, type Ethernet, identifiant VLAN, priorité VLAN, label MPLS, classe de trafic MPLS, adresse IPv4 source, adresse IPv4 destination, protocole IPv4, type de service IPv4 (ToS), port source TCP/UDP/SCTP ou type ICMP, port destination TCP/UDP/SCTP ou code ICMP. Chaque *flow entry* contient des valeurs spécifiques de champs de correspondance qui permettent de faire la comparaison avec l'entête du paquet ainsi que son numéro de port d'entrée. Les instructions associées à chaque *flow entry* décrivent les modifications et la transmission du paquet, le traitement sur

la *group table* et le traitement sur le pipeline des tables. Le traitement sur le pipeline permet aux paquets d'être envoyés aux tables suivantes pour un traitement supplémentaire et permet à des informations sous forme de métadonnées d'être échangées entre les tables. Le traitement du pipeline s'arrête quand l'ensemble des instructions d'une *flow entry* ne spécifie pas de table suivante. A ce stade, le paquet est généralement modifié et transmis.

OpenFlow 1.1.0 propose 5 types d'instructions: appliquer des actions, effacer les actions, écrire des actions, écrire des metadonnées et aller à la table suivante.

Une *group table* se compose de *group entries* tel que chaque *group entry* est composé (1) d'un identifiant de groupe, (2) un type de groupe, (3) des compteurs et (4) des *actions buckets* tel que chaque *action bucket* contient un ensemble d'actions et les paramètres qui leur sont associés. Il s'agit des actions qui seront appliquées sur le paquet. Les actions sont classées selon deux catégories : les actions de type *push/pop* permettent de faire l'encapsulation et la décapsulation des paquets MPLS et et des trames de VLAN alors que les actions de type *set_field* permettent de modifier la valeur d'un champ du paquet en allant de la couche 2 jusqu'à la couche 4.

2.2.2 ForCES

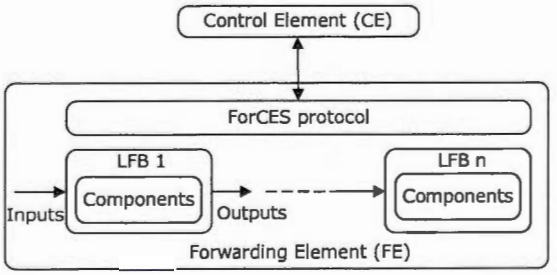

Figure 2.4 L'architecture de ForCES [DGK+10]

ForCES [DGK+10] est un standard de l'*Internet Engineering Task Force* (IETF) qui définit une architecture et un protocole pour standardiser l'échange entre le *control plane* et le *data plane*. Un élément de réseau ForCES contient des éléments de contrôle appelés CEs (*Control Elements*) qui se trouvent au niveau du *control plane* et des éléments de *forwarding* appelés FEs (*Forwarding Elements*) qui opèrent au niveau du *data plane*. La figure 2.4 illustre l'architecture de ForCES: un protocole ForCES permet la communication entre le CE et le FE. Un FE est modélisé sous forme d'un pipeline de LFBs (*Logical Functional Blocks*) de la même façon qu'un routeur OpenFlow est défini à travers un pipeline de tables. Chaque LFB est défini

par son entrée, sa sortie et ses composants. A travers cette figure, on voit que ForCES aussi applique les principes de SDN puisqu'il sépare entre le *control plane* et le *data plane.*

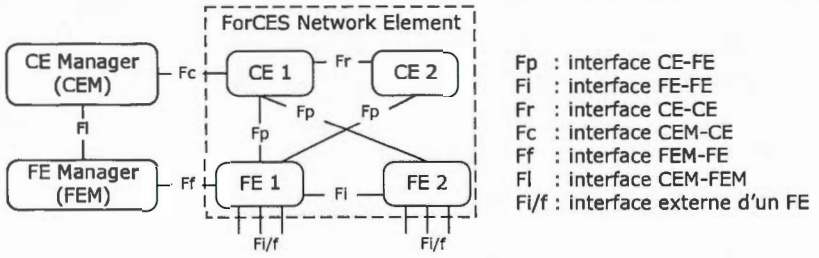

Figure 2.5 Les composantes d'un réseau ForCES [DGK+10]

ForCES permet que plusieurs instances de CEs et de FEs existent dans un élément de réseau appelé NE (*Network Element*). La figure 2.5 représente justement un exemple d'un réseau ForCES où on a un NE composé de deux CEs et de deux FEs. La figure permet de réprésenter l'ensemble des interfaces utilisées dans ce réseau. Chaque FE de l'élément du réseau contient une ou plusieurs interfaces Fi/f pour l'échange des paquets avec le réseau. L'ensemble de ces interfaces forme les interfaces externes d'un FE. En plus de ces interfaces, ForCES définit des interfaces internes notées Fp qui permettent aux différents FEs et CEs de communiquer. D'autres interfaces internes au NE notées Fi sont utilisées pour rendre le transfert des paquets d'un FE à un autre possible. De même, il existe un troisième type d'interfaces internes, noté Fr, qui permet la communication entre les CEs. Ce genre d'interface n'a pas été défini dans OpenFlow. Un *CE manager* et un *FE manager* sont définis pour gérer la configuration des CEs et des FEs respectivement. Une interface Fc est définie entre le *CE manager* et les CEs alors que les FEs sont gérés par le *FE manager* via une interface Ff.

Un élément de contrôle (CE) est une entité logique qui implémente le protocole ForCES et l'utilise pour gérer un ou plusieurs FEs. Plusieurs CEs peuvent être utilisés pour contrôler les différents FEs. L'objectif de la multiplicité est de mettre en place de la redondance et d'avoir le partage de la charge ou la distribution de contrôle. Dans les deux premiers cas, on a un CE actif à la fois mais tous les CEs sont identiques, alors que dans le troisième cas, on a au moins deux CEs actifs en même temps qui supportent des services différents. Les CEs sont responsables d'assurer la coordination entre eux afin d'offrir la consistance et la synchronisation du système.

Un élément de *forwarding* (FE) est une entité logique qui implémente le protocole ForCES. Les FEs utilisent la couche matérielle pour fournir un traitement de paquets comme indiqué par

le(s) CE(s). Les FEs s'exécutent typiquement dans des équipements basés sur des processeurs ASICs, des NPs, ou des *General-Purpose Processors* qui traitent des opérations de *data path* pour chaque paquet. Les FEs assurent toutes les fonctions de traitement de paquets. Ils ne prennent cependant pas d'initiative et agissent en tant qu'esclaves par rapport aux CEs qui les contrôlent. Les FEs n'ont pas conscience de la multiplicité des CEs et ils acceptent naïvement toute commande provenant d'un CE autorisé à les contrôler.

Ce qui est intéressant avec les FEs, c'est le modèle par lequel ils sont définis. Il s'agit d'un modèle formel basé sur le langage XML (*eXtensible Markup Language*) [BPSM+97]. Dans ce modèle, ForCES a introduit la notion de *Logical Functional Block* (LFBs) pour pouvoir représenter un FE comme étant un ensemble de LFBs. Un LFB est en effet un bloc fonctionnel logique qui assure une tâche simple de traitement de paquet. La création d'un LFB implique la définition de l'ensemble de ses aptitudes (par exemple un LFB peut lire des trames Ethernet basiques mais pas des trames Ethernet comportant des tags de VLAN) et la configuration de l'ensemble de ses composantes.

2.3 Langages de programmabilité des réseaux

Dans cette partie, nous allons présenter un ensemble de langages proposés pour les réseaux programmables. Certains de ces langages opèrent au niveau du *control plane*, d'autres sont destinés au *data plane*. Des langages ont été aussi proposés pour le *management plane* pour permettre la configuration des équipements réseaux. D'autres solutions pour les routeurs programmables proposent des outils, des frameworks et des modèles qui visent à simplifier la programmabilité des réseaux.

2.3.1 Langages de description de *headers*

NetPDL [RB06] (*Network Protocol Description Language*) est un langage simple de description des formats de paquets, indépendant du type de l'application réseau qui l'utilise. Son objectif est de fournir une API (*Application Programming Interface*) universelle aux applications réseaux traitant les entêtes de paquets et ayant des besoins différents lors de l'analyse du format des paquets (ces applications utilisent des syntaxes propriétaires pour décrire les entêtes de paquets). Il peut être utilisé autant par des applications de routage de paquets que par des applications d'analyse de trafic, de génération de trafic, de détection d'intrusion ou de capture

de paquets. Cette API est fournie à travers les deux composantes du langage qui sont : le *NetPDL-engine* et la *NetPDL-database*. Le langage étant basé sur le langage XML, la *NetPDL database* est, en effet, un ensemble de fichiers XML qui contiennent la description des entêtes des protocoles réseaux et qui peuvent être dynamiquement mis à jour. Ces fichiers XML sont analysés par les *NetPDL-engines* qui ont pour rôle de construire une représentation interne dans le sens où la structure générée est spécifique à l'*engine* en question.

Pour permettre la description des paquets, NetPDL utilise deux types de description complémentaires : la description du format du paquet (la liste et le format des champs constituant le paquet) et la description des encapsulations de paquets (les règles sur les champs du paquet qui déterminent comment interpréter la séquence d'octets contenu dans les données du paquet selon le protocole utilisé). NetPDL est constitué d'un ensemble de primitives où chaque primitive est composée d'un élément XML caractérisé par plusieurs attributs. Les éléments proposés pour la description d'entête de paquets sont les éléments *<proto>* et *<field>* utilisés pour donner le format du paquet et les éléments *<nextproto>*, *<switch>* et *<case>* qui sont des éléments avancés pour la description des encapsulations de paquets.

Protobuf (Protocol Buffers) [FS] est un langage descriptif open source proposé par Google pour la sérialisation des données structurées pour les protocoles de communication. Protobuf permet de décrire le format des messages échangés dans un protocole et peut donc être utilisé pour la description du format des paquets et leurs entêtes. Le langage permet de spécifier comment l'information à sérialiser est structurée en utilisant des éléments appelés messages. Les messages présentent la structure principale utilisée par Protobuf. Dans le fond, ils ressemblent aux balises XML, mais dans la forme ils ressemblent plus aux structures du langage C [KR88]. Chaque message est caractérisé par un type et un ensemble de paramètres. Chaque paramètre du message est une paire nom−valeur qui a, aussi, un type et qui est précédé par le mot réservé *required* ou *optional* pour indiquer si ce paramètre est obligatoire ou optionnel. Le format du message est très simple. En effet, Protobuf a été défini pour être un XML plus rapide, plus simple et plus efficace [GDK11].

2.3.2 Langages de configuration

NetConf [EBSB11] est un langage de *management plane* qui permet d'installer, de manipuler et de supprimer la configuration des équipements de réseau. Les messages ainsi que

les données du langage sont codés en XML. NetConf est basé sur un ensemble d'opérations telles que l'opération (1) *<get-config>* pour récupérer une configuration spécifique; (2) *<edit-config>* pour charger une nouvelle configuration dans un équipement. Cette opération utilise l'élément *<config>* qui encapsule la définition de la configuration composée principalement des éléments *<interface>*; (3) l'opération *<copy-config>* qui permet de créer une nouvelle configuration ou d'écraser une ancienne configuration si une configuration existe déjà sur l'équipement; (4) *<delete-config>* pour supprimer la configuration d'un équipement. NetConf manipule uniquement des données de configuration (*configuration data*) qui sont des données en écriture qu'il faut modifier pour changer l'état d'un routeur. NetConf n'utilise pas les données d'état (*state data*) puisqu'elles sont des données additionnelles qui décrivent l'état du système mais qui sont en lecture seulement telles que les données de statistiques qui sont calculées sur l'équipement. NetConf utilise un modèle RPC (*Remote Procedure Call*) [SM88] moyennant des messages de type *<rpc>* ou *<rpc-reply>* pour encapsuler les opérations de configuration. Dans ses manipulations, NetConf utilise 3 types de cibles qui sont supposées contenir la nouvelle configuration : (1) *Running Configuration DataStore* pour contenir la configuration active sur l'équipement (2) *Candidate Configuration DataStore* peut supporter des manipulations sans que la configuration actuelle n'en soit touchée et (3) *Startup Configuration DataStore* pour contenir la configuration qui sera chargée lors du démarrage du routeur.

OF-Config [ofc11] est aussi un langage de configuration mais il a été conçu spécifiquement pour les réseaux SDNs. Il fournit une modélisation des données pour OpenFlow. Il est basé sur le langage XML et les opérations de NetConf (les données OF-Config sont encapsulées dans des opérations NetConf). Il est conçu spécialement pour la gestion des routeurs OpenFlow. Dans la logique d'OF-Config, un routeur OpenFlow contient plusieurs ressources OpenFlow. Les types de ressources considérées sont *OpenFlow Queues* et *OpenFlow Ports*. OF-Config définit un point de configuration qui a pour rôle de configurer le routeur OpenFlow. OF-Config utilise une spécification basée sur le langage de modélisation Yang [Bjo10] afin d'exprimer les contraintes liées aux différentes entités du langage. Comme OpenFlow est le langage échangé entre le contrôleur et le routeur, OF-Config représente le langage échangé entre le point de configuration et le routeur. OF-Config exprime l'intégration du *managment plane* dans un réseau OpenFlow.

2.3.3 Outils pour les règles de *forwarding*

SNORTRAN [ES02] est un outil qui permet aux programmeurs d'exprimer des stratégies de gestion et filtrage de paquets. Ces stratégies sont composées de règles de sécurisation des réseaux. Il s'agit, en effet, d'un compilateur optimisant pour les règles de détection d'intrusions. Les règles de détection d'intrusion considérées sont dans la même syntaxe que celles utilisés par Snort [R+99], un NIDS (Network Intrusion Detection System) open-source. Une règle de détection d'intrusion se compose d'un en-tête et d'une liste d'options. L'en-tête contient des tests sur les champs IP/TCP/UDP communs tels que l'adresse IP, le numéro de port et le type du protocole alors que les options permettent de spécifier des tests additionnels. Ces règles sont similaires aux règles OpenFlow qui sont définit à travers les *flow entries* sauf qu'elles ne contiennent pas d'instructions et d'actions. Le moteur de détection d'attaques dans SNORTRAN utilise une combinaison de plusieurs techniques de compilation lui permettant d'avoir de meilleures performances que le Snort original. En effet, dans Snort, les règles sont perçues comme étant une liste d'entrées indépendantes qui sont testés séparément dans un ordre séquentiel. SNORTRAN applique des techniques de compilation connues aux règles de Snort et produit des moteurs de correspondance plus performants. Parmi ces techniques, SNORTRAN utilise les *Cost Optimized Decision Tree*, le *Pattern Matching* et le *String Set Clustering*.

2.3.4 Langages de *control plane* pour OpenFlow

Nettle [VAH10] est conçu pour ête un langage de programmation de *control plane* pour les réseaux OpenFlow. Il s'agit d'un langage basé sur Haskell et utilisant le paradigme *Functional Reactive Programming* (FRP) pour supporter la programmation des systèmes réactifs dans un style déclaratif. Nettle permet de faire la configuration du réseau d'une manière repensée comme expliqué dans *Don't Configure the Network, Program it !* [FVA+10] où on encourage à programmer les réseaux par des langages de haut-niveau. Dans un réseau OpenFlow, on a un contrôleur centralisé qui gère l'ensemble des routeurs. Nettle permet justement d'implémenter les stratégies du réseau, les protocoles et les algorithmes de contrôle. Le langage fournit deux types d'abstractions pour l'implémentation d'une stratégie globale pour le contrôle de la totalité du réseau. Une abstraction discrète basée sur la notion d'évènement est utilisée pour capturer la communication entre le contrôleur et les routeurs. Dans ce niveau d'abstraction, Nettle représente le contrôleur OpenFlow comme étant une boite noire qui transforme le flux des messages arrivant du routeur en un flux de commandes à envoyer au routeur. Une abstrac-

tion continue d'un plus haut niveau permet de capturer des propriétés quantifiables du réseau qui varient avec le temps. Cette abstraction est implémentée en utilisant les techniques de la théorie de contrôle. Ainsi, Nettle fourni un framework pour dissimuler les détails de bas-niveau aux utilisateurs et leur donner un support pour le développement d'applications de contrôle pour les réseaux OpenFlow.

NOX [GKP+08] est surement le framework de *control plane* le plus connu pour les réseaux OpenFlow. Il s'agit, en effet, d'une librairie *open-source* pour le développement de contrôleurs OpenFlow en C++ et en Python. Plusieurs autres contrôleurs OpenFlow ont été proposés. Beacon [bea] est similaire à NOX sauf qu'il fournit une API en Java. Maestro [CCM10] fournit un mécanisme modulaire pour gérer l'état du réseau en utilisant des vues définies par le programmeur. Il a un support pour le *multithreading* ce qui permet d'augmenter considérablement le débit. Onix [KCG+10] fournit des abstractions pour partitionner et distribuer l'état du réseau à travers plusieurs contrôleurs, ce qui permet d'avoir un support à l'évolutivité du réseau, un problème rencontré lors de l'utilisation d'un seul contrôleur centralisé. SNAC [sna] fournit un modèle de haut niveau pour spécifier des stratégies de contrôle d'accès ainsi qu'un outil d'administration graphique.

Frenetic [FFH+10] est aussi un langage de haut-niveau basé sur le modèle FRP (*Functional Reactive Programming*) et dédié au niveau *control plane* des réseaux OpenFlow. Frenetic permet de programmer un ensemble de routeurs OpenFlow distribués. Frenetic a été proposé pour résoudre les conflits et les difficultés à écrire un programme pour la plateforme NOX/OpenFlow, tels que les difficultés à gérer les interactions entre des modules concurrents et les difficultés à gérer des interfaces de bas niveau. Ce langage est implémenté en python. Il fournit un modèle de programmation unifié et compositionnel. Il permet au développeur de se placer à un niveau plus élevé que NOX et d'avoir un vue de tous les paquets du réseau ce qui permet de mieux gérer les *flow entries* lors de leurs installation et désinstallation.

2.3.5 Langages pour le traitement de paquets

PacketC [DJ09] et DPDK [dpd] sont des langages de traitement de paquets qui sont propriétaires et qui ont été conçus spécifiquement pour une plateforme particulière. PacketC est utilisé dans l'environnement de CloudShield Technologies [clo] alors que DPDK est dédié aux architectures des processeurs Intel.

PacketC est un langage de haut-niveau qui repose sur un modèle de traitement parallèle des paquets. Le langage est destiné à une plateforme combinant des mémoires partagées et des mémoires dédiées avec différents processeurs spécialisés supportant l'exécution parallèle de plusieurs instances d'un même programme tel que chaque copie de programme traite un seul paquet. PacketC est l'approche propriétaire de *CloudShield Technologies* pour faire le traitement de paquets à haut débit, particulièrement pour les applications réseaux à données intensives telles que le *Deep Packet Inspection* (DPI), l'interception de paquets et les services sécurisés. Le langage permet d'abstraire les détails relatifs aux dispositifs spécialisés tels que les NPUs (*Network Processing Units*), les FPGAs (*Field Programmable Gate Array*) et les CAMs. L'exécution du langage sur la TCAM et l'utilisation des opérations sur les tables de recherche comme le *Masked Matching* ont été détaillés dans [DJR11].

PacketC déploie des concepts assez intéressants en introduisant de nouvelles structures dans un dialecte C et des méthodes dans un dialecte C++. Les principales structures sont les *Packet Information Blocks* destinées à contenir les adresses de l'emplacement des champs de la couche 2, 3 ou 4 pour chaque paquet, la structure *descriptor* pour la description des entêtes de protocoles standards avec la possibilité d'implémenter une succession de structures *descriptor* pour créer des piles de protocoles et les types *searchset* et *database* conçus pour contenir des *records* formant les entrées d'une base de données. Des méthodes comme *insert*, *delete*, *find* et *match* sont fournies pour faire des manipulations sur les bases de données.

Data Plane Development Kit (DPDK) est la solution propriétaire d'Intel pour faciliter le développement d'applications de traitement de paquets à haut débit qui sont destinées à ses plateforme. L'outil consiste en un ensemble de bibliothèques de *packet processing* visant à accélérer le traitement et améliorer les performances. Les principales composantes de DPDK sont le *Buffer Manager* responsable de l'allocation et la libération des buffers, le *Queue/Ring Manager* offrant des stratégies d'accès aux ressources en minimisant le temps d'attente, la *Flow Classification Library* fournissant des mécanismes pour le rassemblement les paquets dans des *flows* et les pilotes pour l'interfaçage avec des cartes Ethernet haut-débit. Le langage offre une certaine flexibilité en termes d'utilisation des processeurs et de leurs coeurs disponibles permettant l'extensibilité. DPDK offre la portabilité des applications d'une plateforme Intel à une autre ce qui facilite par exemple le passage d'un processeur Intel Atom à un processeur Intel Xeon.

