UNIVERSITÉ DU QUÉBEC À MONTRÉAL

ALLOCATION DES RESSOURCES DANS UN ROUTEUR VIRTUALISÉ

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

ILYAS SNAIKI

AVRIL 2014

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LEXIQUE

Bare Metal (BM): ordinateur sans son système d'exploitation

Bid: enchère

Bundle: faisceau ou paquet

Central Processing Unit (CPU): unité centrale de traitement

Cloud Computing: nuage informatique

Cluster: groupe ou groupement

Cluster Local Scratch (CLS): mémoire haut débit interne par cluster

Double Data Rate (DDR): bus fonctionnant en débit de données double

Dynamic Random Access Memory (DRAM): mémoire vive dynamique

Fast-Path: chemin rapide et matériel de traitement

Field Programmable Gate Array (FPGA): réseau de portes programmables

Global Scratch (GS): mémoire haut débit interne global par processeur

Input/Output (I/O): Entrée/Sortie

Interlaken (IKI): protocole d'interconnexion libre de droits

Internet Protocol (IP): protocole Internet

Line Card (LC): carte de ligne pour le traitement des paquets

Logical Router (LR): routeur logique

Marking: marquage

Media Access Control (MAC): contrôle d'accès au support

Media Switch Fabric (MSF): interface de connexion au fabric de commutation

Micro Engine (ME): micro engin ou cœur de traitement de paquets

Multi-Cores: architecture de processeur à plusieurs cœurs de traitement

Multi-Threading: architecture de processeur à multiple fils d'exécution

Network Processor (NP): processeur réseau

Off-Chip: hors puce électronique

On-Chip: sur puce électronique

Operating System (OS): système d'exploitation

Package: paquetage

Policing: maintien de l'ordre

Quad Data Rate (QDR): bus effectuant quatre transferts de donnée à par cycle

Scratch: mémoire à base de registres

Slice: réseau virtuel

Slow-Path: chemin lent et logiciel de traitement

Static Random Access Memory (SRAM): mémoire vive statique

Supervision Card (SC): carte de supervision

Switch: commutateur réseau

Switch Fabric Card (SFC): carte de fabric de commutation

Testbed: banc de test

Thread: fil d'exécution

Time-To-Live (TTL): durée de vie

Traffic Manager (TM): gestionnaire de trafic

Transmission Control Protocol (TCP): protocole de contrôle de transmissions

Virtual Device Context (VDC): contexte d'équipement virtuelle

Virtual Machine Manager (VMM): gestionnaire de machine virtuelle

X Attachment Unit Interface (XAUI): interface de raccordement à l'unité de 10Gbps

RÉSUMÉ

La venue de la virtualisation des serveurs a amené un besoin de virtualisation pour les réseaux. Comme pour les serveurs, cette virtualisation au niveau des nœuds de réseau (commutateurs, routeurs, etc) offre un environnement dynamique dans lequel de multiples nœuds de réseau virtuels (dit aussi instance virtuelle) peuvent s'exécuter sur les même ressources matérielles sous-jacentes. Sur les serveurs, le concept d'hyperviseur a permis le partage et la répartition des ressources physiques entre les machines virtuelles. Similairement, un réseau virtualisé va requérir le développement de mécanismes d'allocation de ressources tels que les processeurs réseau, les mémoires de paquets, les mémoires de recherche, les accélérateurs et les interfaces d'entrées/sorties, qui seront de natures différentes.

Dans ce mémoire, nous proposerons une amélioration majeure de l'algorithme hiérarchique d'allocation de ressources dans le cadre du projet Netvirt afin de concevoir un nouveau mécanisme basé sur des "*bundles* de ressources" pour assurer une allocation équitable des ressources sur un nœud virtualisé. Cet algorithme hiérarchique est structuré en trois niveaux permettant respectivement d'assurer l'équité, la stabilité et la prévention des blocages dans un nœud virtualisé. Afin de valider notre approche, une implémentation réelle d'un nœud virtualisé a été élaborée sur une plateforme programmable. Nous démontrons à travers cette implémentation la stabilité et la robustesse de notre algorithme d'allocation et nous discutons de l'effet que peut engendrer l'introduction de la virtualisation sur la performance en termes de débit et de latence.

INTRODUCTION

Malgré le succès considérable que connaît la virtualisation au niveau des serveurs, les réseaux, et plus spécifiquement les nœuds de communications tels que les routeurs et les commutateurs, n'ont pas bénéficié suffisamment de cette nouvelle approche et continuent à utiliser généralement des approches classiques, telles que les VLAN et les VPN. L'application de la virtualisation au niveau des nœuds va permettre le déploiement de plusieurs réseaux virtuels complètement isolés les uns des autres sur une même infrastructure physique. Cette démarche de virtualisation aura un effet similaire à celles des serveurs en brisant le couplage étroit entre les opérations et le matériel. Il permettra ainsi de faciliter le déploiement de multiples réseaux hétérogènes sur le même substrat matériel sans être limité par la courante architecture inflexible du réseau Internet[10]. De plus, cette approche se différencie par le fait qu'elle permettra une isolation sur les paquets de chaque réseau virtuel. Le projet Netvirt[46] se situe ainsi dans le contexte de la virtualisation de nœud de communication contrairement aux approches traditionnelles qui consiste à déployer sur le même nœud des services virtualisés tels que les VPNs[35] (Virtual Private Network) et les réseaux superposés[24] (Overlay Network).

Un nœud réseau est un système embarqué dont la fonction principale est la transmission des paquets d'une interface d'entrée vers une autre interface de sortie en utilisant des ressources telles que la mémoire, les processeurs dédiés au traitement de paquets et des processeurs d'accélération (par exemples de cryptage des paquets ou alors de recherche sur les paquets). La principale tâche d'un nœud de communication revient donc à faire l'allocation optimale de ces ressources pour assurer un traitement rapide des paquets. Afin de supporter simultanément plusieurs instances de nœuds virtuels, un nœud virtualisé devra donc étendre ses fonctions d'allocation de ressources afin de permettre tant une isolation des paquets de chacune des instances virtuelles qu'une équité entre ces instances. Cette virtualisation de nœud nécessite le développement de nouveaux mécanismes d'allocation des ressources efficaces qui optimisent l'utilisation de la ressource matérielle entre les instances virtuelles sans compromettre leur performance.

Dans ce mémoire, nous allons proposé une extension du mécanisme d'allocation de ressources de celui proposé dans le cadre du projet NetVirt[46]. Cette extension utilise des *packages* de ressources, ou *bundle*, comme entités de base à partager entre les instances virtuelles. Ces *bundles* sont des regroupements de ressources physiques qui possèdent de liens de dépendance fonctionnels entre elles. Afin de valider notre mécanisme d'allocation basé sur des *bundles*, nous avons implémenté sur une plateforme de traitement de paquets un ensemble d'instances virtuelles et leur mécanisme d'allocation de ressources. Cette plateforme est basée sur un processeur réseau qui fournit la programmabilité et la configurabilité nécessaires pour notre mécanisme d'allocation. Notre mécanisme cherche à maintenir (a) l'équité entre les instances virtuelles, (b) la stabilité de chaque nœud virtuel et (c) la prévention des congestions sur les packages partagés.

Ce mémoire est constitué de cinq chapitres. Nous commencerons par présenter la problématique que nous chercherons à résoudre. Nous expliquerons les défis de l'allocation des ressources dans un environnement virtualisé. Nous donnerons également un aperçu général sur notre contribution en la situant dans le cadre du projet NetVirt, ainsi qu'une revue de littérature qui couvre l'architecture des routeurs, l'allocation des ressources et des solutions de virtualisation de réseau proposées jusqu'à présent.

Ensuite, un aperçu sur les fondements théoriques utilisés pour la conception de notre mécanisme d'allocation de ressources sera donné dans le deuxième chapitre. Nous couvrirons la théorie des jeux pour l'équité et la théorie de contrôle pour la stabilité.

Le troisième chapitre fournira une formalisation du mécanisme d'allocation à trois niveaux, à savoir l'équité, la stabilité et la prévention de blocage, en se basant sur les *bundles*. Nous illustrerons les différents paramètres d'entrée et de sortie de chaque niveau. Nous expliquerons l'implémentation algorithmique des différents niveaux et le lien avec les concepts théoriques.

Puis, l'environnement de déploiement et d'évaluation de notre mécanisme d'allocation sera donné dans le quatrième chapitre. Nous présenterons la plateforme à base du processeur réseau Netronome que nous avons utilisé pour notre implémentation, ainsi que les différentes applications utilisées. Nous caractériserons également le coût de la configuration.

Le dernier chapitre donnera des résultats d'évaluation pour l'équité, la stabilité et la prévention des congestions de notre mécanisme d'allocation. Nous analyserons ces résultats pour fournir des recommandations sur l'implémentation d'un tel mécanisme d'allocation sur une plateforme matérielle.

CHAPITRE I

ALLOCATION DES RESSOURCES SUR UN ROUTEUR VIRTUALISÉ

Il n'existe pas, actuellement, de solution de nœud de communication (routeur/switch) à haut débit virtualisé. Les seules solutions potentielles sont logicielles et utilisent les hyperviseurs sur des processeurs génériques de type Pentium et ARM. Ces solutions ne peuvent pas être utilisées dans un nœud de plus de 100 Gbps. Depuis quelques années, de nombreux projets de recherche ont proposé des approches de virtualisation des nœuds réseau. Dans le but de fournir une solution à cette problématique, nous avons proposé notre mécanisme d'allocation de ressources permettant cette virtualisation au niveau d'un nœud réseau. Dans ce chapitre, nous présenterons les approches de conception des nœuds réseau. Les conceptions sont basées sur un ensemble de mécanismes d'allocation de ressources. Également, nous donnerons un aperçu de l'état du marché des solutions de virtualisation des routeurs/switches pour montrer que la virtualisation n'est pas disponible au niveau du plan de donnée. Cette revue de l'état de l'art nous permettera de comparer notre approche avec celles proposées dans la littérature.

1.1 Problématique et contribution

La notion de virtualisation est une méthode ou un processus dont l'objectif est de diviser l'ensemble des ressources d'un ordinateur (*computing system*) en plusieurs environnements d'exécution à travers un partitionnement matériel ou logiciel, pseudo-parallélisme (temps partagé), pour permettre une simulation partielle ou complète d'une machine[52]. Ce concept a des avantages importants qui permettent à des infrastructures informatiques de fonctionner avec moins de ressources tout en répondant aux besoins des utilisateurs. Avec la virtualisation, la consolidation de plusieurs environnement d'exécution (machine virtuelle par exemple) permet d'optimiser

l'utilisation des ressources matérielles et réduire considérablement les coûts de management et d'administration. De plus, construire des plateformes sécurisées est moins compliqué grâce à l'isolation et l'indépendance entre les environnements d'exécution. Également, elle réduit considérablement les efforts pour les importantes opérations de gestion, à savoir la migration, le recouvrement et la sauvegarde.

Selon le contexte d'implémentation, nous pouvons distinguer trois principales catégories de virtualisation:

- *Virtualisation des serveurs*: Elle consiste à faire un masquage ou une abstraction des ressources physique d'un serveur pour donner une description logique différente de la description physique[22]. Cette abstraction permet la flexibilité à déplacer et aussi de changer la charge de traitement sur un serveur virtuel.
- *Virtualisation des stockages*: Elle permet de donner une abstraction logique à un ensemble de supports de stockage[26]. Ceci permet de créer des stockages virtuels de tailles arbitraires en gérant les stockages physiques comme un utilitaire plutôt qu'un ensemble d'entités physiquement indépendantes.
- *Virtualisation des réseaux*: Elle propose de découpler les fonctionnalités réseau et les mettre en deux catégories avec deux rôles différents, à savoir un fournisseur d'infrastructure et un fournisseur de service[10]. En pratique, cette démarche permet la coexistence de plusieurs réseaux virtuels isolés de différents fournisseurs de service sur la même infrastructure physique.

Dans notre contexte, nous nous intéressons à la dernière catégorie. Virtualiser une infrastructure réseau consiste à virtualiser les éléments qui la constituent, e.g. les nœuds et les liens réseau. Notre démarche se concentre sur la virtualisation des nœuds qui permet la coexistence d'un ensemble de nœuds virtuels complètement isolés sur le même nœud physique. Pour introduire la virtualisation dans un nœud réseau, un mécanisme d'allocation doit être mis en place pour gérer le correspondance des ressources physiques aux différents nœuds virtuels. Ce mappage peut prendre deux formes[18]:

- Allocation statique: Les ressources allouées à un nœud virtuel ne changent pas durant son fonctionnement. Cette forme d'allocation peut être intéressante dans le cas où le profil du trafic à l'entrée du nœud virtuel est statique.

- Allocation dynamique: Les ressources allouées à un nœud virtuel peuvent varier selon le profil du trafic entrant. Contrairement à l'allocation statique, cette méthode d'allocation est plus pratique puisque le profil du trafic est sujet au changement.

Un des défis majeurs est de concevoir un mécanisme d'allocation qui permet d'assurer une utilisation efficace des ressources physiques tout en assurant une équité dans la répartition de ces ressources, une stabilité dans la performance des différents nœuds virtuels et une prévention contre la congestion sur chacune des ressources. D'une part, l'équité permet d'assurer une distribution des ressources répondant aux besoins des différentes instances virtuelles d'une façon équitable. D'autre part, la stabilité permet de contrôler la performance d'un nœud virtuel en la stabilisant autour d'une valeur cible. Finalement, la prévention de blocage ou de congestion agit séparément sur chacune des ressources pour les partager d'une manière à minimiser le risque de congestion des nœuds virtuels qui l'utilisent.

Le niveau d'accessibilité aux ressources du nœud physique constitue un second défi majeur pour la mise en place d'un mécanisme d'allocation. Pour avoir un mécanisme d'allocation efficace, la plateforme matérielle utilisée par le nœud physique doit fournir un niveau important de flexibilité en termes de reconfiguration et de reprogrammation de ses ressources. Une plateforme qui offre un nombre trop limité de ressources reconfigurables limite le nombre de paramètres sur lesquels le mécanisme d'allocation peut agir et, par conséquent, limite la robustesse de la virtualisation. Par contre, avoir une plateforme trop flexible peut impacter l'efficacité du mécanisme d'allocation par le nombre important de ressources à gérer pendant chaque phase d'allocation.

Ce mémoire est réalisé dans le cadre du projet *NetVirt*[46]. Ce projet, en collaboration avec les entreprises Ericsson et Inocybe, examine les principaux défis de la virtualisation réseau en exploitant la découverte des ressources réseau, la création et la gestion des ressources dans un nœud virtualisé, l'extension de la virtualisation aux réseaux sans fil et optique, le déploiement de la virtualisation sur des ressources et des architectures différentes, la spécification des services que le fournisseur d'infrastructures peut offrir au fournisseur de services, et la gestion des nœuds virtuels. Dans ce contexte, notre contribution fournit une preuve-de-concept qui traite la problématique de l'allocation de ressources entre des instances virtuelles sur un nœud physique virtualisé.

Dans ce mémoire, nous nous sommes basé sur le mécanisme d'allocation à trois niveaux déjà disponible et utilisé dans le cadre de ce projet et nous proposons une amélioration de

ce mécanisme d'allocation de ressources dans un nœud virtualisé en se basant sur les *bundles* de ressources. Le nœud virtualisé est implémenté sur une plateforme matérielle à base d'un processeur réseau.

1.2 Conception des routeurs

Dans le but de fournir les capacités de transmission nécessaires pour satisfaire les besoins en terme de bande passante, l'architecture des routeurs a connu plusieurs changements au cours des vingt dernières années[8]. Ceci était le résultat de trois générations de routeurs qui diffèrent en technologie et en performance. Dans les premières générations, un ou plusieurs routeurs sont ajoutés à un *cluster* de routeurs afin d'étendre la capacité de commutation. Cette approche est devenue rapidement inefficace à cause du coût de gestion et de maintenance qui augmente avec l'ajout des nouveaux routeurs. Dans la génération suivante et actuelle, les routeurs sont conçus d'une façon modulaire dont la capacité peut être ajustée en changeant le nombre de modules dans le *boîtier* du routeur. Avec cette approche, étendre la capacité demande moins d'effort avec un coût réduit par rapport aux anciennes approches[13]. La figure 1.1 montre l'architecture d'un routeur de troisième génération. Elle consiste en trois types de cartes, à savoir les *Line Cards* (LC), les *Switch Fabric Cards* (SFC) et les *Supervision Cards* (SC). Les différentes cartes sont interconnectées à travers un module *Backplane*.

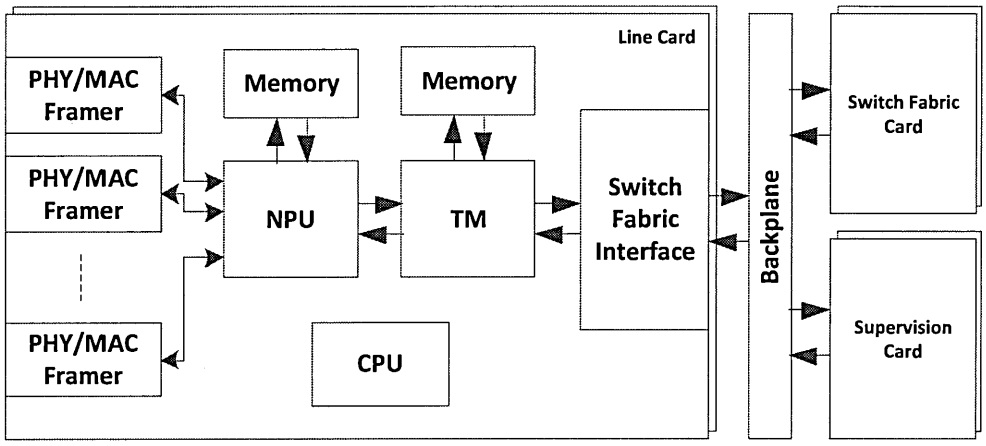

Figure 1.1: Architecture d'un routeur/switch à haut débit.

Les SFCs sont les modules qui assurent la transmission des paquets entre les LCs et les SCs. Par conséquent, les SFCs utilisent plusieurs unités matérielles de commutation configurables, conçues pour fournir une bande passante suffisamment élevée[7] pour que les paquets puissent se déplacer entre les différents modules sans congestion. Ces modules contiennent un contrôle logiciel pour les opérations de management des *buffers* de paquets, de file d'attente ou autres opérations de gestion de trafic. Pour les modules de supervision, ils contiennent l'intelligence qui contrôle et configure les autres modules. Ils exécutent tous les processus logiciels, à savoir les protocoles de routage, et permet la mise à jour des configurations des SFCs et les LCs.

Les LCs sont les modules qui nous intéressent dans notre contexte puisque c'est la partie que nous chercherons à virtualiser. Ces modules reçoivent/transmettent les paquets du/vers le réseau et prennent en charge les opérations de traitement de ces paquets. Ces modules sont connectés au backplane à travers une *Switch Fabric Interface* qui permet de transmettre/recevoir les paquets vers/du SFCs. Du côté du réseau, des interfaces *PHY/MAC/Framer* permettent de faire des conversions optique/électrique ou encore série/parallèle des signaux (qui représentent les données des trames), et des opérations de vérification pour assurer que des paquets ou des datagrammes corrects et complets peuvent être extraits des trames. Pour les opérations de traitement des paquets, un ou plusieurs *processeurs réseau* (NP) peuvent être utilisés. Le processeur réseau effectue des opérations d'analyse d'entêtes des paquets, recherche et classifie avec des tables de recherche et modifie l'entête des paquets. Il interagit avec les mémoires tampon (DRAM), la mémoire de travail (SRAM) et la mémoire de recherche (TCAM) qui peuvent être internes sur le processeur réseau ou externes sur les LCs. Aujourd'hui, en plus des accélérateur de recherche, les processeurs réseau disposent d'un accélérateur de *gestion de trafic* (TM). Cet accélérateur assure la gestion des buffers et l'ordonnancement des flux pour appliquer des configurations de qualité de service.

Dans les architectures des routeurs non-virtualisés, plusieurs approches d'allocation des ressources ont été proposées dans le contexte de qualité de service (QoS). Des outils de QoS, à savoir *Classification*, *Policing* et *Marking*, sont utilisés comme processus d'allocation de ressources pour la prévention contre les congestion ou la régulation de flux[53]. La *Classification* permet d'associer les flux de paquets à différentes classes de service dont chacune a son propre niveau de performance. Un exemple de deux classes de service peut être un *slow-path* déployé sur un *Operating System* (OS) et un *fast-path* implémenté en *Bare Metal* (BM). En redirigeant un flux vers le *fast-path*, il aura un débit maximal avec une latence minimale. Par contre, le coût

en terme de ressources est plus important que celui du *slow-path.* Ensuite, le *Policing* est utilisée pour contrôler un seuil de performance d'un flux afin d'ajuster l'allocation de ressource. Pour finir, le *Marking* consiste à associer aux paquets d'un flux une marque qui indique sa classe de service. Ce mécanisme utilise la classification avec l'ordonnancement pour décider l'ordre d'accès des flux à une ressource partagée. Cependant, ces approches ne tiennent pas compte de l'isolation des ressources allouées entre des instances virtuelles. Les différentes instances, qui sont définies par un groupe de flux, transigent sur l'ensemble des ressources disponibles avec une priorité ou une classe de service que le mécanisme d'allocation utilise pour ordonnancer l'accès au groupement des ressources partagées. Ce modèle ne garantit pas généralement que le comportement d'une instance n'affecte pas substantiellement le reste des instances.

Dans notre contexte, l'utilisation directe de ces approches matérielles n'est pas utile puisque l'isolation entre les ressources allouées aux différentes instances virtuelles est un paramètre important. Également, ces approches ont une visibilité sur une seule dimension du scénario d'allocation. Nous avons besoin d'avoir un mécanisme d'allocation ayant trois niveaux de visibilité: global, par tranche (instance virtuelle) et par ressource. La visibilité globale permet d'offrir un résultat d'allocation d'une instance virtuelle en tenant compte des autres instances. La visibilité par tranche se fait au niveau d'une slice qui représente un *switch* virtuel utilisant une fraction des ressources d'une plateforme physique. Cette visibilité permet de corriger l'allocation d'une instance pour garder sa performance autour d'un point d'équilibre. Finalement, la visibilité par ressource contrôle la charge sur la ressource et ajuste l'allocation pour éviter la congestion. Avec notre approche, nous offrirons ces trois dimensions avec les trois niveaux d'allocation, i.e. équité, stabilité et prévention de congestion. De plus, ces approches sont généralement limitées à un ensemble très réduit de ressources matérielles. Par exemple, ces approches sont souvent liées aux TMs pour appliquer la qualité de service sur les flux de paquets traités. Il est important pour nous d'avoir un mécanisme d'allocation dont la conception n'est pas liée à un type spécifiques de ressources. Il faut être capable d'agir sur un large vecteur de ressources matérielles offrant non pas juste la QoS avec les TMs, mais bien d'autres formes de contrôle, à savoir la capacité de traitement dans les NPs, la dimension des tables de recherche sur les mémoires ou la taille des buffets de paquets.

1.3 Ressources de traitement des paquets

Le traitement d'un paquet reçu sur un LC passe à travers un ensemble de tâches utilisant différentes ressources physiques. À la réception du paquet sur un port d'entrée du LC, il sera transmis à travers des SerDes (Serialization/Deserialization) qui permettent de le de-sérialiser avant de le transmettre vers le processeur réseau. Il commence donc le traitement du paquet en utilisant un ensemble de ressources d'entrée, ou accélérateurs. Le premier accélérateur permet d'allouer un espace sur le buffer des paquets et renvoyer un pointeur. Un descripteur contenant le pointeur, la taille et le port d'entrée est donc associé au paquet et l'accompagnera jusqu'à la fin de son traitement. Ensuite, une classification initiale est effectuée sur le paquet par un accélérateur de pré-classification. Cet accélérateur décode et analyse l'entête du paquet pour déterminer son chemin de traitement. Ces accélérateurs sont configurables et n'offrent pas de flexibilité de programmation.

Après avoir pré-classifié et déterminé les chemin de traitement, le paquet est transmis à un *cluster* de cœurs de traitement sur le NP. Ces cœurs de traitement sont programmables et contiennent les tâches de traitement sous forme d'un code assembleur. Ils utilisent une architecture basée sur le *multi-threading* qui permet d'avoir plusieurs *threads* matériels sur le même cœur. La flexibilité de ces cœurs en terme de programmation offre la possibilité d'associer un cœur, ou un thread, à une tâche spécifique. Autrement dit, il est possible de changer l'association entre les chemins de traitement et les threads sous-jacents. De plus, ces cœurs peuvent être programmés pour envoyer des requêtes de lecture/écriture à des mémoires externes contenant les paquets ou les tables de recherches. Le buffer des paquets utilise généralement la mémoire DRAM qui offre un espace très large et moins coûteux pour enregistrer un nombre important des paquets. Pour les structures de recherche, elles sont souvent déployées sur des mémoires moins profondes mais plus rapides comme la SRAM et la TCAM.

Quand les cœurs du processeur réseau finissent le traitement du paquet, il est ensuite transmis vers les accélérateurs de sortie. Le premier accélérateur effectue des opérations de gestion de trafic avec un TM. Il permet d'associer le paquet à une classe de service qui définit son ordre d'accès à son port destination. Le dernier accélérateur effectue la transmission du paquet vers le port destination et libère son pointeur sur le buffer des paquets.

Pour leur traitement, les paquets passent à travers un séquence de ressources (tels que la mémoire de paquets et le processeur réseau constitués de multi-coeurs, de mémoires de recherche

et de multiples accélérateurs) dont les limites diffèrent en terme de configurabilité et de programmabilité. Cette information permet de déterminer le vecteur de ressources sur lequel notre mécanisme d'allocation peut agir afin d'ajuster la performance d'une slice.

1.4 État du marché

L'architecture d'un équipement réseau est distribuée sur trois plans: plan de données, plan de contrôle et plan de management. Les opérations qui permettent de déterminer la façon avec laquelle les paquets seront traités par le plan de données, sont fournies par le plan de contrôle. De son coté, le plan de management est responsable de la configuration et la supervision de tout l'équipement réseau. Dû au défi majeur que pose la virtualisation du plan de données, la plupart des solutions sur le marché n'offrent qu'une solution de virtualisation au niveau du plan de management et de contrôle.

Cisco[56] offre une des solutions les plus avancées en terme de virtualisation sur leur série Nexus 7000 qui est dédiée aux centres de données. Ils ont défini le concept de *Virtual Device Context* qui permet la création de multiples instances virtuelles sur le même équipement. Cette solution virtualise les plans de management et de contrôle et permet la gestion et la configuration de chaque instance comme étant un équipement physiquement séparé. Un VDC contient deux engins pour les traitements de niveau 2 et 3. En tenant compte des contraintes de compatibilités entre les modules LC du routeur, un VDC peut être déployé sur un ou plusieurs modules. Et selon le type des modules de supervision présent sur le routeur, ce concept peut déployer deux configurations de quatre ou huit VDCs isolés. Deux VDCs ne peuvent pas communiquer directement sans avoir une connexion physique externe entre leurs ports, ce qui assure une isolation complète. Cette technologie repose sur l'allocation dynamique des ressources aux différents VDCs. Suivant le besoin de chaque VDC, un mécanisme d'allocation des ressources supervise périodiquement le taux d'utilisation des ressources allouées à un VDC et la charge du trafic qui lui est associé pour décider d'attribuer plus ou moins de ressources. Pour chaque VDC sur le routeur virtualisé, un profil est créé indiquant la liste des ressources avec la limite minimale sur une ressource devant être garantie, la limite maximale sur une ressource qui ne doit pas être dépassée et le taux d'utilisation de l'allocation sur une ressource. Ces paramètres sont consultés au début de chaque période de re-allocation et mis à jour si une nouvelle allocation est calculée. L'allocation agit principalement sur la taille des tables (TCAM ou SRAM) et la capacité du CPU. Ces ressources peuvent être partagées entre les différents VDCs, tandis que

les interfaces sont allouées statiquement et le mappage ne peut pas changer après l'instanciation d'un VDC.