2.3.6 Framework pour les routeurs programmables

Click [KMC+00] est un modèle de programmabilité pour les routeurs, caractérisé par une architecture modulaire. Un routeur Click est constitué de composants appelés éléments. Un élément est un module qui assure une fonction spécifique de traitement de paquets. L'idée de la plateforme Click est de décomposer les fonctionnalités d'un routeur au niveau le plus élémentaire puis de permettre de faire une composition de ces éléments. Un élément Click est une simple unité de traitement de paquets qui contient une action élémentaire (décrémenter un champ TTL, changer l'adresse IP, incrémenter le compteur de paquets) qui sera appliquée sur le paquet. La configuration d'un routeur Click est définie à travers un graphe orienté dont les sommets représentent ses éléments et les arrêtes sont les connexions entre les éléments. Un élément contient le code qui sera exécuté quand il va recevoir le paquet. Il a des ports d'entrée et des ports de sortie. Des paramètres de configuration peuvent être passés à l'élément au moment de l'initialisation du routeur. Chaque élément supporte également une ou plusieurs interfaces avec lesquelles il communique avec les autres éléments. Click supporte deux types de connexions : les connexions *push* et les connexions *pull*. Dans une connexion *push* les paquets passent de l'élément source vers l'élément destination. Tandis que dans la connexion *pull* les paquets suivent le sens inverse. Le stockage des paquets dans Click se fait dans des *queues*, ce qui permet de décrire explicitement comment les paquets sont mis en mémoire. L'emplacement des *queues* dans le graphe du routeur est important. Plus les *queues* sont placées loin dans le graphe, plus le routeur se verra faire beaucoup de traitement sur chaque paquet entrant avant de pouvoir traiter le paquet suivant. Le routeur ne peut passer à un autre paquet que s'il arrive à stocker le paquet en court dans un espace mémoire. Click utilise un mécanisme d'ordonnancement des tâches qui consiste à placer la liste des tâches sur une *task queue*. Chaque tâche est simplement une demande d'accès d'un élément au CPU. Les configurations dans Click sont décrites à travers un langage de configuration simple basé sur deux structures : (1) des déclarations qui permettent de créer les éléments et (2) des connexions entre les éléments déclarés. Le langage fournit un mécanisme d'abstraction appelé *compound element* qui permet au développeur du routeur de définir ses propres classes *element*. Un *compound element* est une portion de la configuration du routeur qui agit comme une classe *element*. Au moment de l'instanciation du routeur, chaque *compound element* est compilé dans l'ensemble des éléments simples correspondants qui le composent. Le langage est entièrement déclaratif. Il permet de spécifier quels éléments créer et comment les connecter. Le langage ne décrit pas comment traiter les paquets à l'intérieur des

éléments et il ne fournit pas une syntaxe spécifique pour décrire l'ordonnancement des tâches dans le CPU.

NetKit [CBH+03] est un modèle générique de composition de services qui a pour objectif de rendre la programmabilité du réseau indépendante de tout paradigme ou langage de programmation et de toute plateforme matérielle particulière. Dans cette approche basée sur la notion de composant, NetKit introduit des mécanismes pour que les services puissent être configurés dans le cas de déploiement, instanciation ou initialisation de services. Les services peuvent être aussi reconfigurés dynamiquement dans le cas d'une adaptation, extension, évolution ou suppression de services. En effet, ce modèle permet d'ajouter et de supprimer des composants de manière flexible. NetKit définit un composant à travers ses interfaces et ses réceptacles. Les interfaces expriment l'approvisionnement de service alors que les réceptacles indiquent le besoin en service, ce qui permet d'exprimer explicitement les dépendances entre les composants. Le *binding* est une autre notion utilisée dans NetKit pour définir les associations entre les interfaces et les réceptacles. Ces associations permettent de lier les composants dans leur environnement d'exécution qui est appelé capsule. Le support aux reconfigurations dynamiques est fourni à travers un méta-modèle avec des méta-interfaces et l'utilisation de la notion de réflexion qui permet l'inspection, l'adaptation et l'extension des systèmes. La fonctionnalité principale de la réflexion est qu'elle lie le méta-modèle au système sous-jacent par une relation causale de sorte qu'un changement dans le méta-modèle implique implicitement un changement dans le système sous-jacent. Des frameworks de composants sont utilisés pour ajouter la notion de contraintes et de règles spécifiques à un domaine. Ils permettent d'assurer la consistance et l'intégrité des systèmes. Ces frameworks n'acceptent que des *plugins* spécifiques à leur domaine. Les *plugins* appropriés sont sélectionnés de manière transparente au programmeur. Les principaux *plugins* proposés dans NetKit sont le *loader* et le *binder*: un *loader* spécifique est choisi dépendamment du composant à ajouter et un *binder* spécifique est utilisé dépendamment des interfaces à lier. NetKit est destiné à être déployé aussi bien sur des environnements de PC que sur des environnements de routeurs programmables avec des NPs. NetKit a été appliqué à la plateforme de l'IXP1200 où l'ensemble des composants de l'application ont été répartis sur l'ensemble des ressources du NP. Dans ce cas, si on décide de lier deux composants sur des *microengines* séparés, c'est un *binder* basé sur la mémoire *scratch* qui sera sélectionnée.

NetBind [CCK+02] est un outil qui permet de construire dynamiquement des *data paths* dans des routeurs basés sur les NPs. Un *binding* dynamique est réalisé entre des composants

indépendants de *packet processing* pour créer des *data paths* modulaires et extensibles partageant les ressources communes du même NP. NetBind crée des pipelines de *packet processing* à travers un *binding* dynamique de petits morceaux de codes écrits en langage machine. Dans NetBind, les composants du *data path* exportent des symboles qui sont utilisés durant l'opération de *binding*. Un *binder* modifie le code des composants pendant leur exécution. En résultat, les composants sont fusionnés dans un seul code. Le fonctionnement de NetBind rappelle celui des interpréteurs et des optimiseurs dynamiques de code. Bien que son objectif ne soit pas de produire du code optimisé pour une plateforme donnée, cet outil de composition de *data paths*, en réalité, part d'un code source qui est écrit en langage machine, introduit des remplacements et produit un code destination. Les modifications d'adresses physiques dans des blocs d'instructions en mémoires sont des techniques similaires à celles utilisées dans la traduction binaire ou les machines virtuelles des langages de programmation [CCK+02].

Le système NetBind est un ensemble de bibliothèques implémentées en C et C++ dans le host de l'IXP1200. Ces libraires permettent de modifier les instructions du NP pour créer des pipelines de *processing*. Ce système repose sur un modèle de spécification hiérarchique basée sur 3 niveaux : la spécification du routeur, la spécification du NP, la spécification de l'IXP1200 (comme l'outil a été implémenté pour l'IXP1200 d'Intel). Le premier niveau de la spécification permet de décrire le routeur à travers un ensemble de ports d'entrée, des ports de sortie et des *forwarding engines*. Le deuxième niveau est la spécification du NP qui permet de raffiner la spécification du routeur avec des informations sur le nombre de contextes matériels (les *threads*) qui vont exécuter les composants, ainsi que le *binding* de ces composants. Les composants sont groupés dans les pipelines de *processing* où un pipeline de *processing* est défini comme étant un ensemble de composants qui sont exécutés séquentiellement par le même contexte matériel. Les composants sont codés en langage machine et sont groupés dans les pipelines de *processing* qui sont exécutés par des *microengines*. Les composants échangent les paquets entre eux à travers des connexions de type *push* et *pull* comme les éléments dans le routeur Click. Dans ce niveau, on ajoute des symboles aux composants qui permettent d'abstraire les propriétés du *binding* entre les composants telles que les registres et les adresses des blocs d'instructions. Le troisième niveau est la spécification de l'IXP1200. Dans ce niveau, les composants sont reliés en suivant les propriétés associées à l'architecture du de l'IXP1200. Les symboles sont utilisés comme des points d'entrée, des points de sortie, des paramètres d'entrée et des variables globales. Les points d'entrée et de sortie sont des branchements qui permettent d'aller d'un module à un autre.

2.4 Résumé

Ce chapitre nous a permis d'établir une revue de littérature sur les réseaux programmables et d'avoir un aperçu sur les solutions et les langages qui ont été proposés dans ce domaine. Les *Software Defined Networks* ont considérablement révolutionné les réseaux d'aujourd'hui. Grâce aux séparations bien pensées qu'ils ont introduites, en définissant des blocs bien limités qui sont le *control plane*, le *data plane* et le *management plane*, les SDNs ont apporté la flexibilité et les abstractions nécessaires à la programmabilité des réseaux. OpenFlow est arrivé avec ses tables en pipeline pour donner un modèle de traitement de paquets intuitif, dans le sens où il reflète clairement le comportement des paquets à l'intérieur du routeur. D'autres langages ont été proposés traitant, chacun, un niveau donné; à notre connaissance, aucun langage n'a été proposé pour traiter tous les niveaux d'un *Software Defined Network*.

CHAPITRE III

NETLANG

3.1 Introduction

Dans ce chapitre, nous allons présenter notre nouveau langage de programmation NETLANG (*Network Programming Language*). NETLANG est un *Domain Specific Language* dédié aux environnements des *Network Processors* dans le but de permettre le développement d'applications pour les *Software Defined Networks*. NETLANG est un langage qui se veut simple et élégant qui utilise des structures déclaratives dans un style C.

L'objectif de ce chapitre est de présenter les propriétés et les caractéristiques de NETLANG. D'abord, nous présenterons la structure du langage, ses principales fonctionnalités ainsi que les types de données et les primitives qu'il supporte. Ensuite, nous expliquerons son implémentation où nous détaillerons principalement l'architecture de son compilateur.

3.2 Conception de NETLANG

NETLANG est un nouveau langage de programmation de haut-niveau, compilé et dédié au développement des applications réseaux. NETLANG a été conçu dans le but de permettre aux programmeurs des réseaux de développer des applications de traitement de paquets qui vont s'exécuter dans un environnement de NP. Il s'agit d'un langage spécifique au domaine des réseaux SDNs et dédié aux routeurs programmables. NETLANG est un langage de programmation de haut-niveau puisqu'il offre une abstraction du comportement des paquets dans le routeur et dans le NP en exploitant la sémantique d'OpenFlow. Certaines fonctionnalités de NETLANG sont écrites sous forme déclarative et d'autres s'écrivent sous forme procédurale.

NETLANG est une superposition de 4 couches : (1) une couche applicative, (2) une couche

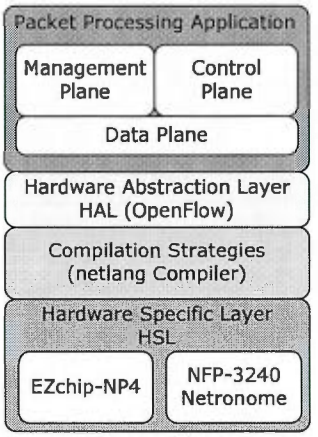

Figure 3.1 Composition de NETLANG

d'abstraction matérielle, (3) une couche de compilation et enfin (4) une couche liée au matériel (voir Figure 3.1). Etant un langage pour les réseaux SDNs, NETLANG permet de décrire les opérations sur un routeur au niveau du *control plane*, du *data plane* et du *management plane*. La couche d'abstraction matérielle utilise OpenFlow pour permettre d'écrire les opérations de traitement de paquets dans une logique de *flow tables* composées de *flow entries* qui présentent les règles de *forwarding* et composent ensemble la stratégie de transmission des paquets. Ensuite, on a la couche du compilateur où NETLANG sera traduit dans le langage machine. La couche HSL (*Hardware Specific Layer*) est la couche spécifique au matériel. Dans cette couche, l'application est générée de manière dépendante de la plateforme sur laquelle elle va s'exécuter.

3.2.1 Caractéristiques de NETLANG

NETLANG est un langage de haut-niveau qui est dédié, unifié et basé sur deux niveaux de langages. Ces caractéristiques sont expliquées dans ce qui suit :

- NETLANG est un langage dédié. Contrairement aux langages de programmation à usage général comme le langage C ou le langage Java, NETLANG est conçu spécifiquement pour permettre le développement d'applications réseau qui font du traitement de paquets. Il est dédié dans le sens qu'il est utilisé dans un contexte spécifique ; celui des réseaux SDNs et des routeurs programmables. En effet, il s'agit d'un langage compilé et il est fait pour s'exécuter dans des environnements de NPs et non pas pour les processeurs à usage général.

- NETLANG est un langage de haut-niveau. NETLANG offre un niveau d'abstraction matérielle

basé sur OpenFlow. En effet, c'est en utilisant la sémantique d'OpenFlow que nous avons pu exprimer le comportement des paquets dans un NP. Le pipeline des *flow-tables* et la structure de *flow-entries*, avec les champs de correspondance qui décrivent le flux de paquets à associer, et les instructions qui donnent les opérations à appliquer aux flux qui correspondent à la règle en question, permet au langage de se placer à un haut-niveau. En effet, c'est un langage de haut-niveau dans la mesure où il reste indépendant de la plateforme matérielle sur laquelle il va s'exécuter. Son adaptabilité aux différentes architectures de NPs sera présentée dans le chapitre 4.

- NETLANG est un langage unifié. NETLANG permet qu'à travers un langage unique de décrire différentes tâches de traitement de paquets comme par exemple des tâches d'extraction des champs de l'entête du paquet, des tâches de recherche dans des tables et des tâches de modification sur le paquet. Contrairement aux TOPs du NP4 d'EZchip qui ont chacun un langage à jeux d'instructions dédiés, plus le langage NPsl (*Network Processor Script Language*) qui est un langage spécial pour la configuration du NP4, NETLANG permet de définir les tables et décrire leurs contenus avec le même langage. De même, on peut intégrer les opérations de configuration du routeur telles que l'activation des ports et la définition des interfaces. Ainsi, NETLANG est un langage SDN unifié qui permet de bâtir une *switch* entière selon 3 vues différentes : *le management plane*, le *control plane* et le *data plane*.

- NETLANG est basé sur deux niveaux de langage. NETLANG est un langage à deux niveaux dont le premier niveau offre un niveau d'abstraction de description de comportement à travers les tables OpenFlow et le deuxième niveau est un langage de bas niveau qui est traduit en matériel de manière spécifique à l'architecture sous-jacente.

3.2.2 Fonctionnalités de NETLANG

NETLANG permet de programmer un noeud du réseau à 3 différents niveaux : i) au niveau du *management plane* ou le plan de gestion, ii) au niveau du *control plane* ou le plan de commande et enfin iii) au niveau du *data plane* ou le plan de données. Ainsi le programmeur peut spécifier la configuration à activer sur le routeur. Il peut aussi, au niveau du *control plane*, définir la stratégie de transmission des paquets basée sur un ensemble de règles de transmission. Enfin, le langage lui permet également de définir le modèle de transmission au niveau du *data plane*, en se basant sur la notion de *Packets Processing Path* (PPP) ou chemin de traitement de

paquets qu'on expliquera plus loin dans cette section.

3.2.2.1 Description de la configuration

NETLANG permet d'activer une configuration spécifique sur un routeur. La configuration détermine le profil de l'équipement en termes de nombre de ports actifs, avec une instanciation des paramètres relatifs à chaque port. La définition des ports du routeur se fait grâce à une structure préétablie appelée *port*. La définition d'une structure *port* se fait par l'instanciation des paramètres suivants (voir Code 1).

1- Le numéro du port : un simple identifiant logique qui permet d'abstraire la désignation physique de ce port qui est strictement liée à la plateforme matérielle. Par exemple SGMII_9 est une désignation utilisée dans le NP4 d'EZchip et MSF0 est utilisé dans NFP-3240.

2- La vitesse de l'interface associée à ce port qui permet de déduire le type de l'interface. Par exemple, une vitesse égale à 1000 mbps (1 GbE) implique que le port associé soit de type SGMII alors qu'une vitesse de 10000 mbps (10 GbE) signifie que le port associé est un port XAUI.

3- La direction de traitement qui peut être soit la transmission ou la réception. La direction de réception (RX) indique que les paquets arrivent sur ce port pour se diriger vers le NP. Dans ce cas, il s'agit d'un port d'entrée. La direction de transmission (TX) signifie que les paquets arrivent du NP et sont redirigés vers ce port qui est un port de sortie. Il est possible qu'un port traite des paquets dans les deux sens. Dans ce cas, les paramètres rx et tx seront activés par *true*.

Code 1 Définition d'un port avec NETLANG

```
port port_1 {
    number = 1;
    speed = 10000;
    rx = true;
    tx = false;
}
```

3.2.2.2 Définition des tables de transmission

Après la création d'une instance de routeur, une description des structures de recherches est nécessaire afin de remplir ses tables de transmission (*forwarding tables*). NETLANG utilise la sémantique d'OpenFlow dans la définition des structures de recherches. Ainsi, le développeur de l'application réseau peut créer facilement des *flow tables*, les remplir par les *flow entries* et les relier dans un ordre de sorte à former le pipeline tel que défini dans OpenFlow v1.1 (voir Code 2).

Code 2 Création d'une table OpenFlow avec NETLANG

```
/* table declaration */
   flowTable table0 = new flowTable(0);

/* entry declaration */
   flowEntry entry1 = new flowEntry() ;

/* match fields assignment */
   entry1.matchField.inputPort = 1 ;
   entry1.matchField.ethDstAddr = 00:1b:77:38:95:c1 ;
   entry1.matchField.ethType = 0x0800 ;

/* instructions assignment */
   entry1.instr(goto,table1);

/* entry insertion */
   table0.addEntry(entry1,1);
```

NETLANG permet aussi de définir des structures de recherche de manière générique. Une structure de recherche générique est composée de deux éléments : la clé de recherche et le résultat de recherche. C'est grâce à cet aspect qu'on peut définir dans NETLANG des tables ARP ou des tables de routage IPv4 ou IPv6 selon leur format conventionnel. Lorsque le programmeur veut définir une structure de recherche générique, il doit spécifier la taille de la clé, la taille du résultat, la composition de la clé et du résultat (par exemple telle séquence d'octets dans la clé représente une adresse IPv4). Enfin, il doit remplir la table en respectant le format qu'il a donné (voir Code 3). Lors de la déclaration d'une structure de recherche, il faut aussi spécifier

la stratégie de recherche qui va s'appliquer sur cette structure comme, par exemple, la stratégie *Exact Match* ou la stratégie *Longest Prefix Match*. C'est grâce à la stratégie de recherche que le compilateur va générer la structure matérielle adéquate. Par exemple, si le développeur indique qu'une stratégie de type *Exact Match* sera appliquée sur la structure, le compilateur va créer une table de hachage, alors que s'il spécifie une stratégie de *Longest Prefix Match* c'est une structure fast IP qui sera générée.

Code 3 Définition d'une Structure de Recherche Générique avec NETLANG

```
lookupStruct unicast_forwarding_table(exact) {
    keyStruct {
        element mac_addr_t:MAC_DA; /* MAC Destination Address*/
        element uint12_t:VID; /* VLAN Identifier */
    }
    resultStruct {
        element uint4_t:OUT_IF; /* Outgoing Interface Identifier */
    }
    /* unicast forwarding entries */
    entries {
        entry <44:55:66:77:00:02,100>,<2>;
        entry <44:55:66:77:00:03,100>,<2>;
        entry <44:55:66:77:00:04,100>,<1>;
        entry <44:55:66:77:00:05,100>,<2>;
        entry <44:55:66:77:00:06,100>,<3>;
        entry <44:55:66:77:00:07,100>,<2>;
    }
}
```

3.2.2.3 Spécification du modèle de transmission des paquets

La programmabilité du *data plane* dans NETLANG permet de définir le modèle de transmission des paquets ou le *forwarding model*. Dans NETLANG, le modèle de *forwarding* est un ensemble de *Packet Processing Path* ou chemin de traitement de paquets. Chaque *packet processing path* est composé d'un ensemble de blocs de traitement où chaque bloc peut contenir un type de tâche bien spécifique comme, par exemple, l'analyse de l'entête du paquet ou la réécriture de certains champs du paquet (voir Code 4). C'est dans ces blocs de traitement, que

le développeur peut écrire des instructions opérationnelles comme des affectations de valeurs à des variables, des opérations arithmétiques, des comparaisons et aussi des instructions *if*, des branchements à travers le mot réservé *goto* et des boucles *for* et *while*. En effet, en dehors du *data plane* le langage est purement déclaratif alors que dans le *data plane* le langage peut être descriptif et procédural. NETLANG permet de regrouper les instructions dans 5 types de blocs prédéfinis.

1- Le bloc *ingress* dans lequel le paquet entrant sera reçu, et dans lequel des opérations préliminaires peuvent être appliquées comme la pré-analyse et la classification. C'est dans ce bloc qu'on peut relever des informations de métadonnées sur le paquet telles que sa taille et le numéro de port sur lequel il est arrivé.

2- Le bloc *parse* qui englobe toutes les opérations de *parsing* et d'analyse des champs de l'entête du paquet dans le cas d'une application de simple *forwarding* simple. L'analyse peut traiter des données utiles du paquet comme dans le cas des applications de sécurité ou de DPI (*Deep Packet Inspection*). Dans ce bloc, on peut extraire les valeurs des champs d'adresses MAC destination et de l'ethertype par exemple pour déduire le type de paquet ce qui permettra de savoir s'il s'agit d'un paquet IPv4, ou si le paquet contient un tag de VLAN ou un label MPLS. Dans le bloc *parse*, on peut faire l'extraction des champs du paquet en se limitant à la couche 2 du modèle OSI comme on peut faire une analyse plus profonde en extrayant les champs de la couche 3, 4 et même 7.

3- Le bloc *search* dans lequel toutes les opérations de recherche sur les tables de *forwarding* vont être lancées et les résultats récupérés pour être passés au bloc *modify* ou au bloc *parse* pour une analyse additionnelle sur le paquet. Ce sont les tables définies dans le *control plane* qui vont être utilisées dans ces opérations de recherche.

4- Le bloc *modify* est responsable d'appliquer les actions trouvées dans les résultats de recherches. Des opérations de réécriture sur le paquet sont appliquées dans ce bloc. La décrémentation du champ TTL est un exemple de modification qui peut s'appliquer sur un paquet IP dans une application de routage.

5- Le bloc *egress* qui assure les opérations de préparation du paquet pour son émission sur un port de sortie. Le réassemblage du paquet et éventuellement le cryptage peuvent s'appliquer dans ce bloc. Ensuite le paquet est placé sur la file de sortie en respectant la politique de priorisation s'il y a lieu.

Code 4 Définition d'un Chemin de traitement de paquets avec NETLANG

```
processingPath path1 {
    ingress processing_bloc1() { ... }
    parse processing_bloc2() { ... }
    search processing_bloc3() { ... }
    modify processing_bloc4() { ... }
    egress processing_bloc5() { ... }
}
```

La figure 3.2 représente un exemple de *forwarding model* implémenté dans un routeur programmable composée de deux *Packet Processing Paths*. Le premier *Packet Processing Path* relie le port d'entrée numéro 1 au port de sortie numéro 4. Il est composé de 5 modules de traitement : un module récepteur, un module *parse*, un module *search*, un module *modify* et enfin un module transmetteur. Ce *path* utilise un seul module mémoire. Le deuxième *Packet Processing Path* est dédié aux paquets qui arrivent sur le port 3 et qui vont être expédiés sur le port 5. Il est composé de 7 modules de traitement. Au cours de l'acheminement d'un paquet à travers ce chemin de traitement, l'accès à deux modules mémoires est effectué. Le traitement du premier *path* est différent du deuxième. De plus, les paquets qui arrivent et qui en sortent du *path1* sont différents des paquets traités par le *path2*.

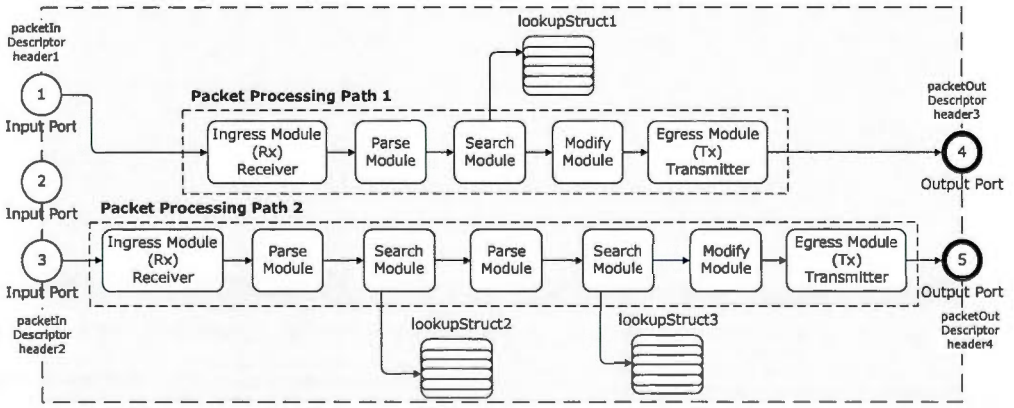

Figure 3.2 Représentation d'un modèle de *forwarding*

Dans NETLANG, un chemin de traitement de paquet est aussi défini par le port d'entrée et le port de sortie qui lui sont associés. Au point d'entrée d'un chemin donné, on associe un descripteur de paquet appelé *header* qui fournit le format des paquets attendus à l'entrée de ce

port. Une strucutre *header* est composée d'une ou plusieurs entrées de type *field*. Un champ est défini à travers son type, sa taille et éventuellement sa valeur. A la sortie, c'est un autre format de *header* qui est produit après le traitement qui a été appliqué sur le paquet. Il est possible de faire l'imbrication des structures *header* avec le mot réservé *next* (voir Code 5).

Code 5 Définition d'un Chemin de traitement de paquets avec NETLANG

```
/* QinQ encapsulation */
header h1 {
    field mac_addr_t:C_DA ; // 6 Bytes
    field mac_addr_t:C_SA ; // 6 Bytes
    field hex:ether_type:0x88a8; // 2 Bytes
    field uint16_t:S_tag; // 2 Bytes
    field hex:ether_type:0x8100; // 2 Bytes
    field uint16_t:C_tag; // 2 Bytes
    field hex:ether_type:0x0800; // 2 Bytes
}
/* MAC in MAC encapsulation */
header h2 {
    field mac_addr_t:B_DA ;
    field mac_addr_t:B_SA ;
    field hex:ether_type:0x88a8;
    field uint16_t:B_VID;
    field hex:ether_type:0x88e7;
    field uint16_t:I_SID;
    next h1 ;
}
```

Le descripteur du paquet à l'entrée est encapsulé dans une structure appelée *packetIn* alors que pour le port de sortie on utilise une structure *packetOut*. Le passage du paquet à travers les différents modules du *Packet Processing Path* permet de le transformer du format *packetIn* au format *packetOut*.

Le code 6 représente la structure d'une application NETLANG. Des exemples d'applications complètes sont donnés dans le chapitre 5.

Code 6 Structure d'un programme NETLANG

```
<constants>
<ports>
 <lookup structures>
    <flow tables>
 <processing paths>
    <processing blocks>
```

3.3 Spécification de NETLANG

3.3.1 Organisation d'un programme NETLANG

Dans un programme NETLANG, tout d'abord, il faut définir les paramètres de configuration du routeur. Ceci permet de donner le profil de l'instance de l'équipement sur lequel l'application réseau va s'exécuter. Les ports définis dans cette première partie du programme vont être utilisés dans la définition des *Packet Processing Paths* et dans les opérations de réception et d'émission des paquets. La première partie du langage traite donc le niveau *management plane*. Ensuite, on trouve la partie *control plane* qui englobe la définition de toutes les structures de recherche qui seront utilisées par les modules *search*. C'est dans cette deuxième partie du programme, qu'on peut définir les tables OpenFlow et les structures de recherche génériques. Enfin, la dernière partie d'un programme NETLANG contient la description du modèle de *forwarding* dans lequel on peut définir un ou plusieurs *Packet Processing Paths*. Un *Packet Processing Path*, ou chemin de traitement de paquet, contient la séquence des opérations qui vont être appliquées sur le paquet. Dans chaque chemin de traitement de paquets, on a une partie descriptive qui donne la structure des paquets destinés à être traités par ce *path* ainsi que le format de ces paquets à la sortie. Cette description permet d'aider le programmeur à bien définir le traitement à l'intérieur du *path*. Les instructions du *Processing Path* forment du code procédural qui donne les opérations de traitement de paquet module par module.

3.3.2 Structures et Types de données

NETLANG supporte 3 catégories de types de données (1) les types de données simples qu'on trouve dans les langages de programmation généraux comme le type *int* ou le type booléen (2) les types de données spécifiques au réseau comme l'adresse IP ou l'adresse MAC (3) et enfin

les types de données liés à OpenFlow telles que la *flow table* et les *flow entries*. Le tableau 3.1 présente l'ensemble des types de données supportés par NETLANG.

Catégorie	Types de données	Signification
Basique	constant	Constante
	boolean	Booléen
	int	Entier
	string	Chaine de caractère
	hex	Hexadécimal
	uint8_t	Entier non signé codé sur 8 bits
	uint16_t	Entier non signé codé sur 16 bits
	uint32_t	Entier non signé codé sur 32 bits
	uint64_t	Entier non signé codé sur 64 bits
Openflow	flowTable	Une table composée de flow entries
	groupTable	Une table composée de groupEntry
	flowEntry	Une entrée dans la flowTable
	groupEntry	Une entrée composée d'actions
	actionBucket	Un ensemble d'actions
	pipeline	Un ensemble de flow tables
Réseau	ipv4_addr_t	Une adresse IPv4
	ipv6_addr_t	Une adresse IPv4
	mac_addr_t	Une adresse MAC
	header	Une structure composée de fields
	port	Une structure pour décrire un port
	packet	Un paquet
	lookupStruct	Une structure de recherche

Tableau 3.1 Types de données de NETLANG

3.3.3 Primitives

NETLANG offre des primitives qui présentent les points d'entrée du langage et qui permettent d'automatiser les opérations basiques sur le paquet. Le langage expose deux types de primitives : les primitives génériques qui peuvent s'appliquer dans le contexte des applications

de *forwarding* de paquets, comme du routage IPv4 par exemple, et des primitives relevées du mécanisme d'OpenFlow. Les primitives de NETLANG sont principalement des fonctions de modification des *flow entries* dans les *flow tables*, des fonctions de lecture de champs spécifiques du paquet sous forme de *getter* et des fonctions d'écriture dans le paquet sous forme de *setter*. Ces fonctions prennent en paramètres les champs de correspondance d'OpenFlow pour, soit les lire, soit les modifier. D'autres fonctions permettent de faire des manipulations sur les tables OpenFlow telles que la suppression ou la mise à jour d'une entrée. L'accès aux structures de recherches et aux *flow tables* se fait par la fonction *lookup()* qui lancera une recherche sur la table spécifiée. La récupération du résultat se fait via la fonction *getResult()*. Les tableaux 3.2 et 3.3 listent les primitives supportées par NETLANG.

Types de primitives	Primitives
Primitives générales	addEntry()
	deleteEntry()
	updateEntry()
	getPacket()
	lookup()
	getResult()
	getHeaderField()
	setHeaderField()
	pushHeaderField()
	popHeaderField()
	sendPacket()

Tableau 3.2 Primitives Génériques de NETLANG

3.4 Implémentation de NETLANG

3.4.1 Grammaire de NETLANG

La grammaire de NETLANG a été définie de façon modulaire et assez générique pour rendre l'extensibilité du langage plus facile. Elle est écrite dans SableCC. Elle est formée de 4 parties : (1) la définition du package qui est le nom du langage, (2) la définition des *helpers* qui vont aider dans la définition des jetons, (3) la partie *tokens* est la troisième partie de la grammaire dans laquelle on définit les jetons du langage et on spécifie ceux qui seront ignorés

Types de primitives	Primitives
Primitives OpenFlow	SetIPSrcAddr()
	getInputPort()
	getEthDstAddress()
	getEthSrcAddress()
	getEtherType()
	getVlanTag()
	setVlanId()
	getProto()
	getSrcIpAddress()
	getDstIpAddress()
	getSrcPort()
	getDstPort()
	pushVlanHeader()
	popVlanHeader()
	pushMPLSHeader()
	popMPLSHeader()
	setVlanPriority()
	setMplslabel()
	setMplsTTL()
	DecrementMplsTTL()
	SetIPDstAddr()
	SetIPtos()
	SetIPecn()
	SetIPttl()
	DecrementIPttl()
	SetSrcPort()
	SetDstPort()

Tableau 3.3 Primitives Spécifiques de NETLANG (Suite)

lors de la compilation (4) et, enfin, les productions qui donnent l'ensemble des règles de la syntaxe de NETLANG. Dans l'implémentation de la grammaire, nous avons respecté la même nomenclature que nous avons utilisée dans ce mémoire.

3.4.2 Architecture du compilateur

Le maintien de la consistance de l'application définie via NETLANG est assuré à travers le compilateur du langage. En effet, le compilateur assure qu'un changement dans un niveau donné ne va pas affecter le reste de l'application. Les vérifications implémentées au sein du compilateur permettent d'éviter qu'un changement au niveau du *control plane* n'affecte le *data plane* ou le *management plane* ou qu'une modification du *management plane* n'entraine pas l'invalidité de tout le système. Le compilateur de NETLANG a été conçu de manière modulaire sous forme de blocs où il devient très facile d'y ajouter ou d'en supprimer des fonctionnalités. Les blocs fonctionnels du compilateur de NETLANG peuvent être classés selon 3 catégories : les blocs de *front-end*, les blocs intermédiaires et les blocs de *back-end* (voir Figure 3.3).

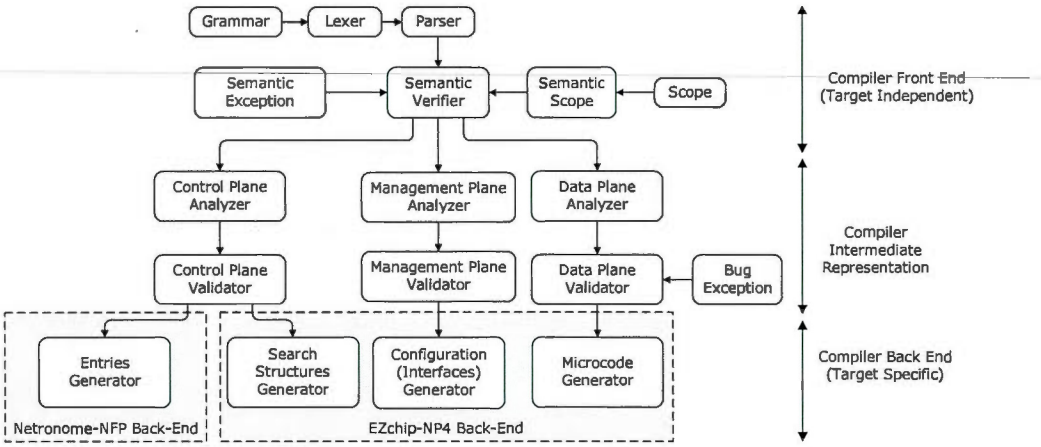

Figure 3.3 Architecture du compilateur de NETLANG

La partie *front-end* du compilateur est totalement indépendante de la plateforme destination. On y trouve la définition du langage NETLANG à travers sa grammaire. Des blocs comme le *lexer* et le *parser* appartiennent aussi à la partie frontale. Ils assurent l'analyse lexicale dans le sens où ils vérifient que le code source respecte les règles définies dans la grammaire. Le *lexer* se charge de décortiquer le code en entrée et en extraire les jetons. Le *parser*, ensuite, parcourt l'arbre syntaxique et vérifie que toutes les séquences de jetons répondent bien à une règle dans

la grammaire du langage. Le bloc de vérification sémantique assure des tests plus judicieux en effectuant une analyse plus pertinente sur le code source. Vérifier qu'une variable ait été déclarée avant son utilisation est un exemple de vérification sémantique. Le bloc de vérification sémantique utilise les blocs *Scope* et *Semantic Scope* pour gérer la portée des variables d'un bloc à un autre à travers le code. Le bloc *Semantic Exception* permet de retourner les erreurs sémantiques sur le code lors de la compilation.

La partie qui est entre le *front-end* et le *back-end* du compilateur permet d'établir une représentation intermédiaire qui va préparer et faciliter la génération du code. Dans cette partie, les blocs *Data Plane Analyzer*, *Management Plane Analyzer* et *Control Plane Analyzer* revisitent le code pour la collecte d'information et la construction d'une représentation simplifiée de l'application. Cette représentation est appelée une représentation intermédiaire parce qu'elle vient sous une forme de plus bas-niveau que le langage source, c'est à dire NETLANG, mais elle reste de plus haut-niveau par rapport au langage machine.

La partie *back-end* du compilateur est strictement spécifique au matériel. Les blocs du *back-end* sont regroupés selon la technologie cible. Deux adaptateurs ont été conçus et implémentés pour NETLANG. Le premier adaptateur est dédié au NP4 d'EZchip et il est composé d'un générateur de structures de recherche dont une partié est dédié à la TCAM NL11000 de *NetLogic*, un générateur des interfaces et un générateur du microcode. Le second adaptateur est pour le NFP-3240 de Netronome et il contient principalement un générateur des entrées pour la TCAM NL7000 de *NetLogic* aussi. L'adaptabilité du langage sera présentée en détails dans le chapitre 4.

La figure 3.3 représente l'architecture du compilateur de NETLANG. Comme le compilateur a été développé en Java, alors chaque boite dans le schéma représente une classe Java. Les flèches permettent de comprendre l'ordre dans lequel le compilateur s'exécute. La partie supérieure dans le schéma représente la partie frontale du compilateur. Ensuite, c'est les classes qui permettent de bâtir la représentation intermédiaire. Enfin, les classes dans le niveau inférieur de l'architecture représente la partie *back-end*.

Dans le code du compilateurs, il y a deux types de classes : les classes générées à partir de la grammaire (celles-ci ne sont pas représenté dans la figure 3.3) et les classes que nous avons dévelopées nous mêmes qui sont au nombre de 15. Ce code n'est pas publique parce que c'est la propriété du laboratoire de téléinformatique à l'UQAM et d'Ericsson Montréal. De plus, la

partie *back-end* du compilateur génère du code pour des plateformes qu'on ne peut acquérir que sous license et avec une entente de confidentialité.

3.4.3 Environnement de Développement

Dans l'implémentation de NETLANG et de son compilateur, nous avons principalement utilisé les outils suivants:

- SableCC [GH98]: SableCC est un compilateur de compilateurs qui permet de générer tout un framework orienté-objet pour le développement des compilateurs, des interpréteurs et des parseurs. En effet, SableCC permet, à partir d'une grammaire, de bâtir un arbre syntaxique fortement typé et il fournit également le patron de conception pour parcourir cet arbre. SableCC offre une séparation claire entre le code généré et le code écrit par le développeur, ce qui réduit le cycle de développement. C'est un outil très efficace qui a énormément facilité le développement du compilateur de NETLANG.
- Le langage de programmation Java [AGH00]: l'ensemble de toutes les classes qui implémentent le compilateur de NETLANG sont écrites en langage Java. En effet, le code généré par SableCC est en Java et son intégration et l'utilisation des méthodes et des classes qu'il crée à partir de la grammaire est plus facile si on reste dans le même environnement, celui de Java.

3.5 Résumé

NETLANG est le langage de programmation de haut-niveau que nous proposons pour le développement d'applications réseaux destinées à s'exécuter dans des environnements de NPs. Dans la définition du langage, nous avons suivi la vision SDNs et nous avons introduit des notions du modèle OpenFlow afin de rendre la description du comportement des paquets à l'intérieur du routeur assez générique pour que le langage puisse être facilement adapté à des plateforme différentes. NETLANG est basé sur deux niveaux de langages: un langage d'abstraction de description du comportement des paquets à travers un pipeline de tables à la OpenFlow et un langage qui traduit ce comportement au niveau matériel. L'adaptabilité de NETLANG aux architectures matérielles est implémentée au niveau de son compilateur dans la partie *back-end* où on retrouve la génération du code spécifique à la technologie cible.

CHAPITRE IV

ADAPTABILITÉ DE NETLANG

4.1 Introduction

Dans l'implémentation du compilateur de NETLANG, nous nous sommes intéressés à deux différentes plateformes matérielles : la carte *NP4-eval System* d'EZchip [Tec] et la plateforme *AMDA-0021-0003* comportant le NFP-3240 de Netronome [Com]. L'objectif de ce chapitre est de présenter l'adaptabilité de NETLANG par rapport aux architectures considérées. L'adaptateur de NETLANG pour EZchip génère du code *NPsl* et du *microcode* qui sont les principaux langages utilisés dans le NP4 d'EZchip alors que l'adaptateur de NETLANG pour le NFP-3240 de Netronome produit un langage propriétaire à Netronome qui ressemble au langage C.

4.2 Adaptateur de NETLANG pour EZchip-NP4

L'adaptateur de NETLANG pour la plateforme d'EZchip est responsable de la génération du code adapté pour EZchip-NP4. Dans ce module du compilateur, le langage est traduit dans ce que EZchip utilise comme outils et langages propriétaires, spécialement conçus pour son NP. Le premier langage est le langage NPsl (*Network Processor Script Language*) qui est un langage déclaratif très simple constitué principalement de structures dans le style du langage C. L'autre language est le *microcode* ou *EZlanguage* qui est une sorte d'assembleur optimisé pour les TOPs avec un jeu d'instructions spécialisées. Dans cette section, nous présenterons, d'abord, la carte d'EZchip ainsi que les langages qui y sont utilisés. Ensuite, nous expliquerons le fonctionnement du module du compilateur qui gère l'adaptabilité à EZchip.