Une autre solution de virtualisation de nœuds réseau utilisée dans l'industrie est le *Logical Router* (LR) de Juniper[25]. Cette solution permet à plusieurs routeurs logiques de s'exécuter sur le même routeur physique en instanciant un contexte logique isolé pour chaque LR. La grande différence entre cette solution et celle de Cisco est la possibilité de partager une interface physique entre plusieurs routeurs logiques. Juniper donne plus d'importance à la granularité des interfaces qu'un routeur peut offrir et considère que le gain est sur le nombre d'interfaces supportées. Sur chaque interface physique, une ou plusieurs interfaces logiques peuvent être instanciées. Afin de transmettre les paquets reçus sur une interface physique vers l'interface logique correspondante, un filtre d'entrée lié à l'interface physique analyse les paquets et détermine leur destination. Quand une interface logique ou physique est allouée à un LR, ce mappage ne change pas dynamiquement. L'allocation des ressources avec cette technologie se fait par tranche. Les ressources du routeur physique sont segmentées et, proportionnellement au profil du trafic entrant, allouées aux différents LRs. Cette solution peut générer l'allocation des ressources d'un routeur physique entre 16 LRs, tandis que celle de Cisco supporte jusqu'à 8 VDCs par nœud physique.

1.5 État de l'art

Durant les six dernières années, de nombreux travaux ont traité de la virtualisation réseau. Une revue complète a été élaboré dans l'article de[10]. Nous remarquerons que la grande majorité des travaux cités ne traite que de l'allocation des ressources pour un réseau virtualisé. Les seuls travaux qui traitent des nœuds réseau, considèrent des nœuds de communication logiciels basés sur les hyperviseurs et des processeurs génériques. Cependant, avec le mécanisme d'allocation proposé dans le cadre du projet NetVirt, notre approche offre une technique d'allocation que nous n'avons pas trouvé dans les approches de la littérature.

Notre mécanisme est conçu pour être dynamique, c-à-d que l'allocation des ressources aux différentes slices n'est pas statique et change dans le temps avec la demande des slices. Des approches comme PlanetLab[11] offrent une démarche d'allocation de ressources statiques. Au moment de la création d'une slice, une découverte des ressources sous-jacentes est faite pour associer statiquement un ensemble des ressources à cette slice. Du fait de la nature du

trafic réseau dont le profil change fréquemment, cette forme d'allocation affecte largement les performances de la slice.

Le partitionnement des ressources et l'isolation au niveau matériel entre les slices est une autre caractéristique importante de notre approche. Trellis[4] et bien d'autres approches se basent sur une isolation de haut niveau sur le système d'exploitation du nœud réseau. Ceci offre un processus d'ordonnancement pour accéder à une ressource matérielle plutôt que de la partitionner. Également, des approches existantes, comme VINI[2] ou FlowVisor[51], utilisent des tunnels ou agissent sur les entrées dans les tables de recherche (table de routage par exemple) pour définir des slices isolées sans agir directement sur les ressources physiques.

Notre mécanisme offre également une structure qui permet d'utiliser différents types de mesures et de fonctionner avec plusieurs objectifs, à savoir l'équité, la stabilité et la prévention des blocages. La plupart des approches, tels que Emulab[21] ou GENI[57], sont conçues avec des mesures de performances fixes, comme le débit par exemple, sur lesquelles elles se basent pour changer l'allocation. En plus, ils cherchent généralement à maintenir un partage équitable sans tenir compte de la stabilité d'une slice et la prévention de congestion sur les ressources.

Le projet FLARE[41] offre une implémentation matérielle d'un commutateur OpenFlow[54] permettant de virtualiser le plan de contrôle ainsi que le plan de données pour pouvoir définir de multiple slices. Également, une autre implémentation matérielle[53] propose une extension du protocole OpenFlow pour supporter une approche d'allocation basée sur les techniques de qualité de service (QoS), telle que la classification, pour assurer un partage équitable des ressources physiques entre les différentes instances virtuelles. Cependant, aucun de ces projets n'a proposé de stratégie d'allocation dynamique des ressources. Leur allocation semble statique et aucune stratégie d'équité ni d'optimalité n'est proposé. Au meilleur de nos connaissances, notre mécanisme est l'unique approche multi-objective qui se base sur des *bundles* et qui est déployé sur un environnement matériel comme celui utilisé dans les architectures courantes des nœuds réseau.

1.6 Conclusion

Virtualiser un nœud réseau nécessite un mécanisme d'allocation capable de fournir un partage optimal des ressources entre les différentes instances virtuelles de façon dynamique. Un tel mécanisme doit fournir un processus d'allocation multi-objectif et tenir compte de l'isolation entre les nœuds virtuels au niveau des paquets de chacun d'eux. L'équité, la stabilité et la

prévention de blocages sont les objectifs que l'on va chercher à atteindre par le mécanisme d'allocation. À travers l'état de l'art, nous avons démontré que notre approche apporte une grande amélioration par rapport aux approches existantes en offrant un mécanisme d'allocation dynamique capable de partitionner les ressources matérielles et d'isoler les paquets des différentes instances virtuelles dans le plan de données d'une plateforme réseau. Au meilleur de nos connaissances, le projet NetVirt est le seul à proposer une approche hiérarchique d'allocation des ressources tenant de l'isolation entre les nœuds virtuels. Nous nous sommes donc basés sur cette approche pour concevoir notre mécanisme se basant sur les *bundles*. Dans le chapitre (2), nous présenterons le fondement théorique que nous utiliserons dans la conception de notre mécanisme d'allocation. Nous illustrerons également comment ces théories serviront dans notre contexte pour fournir l'allocation des ressources aussi équitable et stable que requis.

CHAPITRE II

FONDEMENT THÉORIQUE POUR L'ALLOCATION DES RESSOURCES

L'équité, la stabilité, et la prévention de blocages sont des objectifs importants à garantir dans notre mécanisme d'allocation dans le but de fournir une distribution des ressources efficace et performante. Ces objectifs peuvent être atteints en se basant sur des techniques disciplinées basées sur des approches théoriques. Dans ce chapitre, nous donnerons un aperçu sur le fondement théorique de ces approches disciplinées. Nous couvrirons deux théories, à savoir la *théorie des jeux* et la *théorie de contrôle*. Nous présenterons le principe de base de chaque théorie avec les concepts sous-jacents. Nous montrerons également comment chaque théorie a contribué à notre approche d'allocation des ressources basée sur les bundles pour permettre de garantir l'équité, la stabilité et le non-blocage dans un routeur virtualisé.

2.1 Théorie des jeux

2.1.1 Concept de base

La théorie des jeux est une approche analytique qui permet de décrire les situations d'interaction entre plusieurs décideurs sur un intérêt commun[47][40][32]. Cette théorie donne une modélisation de ces situations d'intéraction et fournit les outils analytiques qui permettent de déterminer, pour un décideur la stratégie à suivre pour maximiser son gain. Les décideurs sont rationnels, c-à-d que chaque décideur cherche à maximiser son propre gain.

On appelle *jeu* un contexte d'intéraction stratégique entre un ensemble de décideurs qu'on appelle *joueurs*. Dans un jeu, un joueur a le choix entre plusieurs *actions* dont chacune lui apporte un gain pré-défini qu'on appelle *utilité* (ou *payoff*). Le jeu peut définir un ensemble

de contraintes sur les joueurs qui spécifient les choix qu'un joueur peut éventuellement prendre durant le jeu. On distingue principalement quatre groupes de modèles théoriques[47] dont chacun a ses propres suppositions et démarches:

- *Jeu stratégique*: Ce groupe est aussi appelé *non-coopératif* et propose un modèle dans lequel les joueurs prennent leurs décisions simultanément et indépendamment. Quand un joueur fait une action, il le fait sans être informé des décisions des autres joueurs. Chaque joueur cherche à maximiser son propre gain sans considérer l'impact de sa décision sur les autres joueurs.

- *Jeu extensif en information parfaite*: Ce groupe propose une description détaillée de la sequence des décisions d'un joueur dans une situation stratégique. Ce modèle permet aux joueurs d'être totalement informés de leurs actions précédentes à n'importe quelle étape du jeu.

- *Jeu extensif en information imparfait*: Ce groupe propose un modèle semblable au modèle du jeu extensif en information parfaite, en informant partiellement les joueurs de leurs actions précédentes à une certaine étape du jeu.

- *Jeu coopératif*: Ce groupe propose un modèle dans lequel un ensemble de joueurs peuvent se joindre pour former un groupe ou une *coalition*. Le jeu devient une interaction entre plusieurs coalitions, au lieu de joueurs individuels, où les joueurs d'une coalition appliquent un comportement coopératif.

Dans notre contexte, nous considérons un jeu stratégique dans lequel chaque slice cherche à maximiser sa propre utilité sans tenir compte des autres slices. Nous utilisons la formulation donnée dans[47]. Le tableau 2.1 donne la notation des différents composants d'un jeu non-coopératif. On suppose que notre jeu est fini, c-à-d que l'ensemble A_i des action du joueur i est

Tableau 2.1: Notation du jeu stratégique.

N	Ensemble des joueurs
A_i	Ensemble non-vide d'actions du joueur i, avec $i \in \mathbb{N}$
$\succsim_i$	Relation de préférence sur $A = \prod_{j \in N} A_j$, avec $i \in \mathbb{N}$

fini. Quand un joueur choisit une action, une conséquence est générée qui représente le profit

du joueur. Le joueur tient compte principalement des conséquences des différentes actions pour montrer ses préférences. On note C l'ensemble des conséquences, $g : A \rightarrow C$ la fonction qui donne l'association entre une action et sa conséquence, et $\succsim_i^*$ la relation de préférence du joueur $i \in N$ entre deux conséquences. Pour qu'une action a soit préférée par rapport à une action b, la conséquence de l'action a doit être préférée par rapport à la conséquence de l'action b. On écrit alors:

$$a \succsim_i b \quad \Rightarrow \quad g(a) \succsim_i^* g(b) \tag{2.1}$$

On représente la relation de préférence par une fonction appelée *fonction d'utilité*. La fonction d'utilité associe le choix d'une action par un joueur à une valeur numérique. Cette valeur numérique reflète le degré de satisfaction d'un joueur par l'action prise, c-à-d que la préférence d'une action est proportionnelle à la valeur numérique donnée par la fonction d'utilité. On note $U_i : A \rightarrow \mathbb{R}$ la fonction d'utilité du joueur $i \in N$. Si une action a est préférée à une action b alors l'utilité de l'action a est supérieure que l'utilité de l'action b. On note:

$$a \succsim_i b \quad \Rightarrow \quad U_i(a) \geqslant U_i(b) \tag{2.2}$$

Par la suite, on note $\langle N, (A_i), (U_i) \rangle$ un jeu stratégique avec un ensemble N de joueurs, un ensemble A_i d'actions pour un joueur $i \in N$ et U_i la fonction d'utilité d'un joueur $i \in N$. Dans notre contexte, les actions, ou profils d'action ou stratégie, d'un joueur correspond au vecteur d'allocation des ressources d'une slice représentant le joueur. Dans ce cas, la fonction d'utilité peut se calculer différemment selon l'objectif que l'allocation doit atteindre. Par exemple, si l'objectif de l'allocation est de corriger le débit d'une slice, la fonction d'utilité reflète le débit que la slice peut atteindre à partir d'un vecteur d'allocation.

2.1.2 Équité

Notre objectif derrière l'utilisation de la théorie des jeux dans notre contexte est de garantir l'équité entre les slices définies sur le même routeur. Cette équité peut être maintenue pour différents objectifs, c-à-d on peut chercher à avoir une équité entre les slices en terme de débit. L'équité est une notion qui a été introduite en exploitant l'abstraction d'une situation d'interactions entre plusieurs joueurs de la théorie des jeux[30] et ne fait pas partie du concept de base. L'équité a été introduite originairement dans[49] et repose sur le principe qu'un joueur est prêt à aider les autres joueurs qui l'aident, et à nuire aux joueurs qui le nuisent.

Une formulation de cette notion pour deux joueurs a été proposée dans[19][36]. Le tableau

2.2 donne la notation d'équité pour[49]. On définit le *gain matériel* (material payoff) comme le profit qu'un joueur peut avoir en fonction d'un contrat avec les autres joueurs. Dans le contexte d'un jeu où les joueurs cherchent à s'aider ou à se nuire, le gain matériel est différent de l'utilité. Autrement, l'utilité et le gain matériel font référence à la même chose[29]. On note $\pi_j^h(b_j)$ et

Tableau 2.2: Notation de l'équité.

A_1	Ensemble des actions du premier joueur
A_2	Ensemble des actions du deuxième joueur
$\pi_i : A_1 \times A_2 \rightarrow \mathbb{R}$	Fonction qui retourne le gain matériel du joueur i
$\Pi(b_j) = \{(\pi_i(a_i, b_j), \pi_j(a_i, b_j)) : a_i \in A_i\}$	Ensemble des pairs de gain matériel quand joueur i suppose que joueur j a choisi b_j

$\pi_j^l(b_j)$ respectivement la paire maximum et minimum qui représentent des optimums de Pareto, c-à-d les pairs dans lesquels on peut pas améliorer le gain matériel d'un joueur sans affecter négativement celui de l'autre joueur. On écrit alors:

$$\pi^e(b_j) = \frac{1}{2}(\pi_j^h(b_j) + \pi_j^l(b_j)) \tag{2.3}$$

le gain matériel équitable. Donc, pour évaluer la volonté du joueur i à aider le joueur j en choisissant a_i, on utilise la fonction suivante:

$$f_i(a_i, b_j) = \frac{\pi_j(b_j, a_i) - \pi_j^e(b_j)}{\pi_j^h(b_j) - \pi_j^{min}(b_j)} \tag{2.4}$$

avec $\pi_j^{min}(b_j)$ est le gain matériel le plus bas du joueur j. On note $f_j(b_j, c_j)$ le prédiction du joueur i sur la volonté du joueur j de l'aider. On change légèrement la notation de l'utilité pour qu'elle tienne compte des choix des deux joueurs et de la prédiction de la volonté d'aide d'un joueur. Donc, l'utilité s'écrit comme suit:

$$U_i(a_i, b_j, c_i) = \pi_i(a_i, b_j) + f_j(c_i, b_i).(1 + f_i(a_i, b_j)) \tag{2.5}$$

Le joueur i a maximisé sa propre utilité en choisissant a_i optimale. On écrit:

$$\begin{aligned} &\text{Maximize}_{a_i} \quad U_i(a_i, b_j, c_i) \\ &\text{subject to} \quad a_i \in A_i \end{aligned} \tag{2.6}$$

Alors, on dit que les deux joueurs sont en équilibre équitable si a_1, l'action prise par le joueur 1, et a_2, l'action prise par le joueur 2, vérifient:

$$a_i \in \arg\max_{a \in A_i} U_i(a, b_j, c_i) \quad \text{avec} \quad i = 1, 2 \quad \text{et} \quad c_i = b_i = a_i \tag{2.7}$$

Généralement, on considère les actions comme les biens que les joueurs demandent et on peut définir l'équité dans un jeu avec le degré de satisfaction de chaque joueur. Des joueurs sont servis équitablement dans un jeu s'ils sont tous satisfaits de la même manière. Malgré la simplicité de la définition, la concrétisation de l'équité en tant qu'objectif est en pratique un problème difficile, notamment quand les biens sur lesquels les joueurs en concurrence sont de natures multiples. Dans notre contexte, les slices sont en concurrence sur un ensemble de ressources de types différents (mémoire, processeur,...). Définir le degré de satisfaction des différentes slices doit se faire indirectement du moment qu'une slice peut avoir accès à un vecteur hétérogène de ressources. Afin d'évaluer cette mesure, on utilise une fonction d'utilité formulée selon le paramètre sur lequel la slice veut être satisfaite. Pour évaluer l'équité dans un jeu, plusieurs démarches peuvent être considérées[6] pour avoir une division équitable des biens du jeu entre les joueurs:

- *Division proportionnelle*: Un joueur obtient une partie des biens dont le ratio entre ce qu'il a demandé et ce qu'il reçoit est égal entre tous les joueurs.
- *Division sans envie*: Un joueur obtient une partie des biens du jeu qui est au moins plus que tous les joueurs.
- *Division exacte*: Un joueur reçoit exactement la partie des biens qu'il nécessite. Cette propriété n'est applicable que si les biens demandées par les joueurs sont inférieurs aux biens du jeu.
- *Division efficace*: Un joueur obtient une partie des biens qu'on ne peut pas améliorer sans nuire aux autres joueurs.
- *Division équitable*: Un joueur reçoit $1/n$ des biens d'un jeu de n joueurs.

Pratiquement, deux métriques d'équité[15][34] sont les plus répandues comme mesures dans la théorie des jeux:

- *Uniformité de l'utilité (utility uniformity)*[15]: Cette mesure est définie comme le ratio de l'utilité maximale et l'utilité minimale de l'ensemble des utilités des différents joueurs. Le degré de l'équité est proportionnel à la valeur de cette mesure, c-à-d quand la valeur de cette mesure augmente, la division des biens entre les joueurs est plus équitable. Cette

mesure s'écrit comme suit:

$$\tau(\omega) = \frac{\min_i U_i(\omega_i)}{\max_i U_i(\omega_i)} \tag{2.8}$$

avec ω_i la partie des biens du jeu données au joueur i.

- *Libre d'envie (envy-freeness)*[58]: Cette mesure évalue pour un joueur sa propre perception des biens que le jeu lui a associé en fonction de ce que les autres joueurs ont eu. Elle évalue l'utilité d'un joueur s'il a eu les biens d'un autre joueur dans le jeu. Cette mesure s'écrit comme suit:

$$\rho(\omega) = \min(\min_{i,j} \frac{U_i(\omega_i)}{U_i(\omega_j)}, 1) \tag{2.9}$$

2.1.3 Équilibre de Nash

L'équilibre de Nash est un des concepts de base dans la théorie des jeux. Il a été introduit dans[42][43] comme solution aux jeux stratégiques, ou non-coopératifs, de plusieurs joueurs. Dans le contexte de l'équilibre de Nash, un joueur connaît les stratégies des autres joueurs dans le jeu afin d'être capable de choisir la stratégie la plus avantageuse pour lui et le reste des joueurs. Un joueur ne peut donc pas maximiser son propre gain en changeant unilatéralement sa propre stratégie[47]. Quand un joueur ne peut pas améliorer son propre gain en changeant sa propre stratégie, sachant que les autres joueurs ont fixés leurs stratégies, on dit que l'équilibre est obtenu. L'ensemble des stratégies est appelé l'équilibre de Nash.

Formellement, on note $\langle N, (A_i), (U_i) \rangle$ un jeu dont N est l'ensemble de joueurs, A_i l'ensemble des actions du joueur i et U_i la fonction d'utilité du joueur i. On note $U_i(a_i, a_{-i})$, avec a_i l'action du joueur i et a_{-i} les actions des joueurs autre que i, l'utilité du joueur i. Le processus de l'équilibre Nash consiste à trouver l'action qui maximise l'utilite du joueur i si un joueur autre que i, c-à-d un joueur $-i$, choisit une action a_{-i}. La solution est la meilleure réponse du jeu[16]. On dit que a_i^* est la meilleure réponse d'un joueur i à l'ensemble d'actions a_{-i} du reste des joueurs si:

$$U_i(a_i^*, a_{-i}) \geqslant U_i(a_i, a_{-i}) \forall a_i \in A \tag{2.10}$$

Pour avoir un équilibre de Nash, il faut que l'ensemble des actions prises par les différents joueurs vérifient la condition de la meilleure réponse, c-à-d que chaque joueur a pris l'action la plus avantageuse pour lui et le reste des joueurs. On écrit alors:

$$U_i(a_i^*, a_{-i}^*) \geqslant U_i(a_i, a_{-i}^*) \forall a_i \in A \tag{2.11}$$

où (a_i^*, a_{-i}^*) est la stratégie la plus avantageuse de prise d'actions des différents joueurs.

Dans notre contexte, les différentes slices présentent leurs demandes aux routeurs qu'on a virtualisé. À cause de la limitation sur les ressources fournies par le routeur, il est probable que les slices ne vont pas recevoir exactement leurs demandes. Dans ce cas, l'équilibre de Nash va être très utile pour déterminer l'allocation des ressources à associer à chaque slice. Chaque slice est informée des demandes du reste des slices et va avoir assez de ressources qui permet d'optimiser sa propre utilité sachant que les slices restantes vont aussi avoir des utilités optimales.

2.1.4 Enchères combinatoires

L'enchère combinatoire est une stratégie utilisée dans le théorie des enchères qui rentre dans le contexte de la théorie des jeux. La théorie des enchères[59][31][28] fournit un ensemble d'outils et de modèles permettant l'analyse des mécanismes des enchères. Une situation d'enchères consiste en un ensemble de personnes qui sont en concurrence pour se procurer un ou plusieurs objets dans une mise aux enchères. Les personnes enchérissent sur les objets selon leurs préférences et leurs budgets en plusieurs étapes. Ce processus est semblable à celui d'un jeu avec des joueurs et des biens, d'où l'utilisation de cette théorie dans la théorie des jeux.

Les mécanismes de la théorie des enchères utilise généralement une représentation graphique pour modéliser l'association entre les joueurs et les biens. Cette représentation est appelée les graphes bipartis[1]. Une graphe biparti peut être noté $\langle U, V, E \rangle$, où U est l'ensemble des nœuds de départ, V est l'ensemble des nœuds d'arrivé et E l'ensemble des arêtes. Dans notre contexte, U est l'ensemble des slices, V est l'ensemble des ressources du routeur et E est la stratégie de mappage entre les slices et les ressources. Ces mécanismes se composent de trois phases[50]:

- *Phase d'initialisation*: La structure du graphe biparti est spécifiée, à savoir les joueurs et leurs budgets initiaux, ainsi que les biens et leurs prix initiaux.
- *Phase des enchères*: Chaque joueur met une enchère sur l'ensemble des biens qui l'intéresse. Après chaque itération, le prix de chaque bien est mis à jour avec un pas pré-défini.
- *Phase d'affectation*: Un mappage est établi entre les joueurs et les biens. Dans chaque itération, ce mappage est mis à jour.

Ce processus se termine quand chaque joueur a une affectation à un bien du jeu. Avec ces affectations, le gain de chaque joueur peut être calculé pour évaluer l'équité de la stratégie du mappage trouvée. Dans notre contexte, on peut évaluer l'utilité de chaque slice selon la stratégie d'allocation.

Une branche dans la théorie des enchères qui nous a servi est les mécanismes des enchères combinatoires[12]. Contrairement au principe de base des enchères, les joueurs dans des enchères combinatoires mettent leurs enchères sur un ensemble des biens, qu'on appelle *package.* Un package regroupe plusieurs biens liés dont la possession ensemble maximise le gain d'un joueur.

Dans le contexte des enchères combinatoires, on définit une fonction $\mathcal{V}$, appelée *fonction d'évaluation*, qui retourne le gain qu'un joueur obtient en se procurant un package[45]. On note $\mathcal{V} : B \rightarrow \mathbb{R}$, avec B l'ensemble des package et $\mathbb{R}$ l'ensemble des nombres réels. Cette fonction vérifie les propriétés suivantes:

- Si un package S est inclus dans un package T, le gain obtenu de S est donc inférieur ou égal à celui de T.

$$S \subseteq T \quad \Rightarrow \quad \mathcal{V}(S) \leqslant \mathcal{V}(T)$$

- Le gain obtenu d'un package vide est nul.

$$\mathcal{V}(\varnothing) = 0$$

- Soit $S \cap T = \varnothing$. Si:

$$\mathcal{V}(S \cup T) > \mathcal{V}(S) + \mathcal{V}(T) \quad \text{Alors } S \text{ et } T \text{ sont des compléments}$$

$$\mathcal{V}(S \cup T) < \mathcal{V}(S) + \mathcal{V}(T) \quad \text{Alors } S \text{ et } T \text{ sont des substituts}$$

On note $U_i(S) = \mathcal{V}_i(S) - p$ l'utilité du joueur i après avoir procuré du package S dont le prix est p. Le processus d'équité dans les enchères combinatoires consiste à trouver la stratégie de mappage dans un graphe biparti, donc les personnes sont les joueurs et les objets sont les packages, qui permet de maximiser l'utilité de chaque joueur tout en gardant les utilités des différents joueurs quasiment égales.

Cette stratégie est très utile dans notre démarche, puisque les ressources sur lesquelles les slices mettent leurs enchères ont des liens de dépendance. Par exemple, si on veut créer

une slice qui demande un engin de traitement de paquets, une base de données dans TCAM et deux interfaces réseaux sur un routeur de plusieurs linecard, il est préférable d'avoir ces trois composants sur le même linecard. Donc, au lieu de mettre des enchères sur les ressources séparément, on met des enchères sur un package de ressources dont la possession maximise l'utilité de la slice.

2.2 Théorie de contrôle

2.2.1 Concept de base

La théorie de contrôle propose une démarche théorique permettant de gérer l'évolution des systèmes dynamiques. L'objectif de cette démarche est de garantir une stabilité dans l'utilité du système contrôlé en supervisant sa sortie pour agir sur son entrée. L'entrée du système est ajustée suivant une référence donnée au système à contrôler. Trois éléments représentent les composants principaux dans un système contrôlé[39]:

- *Signaux d'entrée*: Ce sont les signaux qui proviennent des observations du système à contrôler avec des signaux de référence qui définissent des contraintes sur le comportement du système. Les observations peuvent être des mesures de performance. Les contraintes peuvent être des seuils sur la performance du système contrôlé, par exemple le minimum et le maximum de la latence des paquets.
- *Signaux de sortie*: Ce sont les signaux qui sont générés par le système contrôlé. Selon le contexte, le système contrôlé génère des signaux dont les valeurs sont ajustées pour stabiliser le système. Ces signaux peuvent être des vecteurs d'allocations de ressources pour répondre à un besoin en performance. Normalement, les signaux de sortie doivent respecter les contraintes (les seuils) données au système.
- *Système de contrôle*: C'est un ensemble en deux composants plus le système à contrôler. Les deux blocs sont:
  - *Capteur (sensor)*: Permet de récupérer des mesures observées et les convertir en grandeurs utilisables par le système de contrôle. Par exemple, un capteur peut transformer des valeurs de compteurs sur le nombre de paquets traités en mesure de débit.
  - *Contrôleur (controller)*: Permet d'exploiter le signal de sortie du capteur pour générer une sortie qui va servir pour contrôler le système afin de le stabiliser. Par exemple,

un contrôleur peut recevoir une mesure de débit à utiliser pour générer l'allocation des ressources que le système nécessite pour respecter les contraintes de performance.

Dans notre contexte, chaque slice a un ensemble de compteurs qui se mettent à jour en fonction du trafic traité par la slice. Le capteur transforme les valeurs de ces compteurs en mesures de performance pour chaque slice. Par la suite, le contrôleur propose un nouveau vecteur d'allocation en tenant compte des contraintes de stabilité de la slice. Ces contraintes de stabilité sont représentées pas des intervalles de valeur de ressources que la slice peut prendre.