4.2.1 Description de l'environnement d'EZchip-NP4

Avant d'expliquer comment NETLANG est compilé pour le NP4 d'EZchip, nous allons donner une description des principales composantes de ce *Network Processor*. Nous commencerons par la carte qui le supporte et ensuite nous présenterons les outils avec les quels il s'y interface. En effet, l'environnement de développement utilisé pour EZchip utilise deux langages qui sont le langage NPsl et EZlanguage. La carte *NP4-Eval System* est une troisième composante de cet environnement qu'il faut aussi présenter. Dans les trois points suivants, nous présentons chacune de ces trois composantes du EZchip-NP4 :

- *NP4-Eval System.* Il s'agit d'une plateforme matérielle basée sur le processeur NP4 (voir Figure 4.1). Cette carte est programmable à travers le *microcode* et elle est configurable à travers le langage NPsl. Elle est conçue afin d'être utilisée comme un système d'évaluation pour les vendeurs qui vont créer leurs propres applications sur le NP4. Au coeur du système, on a un NP4 qui permet de faire le traitement de paquets. Le NP4 délivre un débit de 100 Gigabits et permet d'implémenter des applications flexibles de traitement de paquets permettant d'effectuer des manipulations sur l'entête du paquet, créer des tunnels, faire de la commutation au niveau 2, du routage IPv4 ou IPv6 au niveau 3, de la classification de paquets, etc. Le NP4 contient un gestionnaire de trafic intégré qui est responsable de la QoS, la gestion des paquets par flux et par classe de paquets, le contrôle de congestion et les statistiques. *NP4-Eval System* contient également un *Host CPU* (MPC8543) qui offre des fonctionnalités de *control plane* à travers un système d'exploitation *Linux* [Pet00]. La combinaison du NP4 avec un CPU à usage général forme une solution complète de *data plane* et de *control plane.* Ce système est équipé de 5 modules d'interfaces d'entrée/sortie appelés *Slot* et numérotés de *Slot1* à *Slot5.* Le *Slot1* est un module de gestion qui contient 6 ports : 2 ports à 10 GbE SFP+, 2 ports SFP SGMII à 1 GbE, un port RJ45 à 10/100/1000 Mbps et un port console RS-232. Contrairement au *Slot1* qui ne peut être utilisé que pour la gestion du système (réinitialisation du système, connexion au *Host*, configuration), les 4 autres *Slots* (*Slot2* à *Slot5*) sont dédiés aux interfaces réseaux sur lesquelles on peut interconnecter des machines et faire circuler du trafic. Il existe deux types de modules pour les interfaces réseaux : le module SGMII et le module XAUI. Un module SGMII comprend 8 ports SFP à 1 GbE. Il peut être placé sur le *Slot3*, le *Slot4* et/ou le *Slot5.* Quant au module XAUI, il contient deux ports SFP+ à 10GbE et il peut être placé sur le Slot2, le *Slot3*, le *Slot4* et/ou le *Slot5.* Le système

que nous avons utilisé est un système qui contient 2 modules SGMII placé sur le *Slot4* et le *Slot5* et un module XAUI sur le *Slot2*. Le *Slot3* n'est pas utilisé. En plus, des modules d'interfaces I/O, du NP4 et du *Host*, la carte contient également des mémoires externes pour la recherche et les statistiques et une mémoire TCAM. La carte arrive avec du logiciel dont principalement un système d'exploitation Linux ELDK (*Embedded Linux Development Kit*) qui s'active automatiquement lors de la mise sous tension.

- *Network Processor Script Language (NPsl).* Il s'agit d'un langage propriétaire à EZchip qui permet de créer des fichiers script qui sont utilisés pour l'initialisation du NP4. Les scripts NPsl ont une représentation textuelle qui ressemble à des appels d'API avec des paramètres. Il s'agit d'un outil très pratique qui est utilisé pour la configuration du NP4 et l'initialisation des structures de recherche. Le langage NPsl supporte un ensemble de commandes appartenant à différents groupes de routines comme par exemple le *Search Structure Group Commands* pour les structures de recherche et le *Channel Group Commands* pour les interfaces. L'inconvénient avec le langage NPsl est qu'il requiert que les entrées dans les structures de recherche soient écrites en hexadécimal. Coder les entrées à la main est un tâche qui peut introduire des erreurs et entraîner un comportement dysfonctionnel du routeur.

- *EZlanguage.* Le NP4 d'EZchip est basé sur une architecture en pipeline avec des TOPs dans chaque niveau du pipeline. Les TOPs ne sont pas des processeurs RISC standards mais plutôt des moteurs de traitement développés pour être spécialisés. Les instructions assembleur courantes de type *move* ou *alu* sont complémentées par des macros qui assurent des tâches de haut-niveau. Par exemple, le *TOPparse* fourni une instruction *Checksum* qui permet de calculer le champ *Checksum* d'un entête IP en une seule instruction machine. Les instructions *Decode_MAC*, *Decode_MPLS*, *Decode_IPV4*, *Decode_IPV6*, *Decode_TCP*, *Decode_UDP* sont également fournies par le *TOPparse* pour le décodage direct des entêtes MAC, MPLS, IPv4, IPv6, TCP et UDP respectivement. Les valeurs des champs récupérés sont placées dans des registres spécifiques. L'architecture des TOPs réduit considérablement la taille des programmes si on les compare aux architectures RISC standards. Une opération de recherche dans une table est réalisée dans le *TOPsearch* à travers une seule instruction *lookup* où il faut spécifier la clé de recherche, la structure de recherche qu'il faut accéder et le registre qui va contenir le résultat de la recherche.

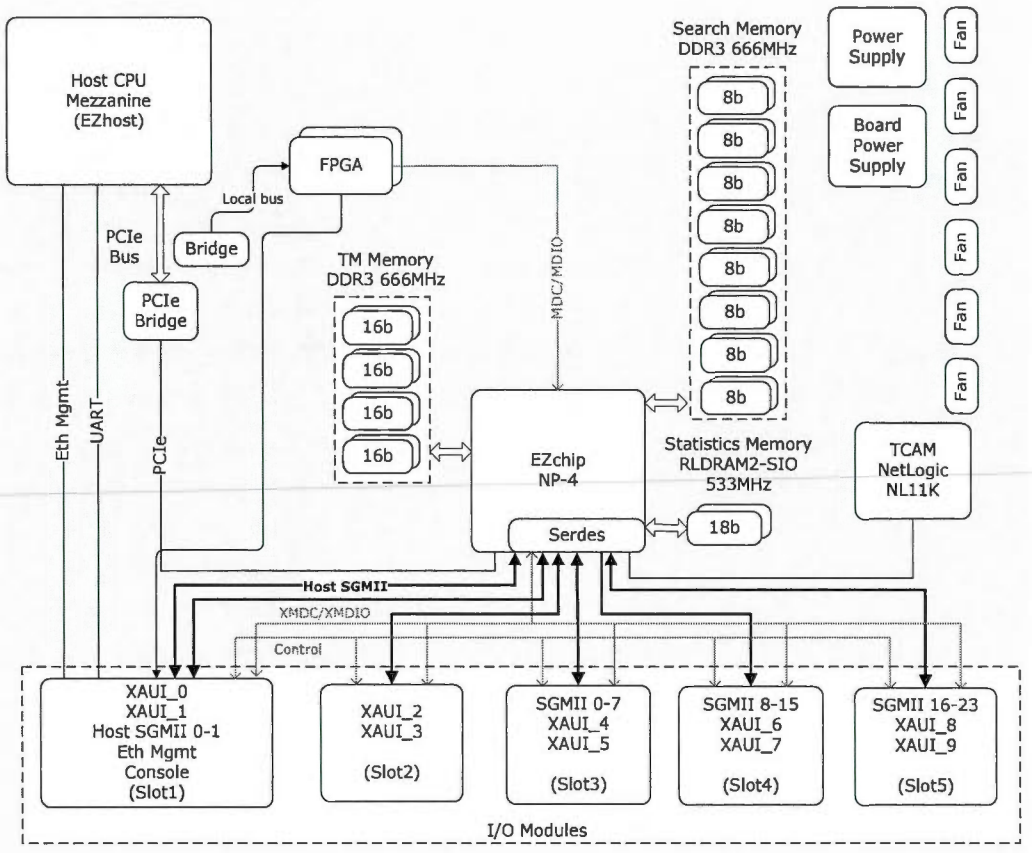

Figure 4.1 Système d'évaluation d'EZchip-NP4 [Tec]

Les 4 TOPs : *TOPparse*, *TOPresolve*, *TOPmodify* et *TOPlearn* utilisent un ***microcode*** similaire. Par contre, le *TOPsearch* utilise des instructions ***macrocode*** qui lui sont propres comme par exemple la macro *LookupTCAM*. Le NP4 d'EZchip repose sur un modèle de programmation linéaire simple qui utilise une image unique concentrée sur un paquet, ce qui permet de faire abstraction du parallélisme en arrière. Le développeur ne prend pas en compte qu'il y a plusieurs moteurs disponibles dans chaque niveau du pipeline. Ceci rend la programmation du NP plus simple par rapport aux environnements qui mettent en oeuvre le *multithreading*. De plus, l'ordonnancement des paquets sur les TOPs, le passage des données d'un TOP à un autre, le maintien de l'ordre des trames dans le pipeline et la gestion de l'accès aux ressources communes, comme les mémoires par exemple, sont toutes des fonctionnalités totalement gérées matériellement et donc transparentes au programmeur. Pour programmer le NP4, il faut écrire 5 petits programmes séparés et indépendants; un programme pour chaque niveau de TOP : un code TOPparse.asm pour le *TOPparse*, un code *TOPsearch.asm* pour le *TOPsearch*, un code *TOPresolve.asm* pour le *TOPresolve*, un code *TOPlearn.asm* pour le *TOPlearn*, et un programme *TOPmodify.asm* pour le *TOPmodify*. Aucune synchronisation n'est requise entre les différents programmes. En effet, la sortie d'un programme est utilisée comme entrée pour le programme suivant.

4.2.2 Les générateurs pour EZchip-NP4

L'adaptateur pour EZchip est composé de 3 modules : le *Search Structure Generator* pour la création et la population des structures de recherche, le *Configuration Generator* pour l'activation des interfaces et le *Microcode Generator* pour produire le code des TOPs (voir le module *EZchip-NP4 Back-End* dans la Figure 3.3). Chaque générateur fait l'analyse du code source NETLANG et produit un code destination au fur et à mesure qu'il parcourt le code de départ. Le générateur des structures de recherche et le générateur des interfaces produisent du code NPsl contrairement au générateur de *microcode* qui est responsable de donner les instructions machine dans *EZlanguage*.

4.2.2.1 Le générateur des interfaces

Lors du parcours du fichier source NETLANG, le premier bloc rencontré est le bloc qui rassemble l'ensemble des interfaces constituant le routeur. On rappelle que dans NETLANG une instance de routeur est définie par un ensemble de ports. Le compilateur récupère les

	Code source NETLANG	Code NPsl généré

Exemple1	port port_1 {
	number = 1;
	speed = 2500;
	rx = true;
	tx = true ;
	}
		EZapiChannel_Config(Channel = 0,
		Command =
		EZapiChannel_ConfigCmd_SetIFParams,
		IF_Type = SGMII,
		IF_Number = 8,
		RX_Enable = TRUE,
		TX_Enable = TRUE,...);

Exemple2	port port_2 {
	number = 2;
	speed = 10000;
	rx = true;
	tx = true ;
	}
		EZapiChannel_Config(Channel = 0,
		Command =
		EZapiChannel_ConfigCmd_SetIFParams,
		IF_Type = XAUI,
		IF_Number = 0,
		RX_Enable = TRUE,
		TX_Enable = TRUE,...);

Tableau 4.1 Génération du code *NPsl* spécifique aux interfaces

paramètres liés à chacune des interfaces qui sont encapsulés dans la structure *port* pour : soit les utiliser directement dans le code généré, soit les interpréter et en déduire de nouvelles valeurs qui sont nécessaires pour compléter l'ensemble des structures créées. L'adaptateur pour EZchip qui gère l'activation et la configuration des interfaces utilise la routine *EZapiChannel_ Config*, en spécifiant la commande *EZapiChannel_ ConfigCmd_ SetIFParams*.

Le mécanisme de traduction de NETLANG vers le langage NPsl est représenté dans le Tableau 4.1 qui présente deux exemples d'activation et de configuration des ports dans un routeur. Bien qu'il s'agisse du port numéro 1 et du port numéro 2, dans le code de NETLANG, le code correspondant en NPsl ne respecte pas cette numérotation puisqu'il doit obéir à des contraintes liées à la carte sur laquelle le code va être exécuté. En effet, pour activer et configurer des interfaces sur la plateforme *NP4-eval System*, l'adaptateur doit tenir compte de deux règles. (Règle 1) Seulement deux types d'interfaces sont possibles : les interfaces XAUI qui correspondent à une vitesse de 10 GbE et les interfaces SGMII qui correspondent à une vitesse de 2.5 GbE. (Règle 2) Il n'y a que 4 interfaces XAUI qui sont numérotées de 0 à 3 et 16 interfaces SGMII numérotées de 8 à 23. Ainsi, si le port défini dans NETLANG, est un port qui a une vitesse égale à 2500 (2500 Mbps = 2.5 Gbps) alors ça sera un port SGMII comme le cas dans l'exemple 1 du Tableau 4.1. Alors que si c'est un port ayant une vitesse égale à 10 000 (10 000 Mbps = 10 Gbps), ça sera une interface XAUI comme dans le cas de l'exemple 2.

Après la détermination du type de l'interface, l'adaptateur va déduire l'identifiant de cette interface. Le port *port_ 1* est le premier port SGMII qui a été déclaré dans le code source, c'est pour ça que le compilateur lui a affecté le numéro d'interface 8 dans le code généré. De même le port *port_ 2* est le premier port XAUI qui a été utilisé dans l'application, donc l'adaptateur lui a assigné l'identifiant 0.

Les valeurs des paramètres *number* et *speed* de la structure *port* ne sont pas utilisées directement dans le code NPsl généré. Par contre, les paramètres *rx* et *tx* sont les mêmes que *RX_ Enable* et *TX_ Enable* dans la routine *EZapiChannel_ Config*, ce qui permet au compilateur d'utiliser directement leurs valeurs sans faire de calcul supplémentaire. Le reste des paramètres qu'on voit sur les structures générées (ainsi que d'autres paramètres omis dans ces examples pour des raisons de clareté pour faciliter la lecture) sont des paramètres par défaut liés à la plateforme utilisée. Par exemple la valeur 0 pour l'attribut *Channel* est due au fait qu'il n'y a qu'un seul NP sur la carte et que l'identifiant de ce NP est 0.

4.2.2.2 Le générateur des structures de recherche

En plus du générateur des interfaces dans le langage NPsl, l'adaptateur de NETLANG pour EZchip contient également un générateur de structure de recherche. Ce module produit aussi du langage NPsl et permet principalement de faire les 3 opérations suivantes:

- Création d'une structure de recherche. En poursuivant le *parsing* du code source, le compilateur va produire une routine *EZapiStruct_ Create* à chaque bloc *lookupStruct{...}* du langage NETLANG. La routine *EZapiStruct_ Create* permet de créer une structure de recherche dans la plateforme d'EZchip. Pour remplir cette routine, le compilateur a besoin de connaître le type de la structure, la taille de la clé, la taille du résultat et le nombre maximal des entrées qu'elle va contenir. Le type de la structure requis est déduit de la stratégie de recherche passée en entrée (voir Tableau 4.2). Il calcule donc ces paramètres. Le type de la recherche est déduit de la stratégie de recherche qui est indiquée en entrée. Dans l'exemple du Tableau 4.2, pour la stratégie *Exact Matching*, le compilateur crée une *Hash Table*. La taille de la clé est calculée à partir de la somme des taille de tous les éléments constituant la clé dans la *lookupStruct*. De même, la taille du résultat est calculée de l'ensemble des éléments formant le résults. Le nombre maximal d'entrées est calculé en fonction de la taille de la clé. Le compilateur doit décider, également, si la structure sera placée dans une mémoire externe ou une mémoire interne. Il doit, de même, assigner un identifiant pour cette structure. Ces paramètres sont générés et regroupés dans la routine *EZapiStruct_ Create*.

- Ajout des entrées dans une structure de recherche. Pour gérer le contenu d'une table de recherche et y ajouter des entrées, le compilateur génère une routine *EZapiStruct_ AddEntry* en spécifiant (1) le numéro de la structure qui est déjà calculé lors de la création de la structure de recherche, (2) la valeur de la clé qui est la concaténation de toutes les valeurs des éléments de la clé, traduit en hexadécimal et (3) la valeur du résultat qui est calculée de la même manière que la clé : on traduit chaque élément en hexadécimale et on le concatène à l'élément suivant aussi écrit en hexadécimal. Le reste des paramètres sont des paramètres générés par défaut. Une routine *EZapiStruct_ AddEntry* est générée pour chaque nouvelle *entry* rencontrée dans le bloc *entries{...}* qu'on trouve dans le bloc *lookupStruct{...}* dans le code source NETLANG.

	Code source NETLANG	Code NPsl généré

Création d'une table de hachage	`lookupStruct forwardingTable(exact)`
	`{`
	`keyStruct {`
	`element mac_addr_t:MAC_DA;`
	`element uint16_t:VID;`
	`}`
	`resultStruct {`
	`element uint16_t:OUT_IF;`
	`}`
	`...}`
		`EZapiStruct_Create (Channel = 0,`
		`Struct_Number = 1,`
		`Partition = 0,`
		`Struct_Type = hash,`
		`Struct_Memory_Area = External,`
		`Key_Size = 8,`
		`Result_Size = 2,`
		`Max_Entries = 512,);`

Ajout d'une entrée dans une table de hachage	`entries {`
	`entry <44:55:66:77:00:02,100>,<2>;`
	`}`
		`EZapiStruct_AddEntry (Channel = 0,`
		`Struct_Number = 1,`
		`Partition = 0,`
		`Key = 0h4455667700020064,`
		`Result = 0h0002, );`

Tableau 4.2 Génération du code *NPsl* spécifique aux structures de recherche

- Support pour la TCAM dans EZchip-NP4. Les *flow-tables* dans NETLANG sont implémentées en utilisant la TCAM avec des structures *hash*. Ce choix de conception repose sur le fait que la TCAM répond intrinsèquement aux caractéristiques des tables OpenFlow. En effet, la TCAM contient un mécanisme de priorisation utile lors de la recherche d'une *flow-entry* puisqu'il assure que c'est la *flow-entry* la plus prioritaire qui sera retournée. De plus, les *wildcards* et le *Masked Matching* fournis dans la TCAM permettent d'ignorer certains champs de correspondance des 14 *tuples* proposés dans OpenFlow 1.1. Par conséquent, les champs de correspondances sont placés comme clé de recherche sur la TCAM. Lors d'une correspondance (un *match*), on récupère l'indice (*rank*) de l'entrée pour aller chercher les instructions liées à cet ensemble de champs de correspondance. Ce *rank* est donc utilisé comme clé de recherche dans une structure *hash* qui contient les instructions correspondant à cette *flow-entry*. Le résultat de cette structure est l'ensemble des instructions suivis de leurs paramètres. Une autre structure *hash* est utilisé comme *group-table* pour stocker les actions par *group-entry*. Bien que la TCAM qu'on a sur la carte du NP4 soit un dispositif propriétaire, EZchip fournit une API pour pouvoir utiliser cette TCAM à travers le langage NPsl. Ceci rend les opérations sur la TCAM plus simples. Afin de pouvoir créer une table dans la TCAM, il faut d'abord créer une instance de TCAM. Lorsque le compilateur analyse le code source NETLANG et qu'une instruction ***new*** *FlowTable()* est trouvée, il utilise la routine *Create_CAM_Device* en utilisant les paramètres par défaut pour l'initialisation d'une TCAM. Ensuite, le compilateur produit le code nécessaire pour créer une table dans la TCAM (1) en utilisant la routine *Create_CAM_Database*, (2) en spécifiant une taille de clé suffisante pour contenir les 14 *tuples* d'OpenFlow et (3) en indiquant le nombre maximal d'entrées dans cette table. Pour ajouter les *flow-entries* dans une *flow-table*, NETLANG propose la primitive *addEntry()* qui prend en paramètres le nom de l'entrée et son index dans la table. La primitive *addEntry()* est transformée par le générateur en une routine *AddCAMEntry*. La valeur de la clé (*key_val*) de cette entrée est la traduction en valeur hexadécimale de l'ensemble des valeurs des champs de correspondance utilisés. La valeur du masque (*mask_val*) est déduite de la manière suivante : pour les positions de la clé où on a des champs de correspondance avec des valeurs spécifiées, on place le masque à 'FFFF...' et pour les positions de la clé où on a des champs de correspondance non spécifiés, on place le masque à '0000...'. Les positions des champs de correspondance respectent le même ordre que celui utilisé dans la spécification d'OpenFlow 1.1 qui va de la couche 2 à la couche 4. Dans le dernier exemple

du Tableau 4.3, nous avons présenté les 4 premiers *tuples* qui sont: le numéro de port d'entrée, l'adresse MAC destination, l'adresse MAC source et enfin, l'*ethertype*. Tous ces champs de correspondance ont été instanciés dans la *flow-entry* de l'exemple, sauf l'adresse MAC source. Comme on respecte l'ordre des champs de correspondance et on utilise les *wildcards*, on place alors la clé et le masque à '0000' pour la position qu'occupe l'adresse MAC source (sa taille est de 6 octets).

	Code source NETLANG	Code NPsl généré
Instanciation de la TCAM	`flowTable table0 = new flowTable(0);`	`Create_CAM_Device(device=0, device_type = NL_11k, device_entries_num = 131072, cascade_depth = 0, Encoding = XY );`
Création d'une table dans la TCAM	`flowTable table1 = new flowTable(1);`	`Create_CAM_Database(device = 0, database = 0, key_size = 320, max_num_entries = 4096, );`
Ajout d'une entrée dans la TCAM	`entry1.matchField.inputPort=1; entry1.matchField.ethDstAddr= 00:1b:77:38:95:c1 ; entry1.matchField.ethType= 0x0800 ;`	`AddCAMEntry (device = 0, Database = 0, Rank = 1, key_val =0h 0001 001b773895c1 000000000000 0800, mask_val=0h FFFF FFFFFFFFFFFF 000000000000 FFFF);`

Tableau 4.3 Génération du code *NPsl* spécifique à la TCAM

4.2.2.3 Le générateur du *microcode*

Le générateur de *microcode* est un composant important du compilateur de NETLANG vu le gain qu'il permet d'offrir dans le cycle de développement. Il est plus facile d'écrire et de débugger du code dans un langage de haut-niveau plutôt qu'en langage assembleur. Cependant,

l'implémentation d'un générateur de code assembleur est une tâche minutieuse qui n'est pas très évidente à réaliser. Néanmoins, nous avons développé un prototype [BLC12] et implémenté un module dans le compilateur qui permet de générer le *microcode* approprié à partir des actions OpenFlow.

Quelques exemples d'opérations OpenFlow écrites dans NETLANG et reproduites en *microcode* sont représentés dans le Tableau 4.4. L'opération *pkt.getEthDstAddress()* est clairement une opération de *parsing* de l'entête du paquet qui vise à récupérer le champ de l'adresse MAC destination. Cette action doit être, alors, nécessairement implémentée au niveau du *TOPparse*. Par contre, une action OpenFlow de type *pushMPLSLabel(pkt, label)* est une instruction d'écriture sur le paquet. Elle est donc réalisée au niveau du *TOPmodify*.

Description	**Code source NETLANG**	***Microcode* généré**

Extraction du numéro de port d'entrée du paquet	`pkt.getInputPort();`	`PutKey 0(KMEM_BASE1)+,`
		`UREG[1].Byte[1],`
		`1;`

Extraction de l'adresse MAC destination	`pkt.getEthDstAddress();`	`Copy 0(KMEM_BASE1)+,`
		`0(FMEM_BASE )+ ,`
		`6;`

Extraction de l'adresse MAC source	`pkt.getEthSrcAddress();`	`Copy 0(KMEM_BASE1)+,` `0(FMEM_BASE )+, 6;`
Extraction de l'EtherType	`pkt.getEtherType();`	`Copy 0(KMEM_BASE1)+,` `0(FMEM_BASE), 2;`

Tableau 4.4 Génération de *microcode* spécifique aux TOPs

L'équivalent de l'instruction *pkt.getEthDstAddress()* dans *EZlanguage* est l'instruction *copy* qui prend comme source la *Frame Memory* (la mémoire pour stocker le paquet) et comme destination la *Key Memory* (la mémoire pour stocker la clé de recherche). En effet, l'instruction *copy* est utilisée dans le *TOPparse* pour copier des données et les passer au *TOPsearch* à travers la *Key Memory*. L'instruction *copy* prend aussi comme paramètre la taille de la donnée à copier.

L'opération *pkt.getEthDstAddress()* permet d'extraire une adresse MAC, ce qui explique la valeur du troisième paramètre qui est de 6 octets.

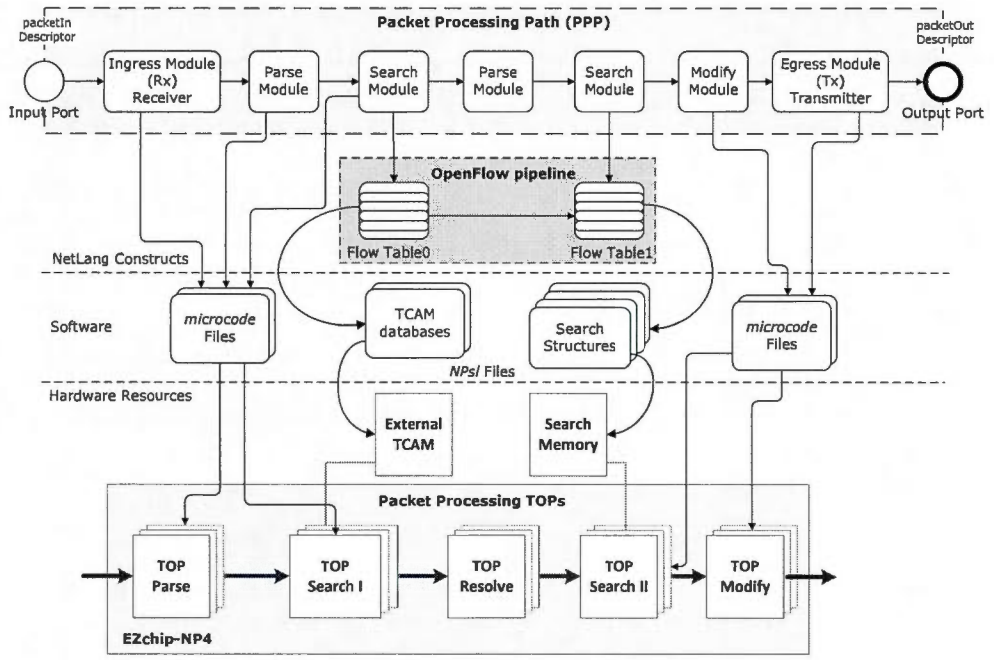

Figure 4.2 *Mapping* de NETLANG avec les ressources matérielles de la plateforme en *back-end*

Chaque bloc de *processing* défini dans un *Packet Processing Path* de l'application est traduit en un programme destiné pour un TOP particulier. Les opérations réalisées dans le bloc *parse* sont en effet générées dans un fichier *TOPparse.asm*. Les instructions écrites dans le bloc *search* sont destinées à aller dans le code *TOPsearch.asm*. De même, le code du bloc *modify* et *transmitter* est placé dans un fichier *TOPmodify.asm*. Le générateur du *microcode* pour le NP4 d'EZchip fait une sorte de démultiplexage des instructions de NETLANG pour les placer sur des petits programmes *EZlanguage* séparés.

Ainsi, nous avons expliqué comment les différentes structures du langage NETLANG se mettaient en correspondance avec les ressources matérielles de la plateforme *NP4-eval System* (voir Figure 4.2). Les ports, les *flow-tables* et les structures de recherche génériques sont générées dans des fichiers NPsl qui permettent de configurer les interfaces du routeur et d'y ajouter des tables dans la TCAM et les autres modules de mémoire. Un *Packet Processing Paths* est découpé sur plusieurs fichiers de code assembleur, où chaque fichier est destiné à un TOP particulier.

4.3 Adaptateur de NETLANG pour Netronome

Le module *Netronome-NFP Back-end* représente le module adaptateur de NETLANG pour Netronome (voir Figure 3.3). La programmabilité du NFP se fait à travers du code C et du code assembleur qui sont plus conventionnels que le *microcode* de EZchip. L'adaptateur de Netronome contient un *Entries Generator* dont le rôle est de traduire les *flow entries* des tables OpenFlow en des entrées de bas-niveau qui vont aller dans la TCAM.

4.3.1 Description de l'environnement du NFP3240

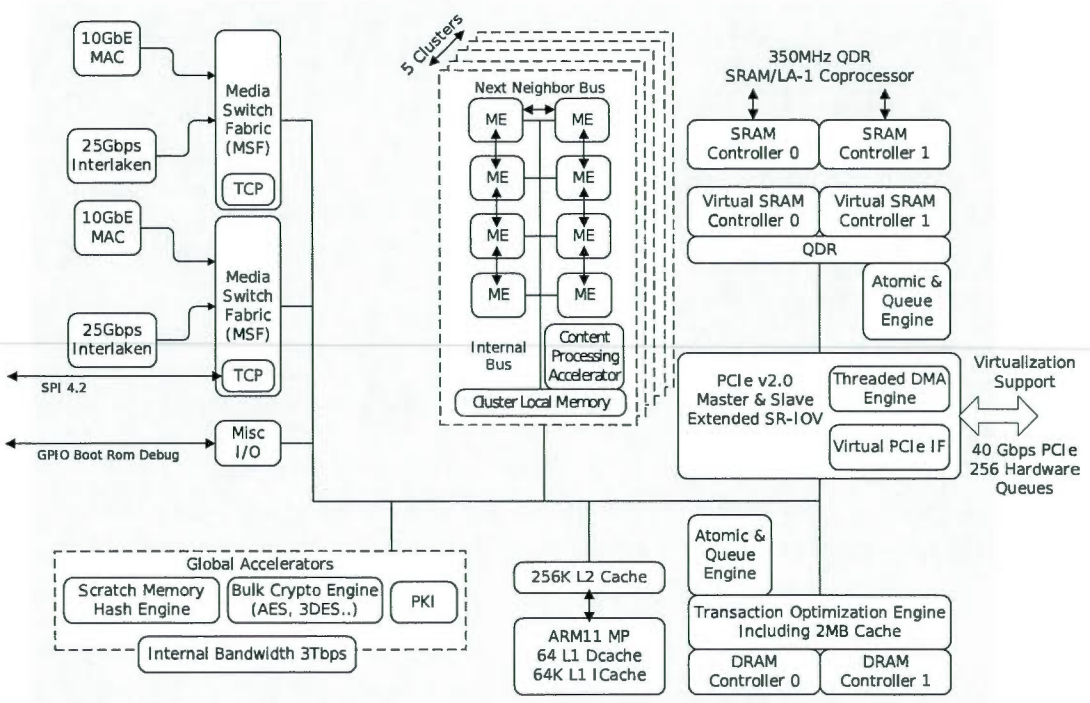

Figure 4.3 La plateforme du NFP-3240 [Com10]

- *AMDA-0021-0003 Plateform.* Il s'agit d'une carte électronique de test pour le NFP-3240 de Netronome. En plus du NFP, la carte contient deux interfaces MSF0 et MSF1 pour la réception et la transmission des paquets, un processeur ARM *Linux* qui permet de configurer et contrôler le NFP à travers un bus PCIe, ainsi que des co-processeurs d'accélération matérielle. La carte est accessible à travers un serveur x86 qui est le *Host CPU* (voir Figure 4.3). Les principales caractéristiques du NFP-3240 sont les *micro-engines* et le

multithreading. Le NFP-3240 est doté de 40 *micro-engines* programmables répartis sur 5 *clusters*, 8 *micro-engines* par *cluster* et 8 *threads* par *micro-engine*, une mémoire de commande commune à tous les *micro-engines* de taille 16k instructions, 256 registres généraux, 512 registres de transfert, 128 *Next Neighbor Registers*, un support pour les opération ALU à un seul cycle horloge, une mémoire locale à 1k mots, un multiplicateur avec 32*8 bits/cycle, un générateur de nombres aléatoires de 16-bit, une communication flexible entre les MEs. Le *micro-engine* est l'unité de traitement principale dans le processeur NFP de Netronome. Tous les MEs ont accès à toutes les ressources partagées (SRAM, DRAM, etc.). Ils ont également accès au bus d'événements internes, ce qui leur permet de communiquet et d'être signalés par les autres périphériques tels que les *timers*, le GPIO (*General Purpose Input/Output*), l'UART (*Universal Asynchronous Receiver/Transmitter*), etc. Le support du *multithreading* est une propriété importante dans les *micro-engines* du NFP. Chaque *thread* s'exécute dans un contexte. Il est possible de changer de contexte quand un *thread* est bloqué, lors d'une opération de lecture sur une mémoire par exemple. Les mécanismes utilisés pour la gestion des contextes sont les signaux, les messages et les sémaphores. Il est possible de configurer les *micro-engines* pour qu'ils s'exécutent selon le mode 8 *threads* ou 4 *threads*. Les MEs d'un même *cluster* bénéficient d'une mémoire de type *Cluster Local Scratch* (CLS) sur une RAM protégée par un ECC (*Error Correcting Code*), avec un support aux opérations atomiques complexes, 16 mémoires *ring* au sein de chaque *Cluster Local Scratch*, une signalisation d'événements, un générateur de nombres aléatoires, des tables de hachage programmables et un échange de données flexible grâce aux *rings* et aux registres de transfert. Les MEs de *clusters* différents communiquent à travers une mémoire *Global Scratch* permettant de déployer jusqu'à 16 *rings*. Le système supporte la signalisation et le transfert de données entre des MEs de *clusters* différents. Les *rings* sont aussi une propriété importante du NFP. Ils présentent à la fois un espace mémoire et une logique d'échange de données: la stratégie appliquée aux *rings* est la *Round Robin*. Configurer un *ring* consiste à lui assigner une taille et un type (*Local Scratch* ou *Global Scratch*). A chaque *ring* est associé un nombre de *threads* producteurs et de *threads* consommateurs.

- *Le langage assembleur de Netronome*. Les *micro-engines* sont des processeurs RISC à 32 bits. Contrairement aux TOPs d'EZchip, les *micro-engines* sont tous similaires, ils ont la même architecture et supportent le même jeu d'instruction. Chaque *micro-engine* possède 4 types de registres: les *Transfer Registers*, les *General Purpose Registers*, les *Next Neigh-*

bor Registers et la *Local Memory*. Le *microcode* utilisé pour programmer les *micro-engines* permet de déclarer des registres, faire des branchements ainsi que des opérations d'I/O sur différentes types de mémoires comme la SRAM, la DRAM ou la mémoire *Sratch*. Le langage assembleur se limite en effet à des instructions de type *alu*, *jump*, *immed*, *nop*... Le langage assembleur utilisé par Netronome est de plus bas niveau que le *microcode* proposé par EZchip. Le code peut être plus volumineux dans le NFP mais il offre plus de flexibilité pour contrôler les *threads* par exemple et pour avoir de la performance. Le modèle de programmation est très différent de celui d'EZchip. En effet, *EZlanguage* est déjà optimisé d'une manière qui ne donne pas de marge de manoeuvre pour extraire plus de performance. Le parallélisme est complétement masqué dans le NP4 mais sa programmabilité est nettement plus facile que le NFP. Un code assembleur dans le NFP est sauvegardé dans un fichier .uc et ce même fichier peut être poussé sur plusieurs *micro-engines* grâce à l'éditeur de liens qui permet d'associer les fichiers *microcode* de l'application aux différents *micro-engines* qu'ils soient sur le même *cluster* ou sur des *clusters* différents.

- *Le langage C de Netronome.* Il est aussi possible de programmer les *micro-engines* en un langage C un peu modifié. Le langage C qui est mis en oeuvre dans le NFP est un langage C parallèle qui expose explicitement le *threading* et la synchronisation au niveau du langage. Le compilateur NFCC (*Network Flow C Compiler*) utilisé pour le NFP n'est pas une implémentation complète de la norme ANSI (*American National Standards Institute*) du langage C. Il y a de nombreuses fonctionnalités qui ne sont pas supportées, comme, par exemple, les opérations à virgule flottante, qui sont en dehors du domaine d'applications du NFP. Le compilateur omet donc ces caractéristiques. Le compilateur ne supporte pas la totalité de la bibliothèque standard du langage C. Il implémente uniquement les fonctions utiles ou nécessaires conformément à la spécification de la librairie C, mais il ne l'implémente pas complètement. Le compilateur n'implémente pas la parallélisation automatique de code. Il s'attend à ce que le *multithreading* soit explicitement mis en oeuvre dans le code en entrée. Le compilateur ne supporte pas C++. Les types de données *float* et *double* ne sont pas, non plus, supportés.

4.3.2 Le générateur pour NFP-3240

L'adaptateur pour Netronome se limite à un générateur pour les *forwarding entries*. En effet, la carte ne donne pas beaucoup de possibilités de configuration puisqu'elle ne dispose que

de deux interfaces. Les structures *port* de NETLANG pour le *management plane* ne sont pas considérées. Par contre, le niveau *control plane* avec la création et la population des *flow tables* est géré par le module *Entries Generator*. Le modèle du *Packet Processing Path* reste valable pour cette plateforme mais son implémentation requiert le développement d'un compilateur au complet pour des processeurs de type RISC. Le niveau *data plane* n'a donc pas été traité.

Le support de NETLANG pour les *flow tables* dans l'environnement de Netronome est une solution de programmabilité suffisante. Comme OpenFlow fournit des portes d'ouverture à l'application réseau à travers la mise à jour des *flow entries*, ceci nous permet de changer le comportement d'une application qui s'exécute sur le NFP, à la volée, en agissant sur les *flow entries* des tables OpenFlow. Bien que le *microcode* reste inchangé, la recompilation de NETLANG et la génération de nouvelles *flow entries* permettent de changer les règles de *forwarding* et donc on change indirectement le traitement sur les paquets dans le NFP.

L'adaptateur du compilateur de NETLANG pour Netronome produit des entrées de bas-niveau pour la TCAM (NetLogic 7000). L'API pour la TCAM fournie par Netronome est une bibliothèque C appelée (*BunkerHill TCAM Library*). L'utilisation de la TCAM se fait à travers une séquence d'opérations un peu similaire à ce qui est fait dans le NPsl et les routines d'EZchip. Tout d'abord, il faut initialiser la TCAM en utilisant la fonction *tcam_ tcam_ init(unsigned int width)*. Cette fonction prend comme paramètre d'entrée une valeur qui représente la largeur de la clé en bits (le masque a la même taille que la clé). Les 4 valeurs possibles qui sont acceptées lors de la configuration de la TCAM sont 72 bits, 144 bits, 288 bits et 576 bits. Nous avons fixé la configuration de la TCAM à la valeur 288 bits afin qu'elle puisse contenir les 14 *tuples* d'OpenFlow v1.1. Après l'instanciation de la TCAM, il faut l'associer à un contexte en utilisant la fonction *tcam_ single_ buffer_ init(unsigned int ctx)*. Dans EZchip, cette opération est similaire à la création d'une *database* dans la TCAM. La population d'une table de la TCAM se fait avec la fonction *tcam_ write_ record_ 288 (unsigned int ctx, unsigned int add, unsigned int *key, unsigned int *mask)* qui prend en paramètres l'identifiant de la table, l'adresse de l'entrée dans la table, la valeur de la clé et la valeur du masque. Cette fonction permet d'ajouter une entrée à une adresse donnée dans une table spécifique de la TCAM. Elle est équivalente à la routine *AddCAMEntry* dans EZchip. La suppression d'une entrée de la TCAM est réalisée par la fonction *tcam_ delete_ record (unsigned int ctx, unsigned int add)* qui prend en paramètres l'identifiant du contexte et l'adresse de l'entrée dans la TCAM.

La traduction des primitives de NETLANG traitant les *flow-tables* d'OpenFlow dans le

	Code source NETLANG	Code C généré

Instanciation de la TCAM	`flowTable table0 = new flowTable(0);`	`unsigned int ctx;`
		`unsigned int key[36];`
		`unsigned int mask[36];`
		`ctx=0;`
		`tcam_tcam_init(288);`
		`tcam_single_buffer_init(ctx);`

Ajout d'une entrée dans la TCAM	`flowEntry entry1 = new flowEntry();`
	`entry1.matchField.IpDstAddr = 128.132.130.23;`
	`table0.addEntry(entry1,1);`
		`key[0]=0x00000000;`
		`key[1]=0x80848217;`
		`key[2]=0x00000000;`
		`mask[0]=0x00000000;`
		`mask[1]=0xFFFFFFFF;`
		`mask[2]=0x00000000;`
		`tcam_write_record_288 (ctx,1,key,mask);`

Suppression d'une entrée de la TCAM	`table0.deleteEntry(entry1);`	`tcam_delete_record(ctx,1);`

Tableau 4.5 Génération du code C spécifique à la TCAM dans le NFP-3240

langage adapté à Netronome est représentée dans le Tableau 4.5. Le premier exemple montre que, lors de la déclaration d'une nouvelle *flow-table* dans NETLANG, le compilateur génère le bout de code permettant d'initialiser la TCAM et d'y créer un nouveau contexte. Le compilateur génère aussi les variables *key* et *mask* avec une taille de 36 octets (36*8 = 288 bits) en préparation à l'ajout des entrées dans la TCAM. Le deuxième exemple explique comment le compilateur gère la définition d'une nouvelle *flow-entry* dans le code source NETLANG. Le compilateur place la traduction hexadécimale des champs de correspondance dans les bonnes position de la clé et produit le masque adéquat en utilisant des 'FFFF' dans les positions où il y a des champs de correspondance spécifiés et des '0000' quand un champ de correspondance n'est pas utilisé. La clé et le masque sont générés sous forme de 9 lignes de 4 octets, les lignes sont indexées de 0 jusqu'à 8. Pour simplifier l'exemple, nous avons placé l'adresse IP dans la deuxième ligne et nous n'avons donné que les 3 premières lignes de la clé et du masque. Le troisième exemple donne le code C généré pour le suppression d'une *flow-entry* de la table OpenFlow. Cette opération est supportée dans NETLANG avec la méthode *deleteEntry()* et elle est traduite par le compilateur en une fonction *tcam_delete_record()*.