Formellement, un système dynamique a une représentation discrète qui fait la liaison entre les états précédents du système contrôlé, les signaux de référence et les signaux de sortie. Le tableau 2.3 donne la notation liée à un système discret[60]. On définit deux fonctions qui

Tableau 2.3: Notation du système dynamique discret.

$\mathcal{X}$	Espace des états
$\mathcal{U}$	Espace des entrées
$\mathcal{Y}$	Espace des sorties
$\mathbb{N}$	Ensemble des nombres naturels

permettent de liées les différents composants de la chaîne de contrôle d'un système dynamique:

- $F : \mathcal{X} \times \mathcal{U} \times \mathbb{N} \rightarrow \mathcal{X}$ permettre de lier les signaux d'entrée et les états précédents avec l'état suivant. Cette fonction donne *l'équation d'état*:

$$x(k+1) = F(x(k), u(k), k) \tag{2.12}$$

- $H : \mathcal{X} \times \mathcal{U} \times \mathbb{N} \rightarrow \mathcal{Y}$ permettre de lier les signaux d'entrée et les états précédents avec les signaux de sortie. Cette fonction donne *l'équation de sortie*:

$$y(k) = H(x(k), u(k), k) \tag{2.13}$$

Dans notre contexte, l'équation d'état permet de faire le lien entre les mesures de performance précédentes et le prochain niveau de performance sur une slice. Pour l'équation de sortie, elle permet de faire le lien entre l'allocation et le niveau de performance à avoir sur la slice.

2.2.2 Stabilité

La stabilité est liée au changement de la sortie du système contrôlé. En gardant le paramétrage du système inchangé, la sortie du système subit des variations dont l'importance dépend de l'importance de la perturbation que le système éprouve. La notion de la stabilité a comme objectif de fournir les moyens pour contrôler ces changements et réagir efficacement pour maintenir la sortie d'un système dynamique constante ou, au pire, dans un intervalle acceptable.

Pour maintenir cette stabilité, un système dynamique utilise des seuils. Ces seuils sont mis sur les niveaux des signaux de sortie pour superviser la variation. Quand l'un de ces seuils est atteint, le système de contrôle se déclenche pour fournir une correction du paramétrage du système contrôlé pour maintenir sa stabilité[39]. On distingue deux types de seuils[37]

- *Seuils statiques*: Ce sont des seuils qui se définissent sur un système à l'instant initial et ne changent pas au cours du temps. Ce type de seuils est facile à utiliser mais il présente un manque de flexibilité et de robustesse.
- *Seuils dynamiques*: Contrairement aux seuils statiques, les seuils dynamiques peuvent changer constamment selon l'état du système. Ces seuils peuvent se calculer en se basant sur des mesures moyennes ou une fonction prédéfinie. Ce type de seuils offre plus de robustesse pour le système sous contrôle mais il peut être difficile à gérer.

Les seuils sont deux valeurs minimale et maximale qui définissent un intervalle dans lequel le système est considéré en stabilité. Atteindre un de ces deux seuils force le système à se comporter suivant des règles prédéfinies. Par exemple, si le trafic qui passe dans une slice est tel qu'une mesure de performance soit inférieure au seuil minimum, le contrôleur peut libérer une partie des ressources de la slice.

Avoir des seuils de contrôle permet d'aider à maintenir la stabilité d'un système dynamique. Par contre, calculer ces seuils sans tenir compte de la perturbation sur le système peut amener à plus d'instabilité[39]. Par exemple, si la perturbation sur le système est importante, les seuils doivent donner un intervalle de contrôle suffisamment large et les ajuster graduellement en fonction du niveau de stabilité atteint par le système contrôlé.

Dans notre contexte, le processus d'allocation et de désallocation des ressources prend un temps important pour le terminer. Choisir des seuils proches pour une slice qui subit des

variations importantes force le système de contrôle à déclencher un processus long qui arrêtera le fonctionnement de la slice pour appliquer ses résultats. Ceci causera plus d'instabilité dans la slice.

En plus de la dimension de l'intervalle de contrôle, d'autres facteurs peuvent causer l'instabilité d'un système dynamique[39]:

- Le temps de réaction du système de contrôle après avoir atteint un seuil. Si le système de contrôle prend trop de temps pour calculer le nouveau paramétrage de système contrôlé, la stabilité du système sera affectée négativement.
- La granularité du système de contrôle. Si le système de contrôle agit sur une granularité plus fine dans le système contrôlé, les corrections ont un impact plus visible sur la stabilité.
- La perturbation du système dynamique. Si la perturbation est importante alors que le système de contrôle est faible, en terme de temps de convergence et d'efficacité, la stabilité du système contrôlé sera largement affectée.

Évaluer la stabilité d'un système dynamique peut se faire d'une façon continue ou discrète dans le temps. La théorie de Lyapunov[27] discute en détails les deux contextes pour offrir une formulation décrivant la stabilité d'un système dynamique. Dans notre contexte, on prélève périodiquement les mesures sur les différentes slices pour le calcul de stabilité. Ceci nous met dans le contexte de la stabilité dans le temps discret. On utilise la notation dans[23].

On considère la fonction $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ définie comme suit:

$$\begin{cases} x(k+1) = f(x(k)) \quad k \in \mathbb{N} \\ f(0) = 0 \end{cases} \tag{2.14}$$

avec k étant le temps discret. La fonction f fait le lien entre l'état précédent du système dynamique et l'état suivant. Pour simplifier la notation, on pose $f^k(p)$ la valeur à $t = k$ avec $p \in \mathbb{R}^n$. Dans la stabilité, on définit x_0 un *point d'équilibre* qui vérifie les conditions suivantes:

$$\begin{cases} x_0 = f(x_0) \\ f^k(x_0) \equiv x_0 \quad k \in \mathbb{N} \end{cases} \tag{2.15}$$

Dans notre contexte, un point d'équilibre représente l'état autour duquel le système doit rester. La théorie de Lyapunov définit deux types de stabilité:

- *Équilibre stable de Lyapunov*: On dit qu'un système dynamique est en équilibre stable au sens de Lyapunov autour d'un point d'équilibre x_0 si:

$$\forall \epsilon > 0 \;\; \exists \delta \geqslant 0 \;\; \text{tel que} \;\; \forall x \in \mathbb{R}^n \;\; \| \, x - x_0 \, \| < \delta \;\; \Rightarrow \;\; \forall k \in \mathbb{N} \;\; \| \, f^k(x) - x_0 \, \| < \epsilon \quad (2.16)$$

- *Équilibre asymptotiquement stable de Lyapunov*: On note $\mathcal{A}$ le domaine d'attraction du point d'équilibre x_0. Un système est attractif au point d'équilibre x_0 si:

$$\lim_{k \to +\infty} f^k(p) = x_0 \;\; \forall p \in \mathcal{A} \quad (2.17)$$

On dit qu'un système dynamique est en équilibre asymptotiquement stable au sens de Lyapunov autour d'un point d'équilibre x_0 si le système est stable et attractif au sens de Lyapunov.

Dans notre contexte, il est pratique d'utiliser le deuxième type de stabilité. Ce type permet de définir le domaine d'attraction depuis lequel on peut définir les seuils de stabilité sur nos slices. Si l'état de la slice est dans le domaine d'attraction, alors les mesures de la slice respectent les seuils de stabilité.

2.2.3 Les filtres adaptatifs

La théorie de contrôle utilise largement des systèmes de commande en boucle fermée[14]. Afin de pouvoir stabiliser le système dynamique contrôlé, un signal de réaction est récupéré du système à travers un capteur afin de l'utiliser comme correcteur de paramétrage selon une référence prédéterminée. Cette forme de contrôle n'utilise pas seulement la mesure à l'instant de correction, mais peut utiliser un historique qui contient plusieurs mesures précédentes. Cet historique permettra de calculer plus précisément les paramètres à donner au système dynamique afin de le stabiliser. Dans notre contexte, une base de mesures est associée à chaque slice, dont la taille dépend de la précision de calcul de l'allocation désirée, et qui donne un couple [mesure,allocation] dans un espace temporel.

La commande en boucle fermée utilise généralement les filtres adaptatifs[20] pour prédire les nouveaux paramètres de stabilisation du système dynamique. Le système dynamique à commande en boucle fermée peut être définit comme suit:

$$\begin{cases} x(k+1) = F(x(k), u(k), k) = A.x(k) + B.u(k) \\ y(k) = H(x(k), u(k), k) = C.x(k) + D.u(k) \end{cases} \quad (2.18)$$

En utilisant les filtres adaptatifs, le nouvel état du système dynamique peut être prédit. Autrement dit, les paramètres A, B, C et D seront calculés en se basant sur l'historique du système dynamique pour trouver le nouveau paramétrage du système qui lui permet de converger vers la stabilité.

L'adaptabilité automatique de ces filtres permet au système de contrôle de réagir automatiquement en cas de dépassement de seuils pour corriger le système dynamique contrôlé. Par contre, le choix du filtre est très lié au contexte dans lequel le système dynamique fonctionne. Des métriques comme le temps de convergence du filtre et la robustesse doivent être pris en considération pour faire le bon choix du filtre. Par exemple, un filtre dont le temps de convergence est grand pour un système fréquemment perturbé peut amener le système à plus d'instabilité.

2.3 Conclusion

Concevoir un mécanisme d'allocation demande de faire appel à des approches théoriques permettant d'assurer les objectifs que nous cherchons à atteindre, à savoir l'équité, la stabilité et la prévention de blocages. À part la prévention prévention de blocages, l'équité et la stabilité peuvent bénéficier d'un fondement théorique solide à travers la théorie des jeux et la théorie de contrôle. Dans ce chapitre, nous avons présenté les fondements théoriques de notre mécanisme d'allocation des ressources dans un routeur virtualisé. Nous avons décrit comment la théorie de jeux et la théorie de contrôle permettent de garantir l'équité entre les slices et la stabilité des différentes slices. La mise en place de ces théories est faite par le biais des algorithmes implémentables. Le choix de l'algorithme à utiliser pour chaque théorie dépend du niveau de performance (la rapidité, la précision,...) désiré. Dans le chapitre (3), nous présenterons une formulation de ces théories dans notre contexte avec une description des implémentations algorithmiques.

CHAPITRE III

FORMULATION DE L'ALLOCATION DES RESSOURCES

La théorie des jeux ainsi que la théorie de contrôle offre un fondement théorique pour notre mécanisme d'allocation permettant d'assurer l'équité et la stabilité dans un routeur virtualisé. Par contre, ces approches doivent être ajustées pour pouvoir les implémenter sur une plateforme matérielle. Dans ce chapitre, nous présenterons la formulation de ces théories dans notre contexte. Nous décrirons la structure de notre mécanisme d'allocation composé de trois niveaux pour l'équité, la stabilité et la prévention des blocages des slices dans un routeur virtualisé tout en invoquant notre amélioration qui tient compte des *bundles* de ressources.

3.1 Description générale

3.1.1 Architecture générale

Notre mécanisme d'allocation de ressources permet de gérer l'affectation des ressources d'un routeur de plusieurs *linecards* à un ensemble de slices. Il se base sur des observations par slice qui peuvent exister sur un ou plusieurs *linecards*. La figure 3.1 fournit une architecture générale montrant les différents composants qui sont utilisés dans le processus d'allocation tout en spécifiant la nature de chaque interaction. On distingue trois blocs principaux:

- *Linecard Controllers*: C'est un ensemble de contrôleurs qui constitue un contrôleur distribué du routeur. Chaque *linecard* a un contrôleur dédié qui reçoit les requêtes d'allocation de ressources de ce *linecard*. Il permet également de donner des valeurs de statistiques des slices créées sur son *linecard*. Ce contrôleur a la visibilité totale sur les ressources de son *linecard* qui lui permet de déployer des configurations, après la réception d'une requête d'allocation des slices, ou lire un compteur, pour monter les statistiques sur les slices.

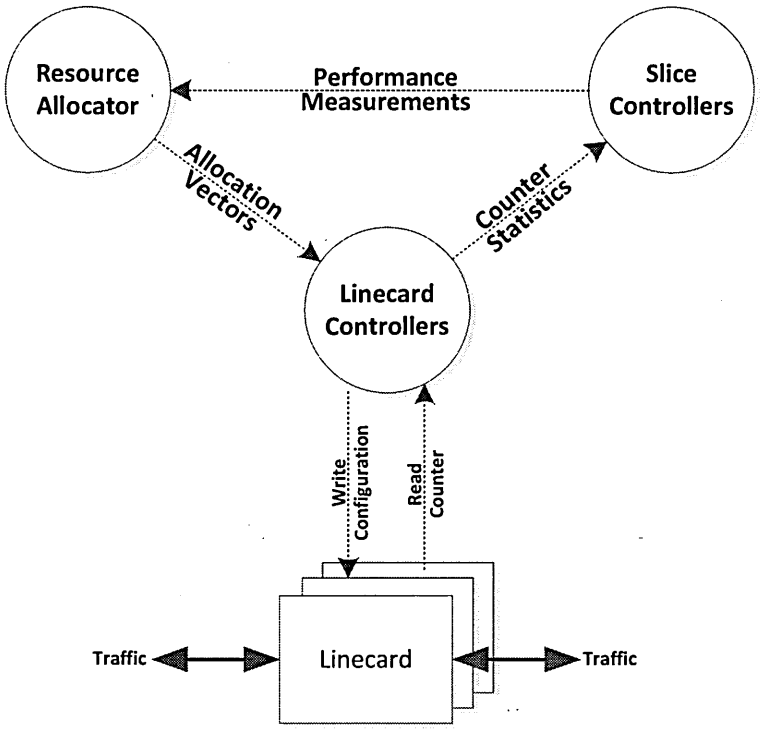

Figure 3.1: Architecture générale.

- *Slice Controllers*: C'est un ensemble de contrôleurs dont chacun est dédié pour une slice. Ce contrôleur reçoit des statistiques de différents contrôleurs de *linecard* et les transforme en mesures de performance. Ceci permet d'avoir une information de performance agrégée d'une slice qui peut exister sur un ou plusieurs *linecards* du routeur.

- *Resource Allocator*: C'est notre mécanisme d'allocation des ressources. Il a une architecture à trois niveaux dont chacun a la responsabilité de fournir l'allocation qui répond à un objectif, à savoir l'équité, la stabilité et la prévention de blocages. Il reçoit les mesures de performance des différentes slices pour produire le vecteur d'allocation de ressources de chaque slice sur chaque *linecard* du routeur.

Chaque slice a un *profil* séparé. Ce profil indique les exigences minimales et maximales d'une slice en terme de ressources. Par exemple, une slice peut indiquer le nombre minimum

et maximum d'engins de traitement de paquets qu'il peut avoir. Dans ce cas, le minimum doit être garanti tout le temps alors que le maximum ne doit jamais être dépassé. Par conséquent, l'allocation doit tenir compte de ces contraintes pour limiter l'allocation des ressources de chaque slice dans l'intervalle défini dans son propre profil.

3.1.2 Processus de fonctionnement

Initialement, le mécanisme d'allocation de ressources fait une allocation transitoire pour les différentes slices afin de pouvoir entrer dans le mode de fonctionnement normal. Les allocations initiales sont basées principalement sur les limites minimales de chaque slice. Après que les allocations initiales sont envoyées aux contrôleurs de slices, ces derniers construisent les configurations correspondantes et les déploient sur leurs *linecards* associés. Le routeur peut donc recevoir les paquets et mettre à jour les compteurs des différentes slices.

Dans le mode de fonctionnement normal, le contrôle de performance des différentes slices se fait périodiquement. Les contrôleurs des slices communiquent avec les contrôleurs des *linecards* pour récupérer les valeurs des compteurs de statistiques afin de calculer les mesures de performances pour les différentes slices. Les contrôleurs des slices peuvent donc faire ce lien entre les compteurs implémentés et les mesures de performance. Ces contrôleurs permettent de fournir non pas juste des mesures correspondantes à l'instant de contrôle, mais aussi un historique des mesures qui comprend des mesures précédentes.

Avec les mesures de performance prêtes, le mécanisme d'allocation de ressources peut fonctionner. Le mécanisme d'allocation est fait en trois niveaux, à savoir l'équité, la stabilité et la prévention des blocages. Chaque niveau a son rythme d'exécution, c-à-d la période après laquelle le re-calcul de ce niveau est fait. Également, chaque niveau est lié à une ou plusieurs mesures de performance qui peuvent être différentes des autres niveaux. Par exemple, le premier niveau peut contrôler le débit des slices alors que le deuxième contrôle la latence des slices. Le mécanisme d'allocation des ressources donne des vecteurs d'allocation de chaque slice qui indique la ressource, le *linecard* sur lequel la ressource existe et le pourcentage de la ressource qui lui est associé. Ces vecteurs seront distribués sur les *linecards* du routeur pour appliquer l'allocation des ressources.

Chaque contrôleur de *linecard* reçoit un ensemble de vecteurs d'allocation de ressources. Ce contrôleur prend en charge le fusionnement de ces vecteurs dans une seule configuration.

Pratiquement, une seule configuration est déployée sur un *linecard* à la fois, même si on peut avoir plusieurs slices sur le même *linecard*. Cette configuration reflète le partitionnement et l'affectation des ressources aux différentes slices. Au début de chaque cycle de contrôle, le contrôleur de *linecard* reçoit plusieurs requêtes des contrôleurs des slices pour lire les compteurs de slices dont ils ont besoin pour ce cycle.

3.1.3 Architecture trois niveaux d'allocation des ressources

Le mécanisme d'allocation est conçu d'une façon hiérarchique en trois niveaux dont chacun a un rôle différent:

- *Niveau d'équité (Fairness level)*: Le premier niveau prend en charge le contrôle de l'équité entre les slices. Selon la mesure de performance définie pour ce niveau, une fonction d'utilité correspondante est calculée pour chaque slice. Par la suite, ce niveau ajuste l'allocation afin d'uniformiser l'utilité de toutes les slices.

- *Niveau de stabilité (Stability level)*: Le deuxième niveau est responsable de la stabilité de chaque slice. Pour chaque slice, la fonction d'utilité correspondante est calculée pour une mesure de performance prédéfinie. Après le calcul de la fonction d'utilité, ce niveau ajuste l'allocation pour stabiliser l'utilité de la slice et minimiser sa variation.

- *Niveau de prévention de blocages (Bottleneck prevention level)*: Le troisième niveau, niveau de prévention de blocages, permet de détecter les blocages sur les différentes slices. Pour chaque slice, la fonction d'utilité est calculée. Avec des seuils prédéfinis sur l'utilité du slice, ce niveau peut détecter si l'utilité franchit un seuil pour corriger l'allocation.

Dans cette structure hiérarchique, chaque niveau a une période d'exécution inférieure au niveau supérieur. Ceci est dû à la complexité de la tâche de chaque niveau. Par exemple, le processus d'exécution du niveau d'équité prend en considération la totalité des ressources sur le système, tandis que celui du niveau de stabilité travaille juste sur une partie de ces ressources. Comme le montre la figure 3.2, le niveau d'équité a une période d'exécution au moins trois fois supérieure à celle du niveau de stabilité. Ceci est juste un exemple, en pratique, les périodes des différents niveaux sont ajustées pour maximiser l'efficacité du mécanisme d'allocation. Par exemple, si le trafic sur les différentes slices est quasiment stable, c-à-d pas de variation importante dans le débit des paquets, les périodes d'exécution des différents niveaux d'allocation

peuvent être élargies.

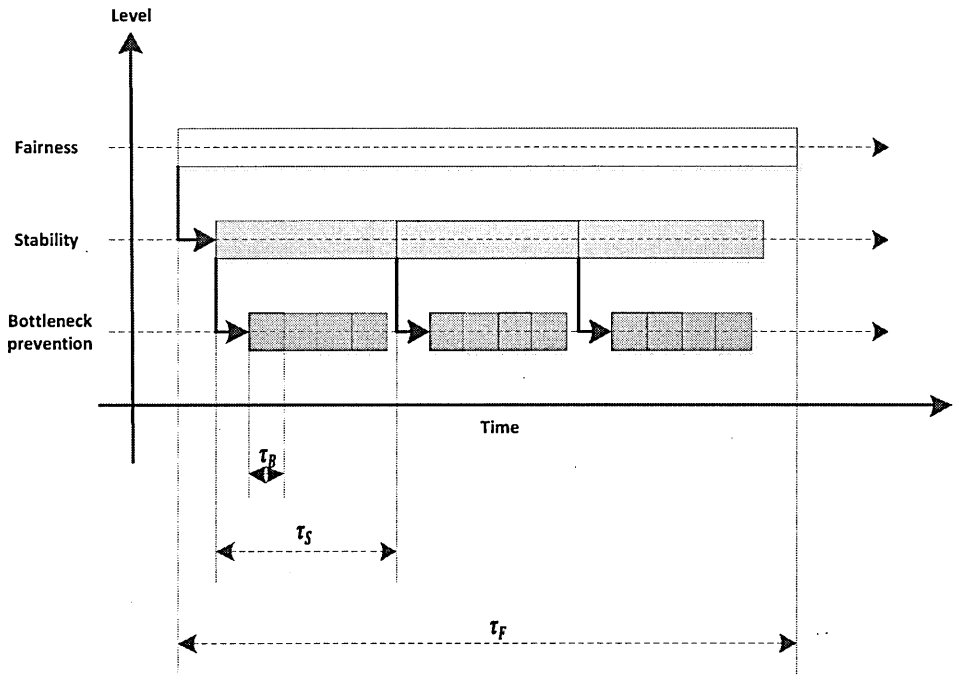

Figure 3.2: Graphe d'exécution des différents niveaux en fonction du temps.

L'architecture à trois niveaux du mécanisme d'allocation est présentée dans la figure 3.3. Les trois niveaux reçoivent des vecteurs de mesures de performance des contrôleurs de slices selon un rythme d'exécution pour générer éventuellement un nouveau vecteur d'allocation à envoyer aux contrôleurs de *linecards*. Les différents niveaux communiquent également entre eux en échangeant principalement deux signaux:

- *Resource Threshold*: Ce signal est envoyé d'un niveau supérieur à un niveau inférieur suivant, c-à-d du niveau d'équité au niveau de stabilité et du niveau de stabilité au niveau de prévention de blocage. Ce signal transmet des seuils sur les ressources qu'un niveau doit respecter dans son processus d'allocation.

- *Violation Event*: Ce signal est envoyé d'un niveau inférieur à un niveau supérieur précédent, c-à-d du niveau de stabilité au niveau d'équité et du niveau de prévention de blocage

au niveau de stabilité. Ce signal transmet un événement qui indique qu'un niveau a tenté de faire une allocation qui dépasse les seuils des ressources qui lui sont indiqués.

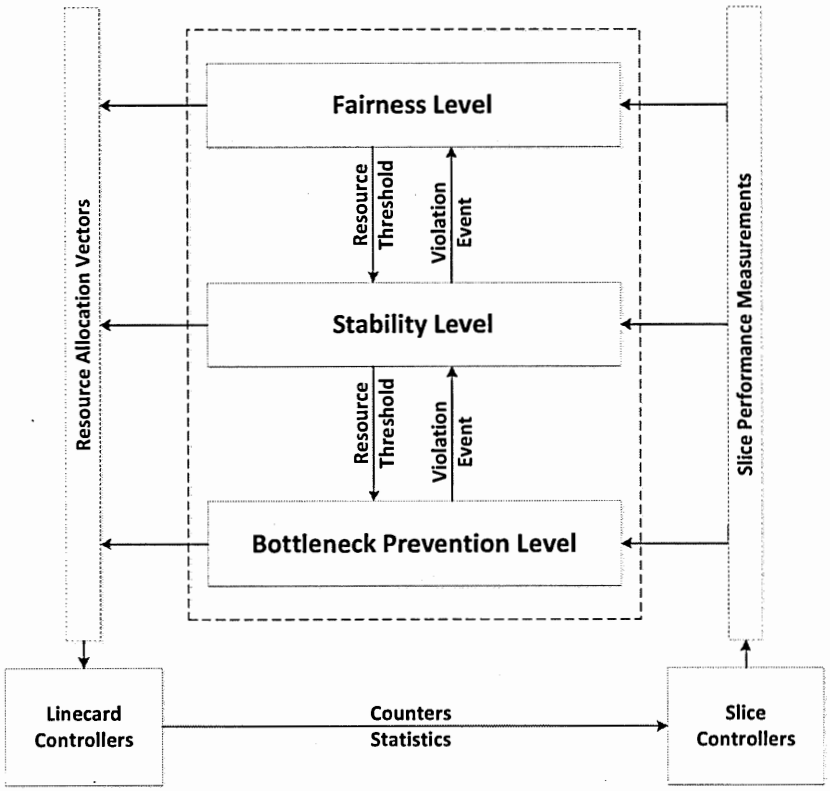

Figure 3.3: Vue générale de l'architecture interne du mécanisme d'allocation.

En utilisant ces signaux pour échanger des informations entre les différents niveaux du mécanisme d'allocation, on peut définir deux modes de déclenchement d'un niveau:

- *Normal Mode*: Ce mode représente le mode normal d'exécution d'un niveau dans lequel il se déclenche en fonction de sa période d'exécution prédéfinie. Au début de chaque période, le niveau propose une nouvelle allocation et passe les seuils d'exécution au niveau inférieur.
- *Forced Mode*: Ce mode représente le mode dans lequel un niveau est forcé à se déclencher indépendamment de sa période d'exécution. Ce mode est lié aux événements de dépassement de seuils dans un niveau. Quand un niveau reçoit un signal de dépassement de seuil du niveau inférieur suivant, il se déclenche pour proposer une nouvelle stratégie d'allocation

et des nouveaux seuils de ressources pour le niveau inférieur suivant.

3.2 Mécanisme d'allocation à trois niveaux

3.2.1 Généralités

3.2.1.1 Circuits et package de ressources

En principe, plusieurs stratégies d'allocation considèrent la ressource, par exemple mémoire ou engin de paquets, comme l'entité de base sur laquelle les instances virtuelles sont en concurrence. Une instance virtuelle spécifie un vecteur des fractions de ressources qu'elle demande. Le mécanisme d'allocation s'engage dans un processus d'allocation permettant d'associer la fraction de ressource que l'instance demande en considérant les ressources comme des entités isolées et indépendantes. En effet, il y a un niveau de dépendance entre les ressources qu'il faut prendre en compte dans la démarche d'allocation des ressources. Ignorer cette dimension peut amener à une allocation inefficace et parfois même illogique. Ces dépendances entre les ressources sont données par les circuits des slices. Comme le montre la figure 3.4, un circuit d'une slice donne la dépendance entre des ressources sur des *linecards* séparés du routeur. Par exemple, circuit-2 définit un lien de dépendance entre des ressources du *linecard*-1 et *linecard*-2, ce qui donne le regroupement $[DRAM_1, NP_1, TCAM_1, NP_2, DRAM_2]$. On appelle ces regroupements des *packages* ou *bundles*. Dans le processus d'allocation, la slice qui contient le circuit-2 doit mettre une enchère sur le *package* au lieu de mettre des enchères sur des ressources séparément. La slice doit obtenir une fraction non nulle pour chacune des ressources qui constitue le circuit pour qu'il fonctionne.