4.4 Résumé

L'adaptabilité du langage est assurée par la partie *back-end* du compilateur. L'adaptateur de Netronome produit du code C alors que l'adaptateur pour EZchip génère du code NPsl et des macros dans *EZlanguage*. Un premier résultat que nous pouvons mentionner c'est que l'implémentation de l'adaptateur de NETLANG pour EZchip s'est bien passée. La compilation du langage était parfaitement adapté à l'environnement du NP4 du fait que leurs langages, avec le modèle des TOPs, sont à des niveaux d'abstraction qui sont assez élevés. Par contre, nous avons rencontré des difficultés lors de l'implémentation de l'adaptateur pour Netronome. L'adaptabilité du langage était un peu plus compliquée pour le NFP à cause que nous n'avons pas pu accéder à temps dans le projet aux bibliothèques qu'ils offrent.

Le deuxième résultat que nous retenons c'est que malgrè que les architectures considérées dans l'implémentation du compilateur de NETLANG soient très différentes, l'une étant basée sur des processeurs en pipeline et l'autre présentant un environnement de *multithreading* avec des processeurs parallèles, nous avons démontré que la sémantique d'OpenFlow permettait une flexibilité de programmation dans les deux plateformes. Les *flow-entries* offrent des points d'ouverture à travers lesquels on peut changer la stratégie de *forwarding*. La programmabilité à

travers les tables d'OpenFlow permet, non seulement, de décrire le traitement des paquets dans le NP, mais elle présente aussi un support pour modifier le comportement du routeur à la volée.

CHAPITRE V

EVALUATION DE NETLANG

5.1 Introduction

Dans ce chapitre, nous allons procéder à l'évaluation de NETLANG en termes d'expressivité. En effet, en plus de l'adaptabilité du langage par rapport aux architectures cibles, l'expressivité est aussi une propriété importante qu'il faut évaluer. Nous avons conçu NETLANG afin qu'il permette d'exprimer diverses applications de traitement de paquets en respectant la vision des *Software Defined Networks* et en utilisant une abstraction de comportement de paquets basée sur OpenFlow. Afin d'évaluer l'expressivité de NETLANG, nous avons étudié des exemples d'applications réseau mettant en ocuvre des logiques et des mécanismes différents, avec des niveaux de complexité variés. Nous présenterons 3 scénarios de tests pour NETLANG où nous donnerons le code correspondant à chaque scénario. Chaque scénario nous permettra de valider une fonctionnalité donnée de NETLANG. L'ensemble de nos résultats seront présentés dans la deuxième section du chapitre.

5.2 Scénarii d'évaluation de NETLANG

L'évaluation de l'expressivité de NETLANG est réalisée à travers trois exemples d'applications (1) une application de *forwarding* basée sur une simple table OpenFlow, (2) une application de *routing* et de *bridging* dans un réseau Metro Ethernet et enfin (3) une application SPBM (*Shortest Path Bridging - MAC mode*) dans un réseau PBBN (*Provider Backbone Bridge Network*). Chaque application a des besoins spécifiques en termes de nombre de ports, de nombre tables et de règles indiquant le traitement des paquets.

5.2.1 Application OpenFlow

Dans le premier scénario de test, nous allons bâtir une application de *forwarding* dans un réseau OpenFlow. L'application est définie dans une *flow table*. Cet exemple d'application nous permettra de valider l'expressivité de NETLANG dans la définition des tables OpenFlow et l'ajout des *flow-entries*.

Considérons un réseau OpenFlow très simple, composé de deux routeurs OpenFlow et d'un contrôleur, NOX par exemple (voir Figure 5.1). Chacun des deux routeurs est constitué de trois interfaces (1) une interface vers le contrôleur, (2) une interface vers l'autre routeur et (3) une interface avec le réseau. La *switch* S1 est connectée au réseau R1 et la *switch* S2 est liée au réseau R2. Considérons la *switch* S1, si on veut la programmer pour la rendre fonctionnelle, l'application NETLANG nécessaire sera celle définie dans le Code 7. En effet, pour rendre le routeur opérationnel, il faut établir la connectivité dans le réseau OpenFlow et rendre l'acheminement des paquets entre le réseau R1 et le réseau R2 possible dans les deux sens, c'est à dire, de R1 vers R2 et de R2 vers R1. L'implémentation de la connectivité se fait par l'injection de la politique de *forwarding* adéquate pour établir les liens entre les routeurs S1 et S2. Il faut aussi activer chacun des routeurs avec la bonne configuration en termes d'activation d'interfaces réseau.

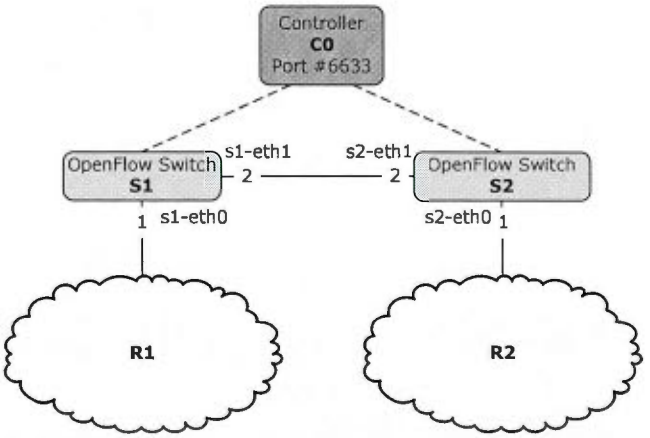

Figure 5.1 Exemple de réseau OpenFlow

Comme présenté dans le code 7, une instance de la *switch* S1 est créée en activant les ports *port_1* et *port_2* correspondant respectivement aux interfaces *s1-eth0* et *s1-eth1*. Ensuite, pour mettre en place la connectivité, nous définissons une *flow-table* (la table0) et nous y ajoutons les

Code 7 Code de l'application OpenFlow

```
/* Creating an OpenFlow Switch instance */
   port port_1 {
       number = 1;
       speed = 10000;
       rx = true;
       tx = true;
   }
   port port_2 {
       number = 2;
       speed = 10000;
       rx = true;
       tx = true;
   }

/* Creating a flow table */
   flowTable table0 = new flowTable(0);

/* Defining the forwarding rules */
   flowEntry entry1 = new flowEntry() ;
   entry1.matchField.inputPort = 1 ;
   entry1.instr(write_action,output_port,2);

   flowEntry entry2 = new flowEntry() ;
   entry2.matchField.inputPort = 2 ;
   entry2.instr(write_action,output_port,1);

   /* Adding flow entries into the flow table */
   table0.addEntry(entry1,1);
   table0.addEntry(entry2,2);
```

flow-entries requises. Dans notre cas, pour permettre le transfert des paquets, nous avons besoin de deux règles de *forwarding* : (1) La première *flow-entry* gère les paquets arrivant sur le *port_2* (en provenance du réseau R2) et souhaitant aller vers le réseau R1. (2) La deuxième *flow-entry* traite les paquets arrivant sur le *port_1* (en provenance du réseau R1) et se dirigeant vers le réseau R2. Ces deux *flow-entries* sont suffisantes pour permettre à la *switch* S1 de gérer son trafic. Elles peuvent avoir la même priorité puisqu'elles ne risquent pas d'entraîner un conflit. En effet, ces deux règles gèrent deux ensembles de paquets disjoints. On peut aussi ajouter une troisième règle qui indique à la *switch* de supprimer tout autre type de trafic. Cette règle, par contre, si elle n'est pas ajoutée au bon emplacement dans la *flow-table*, risque de donner un comportement indésirable. En effet, si on la place en premier, elle aura la plus grande priorité; tous les paquets vont être supprimés et aucun paquet ne sera expédié au réseau R1 ou R2. Il faut donc la placer en dernier.

5.2.2 Application Metro Ethernet

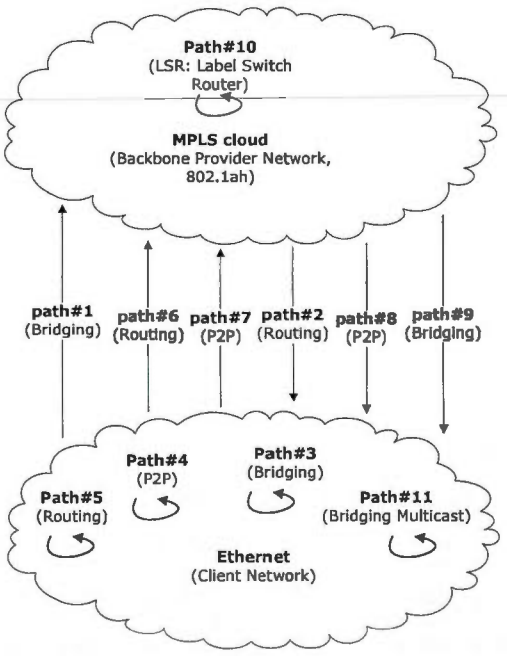

Figure 5.2 Exemple de réseau *Metro Ethernet*

Une application *Metro Ethernet* [HH03] est une application qui s'interface avec deux types de réseaux différents. Un côté de l'application est orienté vers un réseau Ethernet [Hel03]

constituant le réseau client et l'autre côté est orienté vers un nuage MPLS [AA99] qui présente le réseau du fournisseur.

Code 8 Code de l'application *Metro Ethernet* spécifique au *data plane*

```
packet pkt = incomingPacket();
port p ;
p.number = pkt.getInputPort();

if (p.number == 1) { goto path1; } /* Upstream Bridging */
if (p.number == 2) { goto path2; }  /* Downstream Routing */
if (p.number == 3) { goto path3; } /* Ethernet Forwarding-Bridging */
if (p.number == 4) { goto path4; } /* Ethernet Forwarding-P2P*/
if (p.number == 5) { goto path5; } /* Ethernet Forwarding-Routing */
if (p.number == 6) { goto path6; } /* Upstream Routing */
if (p.number == 7) { goto path7; } /* Upstream P2P */
if (p.number == 8) { goto path8; } /* Downstream P2P */
if (p.number == 9) { goto path9; } /* Downstream Bridging */
if (p.number == 10) { goto path10; } /* MPLS LSR Forwarding P2P */
if (p.number == 11) { goto path11; } /* Ethernet Forwarding-Multicast Bridging */
```

Pour traiter les différents types de trafic dans un tel réseau, plusieurs types de traitements doivent être mis en place. Dans notre exemple, l'application *Metro Ethernet* est constituée de 11 *Packets Processing Paths* (voir Figure 5.2). Le *path1* (*Upstream Bridging*) traite des paquets qui arrivent du réseau client et qui sont envoyés vers le nuage MPLS avec un service de *bridging*. Le *path2* (*Downstream Routing*) traite les paquets qui arrivent du nuage MPLS et qui sont envoyés vers le réseau client avec un service de *routing*. Le *path3* (*Ethernet Forwarding and Bridging*) traite les paquets à l'intérieur du réseau Ethernet du client avec un service de *bridging*. Le *path4* (*Ethernet Forwarding and Point-to-Point*) gère les paquets au sein du nuage Ethernet avec un service *Point-to-Point* [Bak03]. Le *path5* (*Ethernet Forwarding and Routing*) traite les paquets au sein du nuage Ethernet en leur appliquant un service de *routing*. Le *path6* (*Upstream Routing*) gère les paquets arrivant du réseau Ethernet et se redirigeant vers le nuage MPLS avec un service de *routing*. Le *path7* (*Upstream Point-to-Point*) traite les paquets qui arrivent du réseau client et qui sont envoyés vers le nuage MPLS avec un service *Point-to-Point*. Le *path8* (*Downstream Point-to-Point*) gère les paquets qui arrivent du réseau MPLS et qui sont envoyés au réseau Ethernet avec un service *Point-to-Point*. Le *path9* (*Downstream Bridging*) gère les paquets

arrivant du nuage MPLS et qui sont envoyés vers le réseau client avec un service de *bridging*. Le *path10* (*MPLS LSR forwarding Point-to-Point*) traite les paquets à l'intérieur du nuage MPLS avec un service *Point-to-Point*. Le *path11* (*Ethernet Forwarding, Bridging Multicast*) traite les paquets à l'intérieur du nuage Ethernet avec un service de *bridging multicast*.

La répartition du traitement sur l'ensemble des *Packet Processing Paths* de cette application est exprimé en NETLANG dans le Code 8. Cette partie du code traite le niveau *data plane*. Les paquets entrant sont dirigés vers le *Packet Processing Path* approprié en fonction du numéro de port par lequel ils sont entrés grâce au mot réservé *goto*. En effet, lors du profilage du routeur, on sait, au préalable, quel type de trafic va arriver sur chaque interface. Chaque type de trafic reçoit un traitement spécifique; par exemple, dans le cas du *bridging*, des tags supplémentaires sont ajoutés à l'entête du paquet.

Code 9 Code de l'application *Metro Ethernet* spécifique au *path3*

```
processingPath path3 {
    packetIn {
        header h1 {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:tpid:0x8100 ;
            field uint16_t:ctag ;
            field hex:etherType:0x0800 ;
        }
    }
    packetOut {
        header h2 {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:tpid:0x88A8 ;
            field uint16_t:stag ;
            field hex:etherType:0x0800 ;
        }
    }
```

Ce qui est intéressant dans cette application, c'est qu'elle manipule un ensemble varié

de paquets avec des formats différents. Elle nous a ainsi permis de tester la fonctionnalité de *header* de NETLANG dans les structures de *packetIn* et *packetOut* (voir Annexe A). Prenons, par exemple, le *path3* qui permet de faire du *bridging* à l'intérieur du réseau Ethernet. Ce *path* transforme des paquets 802.1q en des paquets 802.1ad qu'on appelle aussi des paquets *QinQ* [SRV08]. Le Code 9 montre le format des paquets à l'entrée (*PacketIn*) et à la sortie (*PacketOut*) de ce *path*. La structure du *header PacketIn* est composée de 5 champs : une adresse MAC source, une adresse MAC destination, un *EtherType* ayant la valeur 0x8100 puisqu'il s'agit de paquets 802.1q [Com08], un champ *Customer Tag* et enfin un dernier champ *Network Layer Protocol Identifier* de valeur 0x0800 indiquant que ce qui suit est un paquet IP. Dans la structure du *header paquetOut*, les champs *EtherType* et *Customer Tag* sont supprimés et remplacés par un *EtherType* de valeur 0x88a8 (pour indiquer qu'il s'agit d'un paquet *QinQ*) suivi d'un *Service Tag*.

5.2.3 Application SPBM

Le dernier scénario d'évaluation de NETLANG repose sur une application SPBM [ASF12] permettant de gérer des paquets entre un réseau PBN (Provider Bridge Network) et un réseau PBBN (*Provider Backbone Bridge Network*) (voir Figure 5.3). Cette exemple nous a permis de valider les fonctionalités à l'intérieur du *Packet Processing Path*, les primitives OpenFlow, la définition du traitement sur chaque type de bloc de traitement (*parse*, *search*...), et les structures de recherche génériques qui ne sont pas des tables dans le format d'OpenFlow.

Un nuage PBBN est composé de routeurs BEB (*Backbone Edge Bridge*) à ces extrémités et de routeurs BCB (*Backbone Core Bridge*) en interne. A l'entrée et à la sortie du nuage, les routeurs BEB assurent des fonctions d'encapsulation (à l'entrée) et de décapsulation (à la sortie) *MAC-in-MAC* selon la norme IEEE 802.1ah [BBZ+12] pour ajouter ou supprimer l'entête du réseau *backbone*. Les paquets qu'ils traitent arrive selon le format IEEE 802.1ad. Les routeurs BEB offrent aussi un service de *tunneling* en se basant sur l'identifiant de service I-SID (*Instance Service Identifier*). Des fonctions similaires au multiplexage et démultiplexage sont réalisées par les BEBs pour regrouper plusieurs S-VID (*Service VLAN Identifier*) différents dans un I-SID donné à l'entrée du nuage et faire l'opération inverse à la sortie, pour démultiplexer un tunnel I-SID en plusieurs flux S-VID.

Un routeur BEB est constitué de deux composants *I-Component* et *B-Component* IB-

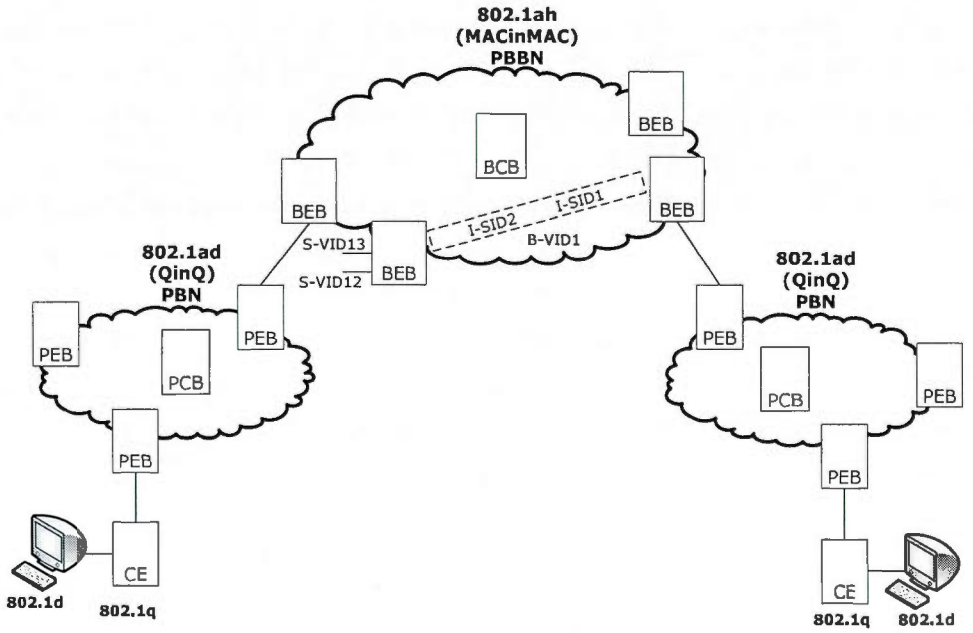

Figure 5.3 Un exemple de réseau PBBN

BEB. Il possède 4 types de port : un port CNP (*Customer Network Port*), un port PIP (*Provider Instance Port*), un port CBP (*Customer Backbone Port*) et un port PNP (*Provider Network Port*) (voir Figure 5.4). Nous allons étudier le comportement d'un routeur BEB en mode MAC. Principalement, le routeur BEB procède à l'ajout d'un tag I-SID ce qui est appelé du *I-tagging*. Ensuite, il ajoute un tag B-VID (*Backbone VLAN Identifier*) et c'est le *B-tagging*. Les transformations introduites par ces opérations de *tagging* sont exprimées dans le code donné en Annexe B. Pour comprendre ce code nous allons expliquer certains points du *data plane* et du *control plane* qui caractérisent une application SPBM.

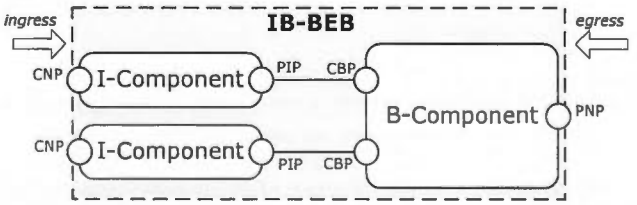

Figure 5.4 Composants d'un routeur BEB

5.2.3.1 Le *data plane* dans SPBM

Au niveau d'un routeur BEB, il peut y avoir deux types de traitement dans le *data plane* selon les deux directions possibles de paquets : la direction *ingress* pour les paquets en provenance d'un réseau PBN et la direction *egress* pour les paquet sortant d'un réseau PBBN.

- *Data path* d'un routeur IB-BEB dans la direction *ingress*. A son arrivée dans un routeur BEB dans la direction *ingress*, le paquet entre par le port CNP et passe d'abord par le module *I-Component*. Les paquets SPBM entrant sont des paquets encapsulés selon le format IEEE 802.1ah. Au niveau du port CNP, le paquet subit une classification basée sur son S-VID ce qui permet d'identifier son nouvel I-SID. Le CNP mantient en effet une table de correspondance entre S-VID et I-SID. Ensuite, le *I-Component* fait du *MAC learning* pour sauvegarder l'adresse C-SA du paquet. Le *I-Component* récupère aussi l'adresse C-DA et lance un *lookup* pour trouver l'adresse B-DA qui lui correspond. L'identification de l'adresse B-DA est effectuée grâce à une table de *mapping* entre C-DA et B-DA. A sa sortie du *I-Component*, le paquet se trouve au niveau du port PIP où l'encapsulation *MAC-in-MAC* est appliquée par l'ajout des champs B-DA et B-SA à l'entête du paquet. Un *I-tagging* est aussi réalisé par l'ajout d'un I-Tag (I-SID et EtherType). Après ces modifications, le paquet atterrit sur le port CBP où un *B-tagging* est appliqué. Après l'insertion du B-Tag, le paquet arrive au *B-Component* du routeur BEB où un *learning* sur la B-SA est réalisé ensuite un *lookup* sur la B-DA est lancé pour l'identification du port PNP. Le port PNP envoie le paquet sur le tunnel B-VLAN.

- *Data path* d'un routeur IB-BEB dans la direction *egress*. Dans ce *Packet Processing Path*, le paquet suit le chemin inverse de la direction *ingress*. Il arrive sur le port PNP où une classification basée sur le B-VID est réalisée. Ensuite, le *B-Component* fait un *learning* de l'adresse B-SA et lance un *lookup* sur la B-DA pour identifier le port CBP. Arrivé au port CBP, le B-tag est retiré et le paquet est envoyé suivant la valeur de son B-VID. Le paquet arrive au port PIP où son I-Tag est supprimé. Dans le *I-Component*, un *learning* de l'association C-SA et B-SA est lancé. Puis, un *lookup* sur l'adresse C-DA permet d'identifier le CNP. Arrivé au port CNP, le paquet est transmis sur le S-VLAN destination.

5.2.3.2 Le *control plane* dans SPBM

Le *forwarding* utilise un ensemble de tables qui donne le support à trois fonctionnalités : la transmission du paquet vers un port de sortie, le MAC *Learning* et la *Loop Avoidance*.

B-DA	B-VID	OUT-IF(s)
44:55:66:77:00:02	0100	if/2
44:55:66:77:00:03	0100	if/2
44:55:66:77:00:04	0100	if/1
44:55:66:77:00:05	0100	if/2
44:55:66:77:00:06	0100	if/3
44:55:66:77:00:07	0100	if/2

Tableau 5.1 Les entrées de *forwarding* dans la FDB du *node1*

- *Filtering DataBase* (FDB). Les entrées de la FDB sont calculées et installées en fonctions des adresses MAC. La FDB est une table qui fait correspondre à l'ensemble des valeurs {B-DA, B-VID} une valeur *next hop*. Elle permet aux *B-Components*, qu'on trouve dans les routeurs BEBs et dans les routeurs BCBs, de transmettre les paquets SPBM à l'intérieur du réseau PBBN. Ces routeurs analysent l'adresse MAC destination et le VLAN *identifier* au niveau *backbone* pour connaître le numéro de l'interface de sortie sur laquelle le paquet va être expédié.

- *Learning Tables*. Le *learning* est effectué à l'extrémité du nuage SPB au niveau des routeurs BEBs dans la direction *ingress*. Les tables de *learning* sont des tables qui se basent sur l'adresse source et non pas l'adresse destination et qui permettent au routeur d'apprendre, pour chaque adresse MAC source, sur quel numéro de port elle est arrivée.

- *Loop Mitigation Table*. Cette table est utilisée pour tester les paquets à leur arrivée. Chaque routeur maintien une table de *Loop Mitigation* ou de *Loop Avoidance* qui lui permet de faire une sorte d'*ingress check*. En effet, cette table fait correspondre à l'ensemble des valeurs {B-SA, B-VID}, une valeur *ingress port*. Si le couple de valeurs *Backbone-Source MAC Address* et *Backbone VLAN Identifier* ne correspond pas au numéro du port d'entrée inscrit dans la table, alors le paquet sera rejeté. Il s'agit d'un mécanisme très puissant de détection de boucle dans le réseau.

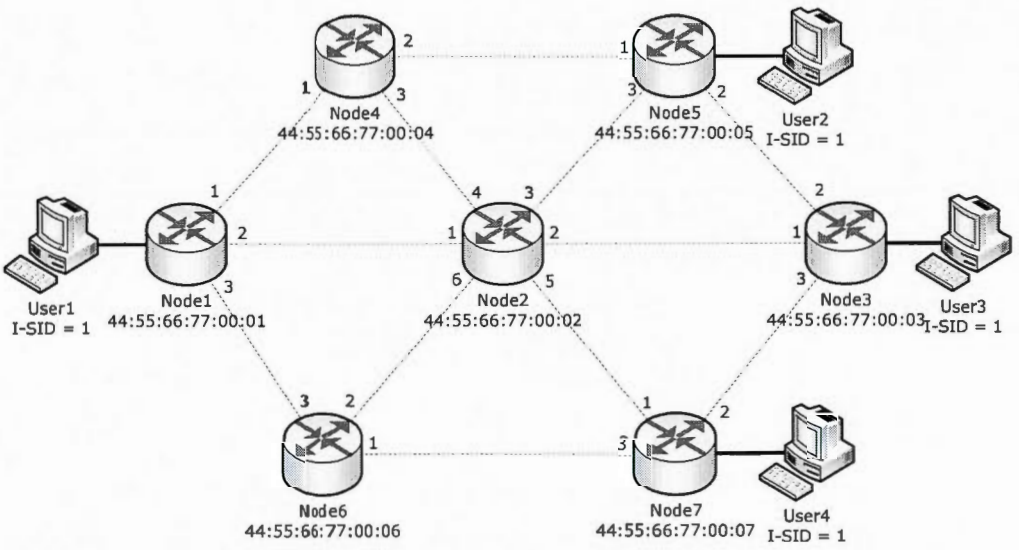

Figure 5.5 Exemple de réseau SPBM

La figure 5.5 représente le réseau considéré pour notre exemple d'application SPBM. L'annexe B donne le code spécifique au routeur 1 (*Node1* dans le schéma). On définit pour ce noeud la table FDB décrite dans le Tableau 5.1. Les mêmes valeurs du tableau sont utilisées dans le code de l'application.

5.3 Validation de NETLANG

NETLANG permet de bâtir une application réseau entièrement définie de manière programmable en respectant le concept des *Software Defined Networks* et en utilisant, dans cette construction, les 3 niveaux de *management plane*, *control plane* et *data plane*. La validation de NETLANG est basée sur les six points suivants :

- NETLANG *est un langage pour les NPs.*

  L'objectif principal de la conception de NETLANG était de répondre à une question : *Est-ce qu'on peut avoir un langage pour les Network Processors?* Les NPs étant des processeurs programmables spécifiques pour les applications réseau, le langage qui leur serait dédié devrait permettre de décrire les différentes tâches appliquées aux paquets passant par le NP. En plus de l'aspect de programmabilité lié aux NPs, il y a aussi l'aspect de configuration lié aux périphériques attachés aux NPs comme les interfaces physiques et

les mémoires externes (la TCAM, par exemple).

Le concept SDN avec le protocole OpenFlow sont arrivés pour nous donner un niveau d'API de description du traitement des paquets à travers le pipeline des *flow-tables* avec leurs règles de *forwarding*, les actions et les instructions OpenFlow. C'est ce qui nous a permis de définir un langage de programmation pour les *Network Processors* et avoir un premier résultat qui est de dire: oui, on est capable d'exprimer des applications de traitement de paquets à travers un langage de haut-niveau dédié aux *Network Processors* en utilisant une abstraction matérielle basée sur OpenFlow.

- NETLANG *est un langage basé sur OpenFlow.*

La deuxième question à laquelle nous avons répondu est la question suivante : *Est-ce qu'OpenFlow est suffisant en terme de description de comportement de paquets pour avoir un langage de deuxième niveau sur les Network Processors?* En effet, avec NETLANG nous avons deux niveaux de langages; le premier niveau du langage est un langage matériel qui s'exécute sur le NP. Le deuxième niveau du langage est un langage descriptif basé sur OpenFlow. L'avantage que nous donne OpenFlow c'est qu'il nous introduit un langage intermédiaire qui est plus riche et qui nous permet de créer plus facilement le *data plane.* OpenFlow est suffisant en termes de description de comportement pour avoir un langage de deuxième niveau sur les NPs. Les autres fonctionnalités comme la configuration et l'activation des interfaces physiques ont été ajoutées pour le *management plane.*

- *Adaptabilité de* NETLANG.

La validation de NETLANG à travers plusieurs types d'implémentation a permis de tester son adaptabilité. L'objectif de cette évaluation était de répondre à la question suivante : *Est-ce que le langage est assez générique pour pouvoir être exécuté sur des plateformes différentes?* Nous avons utilisé des environnements de TOPs en pipeline et des architectures de *microengines* avec du parallélisme et du *multithreading.* Un *back-end* spécifique à chaque plateforme est implémenté pour donner un adaptateur particulier pour un vendeur de NP donné. Au niveau du *data plane*, avec EZchip, on est dans le *forwarding* basique. Les paquets sont traités au niveau de la couche 2 et 3. Alors qu'avec Netronome, on peut aussi avoir des fonctionnalités de sécurité comme du *firewall* par exemple qu'on peut générer avec NETLANG et, toujours, grâce à l'API d'OpenFlow. Les *flow-entries* peuvent être considérées comme une généralisation des ACLs (*Access Control Lists*) utilisées par

les pare-feux dans les réseaux sécuriés. Au niveau du *control plane*, les tables OpenFlow peuvent être facilement implémentées grâce au mécanisme de la TCAM.

- *Expressivité de* NETLANG.

La validation de NETLANG à travers plusieurs types d'application nous a permis de tester son expressivité. Les applications sur lesquelles nous nous sommes basés sont des applications réseaux de domaines variés. Nous avons été capables d'exprimer des applications dans le domaine OpenFlow, comme dans le premier scénario présenté dans ce chapitre, et nous avons également réussi à écrire des applications non-OpenFlow, comme le scénario du réseau Metro Ethernet et l'application SPBM. Le langage est assez éloquent et offre plusieurs fonctionalités qui permettent de bâtir le routeur et le définir sur les 3 plans de *management plane*, *control plane* et *data plane*. En effet, en plus des structures de tables OpenFlow, le langage permet de décrire des structures de recherche génériques avec des clés qui peuvent être composées d'éléments variés et, donc, pas uniquement de champs de correspondance d'OpenFlow. De même, le résultat de la recherche n'est pas restreint aux actions et instructions OpenFlow. Cette flexibilité permet de construire des entrées de n'importe quel format. L'application peut utiliser facilement des tables ARP, par exemple, ou des tables de routage basiques. Les procotoles classiques peuvent être facilement implémentés.

- *Compilateur de* NETLANG.

Le comportement du compilateur permet d'assurer la consistance du système. On peut avoir des implémentations multiples de *data plane* où chaque *data plane* est spécifique à un profil, à un besoin spécifique. Chaque configuration d'un ensemble de *Packet Processing Paths* est associée à un niveau d'API ou à un niveau de *flow-entries* avec lesquelles il pourra changer de comportement. Ce *data plane* prend des commandes pour fonctionner qui viennent de l'OpenFlow, des tables de *flow-entries* qui présentent les points de changements, de modifications et de mise à jour du *data plane*. Un des problèmes vraiment important, que le compilateur traite, est que les trois niveaux, le *data plane*, le *control plane* et le *management plane* doivent être cohérents. Si on en modifie un, ça impacte les autres. Le compilateur permet que les 3 niveaux soient dans une symbiose. En effet, si le programmeur décide de changer la topologie du réseau et, donc, de modifier les interfaces du routeur ce qui touche au niveau *management plane*, cette mise à jour sera prise en

compte par le compilateur dans le reste du programme. De même, si on change le *control plane* et qu'on modifie la forme des *flow-entries* dans les tables, le compilateur permet de retourner les inconsistances que ça peut entraîner sur le *data plane*.

Le compilateur donne la capacité de changer au bon niveau et de trouver la stratégie pour que l'ensemble de l'application soit cohérente. À travers le compilateur de NETLANG, nous avons pu établir les stratégies de modifications globales de ces 3 niveaux et déterminer le niveau de modification qu'il faut faire. Nous avons été capables d'automatiser le processus de validation à tous les niveaux en introduisant les bonnes stratégies de vérifications. Il s'agit, en effet, de stratégies qui déterminent, pour le programmeur, ce qu'il faut qu'il fasse pour garder la cohérence du système.

La cohérence est vérifiée entre les 3 langages, comme si on avait un méta-compilateur qui génère le langage de configuration, le langage de *control plane* et le langage de *data plane*, et qui s'assure que chaque langage est cohérent avec les autres. Même si on a 3 niveaux de langage, on a un seul fichier qui décrit la configuration, qui définit les règles OpenFlow (*le control plane*) et qui donne le *data plane* asscié. La compilation permet que tout soit valide. Ainsi, le compilateur a pour rôle de générer un routeur consistant et cohérent en respectant un besoin spécifique.

- *La Complétude de* NETLANG.

Un dernier résultat, par rapport à notre langage, serait de le comparer par rapport aux fonctionnalités proposées dans les langages que nous avons étudié dans le chapitre 2. Pour établir notre comparaison, nous nous sommes basés sur 5 critères ou propriétés qui sont de dire (1) si le langage permet de faire du *packet processing* pour le *data plane*, (2) si le langage offre une API pour définir les tables de *forwarding* pour le *control plane*, (3) si le langage permet de faire des opérations de configurations pour le *management plane*, (4) si le langage supporte la description des *headers* et enfin (5) si le langage suit la vision SDNs, dans le sens où il tient compte des séparations entre le *data plane*, le *control plane* et le *management plane* (voir Tableau 5.2). NETLANG respecte l'ensemble de ses propriétés dans leur totalité contrairement aux autres langages qui vérifient au plus deux de ces propriétés.

	packet processing	*forwarding tables*	*configuration*	*headers*	*SDN-oriented*
NETLANG	✓	✓	✓	✓	✓
NetPDL				✓	
protobuf				✓	
NetConf			✓		
OF-Config			✓		✓
Snortran		✓			
Nox		✓			✓
Nettle		✓			✓
Frenetic		✓			✓
packetC	✓				
DPDK	✓				
NetKit	✓				
NetBind	✓				

Tableau 5.2 Comparaison entre les fonctionnalités des différents langages

5.4 Résumé

Dans ce chapitre, nous avons présenté des exemples d'applications et nous avons expliqué comment écrire ces application en NETLANG. Bien que notre langage repose sur une abstraction basée sur OpenFlow, il permet d'exprimer des applications réseau dans d'autres domaines qu'OpenFlow, comme les réseaux Metro Ethernet ou les réseaux PBBNs. Cette aptitude révèle la richesse de l'expressivité du langage. Avec le premier exemple de l'application OpenFlow, il était plus facile d'écrire directement la *flow-table* avec NETLANG . Dans le cas de l'application *Metro Ethernet* qui est caractérisée par l'énorme nombre de tables qu'elle utilise, nous avons pu vraiment tirer avantage du langage et de son compilateur. Le langage exprime très bien les tables d'échange OpenFlow. L'application SPBM est aussi une contribution importante. En effet, SPBM utilise plusieurs entités ce qui était intéressant pour nous parce que ça nous a donné un moyen de valider le comportement de chacune des composantes dans le réseau PBBN. Les principaux résultats que nous pouvons extraire de notre contribution, c'est de dire que, oui, nous sommes capables d'obtenir un langage pour les NPs et que la sémantique d'OpenFlow est suffisante en termes de description de comportement de paquets. Le compilateur de NETLANG permet de bâtir une *switch* consistante et complète intégrant les fonctionnalité de *packet processing*, de tables de *forwarding*, de configuration et de description de *header* tout en suivant le modèle SDN.

CONCLUSION

Les réseaux informatiques d'aujourd'hui témoignent d'une révolution importante, que ce soit au niveau du matériel utilisé dans les routeurs qui déploient des technologies de plus en plus sophistiquées, ou au niveau logiciel et applicatif où on voit apparaître de nouveaux paradigmes controversés. En effet, l'émergence des *Network Processors* et leur intégration dans la nouvelle génération de routeurs ont contribué à l'accroissement de la programmabilité de ces plateformes. Les routeurs programmables sont ainsi caractérisés par leur flexibilité et la haute performance qu'ils peuvent délivrer. Les communautés des réseaux programmables sont ensuite arrivées avec des concepts comme SDN (*Software Defined Networks*) pour encourager la programmabilité du réseau.

Dans ce mémoire, nous avons proposé une solution pour les réseaux programmables qui consiste à créer un langage de haut niveau pour les routeurs basés sur les NPs. Nous nous sommes particulièrement intéressé au protocole OpenFlow qui a été défini dans une optique de rendre le réseau plus flexible. Ses principales caractéristiques étant de séparer les fonctions de *control plane* de celles du *data plane*, de centraliser les décisions stratégiques dans un contrôleur et de fournir une interface pour que ce contrôleur puisse agir sur le *data plane*, font de lui un protocole générique et très flexible puisqu'en plus, il a la particularité de définir un routeur par un ensemble de tables en pipeline. Puis, c'est à travers ces tables et les règles de transmission qu'elles contiennent qu'on est capable de définir le traitement qu'on aura sur le paquet. L'autre avantage qu'offre OpenFlow, c'est l'aptitude qu'il donne aux *flow entries* d'être ajoutées et supprimées pendant que la *switch* est en exécution, ce qui nous donne la possibilité de changer le comportement du routeur à la volée. C'est donc la programmabilité des routeurs basés sur les NPs et la flexibilité qu'apporte OpenFlow au réseau qui nous ont amené à chercher s'il était possible d'avoir un langage de haut niveau pour décrire le comportement des paquets dans un NP. OpenFlow semblait être une bonne approche qui allait nous permettre d'avoir une sémantique intéressante en termes de description du traitement des paquets à l'intérieur d'un routeur.

Nous avons donc conçu et implémenté NETLANG, un nouveau langage de programmation basé sur la logique d'OpenFlow et dédié aux routeurs programmables. En effet, NETLANG utilise deux niveaux de langages. Le premier niveau du langage offre une abstraction du traitement des paquets à travers un pipeline de tables contenant un ensemble de règles de transmission. Alors que le deuxième niveau du langage traduit ce comportement au niveau matériel, ce qui le rend spécifique à la plateforme de destination. C'est donc l'introduction du premier niveau du langage qui nous a permis de concevoir un langage de haut niveau pour les NPs.

NETLANG permet d'exprimer des opérations stratégiques au niveau du *control plane* qui consiste à définir et remplir des *flow tables* ou, même, des tables basiques de transmission, comme les tables ARP ou les tables de routage IPv4. L'expressivité du langage a été étendue en lui ajoutant des fonctionnalités de *management plane* permettant de définir le profilage d'un routeur en termes d'activation des interfaces physiques et de configuration de leurs paramètres. Enfin, nous avons fourni un moyen de modéliser les opérations de *data plane* ce qui a contribué à l'enrichissement du langage et, ceci, à travers la notion de *Packet Processing Path*. De plus, des descripteurs de paquets et de *headers* et la manipulation de types de données propres au réseau, comme les adresses IP ou les adresses MAC, font de NETLANG un langage adapté spécifiquement pour les applications de traitement de paquets.

Nous avons par la suite procédé à la validation du langage à travers l'implémentation de compilateurs spécifiques pour deux plateformes différentes dans le but d'évaluer son adaptabilité. Nous avons également utilisé un ensemble d'exemples d'applications réseau comme SPBM et Metro Ethernet pour tester son expressivité. En termes de résultats, nous avons principalement relevé que le compilateur de NETLANG permettait d'assurer la cohérence du système. L'analyse et la vérification sémantique ont permis de prévenir les différentes formes de conflits qu'on pouvait avoir entre ce qui est défini au niveau du *management plane* et du *control plane* et ce qui est utilisé par le *data plane*. Les modules adaptateurs du compilateur de NETLANG ont permis de générer le code spécifique à la plateforme matérielle de destination. L'adaptateur pour le NP4 d'EZchip assure la traduction du langage NETLANG en un langage NPsl. Alors que l'adaptateur pour Netronome a permis de créer le script approprié pour l'environnement du NFP-3240 qui est aussi conforme au processeur IXP2800 d'Intel.

Ainsi, NETLANG s'est révélé facilement portable puisqu'il a aussi bien roulé sur une architecture en pipeline avec des PEs à jeux d'instructions dédiés et hautement optimisés et qui est conçue principalement pour faire du simple *forwarding* de paquet, que sur une architecture

multi-coeurs exploitant le parallélisme et le *multithreading* et qui est dédiée spécialement pour les applications de sécurité réseau comme l'encryptage des paquets et la DPI (*Deep Packet Inspection*).

L'adaptabilité et la portabilité de NETLANG ont été rendues possible grâce à la sémantique d'OpenFlow, sans laquelle nous n'aurions pas pu concevoir de langage pour les routeurs programmables. Chaque vendeur de NPs ayant ses propres spécificités, il est impensable d'avoir aujourd'hui un langage pour n'importe quel type d'application roulant sur les NPs. Cependant, en nous plaçant dans un contexte spécifique nous avons pu assurer la programmabilité à un haut niveau des routeurs basés sur les NPs.

Les travaux futurs pour le langage NETLANG devraient introduire les nouveaux concepts définis dans OpenFlow v1.2 [of211], comme l'extensibilité du support pour les *match fields* à travers les champs TLV (*Type Length Value*) et les classes OXM. NETLANG peut aussi offrir les fonctionnalités proposées dans OpenFlow v1.3.1 [of312], telles que l'introduction du champ priorité dans la *flow entry*, la définition d'une nouvelle table appelée *meter-table* et l'ajout d'autres protocoles comme IPv6.