3.2.1.2 Perturbation de trafic

Les mécanismes d'allocation des ressources se basent sur un processus périodique dont les observations de performance se font au début de chaque période. En se basant sur ces observations, des vecteurs d'allocation se calculent pour atteindre les objectifs du mécanisme d'allocation (objectif d'équité par exemple). Les observations sont prélevées au début de chaque période, ce qui ne reflète pas nécessairement les changements durant la période. Durant chaque période, des changements importants peuvent se produire et affecter négativement les performances de la slice sans que les observations périodiques permettent de les détecter. Dans notre

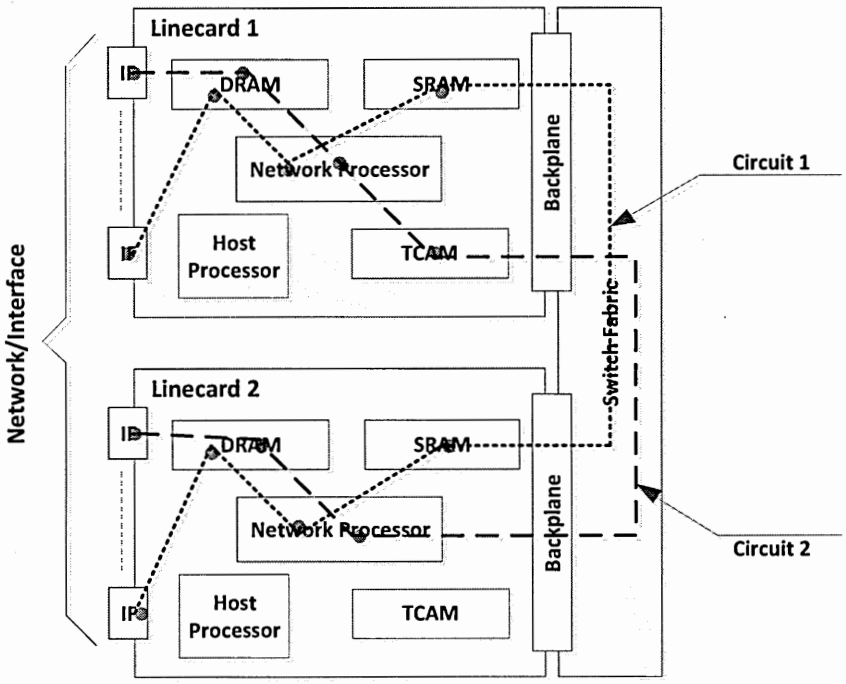

Figure 3.4: Exemple de dépendance des ressources par circuits.

contexte, on considère la *perturbation*. La perturbation est définie au début de chaque période d'allocation afin d'estimer la variation des observations. Les vecteurs d'allocation des ressources vont tenir compte de la perturbation durant cette période.

3.2.2 Notations

On considère un routeur de V *linecards*. Chaque *linecard* u_i peut avoir une spécification particulière d'un ensemble de ressources. Une ressource peut être la mémoire TCAM par exemple. On cherche à créer N slices sur ce routeur dont chaque slice s_i a un profil particulier. Chaque slice s_i comprend un ensemble de circuits P_i. Le mécanisme d'allocation se base sur des mesures de performance qui peuvent être par exemple la latence d'une slice. Cette mesure de performance donne l'utilité de chaque slice que le mécanisme d'allocation utilise. Le tableau 3.1 indique la notation globale à utiliser dans le reste de ce chapitre.

Tableau 3.1: Notations globales.

Symbole	Description
N	Nombre des slices sur le routeur
$S = \{s_1, s_2, ..., s_N\}$	Ensemble de slices sur le routeur
M_i	Nombre de circuits (ou paths) de la slice s_i
$P_i = \{p_{i,1}, p_{i,2}, ..., p_{i,M_i}\}$	Ensemble des circuits de la slice s_i
V	Nombre de *linecards* du routeur
$U = \{u_1, u_2, ..., u_V\}$	Ensemble des *linecards* sur le routeur
K	Nombre de types d'objectif pour le processus d'allocation
$O = \{o_1, o_2, ..., o_K\}$	Ensemble des types d'objectif pour le processus d'allocation, e.g. latence ou quantité de ressource
$\mathbb{R}$	Ensemble des nombres réels

3.2.3 Niveau-A: Niveau d'équité

Ce niveau se sert de la théorie des jeux pour contrôler les ressources à allouer à chaque slice. Un ajustement périodique des ressources allouées à chaque slice permet de maximiser l'utilité des différentes slices tout en garantissant l'équité entre elles. Pour le reste de cette section, on considérera la notation donnée dans le tableau 3.2 et le tableau 3.3.

L'objectif dans ce niveau est de garantir l'équité entre les différentes slices en terme d'utilité. Ce niveau ajuste dynamiquement les utilités des différentes slices en décidant d'une nouvelle allocation pour chaque slice au début de chaque période τ_F. Cette nouvelle allocation permet de maximiser l'utilité de chaque slice tout en tenant compte de l'équité qui doit être vérifiée avec le reste des slices. Formellement, on peut exprimer l'uniformité de l'utilité comme suit:

$$\forall t \quad \alpha = \frac{\min_{i\in\{1,...,N\}} \mathcal{U}_A(s_i, \Gamma_i(t), o_j)}{\max_{i\in\{1,...,N\}} \mathcal{U}_A(s_i, \Gamma_i(t), o_j)} \quad \text{avec} \quad 0 < \alpha \leqslant 1 \tag{3.1}$$

Atteindre l'équité revient à maximiser la quantité α. Si $\alpha \approx 1$, toutes les utilités des différentes slices sont égales. Une slice a un minimum de ressources qu'elle doit nécessairement avoir. En tenant compte de cette dernière contrainte, et on supposons la non-défaillance de toutes les ressources, α ne peut jamais être nul puisque qu'une slice ne doit jamais avoir une utilité nulle.

Tableau 3.2: Notations - Symboles du niveau d'équité.

Symbole	Description
τ_F	Période de temps avant de recontrôler l'équité
$B_i = \{b_{i,1}, b_{i,2}, b_{i,3}, ...\}$	Ensemble des regroupements de ressources demandé par une slice s_i
$A_{i,j} = (a^1_{i,j}, a^2_{i,j}, a^3_{i,j}, ...)$	Vecteur d'allocation allouée (fraction de ressource) pour le niveau-A du regroupement $b_{i,j}$ du slice s_i
$\widehat{A}_{i,j} = (\widehat{a}^1_{i,j}, \widehat{a}^2_{i,j}, \widehat{a}^3_{i,j}, ...)$	Vecteur d'allocation demandée (fraction de ressource) pour le niveau-A du regroupement $b_{i,j}$ du slice s_i
$\underline{A}_i$	Seuil minimum d'allocation des ressources pour slice s_i
$\overline{A}_i$	Seuil maximum d'allocation des ressources pour slice s_i
Γ_i	Nombre de paquets traités par la slice s_i
$\widehat{\Gamma}_i$	Nombre de paquets estimés à traiter par la slice s_i

Au début de chaque cycle τ_F, les différentes slices enchérissent sur un ensemble de *packages*. Les *packages* donnés sont indépendants et une slice enchérit sur un sous-ensemble de l'ensemble des *packages*. La valeur de l'enchère qu'une slice met sur un *package* est proportionnelle à son exigence au début de la période d'exécution. Dans notre mécanisme d'équité, on exploite le concept des enchères combinatoires en considérant des *packages* au lieu de considérer les ressources séparément. De plus, on ajoute le concept de Nash pour pouvoir appliquer une stratégie de partitionnement sur un *package* dans le cas où plusieurs slices cherchent à l'obtenir.

On modélise ce contexte par un graphe biparti, que l'on note $Y = \langle E, F, Q \rangle$, comme le montre la figure 3.5. L'ensemble de départ E du graphe est donné par les slices qu'on désire créer sur notre routeur. L'ensemble d'arrivée F contient les *packages* partageables. Les associations sont données par l'ensemble des arêtes Q. En se basant sur les slices, les circuits des différentes slices et les *packages* de chaque circuit, on construit le graphe biparti comme suit:

- L'ensemble de départ est égale à l'ensemble des slices:

$$E = S$$

- On note θ la fonction qui retourne l'ensemble des *packages* $B_{i,j}$ qui peuvent être associés à un circuit $p_{i,j}$. L'ensemble des *packages* sur lesquels s_i peut mettre une enchère, est donné

Tableau 3.3: Notations - Fonctions du niveau d'équité.

Fonction	**Description**
$\mathcal{U}_A : S\text{x}\mathbb{R}\text{x}O \rightarrow \mathbb{R}$	$\mathcal{U}_A(s_i, \Gamma_i, o_j)$ la fonction d'utilité d'objectif o_j du slice s_i
$\widehat{\mathcal{U}}_A : B_i\text{x}\mathbb{R}^{D_j}\text{x}\mathbb{R}\text{x}O \rightarrow \mathbb{R}$	$\widehat{\mathcal{U}}_A(b_{i,k}, A_{i,k}, \widehat{\Gamma}_i, o_j)$ Fonction d'estimation d'utilité d'objectif o_j du slice s_i
$\phi : B_i\text{x}\mathbb{R}^{D_j}\text{x}U \rightarrow C_k\text{x}\mathbb{R}^Q$	$\phi(b_{i,k}, A_{i,k}, u_j) = (c_{j,l}, G_{i,j,l})$ Fonction qui retourne l'allocation et le regroupement de ressources sur un *linecard* u_j de la slice s_i

par:

$$B_i = \bigcup_{j=1}^{M_i} \theta(p_{i,j}) = \bigcup_{j=1}^{M_i} B_{i,j}$$

- L'ensemble d'arrivée est donné par l'union de l'ensemble des *packages* des différentes slices:

$$F = \bigcup_{i=1}^{N} B_i$$

- L'ensemble des arêtes $Q = \{q_1, q_2, q_3, ...\}$, avec $q_i = [s, b]$ tel que $s \in E$ une slice et $b \in F$ un *package*, est donné comme suit:

$$\forall q \in Q \text{ avec } q = [s_i, b] \; \exists p_{i,j} \text{ tel que } b \in \theta(p_{i,j}) = B_{i,j}$$

c-à-d que pour une arête slice-*package*, il existe un circuit de cette slice qui peut demander ce *package*.

Après avoir construit le graphe biparti à travers la description des différents circuits, chaque slice enchérit sur un sous-ensemble de *packages* disponibles en prenant en considération son profil et la perturbation de son trafic. Le profil permet de spécifier les contraintes minimales et maximales (Annexe B) à prendre en considération avant qu'une slice mette une enchère sur un *package*. La perturbation sur le trafic permet de déterminer l'allocation minimale et maximale à passer au niveau de stabilité.

Mathématiquement, on cherche à maximiser l'utilité de chaque slice. Le maximum est obtenu en trouvant le vecteur d'allocation $\widehat{A}'_{i,k}$ pour chaque *package* $b_{i,k}$ qui maximise l'utilité

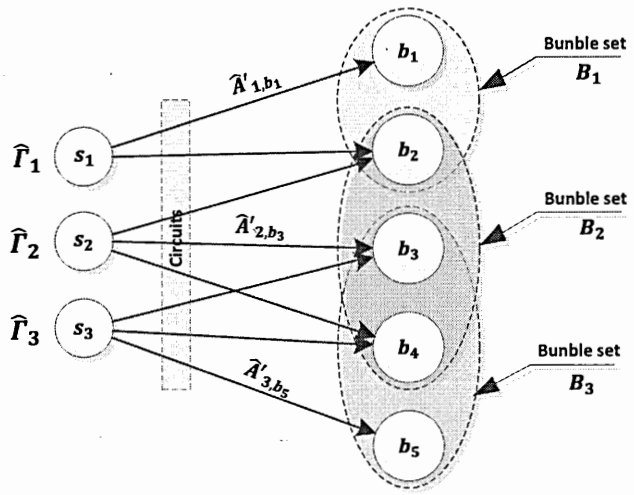

Figure 3.5: Exemple de graphe biparti de trois slices et cinq *packages*.

$\hat{\mathcal{U}}_A(b_{i,k}, \hat{A}'_{i,k}, \hat{\Gamma}_i, o_j)$ pour une perturbation $\hat{\Gamma}_i$. Le vecteur $\hat{A}'_{i,k}$ représente alors l'enchère, ou *bid*, que la slice va mettre sur un *package*.

Les enchères qu'une slice met sur les différents *packages* ne sont pas nécessairement égales. La valeur du bid dépend de la préférence d'un *package* pour une slice. On rappelle la fonction d'évaluation ν_i de la slice s_i que l'on note $\nu_i : B_i \rightarrow \mathbb{R}$. On note alors:

$$\forall i \in \{1, ..., N\} \quad \nu_i(b_{i,k_1}) \leqslant \nu_i(b_{i,k_2}) \quad \Rightarrow \quad \hat{A}'_{i,k_1} \leqslant \hat{A}'_{i,k_2} \tag{3.2}$$

En se basant sur les enchères, le mécanisme d'équité peut s'exécuter pour donner les vecteurs d'allocation acceptables $\hat{A}_{i,k}$. On pose $\mathcal{A}_i = \{\hat{A}_{i,1}, \hat{A}_{i,2}, ...\}$ la meilleure réponse de la slice s_i. La meilleure réponse d'une slice s_i vérifie la conditions suivante:

$$\exists \hat{A}_{i,k} \text{ tel que } \forall \hat{A}'_{i,k} \neq \hat{A}_{i,k} \quad \sum_k \hat{\mathcal{U}}_A(b_{i,k}, \hat{A}_{i,k}, \hat{\Gamma}_i, o_j) \geqslant \sum_k \hat{\mathcal{U}}_A(b_{i,k}, \hat{A}'_{i,k}, \hat{\Gamma}_i, o_j) \tag{3.3}$$

Si $\mathcal{A}_i$ est la meilleure réponse de la slice s_i sachant que $\forall i' \neq i$ l'ensemble $\mathcal{A}_{i'}$ est la meilleure réponse de la slice $s_{i'}$, l'ensemble $\{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_N\}$ représente l'équilibre de Nash.

Algorithmiquement, on utilise l'algorithme de *Best-Response*[15][33] pour chercher l'équité entre les différentes slices du routeur. Cet algorithme est basé sur le concept de la meilleure

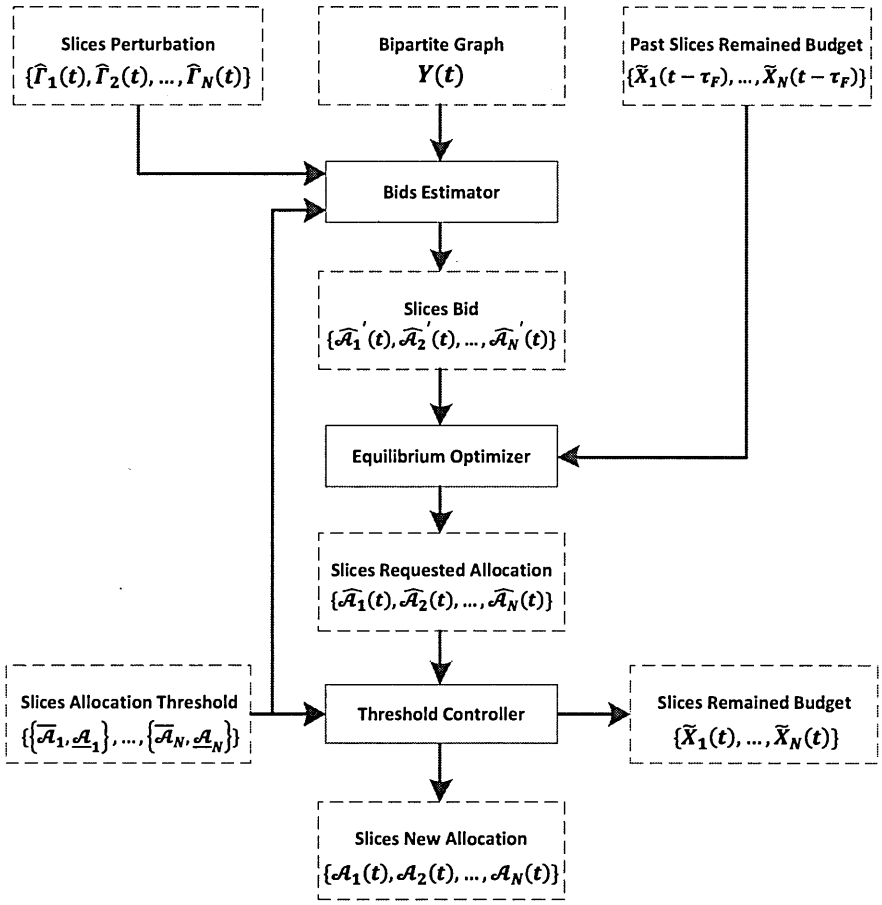

Figure 3.6: Processus algorithmique de calcul du vecteur d'allocation pour l'équité

réponse de l'équilibre de Nash pour déterminer la stratégie d'allocation qui garantit l'équité entre les slices. La figure 3.6 donne le processus qui détermine l'allocation de chaque slice. Ce modèle s'exécute en trois phases:

- *Bids Estimator*: Ce bloc permet d'estimer l'enchère de chaque slice. Le graphe biparti $Y(t)$ est donné à ce bloc pour indiquer l'association entre les slices et les *packages*. Ce bloc permet donc de déterminer l'ensemble d'enchères $\mathcal{A}'_i = \{\widehat{A}'_{i,1}, \widehat{A}'_{i,2}, ...\}$ qu'une slice s_i doit mettre sur son sous-ensemble de *packages*. En tenant compte de la perturbation sur le trafic et des contraintes de ressources, ce processus résout le problème d'optimisation

suivant pour chaque slice:

$$\begin{aligned} &\text{maximise} \quad \sum_k \widehat{\mathcal{U}}_A(b_{i,k}, \widehat{A}'_{i,k}, \widehat{\Gamma}_i, o_j) \\ &\text{sujet à} \quad \underline{A}_i \leqslant \widehat{A}'_{i,k} \leqslant \overline{A}_i \end{aligned} \tag{3.4}$$

- *Equilibrium Optimizer*: Ce bloc permet de déterminer l'allocation optimale permettant aux slices de garantir un équilibre de Nash dans le routeur. Après avoir déterminé les enchères $\widehat{A}'_{i,k}$ pour chaque slice, un prix Q_i se calcule pour chaque *package*. Le prix s'écrit comme suit:

$$Q_j = \sum_{i=1}^{N} \widehat{A}'_{i,b_j} \quad \text{avec} \quad B = \bigcup_{i=1}^{N} B_i = \{b_1, b_2, b_3 ...\} \tag{3.5}$$

$\widehat{A}'_{i,b_j}$ est l'enchère de la slice s_i sur le *package* b_j. Le prix d'un *package* est donc la sommation des enchères que chaque slice a mis sur ce *package*. Au début d'un cycle d'exécution τ_F, une slice s_i a un budget X_i qui représente la sommation de l'ensemble des enchères proposées par la slice plus le budget restant du cycle précédent. On écrit:

$$X_i(t) = \sum_k \widehat{A}'_{i,k}(t) + \widetilde{X}_i(t - \tau_F) \tag{3.6}$$

$\widetilde{X}_i(t)$ est le budget restant à la période précédente. L'algorithme Best-Response a pour objectif de résoudre le problème d'optimisation suivant:

$$\begin{aligned} &\text{maximise} \quad \sum_k \widehat{\mathcal{U}}_A(b_k, \widehat{A}'_{i,b_k}(t), \widehat{\Gamma}_i(t), o_j) = \sum_k \frac{\widehat{A}'_{i,b_k}(t)}{\widehat{A}'_{i,b_k}(t) + Q_k(t)} \\ &\text{sujet à} \quad \sum_k \widehat{A}'_{i,k}(t) = X_i(t) \end{aligned} \tag{3.7}$$

Résoudre ce problème permet de déterminer l'allocation requise $\mathcal{A}_i = \{\widehat{A}_{i,1}, \widehat{A}_{i,2}, ...\}$ pour une slice s_i. Établir l'équité entre les différentes slices ne signifie pas que les utilités des différentes slices sont parfaitement égales. Dans ce cas, des slices sont plus pénalisées en terme d'utilité à la fin de ce processus, c-à-d leurs budgets ont été consommés d'une façon non-optimale. Pour cette raison, ajouter le budget restant de la période précédente au budget courant d'une slice permet de la favoriser si elle a été pénalisée dans le cycle précédent.

- *Threshold Controller*: Ce bloc vérifie la validité de l'allocation proposée. Le contrôle est fait sur la base des contraintes de ressources indiquées sur le profil de chaque slice. Il permet de déterminer l'allocation finale et de calculer le budget restant pour chaque slice.

Le contrôle de la validité d'allocation se fait comme suit:

$$A_{i,j}(t) = \begin{cases} \widehat{A}_{i,j}(t) & \text{if } \underline{A}_i \leqslant \widehat{A}_{i,j}(t) \leqslant \overline{A}_i \\ \underline{A}_i & \text{if } \widehat{A}_{i,j}(t) < \underline{A}_i \\ \overline{A}_i & \text{if } \widehat{A}_{i,j}(t) > \overline{A}_i \end{cases} \tag{3.8}$$

Après avoir déterminé l'allocation valide pour chaque slice, le budget restant $\widetilde{X}_i(t)$ pour chaque slice peut donc être déterminé. Le budget restant est calculé comme suit:

$$\widetilde{X}_i(t) = X_i(t) - \sum_k A_{i,k}(t) \tag{3.9}$$

À la fin de ce processus, les seuils minimum et maximum sont transmis au niveau de stabilité. La distribution des allocation d'une slice sur les différents *linecards* du routeur se fait à travers la fonction ϕ.

3.2.4 Niveau-B: Niveau de stabilité

En se basant sur la théorie de contrôle, ce niveau supervise les performances de chaque slice pour ajuster son allocation. En ajustant dynamiquement les ressources de chaque slice, ce niveau permet de réduire les fluctuations de performances pour garantir un niveau de stabilité acceptable. Pour le reste de cette section, on considère la notation donnée dans le tableau 3.4 et le tableau 3.5.

L'objectif est de stabiliser l'utilité de ce niveau dans le temps. En fonction du trafic reçu, ce niveau doit ajuster dynamiquement l'allocation des ressources de chaque slice pour stabiliser son utilité. Au début de chaque période de contrôle τ_S, une nouvelle allocation est calculée pour corriger la variation de l'utilité d'une slice. Mathématiquement, on peut écrire l'équilibre stable de Lyapunov comme suit:

$$\forall \epsilon \ \ \forall t \quad || \ \mathcal{U}_B(s_i, u_j, \gamma_{i,j}(t), o_k) - \widetilde{\mathcal{U}}_B \ || < \epsilon \ \text{ avec } \ \Gamma_i = \sum_{j=1}^{V} \gamma_{i,j} \tag{3.10}$$

ceci veut dire qu'on cherche à garder l'utilité $\mathcal{U}_B$ d'une slice autour d'une utilité d'équilibre $\widetilde{\mathcal{U}}_B$. L'utilité d'équilibre est fonction des seuils de ressources que le niveau d'équité a passé à ce niveau. On note:

$$\overline{\mathcal{U}}_B(s_i, u_j) = \max_{\widehat{\gamma}_{i,j}} \widehat{\mathcal{U}}_B(c_{j,k}, \overline{G}_{i,j,k}, \widehat{\gamma}_{i,j}, o_l) \tag{3.11}$$

$$\underline{\mathcal{U}}_B(s_i, u_j) = \min_{\widehat{\gamma}_{i,j}} \widehat{\mathcal{U}}_B(c_{j,k}, \underline{G}_{i,j,k}, \widehat{\gamma}_{i,j}, o_l) \tag{3.12}$$

Tableau 3.4: Notations - Symboles du niveau de stabilité.

Symbole	Description
τ_S	Période de temps avant de recontrôler la stabilité
$C_i = \{c_{i,1}, c_{i,2}, c_{i,3}, ...\}$	Ensemble des *bundles* sur *linecard* u_j
$G_{i,j,k} = [g^1_{i,j,k}, g^2_{i,j,k}, g^3_{i,j,k}, ...]$	Vecteur des fractions de ressources allouées du *bundle* $c_{i,k}$ sur *linecard* u_j pour la slice s_i
$\widehat{G}_{i,j,k} = [\widehat{g}^1_{i,j,k}, \widehat{g}^2_{i,j,k}, \widehat{g}^3_{i,j,k}, ...]$	Vecteur des fractions de ressources demandées du *bundle* $c_{i,k}$ sur *linecard* u_j pour la slice s_i
$\underline{G}_{i,j,k}$	Seuil minimum d'allocation des ressources pour slice s_i sur *linecard* u_j
$\overline{G}_{i,j,k}$	Seuil maximum d'allocation des ressources pour slice s_i sur *linecard* u_j
$\gamma_{i,j}$	Nombre de paquets traités par la slice s_i sur *linecard* u_j
$\widehat{\gamma}_{i,j}$	Nombre de paquets estimés à traiter par la slice s_i sur *linecard* u_j

respectivement les seuils minimum et maximum d'utilité en fonction des seuils minimum et maximum $\underline{G}_{i,j,k}$ et $\overline{G}_{i,j,k}$ des ressources à allouer au slice s_i sur *linecard* u_j. Ces seuils sont fixes durant la période τ_F et ne peuvent changer qu'à la prochaine exécution du niveau d'équité. Franchir un de ces deux seuils envoie un événement de dépassement au niveau d'équité. Pour le point d'équilibre, on propose la valeur moyenne. Donc, on note le point d'équilibre comme suit:

$$\widetilde{\mathcal{U}}_B(s_i, u_j) = \frac{\overline{\mathcal{U}}_B(s_i, u_j) + \underline{\mathcal{U}}_B(s_i, u_j)}{2} \tag{3.13}$$

Comme le montre la figure 3.7, le facteur ϵ permet de définir deux espaces: espace de stabilité

Tableau 3.5: Notations - Fonctions du niveau de stabilité.

Fonction	Description
$\mathcal{U}_B : S \text{x} U \text{x} \mathbb{R} \text{x} O \rightarrow \mathbb{R}$	$\mathcal{U}_B(s_i, u_j, \gamma_{i,j}, o_k)$ Fonction d'utilité d'objectif o_k du slice s_i sur *linecard* u_j
$\widehat{\mathcal{U}}_B : C_i \text{x} \mathbb{R}^{D_j} \text{x} \mathbb{R} \text{x} O \rightarrow \mathbb{R}$	$\widehat{\mathcal{U}}_B(c_{j,k}, \widehat{G}_{i,j,k}, \widehat{\gamma}_{i,j}, o_l)$ Fonction d'utilité estimée d'objectif o_l du slice s_i sur *linecard* u_j

et espace d'instabilité. Si à un instant t la slice est dans l'espace d'instabilité, une re-allocation est nécessaire. Autrement, la slice garde la même allocation que la période précédente. Ces

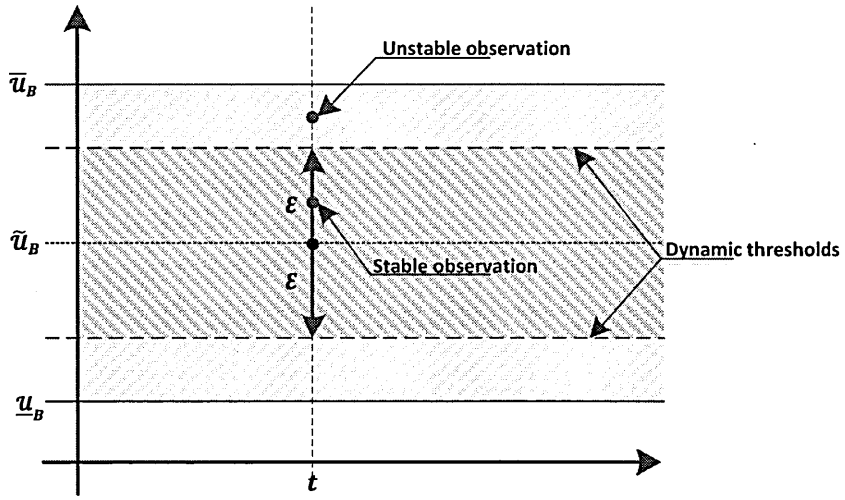

Figure 3.7: Espace de stabilité et d'instabilité autour d'un point d'équilibre.

deux espaces sont définis par un seuil minimum et un seuil maximum sur l'utilité. On note $\mathcal{U}_B^{min}(s_i, u_j, t)$ et $\mathcal{U}_B^{max}(s_i, u_j, t)$ respectivement le seuil minimum et maximum d'une slice s_i sur le *linecard* u_j à un instant t. Contrairement aux seuils $\underline{\mathcal{U}}_B$ et $\overline{\mathcal{U}}_B$ qui sont fixes pour ce niveau, $\mathcal{U}_B^{min}(s_i, u_j, t)$ et $\mathcal{U}_B^{max}(s_i, u_j, t)$ sont ajustables dynamiquement par ce niveau. On suppose que ces deux seuils sont symétriques autour du point d'équilibre et on note:

$$\mathcal{U}_B^{min}(s_i, u_j, t) = \widetilde{\mathcal{U}}_B(s_i, u_j) - \epsilon_{i,j}(t) \tag{3.14}$$

$$\mathcal{U}_B^{max}(s_i, u_j, t) = \widetilde{\mathcal{U}}_B(s_i, u_j) + \epsilon_{i,j}(t) \tag{3.15}$$

Ces seuils vont servir non pas uniquement comme détecteurs d'instabilité mais ils vont aussi permettre d'indiquer les seuils sur les ressources que le niveau de prévention de blocages recevra. Le facteur ϵ permet de définir la sensibilité à la stabilité de ce niveau. Ce facteur vérifie la condition suivante:

$$\forall t \in \mathbb{N} \quad 0 \leqslant \epsilon_{i,j}(t) \leqslant \overline{\mathcal{U}}_B(s_i, u_j) - \widetilde{\mathcal{U}}_B(s_i, u_j) \tag{3.16}$$

La valeur de ce facteur ϵ est proportionnelle à la sensibilité en terme de stabilité. Par exemple, si $[\forall t \ \ \epsilon(s_i, u_j, t) = 0]$, l'observation d'utilité doit être confondue avec l'utilité d'équilibre à chaque instant. Ce facteur doit être choisi de telle sorte à donner à une slice le temps de se stabiliser. Si ϵ est petit alors que la perturbation sur la slice est importante, la slice risque d'être plus instable. On doit donc avoir un facteur de sensibilité ϵ proportionnel à la perturbation. On peut alors noter:

$$\epsilon_{i,j} = f(\gamma_{i,j}) \tag{3.17}$$

f est une fonction croissante. Si la perturbation est grande, la sensibilité à la stabilité est moins importante et inversement.

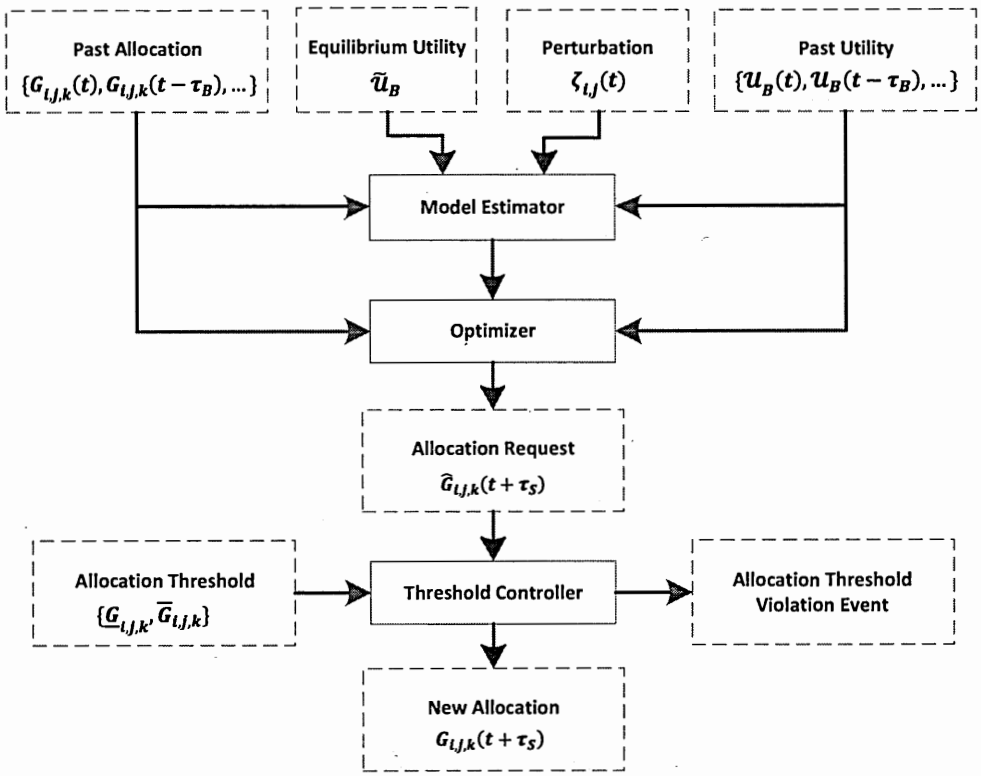

Figure 3.8: Processus algorithmique de calcul du vecteur d'allocation pour la stabilité.

Algorithmiquement, on utilise l'algorithme *Feedback*[38][48] pour stabiliser l'utilité d'une slice. Cet algorithme est basé sur un principe de réaction qui permet d'ajuster l'allocation des

ressources d'une slice en fonction de la mesure de performance et l'utilité d'équilibre. Notre modèle est décrit par la figure 3.8. Ce mécanise passe par trois phases:

- *Model Estimator*: Ce bloc permet de déterminer des paramètres d'estimation des ressources allouées et les mesures performance sur une slice. Le *modèles auto-régressifs et moyenne mobile*, ou *ARMA*, suivant permet de décrire le lien entre les allocations des ressources et les mesures de performance. Pour une slice s_i sur un *linecard* u_j, on a:

$$\begin{aligned} \breve{\mathcal{U}}_B(t) = a_1(t).\breve{\mathcal{U}}_B(t-\tau_S) + a_2(t).\breve{\mathcal{U}}_B(t-2.\tau_S) \\ + b_0^T(t).G(t) + b_1^T(t).G(t-\tau_S) \end{aligned} \tag{3.18}$$

  $\breve{\mathcal{U}}_B(t) = \frac{\mathcal{U}_B(t)}{\tilde{\mathcal{U}}_B - f(\gamma)}$ ou $\breve{\mathcal{U}}_B(t) = \frac{\mathcal{U}_B(t)}{\tilde{\mathcal{U}}_B + f(\gamma)}$ les utilité normalisées pour le calcul de l'allocation minimale et maximale. Ces utilités sont données par la perturbation maximale et minimale. Pour simplifier la notation, on a noté pour une slice s_i sur un *linecard* u_j:

$$\mathcal{U}_B(s_i, u_j, \gamma_{i,j}(t), o_k) \equiv \mathcal{U}_B(t) \ ; \ G_{i,j,k}(t) \equiv G(t) \ ; \ \tilde{\mathcal{U}}_B(s_i, u_j) \equiv \tilde{\mathcal{U}}_B \ ; \ \gamma_{i,j} \equiv \gamma$$

  Les paramètres a_1 et a_2 permettent de capturer la corrélation entre la mesure de performance récente et les deux anciennes mesures de performance. La corrélation entre la mesure de performance récente et les deux anciennes allocations est donnée par b_0 et b_1. Ce modèle de corrélation s'adapte à chaque période d'exécution de ce niveau, c-à-d les paramètres a_1, a_2, b_0 et b_1 ne sont pas fixes et changent dans chaque cycle d'exécution de la stabilité. Ces paramètres sont calculés pour l'allocation minimale et maximale en utilisant l'*algorithmique des moindres carrés récursifs*, ou filtre adaptatif *RLS* (voir Annexe A).

- *Optimizer*: Ce bloc détermine l'allocation des ressources qui permet à une slice d'atteindre son utilité cible. Pour déterminer l'allocation optimale, on cherche à minimiser la fonction suivante:

$$J = (\breve{\mathcal{U}}_B(t) - 1)^2 + q.||\widehat{G}(t) - G(t-\tau_S)||^2 \tag{3.19}$$

  Cette fonction représente deux coûts: le coût d'utilité $J_u = (\breve{\mathcal{U}}_B(t) - 1)^2$ et le coût de contrôle $J_c = ||\widehat{G}(t) - G(t-\tau_S)||^2$. Le coût d'utilité atteint son minimum pour $\breve{\mathcal{U}}_B(t) = 1$, c-à-d quand l'utilité atteint sa cible. Le paramètre q est un facteur de stabilité qui permet d'agir sur la convergence et la réactivité du mécanisme de stabilité. L'allocation des

ressources permettant de minimiser la fonction du coût J est donnée par:

$$\hat{G}(t) = (b_0.b_0^T + q.I)^{-1}.((1 - a_1(t).\breve{\mathcal{U}}_B(t - \tau_S) - a_2(t).\breve{\mathcal{U}}_B(t - 2.\tau_S) \\ - b_1^T(t).G(t - \tau_S)).b_0(t) + q.G(t - \tau_s)) \quad (3.20)$$

- *Threshold Controller*: Ce bloc permet de contrôler la validité de l'allocation proposée. Le contrôle se fait par rapport aux limites minimale et maximale d'allocation de ressources spécifiées par le niveau d'équité. Ce bloc détermine l'allocation finale à considérer tout en notifiant le niveau d'équité en cas de dépassement des seuils de ressources permis. Le contrôle se fait comme suit:

$$G(t) = \begin{cases} \hat{G}(t) & \text{if } \underline{G} \leqslant \hat{G}(t) \leqslant \overline{G} \\ \underline{G} & \text{if } \hat{G}(t) < \underline{G} \text{ avec évènement de violation} \\ \overline{G} & \text{if } \hat{G}(t) > \overline{G} \text{ avec évènement de violation} \end{cases} \quad (3.21)$$

À la fin de ce processus, les limites sur les ressources sont transmises au niveau de prévention de blocages pour qu'il en tienne compte dans son exécution.

3.2.5 Niveau-C: Niveau de détection de blocage

Afin d'éviter les situations de blocage, ce niveau effectue un contrôle dynamique sur les *packages* pour assurer une stratégie d'allocation non-bloquante sur chaque *package*. En se basant sur le concept de *contrôle de flux Max-Min*[3], on supervise chaque *package* séparément pour garantir que sa répartition ait un niveau d'équité de base entre les slices qui lui sont associées. Pour le reste de cette section, on considère la notation donnée dans le tableau 3.6 et le tableau 3.7.

L'objectif de ce niveau est de prévenir les goulots d'étranglement sur un *package*. Des slices associées au même *package* demandent une fraction de ce *package* en fonction du trafic entrant. Donc, au début de chaque période τ_B, ce niveau propose une nouvelle stratégie de partitionnement d'un *package* sur différentes slices afin de prévenir les blocages tout en tenant compte des contraintes de ressources définies par le niveau de stabilité.

Comme le montre la figure 3.9, les différentes slices demandent un vecteur d'allocation du même *package* dont la capacité est limitée. La capacité du *package* doit donc être distribuée sur les différentes slices selon la demande de chacune. Chaque slice demande un vecteur d'allocation

Tableau 3.6: Notations - Symboles du niveau de prévention de blocages.

Symbole	Description
τ_B	Période de temps avant de recontrôler le blocage
$Z_{i,j,k} = [z^1_{i,j,k}, z^2_{i,j,k}, z^3_{i,j,k}, ...]$	Vecteur des fractions de ressources allouées du *bundle* $c_{i,k}$ sur *linecard* u_j pour la slice s_i
$\widehat{Z}_{i,j,k} = [\widehat{z}^1_{i,j,k}, \widehat{z}^2_{i,j,k}, \widehat{z}^3_{i,j,k}, ...]$	Vecteur des fractions de ressources demandées du *bundle* $c_{i,k}$ sur *linecard* u_j pour la slice s_i
$\underline{Z}_{i,j,k}$	Seuil minimum d'allocation des ressources pour slice s_i sur *linecard* u_j
$\overline{Z}_{i,j,k}$	Seuil maximum d'allocation des ressources pour slice s_i sur *linecard* u_j

Tableau 3.7: Notations - Fonctions du niveau de prévention de blocages.

Fonction	Description
$\mathcal{U}_C : S \times U \times \mathbb{R} \times O \rightarrow \mathbb{R}$	$\mathcal{U}_C(s_i, u_j, \gamma_{i,j}, o_k)$ Fonction d'utilité d'objectif o_k du slice s_i sur *linecard* u_j
$\widehat{\mathcal{U}}_C : C_i \times \mathbb{R}^{D_j} \times \mathbb{R} \times O \rightarrow \mathbb{R}$	$\widehat{\mathcal{U}}_C(c_{j,k}, \widehat{Z}_{i,j,k}, \widehat{\gamma}_{i,j}, o_l)$ Fonction d'utilité estimée d'objectif o_l du slice s_i sur *linecard* u_j

qui s'obtient en optimisant le problème suivant:

$$\begin{aligned} &\text{maximize} \quad \widehat{\mathcal{U}}_C(c_{j,k}, \widehat{Z}_{i,j,k}, \widehat{\gamma}_{i,j}, o_l) \\ &\text{subject to} \quad \underline{Z}_{i,j,k} \leqslant \widehat{Z}_{i,j,k} \leqslant \overline{Z}_{i,j,k} \end{aligned} \tag{3.22}$$

On note $D_{j,k}$ la capacité du *package* $c_{j,k}$ du *linecard* u_j. Formellement, on dit qu'une stratégie d'allocation $\{Z_{1,j,k}, Z_{2,j,k}, ..., Z_{N,j,k}\}$ sur un *package* $c_{j,k}$ de capacité $D_{j,k}$ est *faisable* en terme d'utilité quand la condition suivante est vérifiée:

$$\sum_{i=1}^{N} \mathcal{U}_C(s_i, u_j, \gamma_{i,j}(t), o_k) \leqslant \widetilde{\mathcal{U}}_C(s_i, u_j, \sum_{i=1}^{N} \gamma_{i,j}(t), o_k) \tag{3.23}$$

$\widetilde{\mathcal{U}}_C$ est l'utilité maximale que l'on peut atteindre sur le *package* $c_{j,k}$ si toute sa capacité $D_{j,k}$ est utilisée. La stratégie de partition d'un *package* est non-bloquante quand les deux conditions suivantes sont vérifiées:

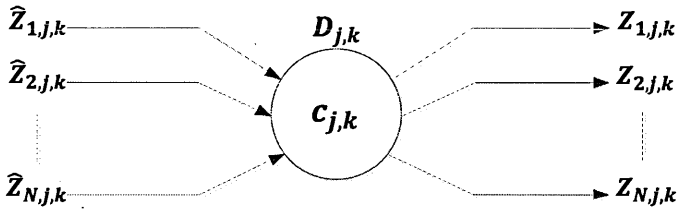

Figure 3.9: Répartition d'un *package* entre différentes slices.

- La stratégie $\{Z_{1,j,k}, Z_{2,j,k}, ..., Z_{N,j,k}\}$ est faisable.
- La stratégie $\{Z_{1,j,k}, Z_{2,j,k}, ..., Z_{N,j,k}\}$ perd sa faisabilité quand une composante $Z_{i,j,k}$ de la stratégie augmente.

On dit alors que la stratégie de partition est Max-Min *équitable.*

Algorithmiquement, on utilise l'algorithme *Max-Min*[9]. Cet algorithme se base sur un mécanisme de distribution itérative de ressource pour assurer une répartition Max-Min équitable (voir Annexe A). L'architecture de ce mécanisme est donnée par la figure 3.10. Trois phases donnent le processus d'exécution de ce niveau:

- *Request Estimator*: Ce bloc permet d'estimer l'allocation qu'une slice demande. En tenant compte du trafic, ce bloc permet de calculer l'allocation de base $\hat{Z}'_{i,j,k}$ en maximisant le problème suivant:

$$\text{maximize} \quad \hat{U}_C(c_{j,k}, \hat{Z}'_{i,j,k}(t), \hat{\gamma}_{i,j}(t), o_l) \tag{3.24}$$

  Ce problème est résolu pour les différentes slices, ce qui donnera une proposition d'une stratégie de partitionnement du *package* $c_{j,k}$.

- *Threshold Controller*: Ce bloc permet de contrôler la validité des requêtes d'allocation proposée. Le contrôle se fait par rapport aux limites minimale et maximale d'allocation de ressources spécifiées par le niveau de stabilité. Ce bloc détermine l'allocation finale en notifiant le niveau de stabilité en cas des dépassement de seuils de ressources permis. Le

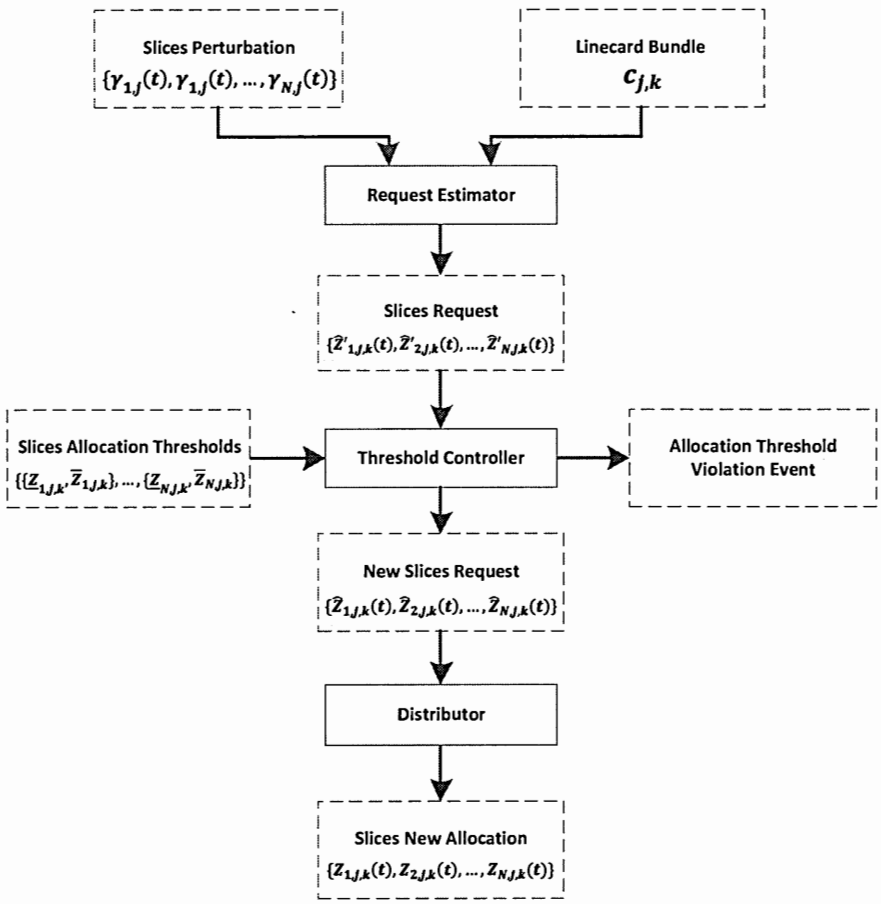

Figure 3.10: Processus de calcul du vecteur d'allocation pour la prévention de blocage.

contrôle se fait comme suit:

$$\hat{Z}_{i,j,k}(t) = \begin{cases} \hat{Z}'_{i,j,k}(t) & \text{if } \underline{Z}_{i,j,k} \leqslant \hat{Z}'_{i,j,k}(t) \leqslant \overline{Z}_{i,j,k} \\ \underline{Z}_{i,j,k} & \text{if } \hat{Z}'_{i,j,k}(t) < \underline{Z}_{i,j,k} \text{ avec évènement de violation} \\ \overline{Z}_{i,j,k} & \text{if } \hat{Z}'_{i,j,k}(t) > \overline{Z}_{i,j,k} \text{ avec évènement de violation} \end{cases} \tag{3.25}$$

- *Distributor*: Ce bloc exécute principalement l'algorithme Max-Min. Il applique un processus itératif permettant de distribuer la capacité d'un *package* $c_{j,k}$ sur l'ensemble des slices qui lui sont associées. Le nombre d'itération est égal au nombre de slices mappées sur le *package*. A la fin de l'algorithme, une stratégie $\{Z_{1,j,k}, Z_{2,j,k}, ..., Z_{N,j,k}\}$ Max-Min

équitable est générée. Cette stratégie vérifie la condition suivante:

$$\sum_{i=1}^{N} \widehat{U}_C(c_{j,k}, Z_{i,j,k}(t), \widehat{\gamma}_{i,j}(t), o_l) \leqslant \widehat{U}_C(c_{j,k}, D_{i,j,k}, \sum_{i=1}^{N} \widehat{\gamma}_{i,j}(t), o_l) \tag{3.26}$$

À la sortie de ce niveau, une stratégie d'allocation Max-Min équitable est générée pour les différents *packages* partagés entre les slices du routeur.

3.3 Conclusion

Mener le fondement théorique de notre mécanisme d'allocation à une démarche pratique nécessite l'utilisation d'algorithmes implémentables. Spécifier les blocs de notre mécanisme ainsi que les entrées/sorties de chacun de ces blocs est crucial pour assurer son déploiement sur une plateforme matérielle. Dans ce chapitre, nous avons présenté la formulation théorique de notre mécanisme d'allocation des ressources à trois niveaux. Pour chaque niveau, nous avons défini les entrées-sorties qui entrent en jeu avec l'architecture algorithmique à utiliser pour l'implémentation. Pour le premier niveau, nous avons utilisé l'algorithme *Best-Response* avec le concept des *packages* pour garantir l'équité entre les slices. Dans le deuxième niveau, nous avons utilisé l'algorithme *Feedback* avec des seuils dynamiques pour assurer la stabilité dans une slice. Et dans le dernier niveau, nous avons utilisé l'algorithme *Max-Min* pour prévenir une slice d'un blocage. Dans le chapitre (4), nous présenterons l'environnement d'expérimentation virtualisé que nous avons implémenté sur une plateforme matérielle. Nous discuterons de l'impact d'une telle démarche sur les performances.

CHAPITRE IV

IMPLÉMENTATION D'UN NŒUD VIRTUALISÉ SUR UNE PLATEFORME MATÉRIELLE

La formalisation de notre mécanisme d'allocation nous a permis de spécifier son architecture et les entrées/sorties de ses différents blocs. Avec les trois algorithmes proposés, notre mécanisme est implémentable et peut donc être déployé sur une plateforme matérielle. Dans ce chapitre, nous présenterons un ensemble d'implémentation que nous utiliserons pour évaluer notre mécanisme d'allocation. Nous utiliserons une implémentation réelle développée sur une plateforme de Netronome dont nous dériverons. Nous présenterons l'architecture de nos implémentations et nous illustrerons l'utilité de cette architecture pour notre mécanisme d'allocation de ressources basé sur les *bundles*. Finalement, nous fournirons une analyse de performance de nos implémentations avec une comparaison entre une approche virtualisée et non-virtualisée.

4.1 Banc de test

4.1.1 Processeur de flux réseau

Les processeurs réseau[17] sont une gamme de processeurs programmables utilisés dans les routeurs de troisième génération et conçus particulièrement pour les applications réseau. Ils utilisent une architecture multi-cores programmables permettant d'effectuer les différentes opérations réseau, à savoir l'extraction des entêtes de paquets, la classification et la modification des entêtes des paquets traités.

Dans notre implémentation, nous avons utilisé le NFP-3200[44] de Netronome. Il s'agit d'un processeur réseau spécialisé dans le traitement des flux de paquets réseau. Ce processeur offre un grand niveau de flexibilité avec une capacité de traitement de paquets pouvant aller

jusqu'à 40 Gbps. Comme le montre la figure 4.1, le NFP-3200 contient plusieurs cœurs de traitement de paquets avec des accélérateurs matériels et des interfaces I/O pour communiquer avec les composantes externes.

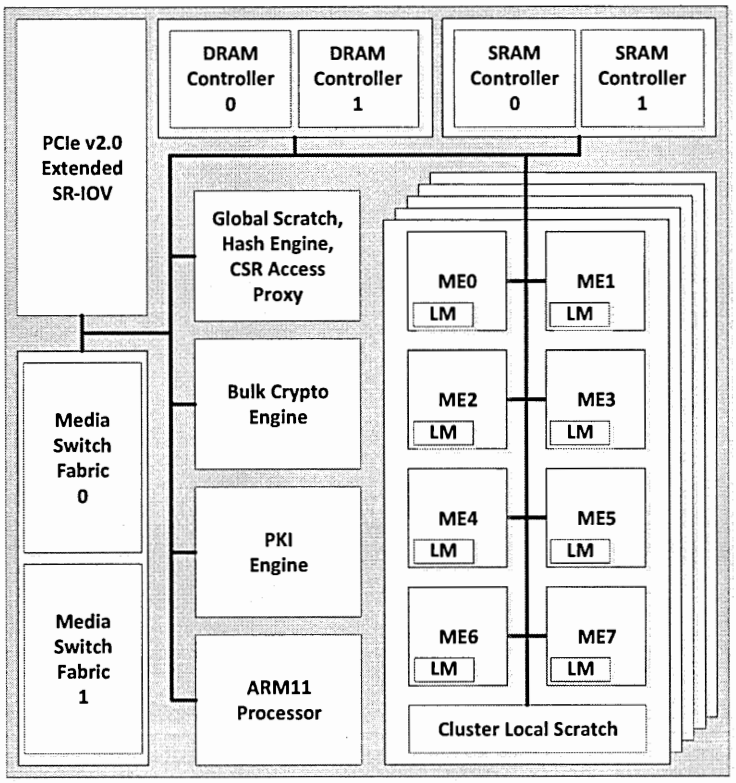

Figure 4.1: Architecture du processeur de flux NFP-3200 de Netronome.

Le NFP-3200 offre 40 cœurs de traitement de paquets (Micro-Engines) répartis uniformément sur 5 *clusters*. Chaque ME contient 8 *threads* avec 8K-mots de mémoire d'instruction. Cette architecture offre un grand niveau de performance grâce au *multi-threading*. Également, cette architecture permet d'implémenter des applications de traitement de paquets différentes sur différents cœurs, voire sur des différents threads, ce qui offre un grand niveau de flexibilité en terme de mappage opération/cœur, ou encore opération/thread.

Le NFP-3200 contient un processeur ARM11 intégré avec deux niveaux de caches (L1 de 32 KB et L2 de 256 KB) qui permet d'implémenter des processus de maintenance des tables

de recherche, de management et de gestion d'allocation de ressource dans notre contexte. Des accélérateurs intégrés dans le NFP-3200 permettent d'offrir des fonctionnalités de cryptage et d'identification pour les applications de sécurité, à savoir les systèmes de détection ou de protection des intrusions réseau. Dans notre démarche, nous avons limité l'utilisation des accélérateurs matériels sur l'engin de hachage pour les tables de hachage.