D'autres concepts ont été également proposés pour OpenFlow v2.0, tels que le support de différents types de pipeline au niveau du *data plane*, où on peut avoir des tables en série, en parallèle ou les deux (des tabes en série et en parallèle combinées) dans le modèle de *forwarding*. Ces trois versions d'OpenFlow sont encore en discussion et restent des propositions dont le groupe de l'ONF (*Open Networking Foundation*) n'en a pas encore annoncé une version finale implémentable.

NETLANG peut aussi être étendu pour d'autres types de plateformes avec d'autres technologies de NPs. Ainsi, il pourrait être adapté avec d'autres routeurs programmables conçus par des vendeurs autres que EZchip et Netronome.

APPENDICE A

UNE APPLICATION METRO ETHERNET

```
/* layer2 MPLS encapsulation, MPLS unicast */
processingPath path1 {
    packetIn {
        header h1  {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:QinQtpid:0x88A8 ;
            field uint16_t:sTag ;
           field hex:etherType:0x0800 ;
        }
    }
    packetOut {
        header h2 {
            field mac_addr_t:O_DA ;
            field mac_addr_t:O_SA ;
            field hex:mplsTpid:0x8847 ;
            field uint32_t:label2 ;
            field uint32_t:label1 ;
            field uint32_t:labelvc ;
            next h1;
        }
    }
}

processingPath path2 {
```

```
    packetIn {
        header h1{
            field mac_addr_t:DA: ;
            field mac_addr_t:SA ;
            field hex:mplsTpid:0x8847 ;
            field uint32_t:label2 ;
            field uint32_t:label1 ;
            field uint32_t:labelvc ;
        }
    }
    packetOut {
        header h2 {
            field mac_addr_t:DA1 ;
            field mac_addr_t:SA1 ;
            field hex:QinQtpid:0x88A8 ;
            field uint16_t:sTag ;
            field hex:QTpid:0x8100 ;
            field uint16_t:sTag ;
        }
    }
}

processingPath path3 {
    packetIn {
        header h1 {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:tpid:0x8100 ;
            field uint16_t:ctag ;
            field hex:etherType:0x0800 ;
        }
    }
    packetOut {
        header h2 {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
```

```
            field hex:tpid:0x88A8 ;
            field uint16_t:stag ;
            field hex:etherType:0x0800 ;
        }
    }
}

processingPath path4 {
    packetIn {
        header h1 {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:tpid:0x88A8 ;
            field uint16_t:stag ;
            field hex:etherType:0x0800 ;
        }
    }
    packetOut {
        header h2 {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:etherType:0x0800 ;
        }
    }
}

processingPath path5 {
    packetIn {
        header h1 {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:etherType:0x0800 ;
        }
    packetOut {
        header h2 {
            field mac_addr_t:O_DA ;
```

```
            field mac_addr_t:O_SA ;
            field hex:etherType:0x0800 ;
        }
    }
}

processingPath path6 {
    packetIn {
        header h1 {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:tpid:0x88A8 ;
            field uint16_t:stag ;
            field hex:tpid:0x8100 ;
            field uint16_t:ctag ;
            field hex:etherType:0x0800 ;
        }
    }
    packetOut{
        header h2 {
            field mac_addr_t:O_DA ;
            field mac_addr_t:O_SA ;
            field hex:tpid:0x8847 ;
            field uint32_t:label1 ;
            field uint32_t:labelvc ;
            field hex:etherType:0x0800 ;
        }
    }
}

processingPath path7 {
    packetIn {
        header h1  {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:tpid:0x88A8 ;
```

```
            field uint16_t:stag ;
            field hex:tpid:0x8100 ;
            field uint16_t:ctag ;
            field hex:etherType:0x0800 ;
        }
    }
    packetOut {
        header h2 {
            field mac_addr_t:O_DA ;
            field mac_addr_t:O_SA ;
            field hex:tpid:0x8100 ;
            field uint16_t:O_ctag ;
            field hex:tpid:0x8847 ;
            field uint32_t:label1 ;
            field uint32_t:labelvc ;
            next h1;
        }
    }
}

processingPath path8 {
    packetIn {
        header h1  {
            field mac_addr_t:O_DA ;
            field mac_addr_t:O_SA ;
            field hex:tpid:0x8847 ;
            field uint32_t:label1 ;
            field uint32_t:labelvc ;
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:tpid:0x88A8 ;
            field uint16_t:stag ;
            field hex:tpid:0x8100 ;
            field uint16_t:ctag ;
        }
    }
```

```
    packetOut {
        header h2  {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
        }
    }
}


processingPath path9 {
    packetIn {
        header h1  {
            field mac_addr_t:O_DA ;
            field mac_addr_t:O_SA ;
            field hex:tpid:0x8100 ;
            field uint16_t:ctag ;
            field hex:tpid:0x8847 ;
            field uint32_t:label2 ;
            field uint32_t:label1 ;
            field uint32_t:labelvc ;
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:tpid:0x88A8 ;
            field uint16_t:stag ;
}
}
    packetOut {
        header h2  {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:tpid:0x8100 ;
            field uint16_t:O_ctag ;
        }
    }
}
```

```
processingPath path10 {
    packetIn {
        header h1  {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:tpid:0x8847 ;
            field uint32_t:label2 ;
            field uint32_t:label1 ;
            field uint32_t:labelvc ;
        }
    }
    packetOut {
        header h2 {
            field mac_addr_t:O_DA ;
            field mac_addr_t:O_SA ;
            field hex:tpid:0x8847 ;
            field uint32_t:label4 ;
            field uint32_t:label3 ;
            field uint32_t:labelvc ;
        }
    }
}

processingPath path11 {
    packetIn {
        header h1 {
            field mac_addr_t:DA ;
            field mac_addr_t:SA ;
            field hex:tpid:0x88A8 ;
            field uint16_t:stag ;
            field hex:etherType:0x0800 ;
        }
    }
    packetOut {
        header h2 {
            field mac_addr_t:DA ;
```

```
            field mac_addr_t:SA ;
            field hex:tpid:0x8100 ;
            field uint16_t:O_ctag ;
            field hex:etherType:0x0800 ;
        }
    }
}
```

APPENDICE B

UNE APPLICATION SPBM

```
/* this is an example of an SPBM application          */
/* this node in an edge (ingress/egress) SPBM device */

constant ethertype_802_1_Q  0x8100; /* vlan tagging */
constant ethertype_802_1_AD  0x88A8; /* QinQ encapsulation */
constant ethertype_802_1_AH  0x88E7; /* MAC in MAC encapsulation */
constant ethertype_802_1_AQ  0xC1; /* is-is SPB */

port p0 { /* CNP :  Customer Network Port */
    number = 1;
    speed = 10000;
    rx = true ;
    tx = true ;
}
port p1 {
    number = 2;
    speed = 2500;
    rx = true ;
    tx = true ;
}
port p2 { /* PNP: Provider Network Port */
    number = 3;
    speed = 2500;
    rx = true ;
    tx = true ;
```

```
}
port p3 {
    number = 4;
    speed = 10000;
    rx = true ;
    tx = true ;
}

/* learning table */
lookupStruct C_SA_to_CNP_learning_table(exact) {
     keyStruct {
         element mac_addr_t:C_SA;
     }
     resultStruct {
         element uint16_t:cnp;
     }
     entries {
         entry <11:22:33:44:55:01>,<1>;
     }
}

lookupStruct B_SA_to_CBP_learning_table(exact) {
    keyStruct {
        element mac_addr_t:B_SA;
    }
    resultStruct {
        element uint16_t:cbp;
    }
    entries {
        entry <44:55:66:77:00:01>,<3>;
    }
}

lookupStruct C_DA_to_B_DA_mapping_table(exact) {
    keyStruct {
        element mac_addr_t:C_DA;
```

```
    }
    resultStruct {
        element mac_addr_t:B_DA;
    }
    entries {
        entry <11:22:33:44:55:04>,<44:55:66:77:00:05>;
        entry <11:22:33:44:55:05>,<44:55:66:77:00:03>;
        entry <11:22:33:44:55:06>,<44:55:66:77:00:07>;
    }
}

lookupStruct S_VID_to_I_SID_mapping_table(exact) {
    keyStruct {
        element uint12_t:S_VID;
    }
    resultStruct {
        element uint24_t:I_SID;
    }
    entries {
        entry <1>,<1>;
    entry <2>,<1>;
        entry <3>,<1>;
    }
}

/* Unicast Filtering DataBase */
lookupStruct unicast_forwarding_table(exact) {
    keyStruct {
        element mac_addr_t:B_MAC_DA; /* Backbone MAC destination address*/
        element uint12_t:B_VID; /* Backbone VLAN Identifier */
    }
    resultStruct {
        element uint4_t:OUT_IF; /* outgoing interface identifier */
    }
    /* unicast forwarding entries */
    entries {
```

```
        entry <44:55:66:77:00:02,100>,<2>;
        entry <44:55:66:77:00:03,100>,<2>;
        entry <44:55:66:77:00:04,100>,<1>;
        entry <44:55:66:77:00:05,100>,<2>;
        entry <44:55:66:77:00:06,100>,<3>;
        entry <44:55:66:77:00:07,100>,<2>;
    }
}


/* Ingress Check Table for Loop Mitigation */
lookupStruct ingress_check_table(exact) {
    keyStruct {
        element mac_addr_t:B_MAC_SA; /* Backbone MAC destination address*/
        element uint12_t:B_VID; /* Backbone VLAN Identifier */
    }
    resultStruct {
        element uint4_t:IN_IF; /* outgoing interface identifier */
    }
    /* ingress check entries */
    entries {
        entry <44:55:66:77:00:02,100>,<2>;
        entry <44:55:66:77:00:04,100>,<1>;
        entry <44:55:66:77:00:06,100>,<3>;
    }
}

 packet pkt = incomingPacket();
 port p;
 p.number = pkt.getInputPort() ;
 if (p.number == 1) { goto path1; }

processingPath path1 {

    int s_vid, i_sid;
    int b_vid = 100;
```

```
int out_if ;  /* outgoing interface number */
mac_addr_t c_sa, c_da, b_da;
mac_addr_t b_sa = 00:11:e2:a3:1f:55 ;
boolean unresolved = false ;

packetIn {
    header h1 {
        field mac_addr_t:C_DA ; // 6 Bytes
        field mac_addr_t:C_SA ; // 6 Bytes
        field hex:ether_type:0x88a8; // 2 Bytes
        field uint16_t:S_tag; // 2 Bytes
        field hex:ether_type:0x8100; // 2 Bytes
        field uint16_t:C_tag; // 2 Bytes
        field hex:ether_type:0x0800; // 2 Bytes
   }
}

packetOut {
    header h2 {
        field mac_addr_t:B_DA ;
        field mac_addr_t:B_SA ;
        field hex:ether_type:0x88a8;
        field uint16_t:B_VID;
        field hex:ether_type:0x88e7;
        field uint16_t:I_SID;
        next h1 ;
    }
}

/*CNP : Customer Network Port*/
ingress processing_bloc1() {
    pkt.decodeLayer2();

}

parsing processing_bloc2() {
```

```
        s_vid = pkt.popVlanHeader();
    }

    search processing_bloc3() {
        lookup search1 = new lookup(S_VID_to_I_SID_mapping_table,s_vid);
        if (search1.match() == true) {
            i_sid = search1.getResult();
        }
        else {
            unresolved = true ;
        }
    }

    /* I-Component */
    parsing processing_bloc4() {
        c_da = pkt.getEthDstAddress();
        c_sa = pkt.getEthSrcAddress();
    }

    search processing_bloc5() {
        lookup search1 = new lookup(C_SA_to_CNP_learning_table,c_sa);
        if (search1.match() == false) {
            C_SA_to_CNP_learning_table.addEntry(c_sa,p.number);
        }
        lookup search2 = new lookup(C_DA_to_B_DA_mapping_table,c_da);
        if (search2.match() == true) {
            b_da = search2.getResult();
        }
        else {
            unresolved = true ;
        }
    }

    /* PIP : Provider Instance Port*/
    editor processing_bloc6() {     /* I-Tagging */
        pkt.pushHeaderField(0,b_da);
```

```
        pkt.pushHeaderField(6,b_sa);
        pkt.pushHeaderField(12,ethertype_802_1_AH);
        pkt.pushHeaderField(14,i_sid);
    }

    /* CBP :  Customer Backbone Port*/
    editor processing_bloc7() {     /* B-Tagging */
        pkt.pushHeaderField(12,ethertype_802_1_AD); /* ethertype */
        pkt.pushHeaderField(14,b_vid); /* B_VID */
    }

    /* B-Component Relay */
    search processing_bloc8() {
        lookup search1 = new lookup(B_SA_to_CBP_learning_table,b_sa);
        if (search1.match() == false) {
            B_SA_to_CBP_learning_table.addEntry(b_sa,p3.number);
        }
        lookup search2 = new lookup(unicast_forwarding_table,b_da,b_vid);
        if (search2.match() == true) {
            out_if = search1.getResult(); /* PNP identifier */
        }
        else  {
            unresolved = true ;
        }
    }

    /* PNP : Provider Network Port */
    egress processing_bloc9() {
        if (unresolved == true) {
            pkt.discard();
        }
        else {
            pkt.send(out_if);
        }
    }
}
```

BIBLIOGRAPHIE

[AA99] D.O. Awduche and J. Agogbua. Requirements for traffic engineering over mpls. RFC 2702, September 1999.

[AGH00] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*, volume 2. Addison-wesley Reading, MA, 2000.

[ASF12] P. Ashwood-Smith and D. Fedyk. Is-is extensions supporting ieee 802.1 aq shortest path bridging. RFC 6329, April 2012.

[Bak03] F. Baker. Point-to-point protocol (ppp) bridging control protocol (bcp). RFC 3518, April 2003.

[BBZ+12] F. Balus, M. Bocci, R. Zhang, A. Sajassi, and N. Bitar. Extensions to vpls pe model for provider backbone bridging. RFC 2702, August 2012.

[bea] Beacon controller. https://openflow.stanford.edu/display/Beacon/Home.

[Bjo10] M. Bjorklund. Yang-a data modeling language for the network configuration protocol (netconf). RFC 6020, October 2010.

[BLC12] B. Boughzala, Y. Lemieux, and O. Cherkaoui. Programming openflow switches using a unified and a dedicated high-level language (poster). IEEE INFOCOM, March 2012.

[BPSM+97] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66, 1997.

[CBH+03] G. Coulson, G. Blair, D. Hutchison, A. Joolia, K. Lee, J. Ueyama, A. Gomes, and Y. Ye. Netkit: a software component-based approach to programmable networking. *ACM SIGCOMM Computer Communication Review*, 33(5):55–66, 2003.

[CCK+02] A.T. Campbell, S.T. Chou, M.E. Kounavis, V.D. Stachtos, and J. Vicente. Netbind: A binding tool for constructing data paths in network processor-based routers. In *Open Architectures and Network Programming Proceedings*, pages 91–103. IEEE, 2002.

[CCM10] Z. Cai, A.L. Cox, and T.S.E.N. Maestro. Maestro: A system for scalable openflow control. Technical report, Technical Report TR10-08, Rice University, 2010.

[CL07] H.J. Chao and B. Liu. *High performance switches and routers*. Wiley-IEEE Press, 2007.

[clo] The cloudshield technologies. http://www.cloudshield.com/.

[Com] The Flow Processing Company. Network flow processing platforms. http://www.netronome.com/pages/reference-hardware/.

[Com08] LAN MAN Standards Committee. 802.1 q/d10, ieee standards for local and metropolitan area networks: Virtual bridged local area networks. http://www.ieee802.org/21/doctree/Temp/P802-21-D11.pdf, April 2008.

[Com10] The Flow Processing Company. Nfp-3xxx netronome flow processor. http://www.netronome.com/files/image/Products/nfp-32xx-architecture.jpg, 2010.

[DGK+10] A. Doria, R. Gopal, H. Khosravi, L. Dong, J. Salim, and W. Wang. Forwarding and control element separation (forces) protocol specification. RFC 5810, March 2010.

[DJ09] R. Duncan and P. Jungck. packetc language for high performance packet processing. In *High Performance Computing and Communications, 2009. HPCC'09. 11th IEEE International Conference on*, pages 450–457. IEEE, 2009.

[DJR11] Ralph Duncan, Peder Jungck, and Kenneth Ross. packetc language and parallel processing of masked databases. In *packetC Programming*, pages 335–344. Springer, 2011.

[dpd] Data plane development kit. http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/packet-processing-is-enhanced-with-software-from-intel-dpdk.html.

[EBSB11] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. Network configuration protocol (netconf). RFC 6241, June 2011.

[ES02] S. Egorov and G. Savchuk. Snortran: An optimizing compiler for snort rules. *Fidelis Security Systems*, 2002.

[ezc99] Ezchip technologies white paper. network processor designs for next generation networking equipement. http://www.ezchip.com/, 1999.

[FFH+10] N. Foster, M.J. Freedman, R. Harrison, J. Rexford, M.L. Meola, and D. Walker. Frenetic: a high-level language for openflow networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, pages 6–12. ACM, 2010.

[FS] R. Fernando and S. Stuart. Encoding rules and mime type for protocol buffers. http://code.google.com/p/protobuf/.

[FVA+10] N. Feamster, A. Voellmy, A. Agarwal, P. Hudak, S. Burnett, and J. Launchbury. Don't configure the network, program it! domain-specific programming languages for network systems. Technical report, Yale University, 2010.

[GDK11] N. Gligoric, I. Dejanovic, and S. Krco. Performance evaluation of compact binary xml representation for constrained devices. In *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on*, pages 1–5. IEEE, 2011.

[GH98] E.M. Gagnon and L.J. Hendren. Sablecc, an object-oriented compiler framework. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pages 140–154. IEEE, 1998.

[Gil08] R. Giladi. *Network processors: architecture, programming, and implementation.* Morgan Kaufmann Pub, 2008.

[GKP+08] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[Hed88] C.L. Hedrick. Rip version 2. RFC 2453, November 1988.

[Hel03] G. Held. *Ethernet networks.* Wiley Online Library, 2003.

[HH03] S. Halabi and B. Halabi. *Metro ethernet.* Cisco Systems, 2003.

[KCG+10] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A distributed control platform for large-scale production networks. *OSDI, Oct*, pages 1–6, 2010.

[KMC+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C programming language.* Prentice Hall, 1988.

[KWGT12] M. Kind, F. Westphal, A. Gladisch, and S. Topp. Split architecture: Applying the software defined networking concept to carrier networks. In *World Telecommunications Congress (WTC), 2012*, pages 1–6. IEEE, 2012.

[MAB+08] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[Moy09] J. Moy. Ospfv2 hmac-sha cryptographic authentication. RFC 5709, October 2009.

[net10] Netronome white paper. 40 gbps regular expression matching for network appliances. http://www.netronome.com/, 2010.

[of211] OpenFlow Switch Specification. Version 1.2 (Wire Protocol 0x03). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf, December 2011.

[of312] OpenFlow Switch Specification. Version 1.3.1 (Wire Protocol 0x04). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf, September 2012.

[ofc11] OpenFlow Configuration and Management Protocol OF-CONFIG 1.0. https://www.opennetworking.org/images/stories/downloads/of-config/of-config1dot0-final.pdf, November 2011.

[off11] OpenFlow Switch Specification. Version 1.1.0 Implemented (Wire Protocol 0x02). http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf, February 2011.

[onf] Open Networking Foundation. https://www.opennetworking.org/.

[Pet00] Richard Petersen. *Linux: the complete reference.* McGraw-Hill Professional, 2000.

[R+99] M. Roesch et al. Snort-lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*, pages 229–238. Seattle, Washington, 1999.

[RB06] F. Risso and M. Baldi. Netpdl: an extensible xml-based language for packet header description. *Computer Networks*, 50(5):688–706, 2006.

[RFRW11] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, pages 7–13. ACM, 2011.

[RL09] Y. Rekhter and T. Li. A border gateway protocol 4 (bgp-4). RFC 5709, October 2009.

[She13] S. Shenker. Software-defined networking at the crossroads. http://www.stanford.edu/class/ee380/Abstracts/130515.html, May 2013.

[SM88] Inc. Sun Microsystems. Rpc: Remote procedure call protocol specification version 2. RFC 1057, June 1988.

[sna] Snac controller. http://www.openflowhub.org/display/Snac/SNAC+Home.

[SRV08] R. Sanchez, L. Raptis, and K. Vaxevanakis. Ethernet as a carrier grade technology: developments and innovations. *Communications Magazine, IEEE*, 46(9):88–94, 2008.

[Tec] EZchip Technologies. Ezchip np-4 evaluation system. http://www.ezchip.com/p_ezsystem.htm.

[Tec11] EZchip Technologies. Ezchip np-4 product brief. http://www.ezchip.com/Images/pdf/NP-4_Short_Brief_online.pdf, April 2011.

[VAH10] A. Voellmy, A. Agarwal, and P. Hudak. Nettle: Functional reactive programming for openflow networks. http://www.cs.yale.edu/publications/techreports/tr1431.pdf, 2010.

[Zim80] Hubert Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, 1980.