Plusieurs niveaux de mémoire sont offerts par le NFP-3200. Les supports mémoire sont distribués *on-chip* et *off-chip* avec des tailles et des vitesses d'accès différentes. Sur chaque ME, une mémoire locale (Local Memory) est disponible et accessible seulement par les threads de ce ME. Sur chaque *cluster*, une *scratch* locale (Cluster Local Scratch) est intégrée et est accessible uniquement par les MEs de ce *cluster*. Les CLS peuvent être utilisées pour des registres circulaires qui permettent d'échanger des données entre les MEs du même *cluster*. Pour communiquer entre des MEs sur des *clusters* differents, une *scratch* globale (Global Scratch) est disponible on-chip et est accessible pour tous les MEs. Les mémoires *on-chip* offrent une grande vitesse d'accès mais sont limitées en taille. Dans notre contexte, nous avons utilisé ces mémoires pour les compteurs de statistiques et les registres circulaires pour transmettre des messages entre les MEs. Pour les tables de recherche, nous avons utilisé des mémoires *off-chip*. Pour communiquer avec ces mémoires externes, le NFP-3200 offre des contrôleurs DDR3 et QDR pour interfacer respectivement avec la DRAM et la SRAM. Le NFP-3200 permet également d'interfacer avec la TCAM à travers le contrôleur QDR par le biais d'un FPGA.

Pour recevoir et/ou transmettre des paquets, le NFP-3200 a deux interfaces *Media Switch Fabric* qui offrent jusqu'à 20 Gbps de trafic bidirectionnel. Ces interfaces MSF peuvent être configurées en deux modes différents. Le premier mode est le mode *XAUI* qui offre deux interfaces de 10 Gbps chacune. Le deuxième mode est le mode *IKL* utilisant le standard *interlaken* qui offre jusqu'à 20 Gbps sur chaque MSF. Dans notre contexte, la plateforme Netronome fonctionne avec un mode XAUI. Également, le NFP-3200 offre une extension du plan de contrôle et de management en permettant l'interfaçage avec des systèmes x86 à travers une interface PCIv2.0. Ce standard de communication permet de faciliter le développement des applications de contrôle et de management, et offre une large bande passante de 40 Gbps entre le NFP-3200 et un serveur d'architecture x86. Cette caractéristique est très utile dans notre contexte pour pouvoir développer un mécanisme performant d'allocation des ressources.

4.1.2 Architecture de test

Pour évaluer notre implémentation, nous avons mis en place une topologie de test comme le montre la figure 4.2. La topologie de test comprend principalement deux blocs: le serveur Netronome[44] et le générateur de trafic Blaster[5].

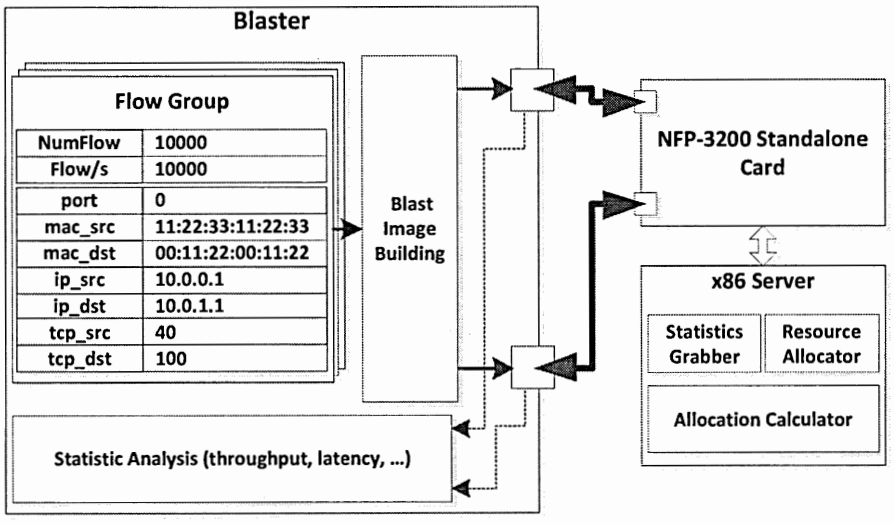

Figure 4.2: Architecture de test avec le Blaster.

4.1.2.1 Serveur Netronome

Ce bloc (*x86 server + NFP-3200 standalone card*) contient l'implémentation matérielle de notre nœud virtualisé avec les plans de gestion des ressources entre les différentes instances virtuelles. Il est composé de deux éléments:

- Le premier élément est la carte Netronome. Cette carte contient principalement le processeur de flux réseau NFP-3200 pour le traitement des paquets, un système mémoire composé des mémoires SRAM, DRAM et TCAM afin de stocker les paquets et les tables de recherches, deux interfaces XAUI de 10 Gbps chacune pour la réception et la transmission des paquets et une interface PCIe v2.0 pour communiquer avec un système x86. Cet élément implémente le code matériel (assembleur du plan de données) de notre nœud virtualisé.

- Le deuxième élément est un serveur x86. Ce serveur contient une fente PCIe v2.0 contenant la carte Netronome qui lui permet de communiquer avec le plan de données (datapath) en envoyant des nouvelles configurations ou en recevant des statistiques. Cet élément contient également les composants de management, à savoir *Statistics Grabber* pour récupérer les statistiques des différents slices, *Resource Allocator* pour appliquer les nouvelles configurations et *Allocation Calculator* contenant les différents algorithmes d'allocation de ressources. Ces composants de management s'exécutent périodiquement ou suite à un événement pour reconfigurer le nœud virtualisé.

Contrairement à d'autres plateformes réseau qui utilisent des processeurs de management de faible performance, la solution Netronome offre une interface PCIe v2.0 pouvant offrir un débit de 40 Gbps et permettant de connecter la carte à des systèmes hétérogènes plus flexibles et plus performants. Cette caractéristique est très importante pour pouvoir exécuter des algorithmes complexes pour plusieurs instances virtuelles en un temps réduit. Ceci est très important pour améliorer la réactivité de notre mécanisme d'allocation au changement du trafic sur le nœud virtualisé.

4.1.2.2 Générateur de trafic Blaster

Ce bloc permet de générer des flux de paquets pour les différentes instances virtuelles de notre nœud. Le Blaster est conçu pour rouler sur une carte PCIe standard qui utilise un processeur de flux de Netronome. Cette solution peut générer théoriquement jusqu'à $\approx$8 Gbps de trafic sur deux interfaces XAUI de 10 Gbps chacune. Le Blaster donne la possibilité de créer plusieurs groupes de flux en se basant sur des fichiers de captures des paquets (fichiers .pcap) et les synthétiser avant les générer sur les ports réseau. Après avoir spécifié un fichier pcap pour un groupe de flux, le Blaster permet d'associer un profil pour chaque groupe. Un profil de groupe de flux a les paramètres suivants:

- *Fichier pcap*: Il spécifie le fichier de capture des paquets qui représente la référence du trafic à utiliser pour le système sous test. Son flux de paquets capturés est copié sur l'interface de sortie du Blaster.
- *Nombre de flux et de flux/seconde*: Il spécifie la longueur et la vitesse d'émission du groupe de flux. Le nombre de flux détermine le nombre de fois que le flux (la capture des paquets) sera répété. Le nombre de flux par seconde spécifie la vitesse avec laquelle un flux sera

envoyé.

- *Adresses MAC*: Il spécifie l'adresse MAC source et destination qui seront données aux différents paquets du flux. Ce paramètre est optionnel; s'il n'est pas spécifié, les paquets gardent leurs adresses MAC.
- *Adresses IP et ports TCP*: Il spécifie l'adresse IP source et destination avec le numéro de port TCP source et destination qui seront associés aux paquets du flux. Ce paramètre est optionnel.
- *Port de sortie*: Il spécifie le port sur lequel le groupe de flux sera envoyé.

La possibilité d'avoir des profils de flux nous permet de générer des trafics qui correspondent aux besoins de nos implémentations. Chaque instance virtuelle supporte un ensemble de circuits qui traitent un type spécifique de paquet. Par exemple, si on a une instance avec un circuit de routage IPv4 niveau trois, on peut spécifier une capture de paquets IPv4 pour pouvoir tester notre circuit.

Après avoir spécifié un ensemble de groupe de flux, le Blaster combine les différents groupes de flux dans une image *blast*, qui sera utilisée pour générer le trafic sur les interfaces de sortie. Quand un blast se termine, c-à-d que tous les paquets sont envoyés, des paquets sont reçus sur les interfaces du Blaster du système sous test. Le Blaster offre des moyens d'analyse de performance depuis les statistiques récupérées des paquets reçus. Cette caractéristique d'analyse est très intéressante pour faciliter le processus d'évaluation de performance. Cependant, nous n'avons pas utilisé cette propriété dans nos évaluations. Dans notre contexte, notre système contient un ensemble de sous-systèmes séparés (instances virtuelles) et nous avons besoin de les évaluer séparément. Nous avons utilisé des compteurs de statistiques dans nos implémentations qui nous ont permis d'évaluer la performance des différents sous-systèmes et ont servi comme points de supervision pour notre mécanisme d'allocation.

4.2 Implémentations

4.2.1 Aperçue général

Pour valider notre mécanisme d'allocation, nous avons développé un ensemble d'implémentations sur le système NFP-3200. Les implémentations doivent fournir un ensemble de slices

et de circuits dont nous pouvons gérer les ressources.

La figure 4.3 montre la partie de l'implémentation en charge de la gestion des ressources entre les slices. Cette partie permet de superviser les slices à travers des statistiques et d'ajuster l'allocation via des configurations si nécessaire. Elle contient trois blocs principaux:

- *Statistics Grabber*: Il permet de récupérer les statistiques des slices. Ce bloc a accès aux blocs mémoire qui contiennent les compteurs de statistiques. Il peut récupérer des statistiques par circuit (*Per Circuit Statistics Grabber*) ou par slice (*Per Slice Statistics Grabber*). Les statistiques sont consultées périodiquement et enregistrées dans des fichiers cycliques pour chaque slice.

- *Allocation Calculator*: Il contient les algorithmes qui constituent notre mécanisme d'allocation. Il contient trois blocs pour le calcul d'équité (*BestResponse*), de la stabilite (*Feed-Back*) et de la détection du blocage (*MaxMin*). Chaque slice a son propre contrôleur lié à son profil. Il calcule l'utilité à travers les statistiques et envoie un signal de déclenchement à ce bloc quand un seuil est franchi afin de calculer une nouvelle allocation. Une nouvelle allocation est donc calculée pour chaque *package* de ressources (*bundle*).

- *Resource Allocator*: Il permet d'appliquer l'allocation au plan de données et de rendre une description abstraite d'allocation à une configuration matérielle qui reflète cette description. Ce bloc contient principalement des outils développés par la bibliothèque de Netronome (NFP API) qui permettent la configuration des threads (pour la puissance du traitement) et des mémoires de recherche (pour la capacité en terme d'entrées de recherche).

La figure 4.4 montre la partie de l'implémentation en charge du traitement du trafic réseau. Cette partie contient des instances virtuelles (slices) implémentées de telle sorte que leurs ressources peuvent être ajustées. L'implémentation contient essentiellement un ensemble de *packages* de ressources dont chacun représente les ressources d'un circuit. Les slices peuvent avoir une fraction de ressources d'un ou plusieurs *packages* selon ses besoins, son profil et les besoins des autres slices. Pour diriger un flux de paquets vers son slice, une couche d'isolation est mise en place pour permettre de distribuer le trafic sur les slices. Des compteurs matériels de 32-bits sont placés sur les *bundles* et les slices permettant de mettre à jour les blocs de statistiques (*Per Circuit Statistics Block* et *Per Slice Statistics Block*). Des blocs mémoires

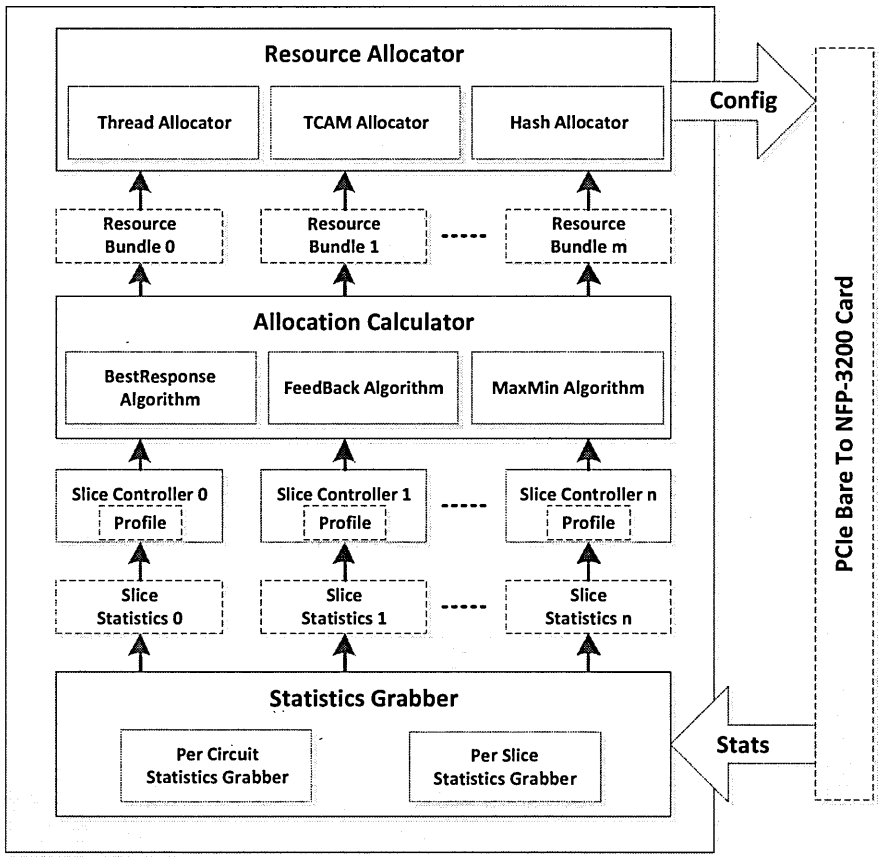

Figure 4.3: Aperçu général de l'implémentation - gestion d'allocation.

sont aussi dédiés pour stocker la configuration des slices (*Thread Configuration Block*, *TCAM Configuration Block* et *Hash Configuration Block*). Chaque fois que le plan de données est partiellement ou complètement arrêté, ces blocs de configuration sont consultés pour recharger la configuration.

4.2.2 Chemins de traitement et packages de ressources

Dans nos implémentations, nous avons utilisé trois types de *bundles* qui correspondent à trois chemins de traitement différents. Chaque chemin est caractérisé par un ensemble de tâches à exécuter et un ensemble de ressources à utiliser.

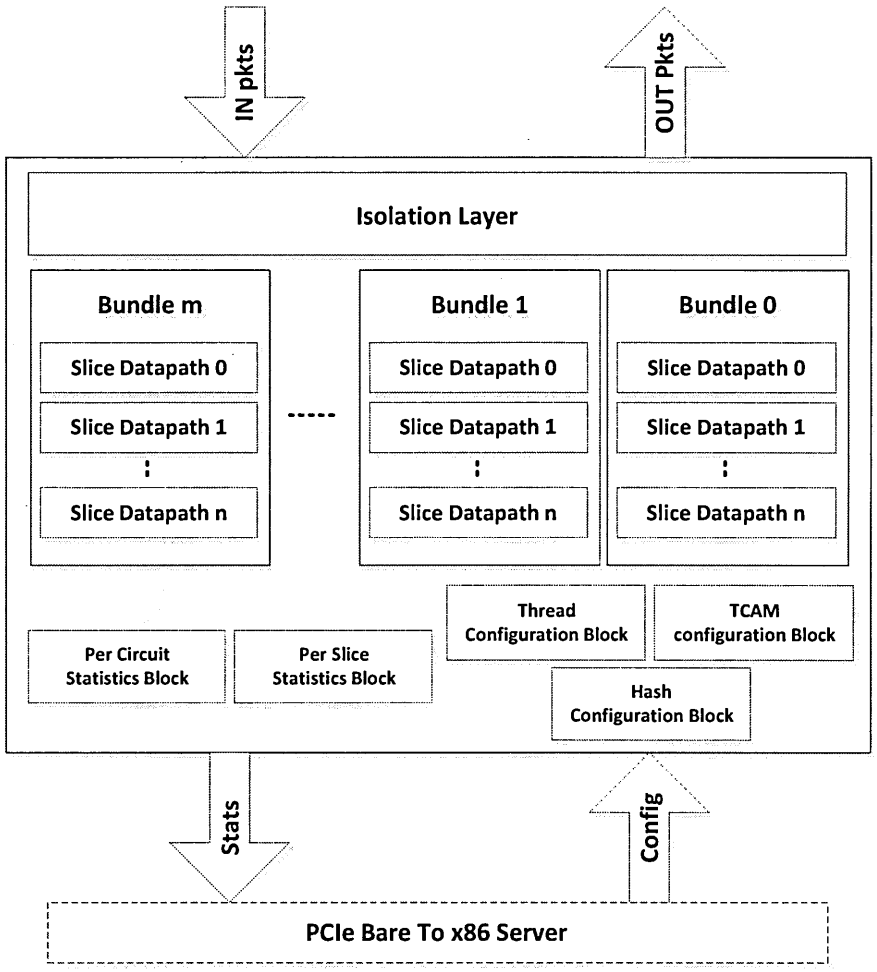

Figure 4.4: Aperçu général de l'implémentation - plan de données.

Notre premier *package* de ressources correspond à un chemin de traitement OpenFlow[54]. Comme le montre la figure 4.5, ce chemin de traitement utilise trois engins de traitement de paquets. Le *OF-Header Parse Engine* permet l'extraction de 14 champs du paquet reçu constituant le descripteurs OpenFlow. Cet engin nécessite plusieurs accès au buffer des paquets sur la DRAM. Le *OF-Table Search Engine* effectue la tâche de classification d'un paquet via une table de flux. Il utilise le descripteur extrait pour effectuer une première recherche dans une table TCAM. En cas de match (correspondance trouvée), une deuxième recherche dans une table à

accès direct est effectuée pour déterminer les actions à appliquer sur le paquet. Le *OF Forwarding Engine* applique les actions indiquées par le résultat de la classification sur le paquet. Nous notons p_{of} et b_{of} respectivement le circuit et le *bundle* de ce chemin de traitement.

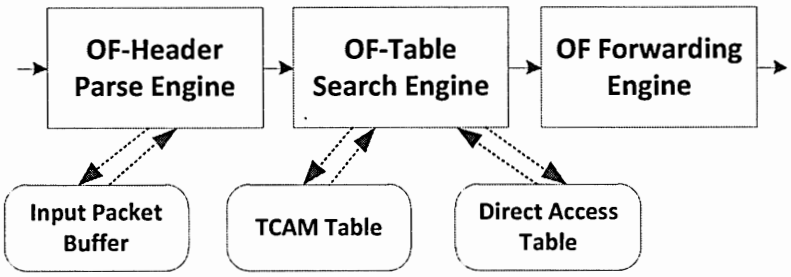

Figure 4.5: Chemin de traitement OpenFlow.

Le deuxième *package* de ressources fonctionne avec un chemin de traitement pour le routage niveau 3. Trois engins de traitement sont utilisés, comme le montre la figure 4.6. Le *L3-Header Parse Engine* permet l'extraction de l'adresse IP destination après avoir vérifié que le TTL du paquet reçu n'est pas expiré. Le *L3-Table Search Engine* permet de déterminer le port de sortie du paquet via une table de routage. Deux recherches successives dans une table TCAM et une table à accès direct permettent de déterminer le port de sortie du paquet. Le *L3 Forwarding Engine* effectue une recherche dans la table à accès direct pour déterminer les nouvelles adresses MACs du paquet avant de le modifier et le mettre sur le port correspondant. Nous notons p_{l3} et b_{l3} respectivement le circuit et le *bundle* de ce chemin de traitement.

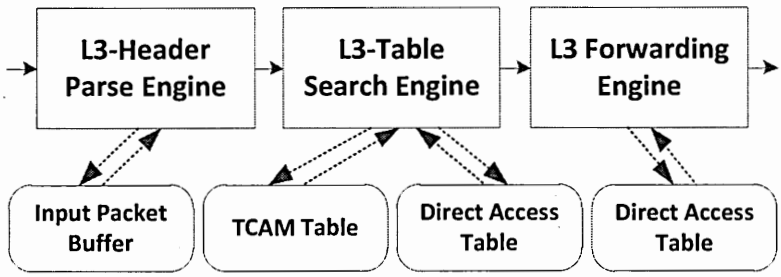

Figure 4.6: Chemin de traitement L3.

Le troisième *package* de ressources donne un chemin de traitement pour la commutation niveau 2. Il contient également trois engins de traitement comme le montre la figure 4.7. Le *L2-Header Parse Engine* extrait l'adresse MAC destination du paquet en accédant au buffer des paquets. Le *L2-Table Search Engine* effectue une recherche permettant de déterminer l'action à appliquer sur le paquet. La recherche se fait sur une table de hachage sur la SRAM. Le *L2 Forwarding Engine* utilise le résultat de recherche pour appliquer une ou plusieurs actions sur le paquet. Nous notons p_{l2} et b_{l2} respectivement le circuit et le *bundle* de ce chemin de traitement.

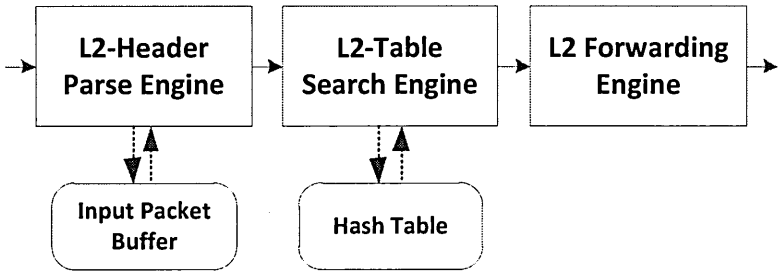

Figure 4.7: Chemin de traitement L2.

Choisir des chemins de traitement, qui diffèrent en termes de type d'opérations et de type de ressources, nous permet d'avoir des *bundles* de prix différents. Au moment d'une enchère, par exemple, une slice doit ajuster son enchère sur un en fonction de ses besoins et du prix de ce *bundle*. Ceci permet de donner un graphe biparti dont les objets (l'ensemble d'arrivée) sont différents.

4.2.3 Niveaux d'isolation

Dans nos implémentations, les deux interfaces physiques de la plateforme NFP-3200 sont partagées entre les différentes slices. Partager ces interfaces implique que les paquets correspondants à toutes les slices arrivent de la même source, d'où le besoin d'avoir une couche d'isolation entre les ports physiques et les chemins de traitement des paquets.

Quand un paquet arrive sur le port physique (MSF0 ou MSF1), il est mis dans le buffer des paquets dans la DRAM et un descripteur qui lui est associé est mis dans un registre circulaire (*physical Port* ou *pP*). Notre couche d'isolation effectue une pré-classification pour déplacer le

descripteur de paquet vers un autre registre circulaire. Comme le montre la figure 4.8, nous avons proposé deux stratégies d'isolation:

- *Un niveau d'isolation*: Cette stratégie d'isolation (à droit) permet d'offrir une isolation par slice. Pour chaque slice, un port virtuel *vP* (*virtual Port*) lui est associée contenant ses paquets. Les circuits de slice partagent donc le port virtuel *vP*. La pré-classification se fait sur l'adresse MAC source des paquets dans le port physique *pP*. La répartition des paquets sur les circuits est effectuée par la slice.

- *Deux niveaux d'isolation*: Cette stratégie d'isolation (à gauche) donne une isolation par circuit. Les circuits d'une slice ont des ports circuit *cP* (*circuit Port*) séparés. Dans le premier niveau, une pré-classification est effectuée sur une partie de l'adresse MAC source (2-octets) des paquets dans le port physique *pP*. Cette phase décide le port virtuel *vP* d'un paquet. La deuxième phase effectue une autre pré-classification sur l'autre partie de l'adresse MAC destination (4-octets) pour déterminer le port circuit *cP* d'un paquet.

Le premier niveau d'isolation est généré par un seul engin, tandis que le deuxième est implémenté sur plusieurs engins. Le nombre d'engins du deuxième niveau est égal au nombre de ports virtuels. Chaque engin est lié à un port virtuel, ce qui permet de mettre les circuits dans des regroupements par slice avec une isolation plus forte. Également, la nécessité d'avoir un pré-classificateur dans notre contexte est due au nombre limité d'interfaces physiques de la carte NFP-3200 (deux ports). Avec plusieurs instances virtuelles qui partagent un nombre limité d'interfaces physiques, l'utilisation d'un pré-classificateur permet de filtrer et distribuer les paquets reçu sur les ports virtuels de chaque instance. Par conséquent, une instance virtuelle peut avoir plusieurs ports virtuels au lieu d'être limité par le nombre réduit des interfaces physiques.

4.3 Analyse de performance

Nous présenterons les résultats d'évaluation de nos implémentations qui utilisent trois slices sur trois *packages* de ressources. Ces résultats illustrent le coût de la virtualisation. Le tableau 4.1 donne les composants de nos implémentations suivant la notation du chapitre (3).

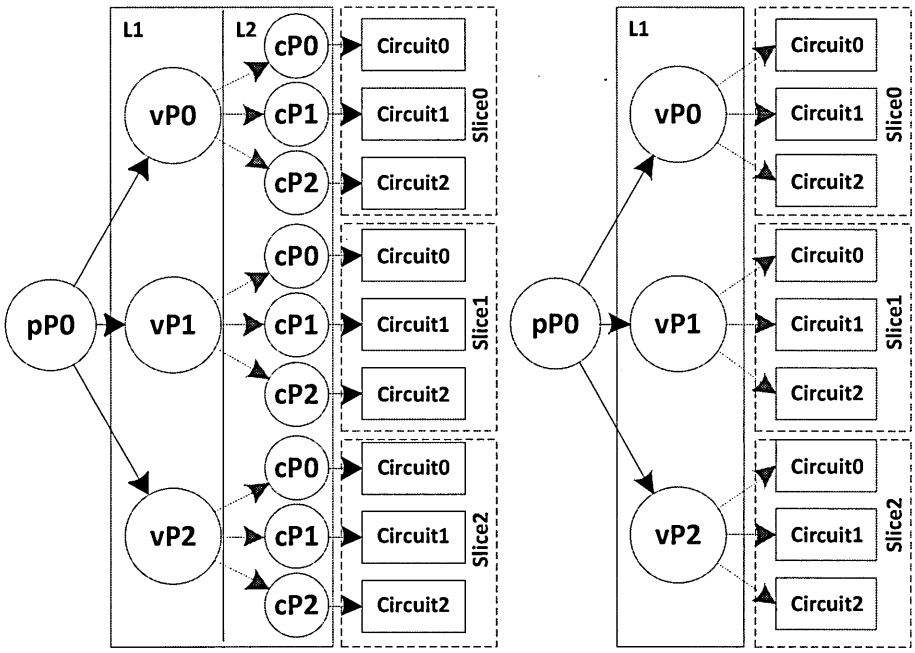

Figure 4.8: Stratégies d'isolation du trafic par circuit et par slice.

4.3.1 Performance avec et sans virtualisation

Introduire la virtualisation dans une plateforme informatique vient souvent avec des coûts additionnels. Pour un regroupement fixe de ressources, la virtualisation apporte une réduction de performance que ce regroupement peut offrir sans virtualisation. Mettre en place de la virtualisation rend l'existence des compteurs de statistiques, les points d'entrée de configuration et l'isolation nécessaires.

Le tableau 4.2 donne une comparaison de performances entre un circuit (OpenFlow, routage niveau 3 et commutation niveau 2) dans un environnement virtualisé et non-virtualisé. Pour pouvoir faire la comparaison, nous avons utilisé un seul cœur de traitement dans les deux contextes. Nous remarquons qu'il y a un rapport de $\approx$50 % entre une approche virtualisée et non-virtualisée. Malgré cette différence, ceci n'affecte pas le gain que nous pouvons avoir à travers la virtualisation:

Tableau 4.1: Notations

$N = 3$	Nombre de slices
$M_0 = M_1 = M_2 = 3$	Nombre de circuits par slice
$V = 1$	Nombre de linecards
$U = \{u_{nfp3200}\}$	Linecard NFP-3200
$B = \{b_{of}, b_{l3}, b_{l2}\}$	*packages* de ressources
$P = \{p_{of}, p_{l3}, p_{l2}\}$	Circuits de traitement de paquets

Tableau 4.2: Performance avec et sans virtualisation

Mode	**OpenFlow**	**L3**	**L2**
Sans virtualisation	2.5 Gbps	3.6 Gbps	5.71 Gbps
Avec virtualisation	1.15 Gbps	1.64 Gbps	2.59 Gbps

- Dans l'implémentation non-virtualisée, le circuit occupe toutes les ressources disponibles. Ceci offre une capacité de traitement qui dépasse la capacité de trafic que la plateforme peut supporter. Des ressources sont donc perdues.
- Dans l'implémentation virtualisée, le circuit coexiste avec des circuits d'autres slices. Le coût supplémentaire de cette démarche peut être compensé à travers les ressources qui sont généralement inutilisées. Ceci permet d'avoir un contexte virtualisé dont le niveau de performance de ses instances peut s'approcher considérablement celui d'un environnement non-virtualisé.

4.3.2 Coût de la configuration

Dans nos implémentations, nous avons considéré les cœurs de traitement de paquets, la TCAM et la Hash comme ressources à reconfigurer. Ces ressources permettent d'ajuster la performance d'une slice en termes de nombre de paquets à traiter par seconde et de nombre de descripteurs de paquets à traiter. Le tableau 4.3 présente le temps nécessaire pour ajuster la configuration en fonction de différents types de ressources. Ces chiffres ont été estimés en

Tableau 4.3: Temps de configuration pour différentes ressources

Ressource	Temps de configuration (ms)
Thread	19.86
1k entrées TCAM	494.78
10k entrées TCAM	3500
1k entrées Hash	180.80
10k entrées Hash	916.91

observant le taux de pertes de paquets sous un débit élevé quand une configuration est appliquée.

Configurer les threads d'un cœur nécessite tout d'abord de l'arrêter. Avec un temps de configuration de $\approx$ 19 ms, une perte de $\approx$ 15000 paquets peut se produire sous un debit de $\approx$ 0.5 Mpps. Ceci peut affecter la performance de notre système sous de grands débits variables. Pour masquer cet effet, nous avons ajusté nos implémentations pour qu'elles appliquent l'allocation avec un mécanisme qui ressemble au ISSU[55] (In-Service Software Upgrade). Au lieu d'arrêter tous les cœurs au moment de la configuration, nous avons fait l'allocation sur deux étages avec deux groupes de cœurs. Pour appliquer la configuration, nous arrêtons le premier groupe pour ajuster l'allocation tandis que le deuxième groupe continue de traiter les paquets. Ceci permet de minimiser le taux de perte des paquets durant la phase de reconfiguration.

La configuration des tables de recherches consiste à ajuster le mappage entre les slices/-circuits et des régions mémoire. Changer un mappage mémoire consiste à déplacer l'ensemble des entrées d'une table d'un offset mémoire à un autre. Ceci est réalisé à travers plusieurs écritures mémoire. Comme le montre le tableau 4.3, le temps de configuration de la TCAM ou de la Hash augmente avec le nombre d'entrées à écrire (494 ms et 3500 ms pour 1000 et 10000 entrées TCAM). Avec notre plateforme contenant une TCAM et une SRAM, nous ne pouvons pas masquer l'impact de cette configuration en utilisant le même mécanisme que précédemment (ISSU).

Mode d'isolation	Cycles d'horloge	Temps de configuration (μs)
L1	531	0.442
L2	669	0.574

Tableau 4.4: Temps de configuration pour différentes ressources

4.3.3 Coût de pré-classification

Avoir une couche de pré-classification est nécessaire pour isoler le trafic des différentes slices. En ajoutant de la pré-classification, chaque paquet doit être analysé en extrayant et classifiant son adresse MAC. Ceci coûte des cycles d'horloge supplémentaires qui s'ajoutent au coût de traitement d'un paquet.

Le tableau 4.4 montre le coût en terme de cycles d'horloge de chaque niveau d'isolation. Avec un horloge de 1.2 GHz (cycle d'horloge de 0.833 ns), un niveau d'isolation ajoute une latence de 0.833 ns $\times$ 531 cycles = 0.442 μs et double avec deux niveaux d'isolation. Tenant compte de cette évaluation, les buffers de paquets doivent être dimensionnés correctement pour éviter les pertes. En augmentant le niveau d'isolation, les buffers doivent être plus larges pour supporter plus de descripteurs de paquets.

4.3.4 Coût de compteurs de statistique

Contrairement aux approches non-virtualisées dont les statistiques peuvent être optionnelles, avoir des compteurs matériels de statistiques dans un environnement virtualisé est nécessaire. Ceci rend le coût de supervision des slices inévitable. Dans nos implémentations, nous avons 14 compteurs par slice de trois circuits dont:

- un compteur de paquets entrants par circuit;
- un compteur de paquets matchés par circuit;
- un compteur de paquets non-matchés par circuit;
- un compteur de paquets sortants par circuit;
- un compteur de paquets entrants par slice;

- un compteur de paquets rejetés pour congestion par slice.

NFP-3200 offre le moyen d'effectuer les incrémentations des compteurs atomiquement (pas d'accusé d'écriture). Ce mécanisme optimise le coût de supervision à 12 cycles par compteur, ce qui permet d'atteindre $12_{cycle\ par\ compteur} \times 14_{compteur} = 168$ cycles par paquets ($0.140\mu s$) dans le pire des cas. Par conséquent, ce coût augmente proportionnellement avec le nombre de slices et de circuits qu'une implémentation contient.

4.4 Conclusion

Afin de tester notre mécanisme d'allocation, avoir un nœud réseau virtualisé est une nécessité. Ce nœud virtualisé doit offrir les outils de configuration, de programmation et de supervision avec un ensemble de *bundles* pour pouvoir faire fonctionner notre processus d'allocation. Puisque l'implémentation de ce nœud est faite sur une plateforme matérielle, une connaissance approfondie des composants et l'environnement logiciel de cette plateforme est importante pour pouvoir mapper les fonctionnalités de notre mécanisme sur le nœud virtualisé. Dans ce chapitre, nous avons illustré l'architecture des implémentations que nous avons fourni pour l'évaluation de notre mécanisme d'allocation. Nous avons présenté la plateforme réseau NFP-3200 de Netronome et comment nous l'avons utilisé pour développer des moyens d'allocation de ressources. Nous avons montré que la virtualisation vient avec un coût supplémentaire qu'il faut considérer pour bien dimensionner un système réseau. Dans le chapitre (5), nous évaluerons l'efficacité notre mécanisme d'allocation en utilisant les implémentations que nous avons présenté dans ce chapitre. Nous conclurons sur l'effet d'avoir un tel mécanisme déployé sur une plateforme matérielle pour sortir avec des recommandations.

CHAPITRE V

ÉVALUATIONS ET ANALYSES DU MÉCANISME D'ALLOCATION DE RESSOURCES

Après avoir déployé notre mécanisme d'allocation sur notre implémentation de nœud virtualisé, il est important d'évaluer le comportement de ses différents niveaux, afin d'obtenir des conclusions sur nos expérimentations. Dans ce chapitre, nous présenterons une évaluation de notre mécanisme d'allocation de ressources telle que présentée dans le chapitre (3). Nous utiliserons les implémentations du chapitre (4) pour fournir une analyse d'efficacité de notre approche d'allocation. Nous illustrerons le plan d'expérimentations ainsi que les paramètres à observer pour l'évaluation. Nous discuterons à la fin des résultats de nos tests pour fournir des recommandations sur les contextes d'utilisation de notre approche.

5.1 Contexte d'expérimentations

Dans nos évaluations, nous cherchons à acquérir des connaissances sur le comportement de notre mécanisme d'allocation dans des contextes particuliers et sur une implémentation matérielle. Nous avons choisi un contexte avec un haut niveau d'isolation (isolation par circuit) pour pouvoir faire une évaluation plus fine et observer l'effet d'une perturbation de trafic par circuit sur une slice. Du moment que nous n'avons pas intérêt d'évaluer l'effet de la priorité entre les slices, nous avons utilisé trois slices identiques de même priorité. Donc, nous utiliserons une implémentation paramétrée comme suit:

- Stratégie d'isolation à deux niveaux;
- Trois slices identiques;
- Trois circuits par slice (circuit OF, circuit L3 et circuit L2);

- Pas de priorité entre les slices ou les circuits des slices durant l'allocation.

Avant de tester notre implémentation, des configurations initiales sont associées à chaque bloc de notre mécanisme d'allocation suivant l'objectif de l'expérimentation. À travers un trafic réseau personnalisé et généré par le Blaster, nous cherchons à évaluer quelques caractéristiques de notre mécanisme d'allocation:

- Temps de convergence: il caractérise la durée minimale pour retourner à l'état stable après une perturbation du trafic. Cette évaluation permet de savoir le niveau de perturbation à permettre pour éviter la divergence de notre système.
- Sensibilité aux perturbations: elle caractérise la capacité d'un changement de trafic à perturber le système. Cette évaluation montre le rapport entre la valeur de la perturbation et la convergence du système.
- Intervalle de stabilité: il caractérise la différence en les seuils minimum et maximum sur la fonction d'utilité contrôlée. Cette évaluation permet de déduire l'influence de la largeur de cette intervalle sur la stabilité et la convergence du système.
- Période d'exécution: elle caractérise le temps minimum pour re-déclencher l'exécution d'un algorithme. Cette évaluation permet de mettre en évidence l'impact de cette période sur le temps de convergence vers l'intervalle de stabilité.
- Ratio d'équité: il caractérise la qualité de répartition des ressources entre les slices. Ce paramètre est lié au niveau d'équité de notre mécanisme d'allocation.
- Facteur de stabilité: il caractérise le rapport entre la stabilité et la vitesse de réponse du système. Ce paramètre est lié au niveau de stabilité de note mécanisme d'allocation des ressources.

Dans notre évaluation, la perturbation du trafic est obtenue en augmentant ou diminuant le débit du trafic avec un pourcentage particulier. Par exemple, pour un trafic de 10 Gbps, une perturbation de +25 % consiste à changer le débit du trafic à 12.5 Gbps.

5.2 Résultats d'expérimentations

Nous cherchons à évaluer les algorithmes des différents niveaux de notre mécanisme d'allocation à travers un ensemble de tests unitaires pour chaque niveau sous des configura-

tions particulières. L'objectif consiste à évaluer les caractéristiques listées préalablement pour pouvoir extraire des conclusions et fournir des recommandations.

5.2.1 Équité

Nous cherchons à déterminer l'impact de la perturbation du trafic en fonction de la période d'exécution sur le temps de convergence. Pour l'évaluation, nous avons utilisé neuf flux de trafic de 0.5 Gbps pour les différents circuits de chaque slice. Avec les configurations matérielles de notre plan de données, ce débit met notre système dans un état de surcharge, c-à-d la demande dépasse la capacité disponible. Ceci nous permettra d'évaluer le premier niveau de notre mécanisme d'allocation. La fonction d'utilité à contrôler est la suivante:

$$\mathcal{U}_A(s_i) = \mathcal{U}_A(b_{i,1}) + \mathcal{U}_A(b_{i,2}) + \mathcal{U}_A(b_{i,3}) \text{ avec } \mathcal{U}_A(b_{i,k}) = \frac{\Gamma_{i,k}^{in} - \Gamma_{i,k}^{drop}}{\Gamma_{i,k}^{in}}$$

L'utilité d'une slice est la somme des utilités des différents circuits de cette slice. $\Gamma_{i,k}^{in}$ et $\Gamma_{i,k}^{drop}$ sont respectivement le nombre de paquets entrants et rejetés par congestion. Nous rappelons que l'équité (fairness) est donnée par:

$$\alpha = \frac{\min_{i\in\{1,\ldots,3\}} \mathcal{U}_A(s_i)}{\max_{i\in\{1,\ldots,3\}} \mathcal{U}_A(s_i)}$$

Sous un seuil d'équité de $\alpha = 95$ %, nous évaluons l'effet de la valeur de perturbation sur l'équité pour des périodes d'exécution différentes. Dans chaque expérimentation, une perturbation a été appliqué sur un circuit pour faire sortir le système de son intervalle de stabilité, tandis que le trafic sur le reste des circuits reste constant.

Sur les Figures 5.1 et 5.2, nous avons fixé la période d'exécution à $\tau_f = 2$ s et nous avons changé la valeur de la perturbation. Nous observons qu'avec l'augmentation de la perturbation, le niveau d'équité prend plus de temps pour remettre la système dans l'intervalle de stabilité.

Sur les Figures 5.3 et 5.4, nous avons fixé la période d'exécution à $\tau_f = 0.5$ s et nous avons appliqué deux perturbations à 50 % et 100 % respectivement. Nous observons également que le temps pour ramener le système dans son espace de stabilité entre 95 % et 100 % est proportionnel à la valeur de perturbation de trafic.

À travers les quatre configurations précédentes, nous observons qu'en diminuant la période d'exécution de l'algorithme, le mécanisme d'allocation prend plus de temps pour ramener le système à sa stabilité. Ceci est dû au fait que le système doit garder une nouvelle allocation

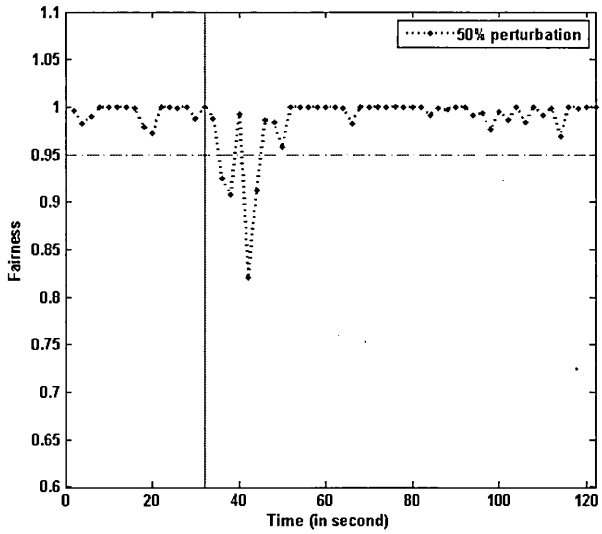

Figure 5.1: Équité pour une période de 2 s à 50 % de perturbation.

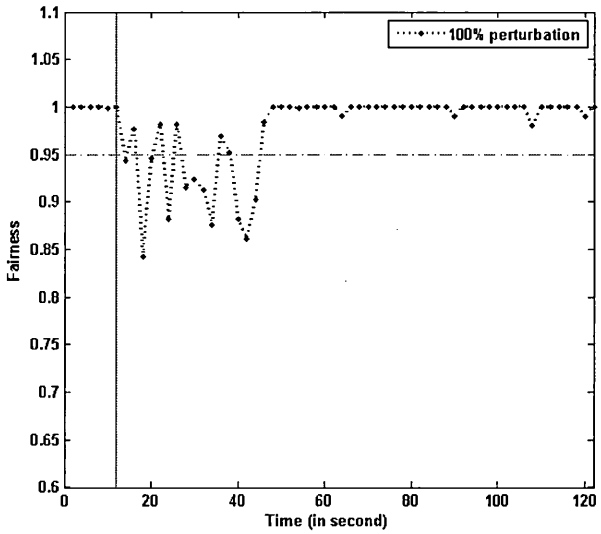

Figure 5.2: Équité pour une période de 2 s à 100 % de perturbation.

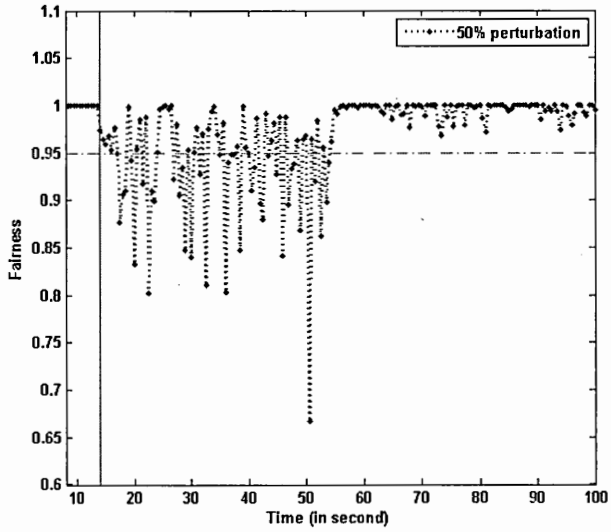

Figure 5.3: Équité pour une période de 0.5 s à 50 % de perturbation.

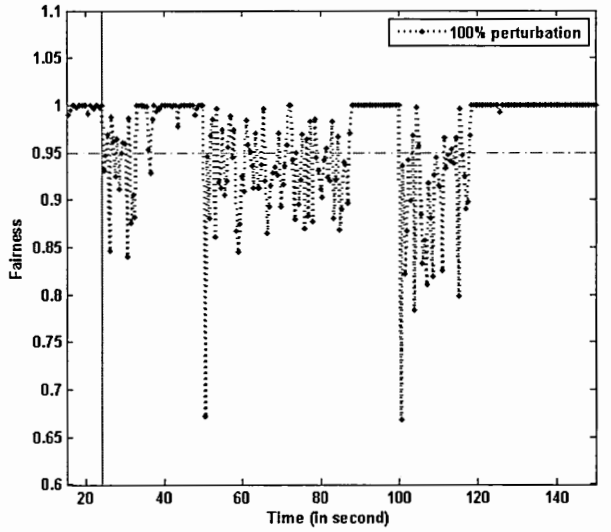

Figure 5.4: Équité pour une période de 0.5 s à 50 % de perturbation.

pour une durée suffisante pour pouvoir l'utiliser. Si l'allocation change rapidement, le système reste dans un état d'instabilité et le temps de convergence devient plus grand. Également, une faible période d'exécution force le mécanisme d'allocation à appliquer un nombre important d'allocation, ce qui augmente le nombre de paquets perdus.

5.2.2 Stabilité

Nous cherchons à montrer l'impact du facteur de stabilité et de l'intervalle de stabilité sur la convergence de notre système. Nous avons utilisé les même flux de trafic qu'auparavant pour surcharger notre système. Nous avons utilisé la même fonction d'utilité que le niveau d'équité pour la stabiliser autour d'une utilité cible de $\widetilde{\mathcal{U}}_B(s_i) = 50$ %. La fonction d'utilité s'écrit comme suit:

$$\mathcal{U}_B(s_i) = \mathcal{U}_B(c_{i,1}) + \mathcal{U}_B(c_{i,2}) + \mathcal{U}_B(c_{i,3}) \quad \text{Avec} \quad \mathcal{U}_B(c_{i,k}) = \frac{\gamma_{i,k}^{in} - \gamma_{i,k}^{drop}}{\gamma_{i,k}^{in}}$$

Nous travaillons sur la même linecard (carte Netronome), alors nous avons $\gamma_{i,k}^{in} = \Gamma_{i,k}^{in}$, $\gamma_{i,k}^{drop} = \Gamma_{i,k}^{drop}$ et $c_{i,k} = b_{i,k}$.

Avec une période d'exécution de $\tau_s = 0.5$ s, nous évaluons l'effet du facteur de stabilité et de l'intervalle de stabilité sur la convergence de notre système. Nous appliquons une perturbation de 50 % sur un circuit tandis que le trafic sur le reste des circuits est stable.

Sur les Figures 5.5 et 5.6, nous avons fixé le facteur de stabilité à $q = 0.00001$. Nous avons fixé l'intervalle de stabilité à respectivement 10 % et 20 %. Les utilités doivent être respectivement entre 40 % $\leqslant \mathcal{U}_B(s_i) \leqslant$ 60 % et entre 30 % $\leqslant \mathcal{U}_B(s_i) \leqslant$ 70 %. Nous observons qu'avec un intervalle étroit, la convergence devient plus difficile et ceci peut faire entrer le système dans une instabilité permanente.

Sur les Figures 5.7 et 5.8, nous avons fixé l'intervalle de stabilité afin de pouvoir évaluer l'effet du facteur de stabilité q. La valeur de q décide le comportement de convergence de l'utilité après une perturbation de trafic. Avec une valeur faible de q, nous avons observé une convergence rapide mais qui se caractérise par des variations très importante. En augmentant q, la vitesse de convergence diminue tandis que l'oscillation autour de la cible devient moins importante.

En observant ces quatre expérimentations, nous avons conclu qu'il y a une limite sur la largeur de l'intervalle de stabilité que le système peut supporter. Dû à la nature discrète de l'allocation des ressources, changer la performance se fait par tranche au lieu d'être continue.

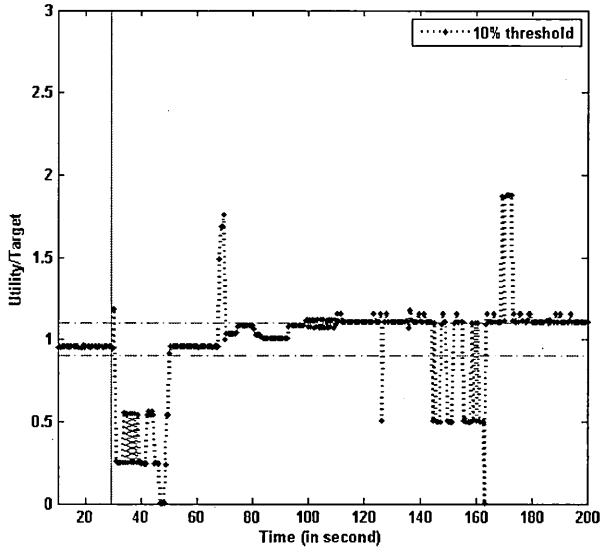

Figure 5.5: Ratio utilité/cible pour un intervalle de 10 %.

Par exemple, ajouter un thread à une slice donne une capacité de traitement de 140000 paquets/seconde. En conséquence, si le pas d'allocation est grand, l'intervalle de stabilité doit être suffisamment grand pour permettre au système de trouver un état de stabilité.

5.2.3 Prévention de blocages

Avec cette expérimentation, nous cherchons à montrer le comportement de répartition d'un bundle entre nos trois slices en fonction du trafic entrant. Le processus de fonctionnement du niveau de prévention de blocage est simpliste et consiste principalement à répartir les ressources d'un package d'une façon équitable en cas de surcharge. Pour ce niveau, nous avons utilisé la fonction d'utilité suivante:

$$\mathcal{U}_C(s_i) = \mathcal{U}_C(c_{i,1}) + \mathcal{U}_C(c_{i,2}) + \mathcal{U}_C(c_{i,3}) \quad \text{Avec} \quad \mathcal{U}_B(c_{i,k}) = \gamma_{i,k}^{in}$$

La Figure 5.9 montre le débit alloué avec le débit actuel à l'entrée pour chacune des slices. Sous une période d'exécution de 0.5 s, ce niveau répartit la capacité d'un bundle selon la demande. Nous observons qu'une fois la capacité allouée est consommée à $\approx$ 100 %, une

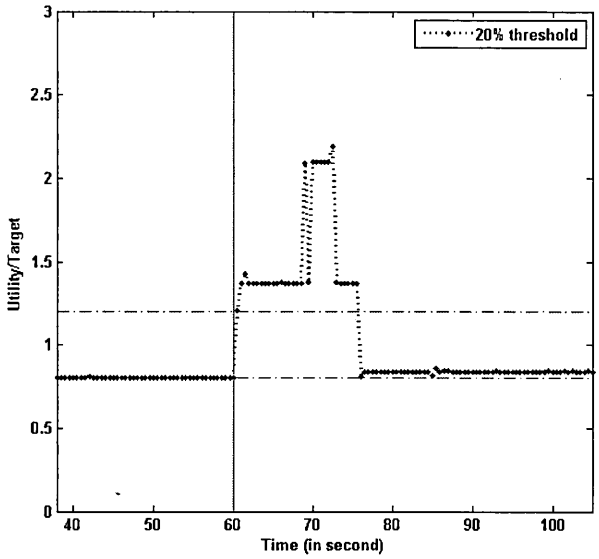

Figure 5.6: Ratio utilité/cible pour un intervalle de 20 %.

nouvelle allocation est calculée pour répondre au besoin d'une slice. Nous observons qu'en cas de congestion du bundle, la capacité de traitement est répartie équitablement entre les différentes slices.

5.3 Discussion des résultats d'expérimentations

Notre mécanisme d'allocation contrôle l'allocation des ressources entre différentes instances virtuelles implémentées sur une plate-forme matérielle. Selon la technologie de la plate-forme utilisée, le pas d'allocation change. Nous avons observé à travers les expérimentations précédentes que l'allocation se fait par tranche (ensemble de threads par exemple), ce qui est très critique à prendre en considération quand on spécifie l'intervalle de stabilité. Fixer un intervalle de stabilité très étroit par rapport au pas d'allocation force à système de rester dans un état d'instabilité permanente. Par conséquent, le pas d'allocation des ressources de la plateforme utilisée permet de donner la largeur minimale à avoir dans les intervalles de stabilité. D'une autre part, la largeur maximale de cet intervalle est proportionnelle à l'état du trafic entrant dans le système. Si le système reçoit un trafic trop perturbé, il est préférable d'élargir l'intervalle

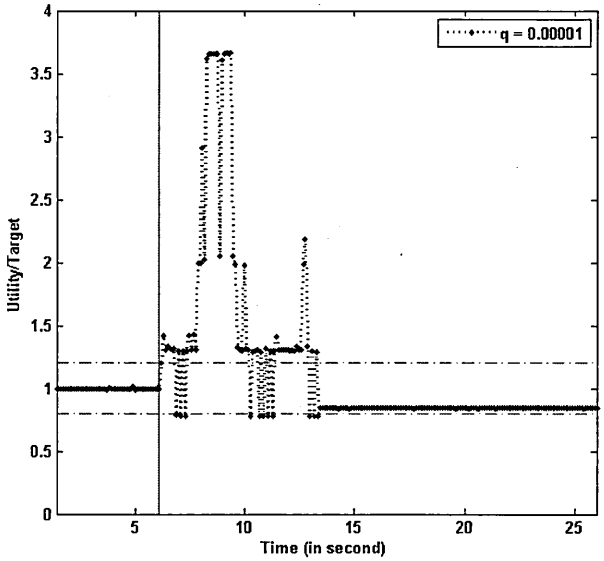

Figure 5.7: Ratio utilité/cible pour $q = 0.00001$.

de la stabilité pour ne pas forcer le mécanisme d'allocation à appliquer des nouvelles configurations trop fréquemment. Ceci amène à un taux de perte de paquets important et empêche le système de trouver son rythme après une nouvelle allocation.

La perturbation du trafic est également liée à la période d'exécution du mécanisme d'allocation. Ramener le système à son intervalle de stabilité se fait graduellement et ceci prend un temps proportionnel à la valeur de la perturbation. Si la perturbation est importante, le temps de convergence devient important. La période d'exécution doit être choisie de telle sorte à permettre au système d'utiliser chaque nouvelle allocation pour garantir sa convergence. Selon le vecteur de ressource à configurer, la période d'exécution ne doit pas être inférieure à la somme des temps de configuration des différentes ressources. Également, la période d'exécution ne doit pas être trop grande. Ceci risque d'impacter la réactivité du mécanisme d'allocation au changement de trafic du système.

Au cours d'une nouvelle configuration, des pertes de paquets peuvent être inévitable. Pour minimiser ces pertes, il est recommandé d'avoir des buffers de paquets suffisamment larges. Nous avons mentionné qu'au moment d'une configuration, les slices ou les circuits à configurer sont

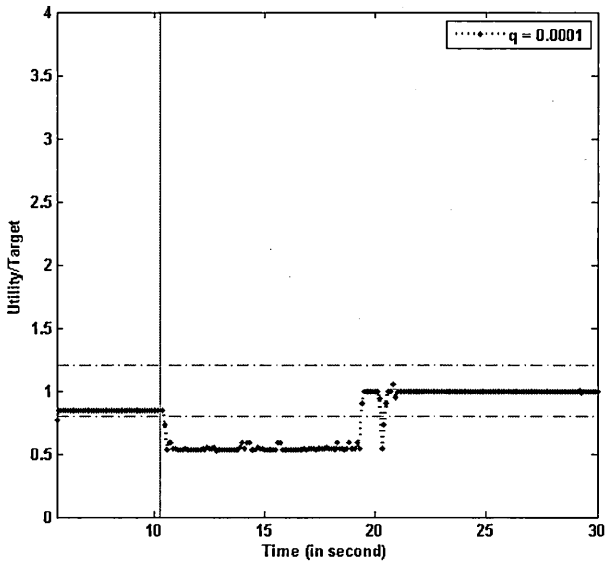

Figure 5.8: Ratio utilité/cible pour $q = 0.0001$.

arrêtés. Si le débit est important, le système est obligé de rejeter des paquets dans le cas des petits buffers. La taille des buffers doit être proportionnelle au débit que le système doit supporter. Par contre, la taille que le système peut offrir aux buffers est limitée, donc il faut toujours prévoir un taux de perte dans des cas extrêmes.

5.4 Conclusion

Contrairement aux plateformes de simulation, implémenter notre mécanisme d'allocation de ressources sur une plateforme réseau présente un grand défi dû à la nature matérielle de l'environnement de déploiement. Par conséquent, il est nécessaire de fournir des évaluations de notre mécanisme dans un tel environnement pour pouvoir observer son comportement, la corrélation entre différents paramètres et l'effet de la plateforme sur les résultats d'allocation des différents niveaux. Dans ce chapitre, nous avons fourni un ensemble d'expérimentations pour l'évaluation de notre mécanisme d'allocation. Nous avons mis en évidence l'impact de différents paramètres, à savoir la période d'exécution, l'intervalle de stabilité et le facteur de stabilité. Pour la période d'exécution, nous avons conclu qu'elle est inversement proportionnelle

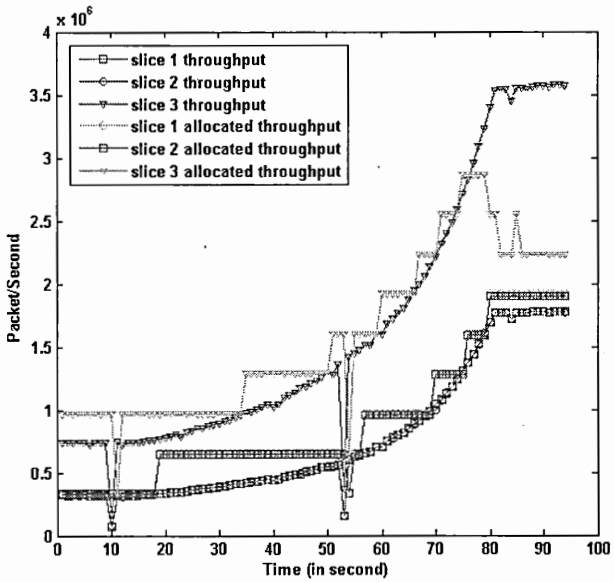

Figure 5.9: Allocation de capacité de traitement de paquets entre trois slices.

à la rapidité de convergence d'un système virtualisé après une perturbation de trafic. Ceci est dû au temps que le système prend pour utiliser efficacement une nouvelle allocation. D'une autre part, nous avons observé la largeur de l'intervalle de stabilité affecte également la convergence du système après une perturbation à cause de l'allocation des ressources qui ce fait par tranche. Finalement, nous avons mis en évidence l'impact du facteur de stabilité sur le comportement de convergence vers l'état de stabilité. Nous avons montré que l'architecture matérielle de la plate-forme utilisée a un rôle important dans le dimensionnement de ces paramètres. Il est crucial de faire une évaluation préalable de la plateforme physique à virtualiser. Il faut s'assurer que la plateforme offre une abstraction logicielle qui permet d'accéder à toutes les ressources et les configurer afin de pouvoir appliquer les résultats d'allocation. Pour finir, nous avons fourni des conseils aidant à choisir la bonne valeur de ces paramètres afin d'assurer une convergence optimale.

CONCLUSION

En se basant sur le mécanisme d'allocation dans le projet NetVirt, nous avons proposé un nouveau mécanisme d'allocation permettant la distribution des ressources physiques d'une plateforme réseau virtualisée entre un ensemble des slices. Notre mécanisme propose une stratégie d'allocation basée sur des packages de ressources, ou bundle, permettant d'agir simultanément sur des groupements de ressources plutôt que sur les ressources physiques séparément. Les ressources physiques appartenant au même bundle possèdent des liens de dépendances fonctionnelles qui se déterminent à travers les chemins de traitement déployés sur le nœud virtualisé. Ces chemins de traitement contiennent un ensemble de tâches s'exécutant sur un regroupement de ressources, ce qui nécessite, logiquement, d'agir sur toutes les ressources de ce groupement afin d'ajuster le niveau de performance de ce chemin de traitement. Par conséquent, allouer une fraction d'un bundle à une slice offre automatiquement une fraction de chacune des ressources physiques de ce bundle. Par notre formulation d'allocation basée sur la structure hiérarchique à trois niveaux, notre mécanisme a permis d'assurer l'équité, la stabilité et la prévention des congestions dans un noeud virtualisé en tenant compte de notre stratégie basée sur les packages de ressources.

Dans le but d'évaluer les différents blocs de notre mécanisme, nous avons développé une implémentation matérielle d'un nœud virtualisé. Notre implémentation consiste en plusieurs nœuds virtuels sur une plateforme matérielle basée sur un processeur réseau de Netronome. Elle utilise trois types de nœud: commutateur OpenFlow, routeur niveau trois et commutateur niveau deux. À travers cette implémentation, nous avons observé qu'introduire la virtualisation sur notre plateforme virtualisée entraîne une réduction de performance d'environ 50% par rapport à un environnement non-virtualisé. Ceci est dû à l'ensemble des blocs de statistiques, de configuration et d'isolation à ajouter pour pouvoir supporter un mécanisme d'allocation de ressources. Par contre, cette réduction de performance peut être compensée en exploitant les ressources généralement perdues, ce qui permet de ne pas affecter le gain de la virtualisation.

Cet environnement virtualisé a été utilisé pour évaluer notre mécanisme d'allocation à travers plusieurs tests unitaires. À partir de ces tests, nous avons observé qu'il y a un impact

important de l'architecture de notre plateforme sur l'efficacité des différents algorithmes. Nous avons remarqué que l'allocation des ressources se fait par tranche. Cette nature discrète de partage de ressources doit être prise en considération pour spécifier les intervalles de stabilité qui permettent au système de converger après une perturbation de trafic. Également, nous avons observé que la période d'exécution des algorithmes doit être choisie en fonction de la perturbation de trafic que le système peut subir. Une grande perturbation avec une courte période d'exécution ne donne pas au système le temps de bénéficier d'une nouvelle allocation et, par conséquent, assurer un temps de convergence raisonnable. Finalement, nous avons remarqué que la dimension des buffers de paquets est critique pour minimiser les pertes de notre mécanisme d'allocation. Les buffers doivent être dimensionnés en fonction du trafic que le système doit supporter ainsi que le nombre de ressources qui doivent être configurées. Ceci est dû au fait que le système doit être mis en suspens durant une nouvelle allocation, ce qui laisse des paquets en attente.

Nos expérimentations ont permis de mettre en évidence les défis qu'un mécanisme d'allocation de ressources rencontre afin d'être déployé sur une plateforme réseau matérielle. Actuellement, un nombre important d'approches peuvent bénéficier de notre contribution. *Network Function Virtualization* (NFV) est l'une de ces approches réseau émergentes reposant principalement sur la virtualisation en donnant une abstraction des fonctions de traitement des paquets, ou *Virtual Network Function* (VNF), d'un nœud réseau. En principe, NFV encourage l'utilisation des systèmes informatiques génériques, à savoir les serveurs, comme nœud réseau dans une infrastructure IT. Bien que cette démarche permette de réduire la complexité de déploiement et d'extension, elle affecte la performance du réseau en utilisant des architectures génériques à la place des architectures réseau performantes. Avec notre approche, NFV peut bénéficier de l'abstraction en bundle/circuit qui peut être utilisée comme VNFs, avec la capacité de faire une gestion des ressources entre les différentes instances qui utilisent ces VNFs. Par contre, notre approche doit être ajustée pour permettre la gestion des ressources avec une visibilité sur tous les nœuds d'une topologie réseau au lieu d'avoir une visibilité locale sur un nœud réseau. Cette ajustement va être introduite principalement dans notre formulation. Notre formulation doit donc tenir compte d'une autre dimension, à savoir les nœuds réseau, ainsi que les liens physiques entre les différents nœuds.

APPENDICE A

ALGORITHMES

Dans cette annexe, nous présenterons les pseudo-codes des algorithmes principaux que nous avons utilisé dans note mécanisme d'allocation. Nous présenterons l'algorithme *Best-Response*, *Recursive-Least Squares* et *Max-Min* que nous avons utilisé respectivement pour l'équité, la stabilité et la prévention de blocages.

A.1 Algorithme Best-Response

Le pseudo-code 1 donne le processus de l'algorithme Best-Response utilisé dans le niveau d'équité.

Algorithm 1 Best-Response algorithm

Require: $\{Q_1, Q_2, ...\}$ {bundles price}

Require: $\widetilde{X}_i$ {initial budget}

Require: $\{\widehat{A}'_{i,1}, \widehat{A}'_{i,2}, ...\}$ {bundles bid}

$X_i \leftarrow \sum_j \widehat{A}'_{i,j} + \widetilde{X}_i$

Sort bundles with price $\{Q_1, Q_2, ...\}$

Compute the largest k such as:

$\frac{\sqrt{Q_k}}{\sum_{j=1}^{k} \sqrt{Q_j}}(X_i + \sum_{j=1}^{k} Q_j) - Q_k \geqslant 0$

Set $\widehat{A}_{i,j} = 0$ for $j > k$ and for $1 \leqslant j \leqslant k$ set

$\widehat{A}_{i,j} = \frac{\sqrt{Q_j}}{\sum_{l=1}^{k} \sqrt{Q_l}}(X_i + \sum_{l=1}^{k} Q_l) - Q_j$

A.2 Algorithme Max-Min

L'algorithme Max-Min utilisé dans le niveau de prévention de blocages est donné par le pseudo-code 2.

Algorithm 2 Max-Min algorithm

Require: $c_{j,k}$ {bundle to control}

Require: $D_{j,k}$ {ressource capacity of the bundle}

Require: $\{\widehat{Z}_{1,j,k}, \widehat{Z}_{2,j,k}, ..., \widehat{Z}_{N,j,k}\}$ {slices resource request}

Require: $\{\widehat{\gamma}_{1,j}, \widehat{\gamma}_{2,j}, ..., \widehat{\gamma}_{N,j}\}$ {slices trafic perturbation}

$x \leftarrow N$

$y \leftarrow D_{j,k}$

while $x > 0$ **do**

if $\widehat{\mathcal{U}}_C(c_{j,k}, \widehat{Z}_{x,j,k}, \widehat{\gamma}_{x,j}, o_l) \leqslant \widehat{\mathcal{U}}_C(c_{j,k}, y, \sum_{i=1}^{N} \widehat{\gamma}_{i,j}, o_l)/x$ **then**

$Z_{x,j,k} \leftarrow \widehat{Z}_{x,j,k}$

else

$Z_{x,j,k}$ such that $\widehat{\mathcal{U}}_C(c_{j,k}, Z_{x,j,k}, \widehat{\gamma}_{x,j}, o_l) = \widehat{\mathcal{U}}_C(c_{j,k}, y, \sum_{i=1}^{N} \widehat{\gamma}_{i,j}, o_l)/x$

end if

$y \leftarrow y - Z_{x,j,k}$

$x \leftarrow x - 1$

end while

A.3 Algorithme Recursive-Least Squares

Le processus de l'algorithme Recursive-Least Squares utilisé dans le niveau de stabilité est donné par le pseudo-code 3 (RLS avec bruit nul).

Algorithm 3 Recursive-Least Squares algorithm

Require: $\{\mathcal{U}_B(t-1), \mathcal{U}_B(t-2), ...\}$ {past utility observations}

Require: $\{G(t-1), G(t-2), ...\}$ {past resource allocation vectors}

$U(L) = [\mathcal{U}_B(1), \mathcal{U}_B(2), ..., \mathcal{U}_B(L)]$

$\widehat{\varphi}(t) = [\mathcal{U}_B(t-1), \mathcal{U}_B(t-2), ..., G(t-1), G(t-2), ...]$

$\widehat{\Phi}(L) = [\widehat{\varphi}(1), \widehat{\varphi}(2), ..., \widehat{\varphi}(L)]$

$\widehat{\theta} \leftarrow [\widehat{\Phi}(L).\widehat{\Phi}^T(L)]^{-1}.\widehat{\Phi}(L).U(L)$ with $\widehat{\theta} = [a_1, a_2, ..., b_1, b_2, ...]$

APPENDICE B

PROFIL DE LA SLICE

Dans cette annexe, nous présenterons la description XML à utiliser pour définir l'environnement virtualisé. Nous illustrerons les différents éléments utilisés pour décrire les slices et le mappage sur les ressources disponibles. Nous décrirons également les éléments du linecard qui fournit l'ensemble des ressources matérielles à partager entre les slices.

B.1 Description de la slice

Ci-dessous la description XML d'une slice. Il consiste principalement de trois éléments: les ports d'interfaçage, les ressources et les circuits.

```
<slice id="v0" name="slice A" description="all resources are dedicated">
 <ports-interfaces>
 <resources>
 <circuits>
</slice>
```

B.1.1 Ports d'interfaçage

La description des ports d'interfaçage spécifie l'ensemble des interfaces qu'une slice peut utiliser. Chaque description de port d'interfaçage donne le mappage entre un port virtuel *id* et le port physique *port-interface-ref* du linecard. Il indique également le mode de partage *share-mode* port spécifié.

```
<port-interface id="v0pi1" port-interface-ref="m0fpp1"
        max-oversubscribing-ratio="2:1" share-mode="dedicated"/>
```

B.1.2 Ressources

La description des ressources liste l'ensemble des ressources qu'une slice peut exploiter. Chaque description de ressource *id* spécifie l'indexe de la ressource physique *UPC-resource-id* à utiliser, le module linecard *UPC-id* de la ressource physique, le mode de partage *share-mode* et la limite *limit* de la ressource virtuelle sur de la ressource physique.

```
<resource id="v0r7" name="L2-Engine" UPC-id="m0"
    UPC-resource-id="m0fe012e0" share-mode="dedicated"
    limit="30" limitUnit="Mpps" packet-processing-id="pp1"/>
```

B.1.3 Circuits

La description des circuits fournit les éléments des différents chemins de traitement d'une slice. Chaque description d'un circuit *id* donne sa direction *direction* dans le routeur, les ports virtuels *ingress-port-ref* sur lequel est mappés et l'ensemble des tâches du circuit. Chaque tâche est définie une ressource *resources* avec la mémoire d'entrée *input_mem* et la mémoire de sortie *output_mem*.

```
<circuit id="v0c0" direction="south-north">
 <ingress-port id="v0c0ip0" ingress-port-ref="v0pi1"/>
 <ingress-port id="v0c0ip1" ingress-port-ref="v0pi2"/>
  <task-processing>
   <task id="v0c0t0" resource="v0r7" name="parse_MAC_DST"
   input_mem="FMEM" input_offset="FMEM_MAC_DST_OFFSET"
   input_size="FMEM_MAC_DST_SIZE" output_size="MAC_DST_SIZE"
   output_mem="KMEM" output_offset="0"/>

   <task id="v0c0t1" resource="v0r6" name="lookup_MAC_Table"
   input_mem="KMEM" input_offset="KMEM_MAC_DST_OFFSET"
   input_size="MAC_DST_SIZE" output_mem="OREG"
   output_offset="PORT_NUM_OFFSET" output_size="PORT_NUM_SIZE"/>

   <task id="v0c0t2" resource="v0r3" name="lookup_VLAN_Table"
   input_mem="KMEM" input_offset="0" input_size="VLAN_SIZE"
```

```
    output_mem="OREG" output_offset="PORT_NUM_OFFSET"
    output_size="PORT_NUM_SIZE"/>

    <task id="v0c0t3" resource="v0r1" name="lookup_VRF_Table"
    input_mem="KMEM" input_offset="{KMEM_VLAN_SIZE+KMEM_IP_DST_OFFSET}"
    input_size="{VLAN_SIZE+IP_DST_OFFSET}" output_mem="OREG"
    output_offset="PORT_NUM_OFFSET" output_size="PORT_NUM_SIZE"/>

    <task id="v0c0t5" resource="v0r8" name="forward_outputport"
    input_mem="RMEM" input_offset="RMEM_OUTPUT_PORT_OFFSET"
    input_size="RMEM_OUTPUT_PORT_SIZE" output_mem="SEND_REG2"
    output_offset="sSend_byDstPort"
    output_size="sSend_byDstPort_SIZE"/>
  </task-processing>
</circuit>
```

B.2 Description du linecard

Le linecard représente la plate-forme matérielle qui forme le module sur lequel les slices sont instanciés. Une routeur peut avoir plusieurs linecards interconnectés par un switch-fabric. Nous donnerons un exemple utilisant le module d'évaluation de EZchip pour avoir un routeur à un seul linecard. Nous donnerons la description des ressources principales dans un linecard.

B.2.1 Engin de traitement de paquets

Ci-dessous la description d'un engin de traitement de paquets. Elle fournit principalement la capacité de traitement en *Mpps* (Million de paquet par second) avec son mode de partage. Les engins de paquets peuvent être de type couche 2 ou couche 3 (L2 ou L3).

```
<L2-Engine id="m1fe0l2e0" bandwidth="30" bandwidthUnit="Mpps"
share-mode="dedicated" config-mode="configurable"/>
```

B.2.2 Port d'interfaçage

La description ci-dessous donne la définition d'un port d'interfaçage réseau sur un linecard. Elle spécifie la vitesse de l'interface en *Gbps*.

```
<port-interface id="m1pi0" rate="10" rateUnit="Gbps"/>
```

B.2.3 Tables de recherche

Ci-dessous la description d'une structure de recherche sur un linecard. La description fournit la capacité de la structure *entries-number*, son mode de partage et le type d'application *type-application* qui peut l'utiliser. Une table de recherche peut être de plusieurs types, à savoir TCAM de classification, FIB-TCAM de IPv4, TCAM de niveau L2 ou table de hachage MAC.

```
<Classification-TCAM entries-number="16000" config-mode="static"
share-mode="dedicated" id="m1t4" type-application="L3/L4-services"/>
```

BIBLIOGRAPHIE

[1] Armen S Asratian : *Bipartite graphs and their applications.* Numéro 131. Cambridge University Press, 1998.

[2] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson et Jennifer Rexford : In vini veritas: realistic and controlled network experimentation. *In ACM SIGCOMM Computer Communication Review*, volume 36, pages 3–14. ACM, 2006.

[3] Dimitri P Bertsekas, Robert G Gallager et Pierre Humblet : *Data networks*, volume 2. Prentice-Hall International, 1992.

[4] Sapan Bhatia, Murtaza Motiwala, Wolfgang Muhlbauer, Vytautas Valancius, Andy Bavier, Nick Feamster, Larry Peterson et Jennifer Rexford : Hosting virtual networks on commodity hardware. *Georgia Tech. University., Tech. Rep. GT-CS-07-10*, 2008.

[5] Argon Blaster : Flow simulator and traffic generator. http://www.argonblaster.com/blaster_product_brief.pdf, October 2012.

[6] Steven J Brams, Michael A Jones, Christian Klamler *et al.* : Better ways to cut a cake. *Notices of the AMS*, 53(11):1314–1321, 2006.

[7] Broadcom : High-performance packet switch fabric. http://www.broadcom.com/collateral/pb/88130-PB00-R.pdf, 2007.

[8] H Jonathan Chao : Next generation routers. *Proceedings of the IEEE*, 90(9):1518–1558, 2002.

[9] H Jonathan Chao et Bin Liu : *High performance switches and routers.* John Wiley & Sons, 2007.

[10] NM Chowdhury et Raouf Boutaba : A survey of network virtualization. *Computer Networks*, 54(5):862–876, 2010.

[11] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak et Mic Bowman : Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[12] Peter Cramton, Yoav Shoham et Richard Steinberg : Combinatorial auctions. 2006.

[13] András Császár, Gábor Enyedi, Gábor Rétvári et Markus Hidell : Converging the evolution of router architectures and ip networks. *Network, IEEE*, 21(4):8–14, 2007.

[14] John Comstock Doyle, Bruce A Francis et Allen Tannenbaum : *Feedback control theory*, volume 1. Macmillan Publishing Company New York, 1992.

[15] Michal Feldman, Kevin Lai et Li Zhang : The proportional-share allocation market for computational resources. *Parallel and Distributed Systems, IEEE Transactions on*, 20(8):1075–1088, 2009.

[16] Drew Fudenberg et Jean Tirole : Game theory. 1991, 1991.

[17] Ran Giladi : *Network processors: architecture, programming, and implementation.* Morgan Kaufmann, 2008.

[18] Aun Haider, Richard Potter et Akihiro Nakao : Challenges in resource allocation in network virtualization. *In 20th ITC Specialist Seminar*, volume 18, page 20, 2009.

[19] Daniel Hausman : Fairness and trust in game theory. *Unpublished paper, London School of Economics and University of Wisconsin*, 1998.

[20] Simon S Haykin : *Adaptive Filter Theory, 4/e*. Pearson Education India, 2005.

[21] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb et Jay Lepreau : Large-scale virtualization in the emulab network testbed. *In USENIX Annual Technical Conference*, pages 113–128, 2008.

[22] HP : Server virtualization technologies for x86-based hp blade system and hp proliant servers, technology brief, 3rd edition. http://h10032.www1.hp.com/ctg/Manual/c01067846.pdf, 2009.

[23] A Iggidr et M Bensoubaya : New results on the stability of discrete-time systems and applications to control problems. *Journal of mathematical analysis and applications*, 219(2):392–414, 1998.

[24] John Jannotti, David K Gifford, Kirk L Johnson, M Frans Kaashoek *et al.* : Overcast: reliable multicasting with on overlay network. *In Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, pages 14–14. USENIX Association, 2000.

[25] Juniper : Logical routers. http://www.juniper.net/techpubs/software/junos/junos85/feature-guide-85/id-11139212.html, 2010.

[26] W. Karpoff et B. Lake : Storage virtualization system and methods, février 15 2005. US Patent 6,857,059.

[27] Hassan K Khalil : *Nonlinear systems*, volume 3. Prentice hall Upper Saddle River, 2002.

[28] Paul Klemperer : Auction theory: A guide to the literature. *Journal of economic surveys*, 13(3):227–286, 1999.

[29] Kai A Konrad : Altruism and envy in contests: an evolutionarily stable symbiosis. *Social Choice and Welfare*, 22(3):479–490, 2004.

[30] Christian Korth : Game theory and fairness preferences. *In Fairness in Bargaining and Markets*, pages 19–34. Springer, 2009.

[31] Vijay Krishna : *Auction theory*. Academic press, 2009.

[32] Masahiro Kumabe et H Reiju Mihara : Computability of simple games: A complete investigation of the sixty-four possibilities. *Journal of Mathematical Economics*, 47(2): 150–158, 2011.

[33] Xavier León et Leandro Navarro : Limits of energy saving for the allocation of data center resources to networked applications. *In INFOCOM, 2011 Proceedings IEEE*, pages 216–220. IEEE, 2011.

[34] Xavier León Gutiérrez, Leandro Navarro Moldes *et al.* : Macro-economic regulation for workload redistribution in large-scale shared infrastructure. 2010.

[35] ALbert Lespagnol et Howard A Seid : Virtual private network, juin 16 1998. US Patent 5,768,271.

[36] Jonathan Levin : Fairness and reciprocity. 2006.

[37] Harold C Lim, Shivnath Babu, Jeffrey S Chase et Sujay S Parekh : Automated control in cloud computing: challenges and opportunities. *In Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 13–18. ACM, 2009.

[38] Xue Liu, Xiaoyun Zhu, Pradeep Padala, Zhikui Wang et Sharad Singhal : Optimal multivariate control for differentiated services on a shared hosting platform. *In Decision and Control, 2007 46th IEEE Conference on*, pages 3792–3799. IEEE, 2007.

[39] Dan C Marinescu : Cloud computing: Theory and practice. Rapport de projet, Working paper. Computer science Division, Department of electrical engineering & Computer science, university of Central florida, orlando, fl, 2012.

[40] Roger B Myerson : *Game theory: analysis of conflict.* Harvard university press, 2013.

[41] Akihiro Nakao : Flare: Open deeply programmable network node architecture, 2012.

[42] John Nash : Non-cooperative games. *The Annals of Mathematics*, 54(2):286–295, 1951.

[43] John F Nash *et al.* : Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36(1):48–49, 1950.

[44] Netronome : nfp-32xx flow processor. http://netronome.com/wp-content/uploads/2013/12/NFP-32xx-Product-Brief-10-22-12.pdf, October 2012.

[45] Noam Nisan : *Algorithmic game theory.* Cambridge University Press, 2007.

[46] LTIR Laboratory University of Quebec At Montreal : Netvirt project. http://www.netvirt.ca/, 2008.

[47] Martin J Osborne : *A course in game theory.* Cambridge, Mass.: MIT Press, 1994.

[48] Pradeep Padala, Kai-Yuan Hou, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal et Arif Merchant : Automated control of multiple virtualized resources. *In Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26. ACM, 2009.

[49] Matthew Rabin : Incorporating fairness into game theory and economics. *The American economic review*, pages 1281–1302, 1993.

[50] Madan Sathe : *Parallel graph algorithms for finding weighted matchings and subgraphs in computational science.* Thèse de doctorat, University of Basel, 2012.

[51] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown et Guru Parulkar : Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[52] Amit Singh : An introduction to virtualization. http://www.kernelthread.com/publications/virtualization/, 2009.

[53] P Skoldstrom et Kiran Yedavalli : Network virtualization and resource allocation in openflow-based wide area networks. *In Communications (ICC), 2012 IEEE International Conference on*, pages 6622–6626. IEEE, 2012.

[54] OpenFlow Switch Specification : Version 1.3. 1. *Open Networking Foundation*, 2012.

[55] Cisco Systems : Cisco ios high availability (ha) - in-service software upgrade (issu). http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6550/prod_ presentation0900aecd80456cb8.pdf, 2006.

[56] Cisco systems : Technical overview of virtual device contexts. http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9402/White_Paper_Tech_Overview_Virtual_Device_Contexts.pdf, June 2012.

[57] Jonathan S Turner : A proposed architecture for the geni backbone platform. *In Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*, pages 1–10. IEEE, 2006.

[58] Hal R Varian : Equity, envy, and efficiency. *Journal of economic theory*, 9(1):63–91, 1974.

[59] William Vickrey : Auctions and bidding games. *Recent advances in game theory*, 29:15–27, 1962.

[60] Mathukumalli Vidyasagar : *Nonlinear systems analysis*, volume 42. Siam, 2002.